UNIT – IV

FILES

- ✓ **A file is an external collection of related data treated as a unit.**
- ✓ The purpose of a file is to keep a record of data.
- ✓ We need to store data permanently since data present in the main memory is lost when the computer is shut down.
- ✓ Files are stored in secondary storage device like disk and tapes.
- ✓ When the computer reads the data it is moved from external device to memory and memory to external device.
- ✓ The movement of data uses a special work area called buffer.
- ✓ A buffer is a temporary storage area that holds data for a short period of time.
- ✓ The data is read from the file until it reaches end of file marker.
- ✓ The operating system should have a set of rules to name a file.
- ✓ A program reads or writes files need to know different information about the file like filename, position, etc.
- ✓ It has predefined file structure present in **stdio.h** header file and its type is FILE.
- ✓ **"A file is a place on the disk where a group of related data is stored"**.

STREAMS

- ✓ A stream is a sequence of characters.
- ✓ It can be data input and data output.
- ✓ It is associated with physical devices.

Text and Binary Streams

- ✓ There are two types of streams: Text and Binary.
- ✓ A text streams consists of sequence of characters divided into lines with each line terminated by a new line.
- ✓ A binary stream consists of a sequence of data values such as integers, real, etc.

Stream File Processing

- ✓ A file exists with a name known to operating system.
- ✓ A stream is an entity created by program.
- ✓ To use a file we must associate the stream with the file name.
- ✓ There are four steps for processing a file:
    - ✓ Creating a stream
    - ✓ Opening a file
    - ✓ Using the stream name
    - ✓ Closing the stream

Creating a Stream

Stream is created when it is declared. The declaration uses the FILE type.

FILE *file_pointer;

Opening a File

When a stream is created we associate it to file. We use a standard function to open a file.

FILE * file_pointer;

file_pointer = fopen("filename", "mode");

Using the Stream File

After creating a stream we can use the stream pointer in all functions to access the file for input or output.

Closing the Stream

When the file processing is completed then we close the file. Closing a file will break the link of the stream pointer of the file. It is done by using fclose() function.

fclose(file_pointer);

System Created Streams

- ✓ A terminal can be a keyboard or monitor.
- ✓ It can be a source or destination of a text stream.
- ✓ Three streams are created and associated to communicate with a terminal.
- ✓ The stream pointers are defined in the stdio.h header file. They are:
    - ✓ stdin – points to standard input stream
    - ✓ stdout – points to standard output stream
    - ✓ stderr – points to standard error stream
- ✓ The association is done automatically.

STANDARD LIBRARY INPUT / OUTPUT FUNCTIONS

- ✓ The stdio.h header file contains several input / output functions.
- ✓ They are grouped into eight categories.
    - ✓ File open / close
    - ✓ Formatted input / output
    - ✓ Character input / output
    - ✓ Line input / output
    - ✓ Block input / output
    - ✓ File positioning
    - ✓ System file operations
    - ✓ File status

FUNCTIONS FOR FILE HANDLING

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from the file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

File Open / Close Functions

If we want to store data in a file in the secondary memory, we must specify certain things about the file to the operating system. They are

1. **Filename**
2. **Data structure**
3. **Purpose**

Filename is a string of characters that make up a valid filename. Data structure of a file is defined as FILE in the library of standard I/O function definition. Therefore all files should be declared as type FILE before they are used. FILE is a defined data type. When we open a file we must specify what we want to do with the file. For example we may write data to the file or read the already existing data.

The General Format for Declaring and Opening a File

The fopen( ) function has the following prototype:

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

Example:

```
FILE * fp;
fp = fopen("TEST", "w");
```

The above code uses **fopen( )** to open a file named TEST for output. The first statement declares the variable fp as a pointer to the data type FILE. The second statement opens the file, named file name and assigns an identifier to the FILE type pointer fp. This pointer contains all the information about the file when subsequently used as a communication link between the system and the program. The second statement also specifies the purpose of opening this file. Mode can be one of the following:

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and append mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and append mode

The fopen function works in the following way.

- ✓ Firstly, it searches the file to be opened.
- ✓ Then, it loads the file from the disk and place it into the buffer.
- ✓ It sets up a character pointer which points to the first character of the file.

Both the filename and mode are specified as strings. They should be enclosed in double quotation marks. When trying to open the file, the following things may happen:

1. When the purpose is writing, a file with the specified name is created if the file does not exist. The contents are deleted if the file already exists.
2. When the purpose is appending, the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is reading and if it exists then the file is opened with the current contents safe. Otherwise an error occurs.

Closing a File: A file must be closed as soon as all operations on it have been completed. We have to close a file so that we can reopen the same file in different mode. The syntax is as follows:

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

Example:

```
FILE *fp1,*fp2;
fp1=fopen("salary.txt","r");
fp2=fopen("employee.txt","w");
…….
…….
fclose(fp1);
fclose(fp2);
```

All files are closed automatically whenever a program terminates. However closing a file as soon as you are done with it is good programming habit.

Terminal and File Character Input / Output Functions

getc and fgetc Functions

They read next character from the file stream which can be a user defined stream or stdin and convert it to integer. If read detects end of file it returns EOF. The syntax of these functions are:

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

putc and fputc Functions

They are used to write a character to the file stream specified which can be a user defined stream or stdout or stderr. The first parameter is a character and second parameter is the file pointer. The syntax of these functions are:

```
int putc(int ch, FILE *fp);
int fputc(int ch, FILE *fp);
```

Working with Strings: fputs( ) and fgets( )

In addition to **getc( )** and **putc( )**, C supports the related functions **fgets( )** and **fputs( )**, which read and write character strings from and to a disk file. These functions work just like **putc( ) and getc( )**, but instead of reading or writing a single character, they read or write strings. They have the following prototypes:

int fputs(const char **str*, FILE **fp*);
char *fgets(char **str*, int *length*, FILE **fp*);

The **fputs( )** function writes the string pointed to by *str* to the specified stream. It returns **EOF** if an error occurs.

The **fgets( )** function reads a string from the specified stream until either a newline character is read or *length*–1 characters have been read. If a newline is read, it will be part of the string (unlike the **gets( )** function). The resultant string will be null terminated. The function returns *str* if successful and a null pointer if an error occurs.

The following program demonstrates **fputs( )**. It reads strings from the keyboard and writes them to the file called TEST. To terminate the program, enter a blank line. Since **gets( )** does not store the newline character, one is added before each string is written to the file so that the file can be read more easily.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
    char str[80];
    FILE *fp;
    if((fp = fopen("TEST", "w"))==NULL)
    {
        printf("Cannot open file.\n");
        exit(1);
    }
    do
     {
        printf("Enter a string (CR to quit):\n");
        gets(str);
        strcat(str, "\n"); /* add a newline */
        fputs(str, fp);
    } while(*str!='\n');
    return 0;
}
```

Formatted File Input / Output Functions

fscanf() and fprintf() functions can be used with any text files. These functions behave exactly like **printf( )** and **scanf( )** except that they operate with files. The prototypes of **fprintf( )** and **fscanf( )** are:

int fprintf(FILE **fp*, const char **control_string*, . . .);
int fscanf(FILE **fp*, const char **control_string*, . . .);

where *fp* is a file pointer returned by a call to **fopen( )**. **fprintf( )** and **fscanf( )** direct their I/O operations to the file pointed to by *fp*.

The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

As an example, the following program reads a string and an integer from the keyboard and writes them to a disk file called TEST. The program then reads the file and displays the information on the screen. After running this program, examine the TEST file. As you will see, it contains human readable text.

```
/* fscanf() - fprintf() example */
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char s[80];
    int t;
    if((fp=fopen("test", "w")) == NULL)
    {
        printf("Cannot open file.\n");
        exit(1);
    }
    printf("Enter a string and a number: ");
    fscanf(stdin, "%s%d", s, &t); /* read from keyboard */
    fprintf(fp, "%s %d", s, t); /* write to file */
    fclose(fp);
    if((fp=fopen("test","r")) == NULL)
    {
        printf("Cannot open file.\n");
        exit(1);
    }
    fscanf(fp, "%s%d", s, &t); /* read from file */
    fprintf(stdout, "%s %d", s, t); /* print on screen */
    return 0;
}
```

BINARY INPUT / OUTPUT

TEXT VERSUS BINARY STREAMS

For text input/output, we convert a data type to a sequence of characters but for binary input/output we transfer data without changing memory representation.

Text and Binary Files

Files are used for storage and extraction of data. Files are divided into two types based on the storing format. They are as follows:

- ✓ Text files
- ✓ Binary files

Text Files

- ✓ The text file is a file in which data is stored using characters only.
- ✓ They are used to store textual information such as alphabets, numbers, special symbols, etc.
- ✓ This data is easily understood by humans.
- ✓ Actually the ASCII code of the textual characters is stored in text files. Since data in the storage device is stored in binary format hence the text file contents have to be converted first into binary format and then stored in the storage device.
- ✓ A text file can store different character sets such as
    - ✓ Upper case letters (A…Z)
    - ✓ Lower case letters (a...z)
    - ✓ Numeric characters (1,2,...9)
    - ✓ Special symbols (%,$,…)
    - ✓ Punctuation characters (, : ? …)
- ✓ The examples of text files are **C source program code** and files with **.txt** extension.
- ✓ Several operations that are used to manipulate data in text files with the help of in built functions of C language are creating a new file, open an existing file, reading a file, writing a file etc.

Binary files

- ✓ A binary file is a collection of data stored in internal format of the computer.
- ✓ They are read and written using block functions.
- ✓ The binary files stores information in binary form that is understood by the machine.

- ✓ The use of binary file will eliminate the conversion from text to binary format for storage purpose.
- ✓ The disadvantage of binary files is that it cannot be understood by the humans.
- ✓ Any data that is stored in the form of bytes is a binary file. Every **exe** file is a binary file.
- ✓ The examples of binary files are video streams, image files, etc. The operations performed on binary files are read, write and append with the help of inbuilt functions.

Differences between Text and Binary Files

Text Files

- ✓ All data in text file is human readable form
- ✓ Each line of data is ended with a new line character
- ✓ There is a special character called EOF at the end of the file

Binary Files

- ✓ Data are stored in same format as they are stored in memory
- ✓ There are no new line characters
- ✓ There is a special character called EOF at the end of the file

State of a File

- ✓ A file that is opened can be in read or write or error state.
- ✓ If we want to read then the file must be in read state
- ✓ If we want to write then the file must be in write state
- ✓ The error state result when error occurs. For example, if file is opened in write state and if we want to perform read.
- ✓ If the file is error state then we cannot read or write.
- ✓ To open a file in read state we must specify read mode in the open statement.
  - ✓ If file exists then read mode will be success otherwise it fails.
  - ✓ A file can be opened in write state using write or append mode.
- ✓ Apart from these three modes we also have the update mode. It allows both reading and writing. It is used with + sign.

Opening Binary Files

- ✓ The open operation for the binary files is the only change in the mode.
- ✓ The syntax is: **FILE *fopen(const char *filename, const char *mode);**

- ✓ The file name is the name of the file as first parameter and the second parameter is the mode that can be read, write or both and append
- ✓ To indicate that a file is binary, we add a binary indicator 'b' to the mode.
- ✓ The binary file modes are:
    - ✓ read binary (rb)
    - ✓ write binary (wb)
    - ✓ append binary (ab)
    - ✓ update read binary (rb+ or r+b)
    - ✓ update write binary (wb+ or w+b)
    - ✓ update append binary (ab+ or a+b)

Closing Binary Files

- ✓ The binary files are closed as they are needed.
- ✓ The syntax is: **int fclose(FILE *sp);**

STANDARD LIBRARY FUNCTIONS FOR BIRNARY FILES

Block Input / Output Functions

- ✓ These functions are used for read and write data to binary files
- ✓ When we read and write binary files the data are transferred as found in memory
- ✓ No format conversion is required
- ✓ The block read function is fread and block write function is fwrite

File Read: fread()

- ✓ The fread function reads a specified number of bytes from a binary file and places them into memory at the specified location
- ✓ The syntax is: **size_t fread(void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);**
- ✓ The first parameter, buffer is a pointer to the input area in memory. It is a generic pointer and any pointer type can be passed to the function
- ✓ The num_bytes and count are multiplied to determine how much data are to be transferred
- ✓ The size is determined by using sizeof operator and count is used when reading structures
- ✓ The last parameter is associated stream

File Write: fwrite()

- ✓ The fwrite function writes a specified number of items to a binary file
- ✓ The syntax is:

  size_t fwrite(const void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);

- ✓ The first parameter, buffer is a pointer to the output area in memory. It is a generic pointer and any pointer type can be passed to the function
- ✓ The num_bytes and count are multiplied to determine how much data are to be transferred
- ✓ The size is determined by using sizeof operator and count is used when reading structures
- ✓ The last parameter is associated stream

File Status Functions

- ✓ There are three status functions:
  - ✓ Test end of file
  - ✓ Test error
  - ✓ Clear error

Test end of file: feof function

- ✓ This function can be used to test for an end of file condition.
- ✓ It takes a file pointer as an argument and returns a non-zero integer value if all of the data from the specified file has been read and returns zero otherwise. The syntax is:

int feof(FILE *fp);

The following routine reads a file until the end of the file is encountered:

while(!feof(fp)) ch = getc(fp);

This method can be applied to text files as well as binary files.

Test error: ferror Function

- ✓ The **ferror( )** function determines whether a file operation has produced an error. The **ferror( )** function has this prototype,

  ✓ **int ferror(FILE *fp);**

- ✓ It also takes a file pointer as its argument and returns a non-zero integer value if an error has been detected upto that point during processing it and returns zero otherwise. Because each file operation sets the error condition, **ferror( )** should be called immediately after each file operation; otherwise, an error may be lost.

Positioning Functions

- ✓ It is used for two purposes:
  - ✓ Random processing of data in disk files
  - ✓ To change the file state

Rewind Files: rewind Function

- ✓ The **rewind( )** function resets the file position indicator to the beginning of the file specified as its argument. That is, it "rewinds" the file. Its prototype is:

  void rewind(FILE **fp*); where *fp* is a valid file pointer.

Current Location: ftell Function

- ✓ You can determine the current location of a file using **ftell( )**. Its prototype is:

  long int ftell(FILE **fp*);

- ✓ It returns the location of the current position of the file associated with *fp*. If a failure occurs, it returns –1.

Set Position: fseek() and Random-Access I/O

- ✓ It is used to move the file position to a desired location within the file. It takes the following form:

  int fseek(FILE *fp, long int offset, int position);

- ✓ Here '**fp'** is a pointer to the file, **'offset'** is a number or variable of type **'long int'**, and position is an integer variable. The offset specifies the positions (bytes) to be moved from the location specified by position.

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

- ✓ The Offset may be '+'ve meaning move forward, or '-' ve meaning move backward.

Example:

fseek(fp,0L,0)	Go to the beginning (Similar to rewind)
fseek(fp,m,0)	Move (m+1)th byte in the file from the beginning
fseek (fp,m,1)	Go forward by 'm' bytes from the current position
fseek (fp,-m,1)	Go backward by 'm' bytes from the current position
fseek(fp,-m,2)	Go backward by m bytes from the end

The getw and putw functions

The getw and putw are integer oriented functions. They are similar to getc and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of getw and putw are:

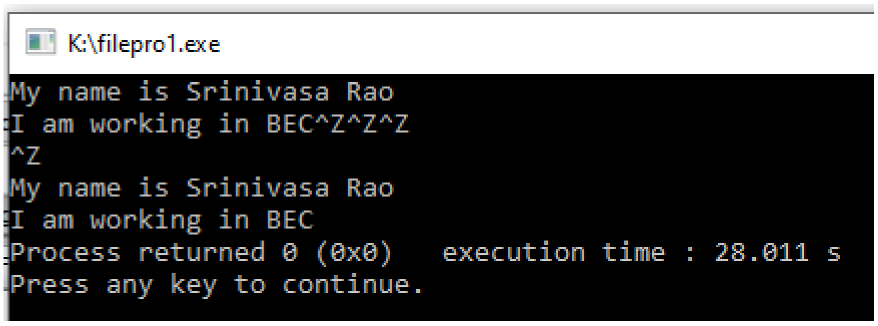
putw(integer, fp);

getw(fp);

/*Program to write contents to a file and display them on the screen*/

```
#include<stdio.h>
int main()
{
   FILE *fp;
   char ch;
   fp=fopen("data1.txt","w");
   while((ch=getchar())!=EOF)
     putc(ch,fp);
   fclose(fp);
   fp=fopen("data1.txt","r");
   while((ch=getc(fp))!=EOF)
     putchar(ch);
   fclose(fp);
   return 0;
}
```

OUTPUT:

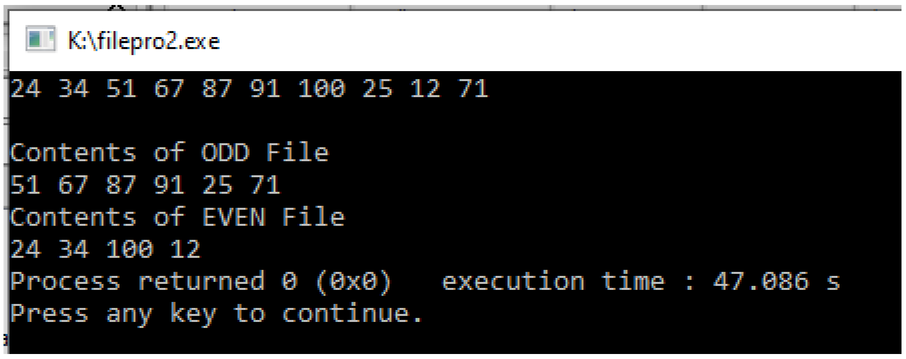

K:\filepro1.exe

```
My name is Srinivasa Rao
I am working in BEC^Z^Z^Z
^Z
My name is Srinivasa Rao
I am working in BEC
Process returned 0 (0x0)   execution time : 28.011 s
Press any key to continue.
```

/*Program to read the numbers from DATA file and then to write all odd numbers to ODD file and all even numbers to EVEN file*/

```
include<stdio.h>
int main()
{
   FILE *fp1,*fp2,*fp3;
   int x,i;
   fp1=fopen("DATA.txt","w");
   for(i=1;i<=10;i++)
   {
      scanf("%d",&x);
      putw(x,fp1);
   }
   fclose(fp1);
   fp1=fopen("DATA.txt","r");
   fp2=fopen("ODD.txt","w");
   fp3=fopen("EVEN.txt","w");
   while((x=getw(fp1))!=EOF)
   {
      if(x%2==0)
         putw(x,fp3);
      else
         putw(x,fp2);
   }
   fclose(fp1);
   fclose(fp2);
   fclose(fp3);
   fp2=fopen("ODD.txt","r");
   fp3=fopen("EVEN.txt","r");
   printf("\nContents of ODD File\n");
   while((x=getw(fp2))!=EOF)
      printf("%d ",x);
   printf("\nContents of EVEN File\n");
   while((x=getw(fp3))!=EOF)
      printf("%d ",x);
   fclose(fp2);
   fclose(fp3);
   return 0;
}
```

OUTPUT:



```
K:\filepro2.exe
24 34 51 67 87 91 100 25 12 71

Contents of ODD File
51 67 87 91 25 71
Contents of EVEN File
24 34 100 12
Process returned 0 (0x0)   execution time : 47.086 s
Press any key to continue.
```

/*Program that uses the functions ftell and fseek*/

```
#include<stdio.h>

int main()
{
   FILE *fp;
   long int n;
   char ch;
   fp=fopen("RANDOM.txt","w");
   while((ch=getchar())!=EOF)
     putc(ch,fp);
   printf("Number of characters entered = %ld\n",ftell(fp));
   fclose(fp);
   fp=fopen("RANDOM.txt","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);
   printf("\nPosition is %ld\n",ftell(fp));
   do
   {
      putchar(getc(fp));
   }while(!fseek(fp,-2L,1));
   fclose(fp);
   return 0;
}
```

OUTPUT:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
Number 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
```

/* Write a program to copy the contents of one file into another file*/

```
#include<stdio.h>
main()
{
	FILE *fs,*ft;
	char ch;
	fs=fopen(“Source.txt”, “r”);
	if(fs==NULL)
	{
		printf(“Source.txt file cannot be opened”);
		exit(0);
	}
	ft=fopen(“Target.txt”, “w”);
	if(ft==NULL)
	{
		printf(“Target.txt file cannot be opened”);
		fclose(fs);
		exit(0);
	}
	while((ch=getc(fs))!=EOF)
	{
		putc(ch,ft);
	}
	fclose(fs);
	fclose(ft);
	printf(“the file copy operation performed successfully”);
}
```

/* Write a program to read the list of integers from two different files and merge and store them in a single file/third file */

```
#include<stdio.h>
main()
{
	FILE *fs,*ft;
	int c;
	fs=fopen(“Source1.txt”, “r”);
	if(fs==NULL)
	{
		printf(“Source1.txt file cannot be opened”);
		exit(0);
	}
	ft=fopen(“Target.txt”, “w”);
	if(ft==NULL)
	{
			printf("Target.txt file cannot be opened");
			fclose(fs);
			exit(0);
		}
		for(; fscanf(fs, "%d", &c)!=EOF ;)
			fprintf(ft, "%d", c);
		fclose(fs);
		fclose(ft);
		fs=fopen("Source2.txt", "r");
		if(fs==NULL)
		{
			printf("Source2.txt file cannot be opened");
			exit(0);
		}
		ft=fopen("Target.txt", "r+");
		if(ft==NULL)
		{
			printf("Target.txt file cannot be opened");
			fclose(fs);
			exit(0);
		}
		fseek(ft,0L,2);
		for(; fscanf(fs, "%d", &c)!=EOF ;)
			fprintf(ft, "%d", c);
		fclose(fs);
		fclose(ft);
		printf("Files merged successfully now you can open target.txt file to check the
		contents");
}
```

MEMORY ALLOCATION FUNCTIONS

- ✓ There are two types of memory allocations: static allocation and dynamic allocation
- ✓ Static allocation refers to allocation of memory for a variable at compile time. They need declarations and definitions specified in the source program. The number of bytes reserved cannot be changed.
- ✓ Dynamic allocation refers to allocation of memory for a variable at run time. They use predefined functions to allocate and de-allocate memory for data while the program is running. Memory is allocated from heap through pointer.
- ✓ stdlib.h header file is included when dynamic memory allocation functions are used in the program.

Dynamic Memory Management

Consider an array: **int marks [100];**

Such a declaration is used to store 100 student's marks in memory. The moment we make declaration, 200 bytes are reserved in memory for storing 100 integers in it. However it may so happen that, when we actually run the program we might be interested in storing only 30 students' marks, which would result in wastage of memory.

Other way round there always exists a possibility that when you run the program you need to store more than 100 students' marks, in this case the array would fall short in size. Moreover there is no way to increase or decrease the array size during execution. This can done by dynamic data structures along with dynamic memory management techniques.

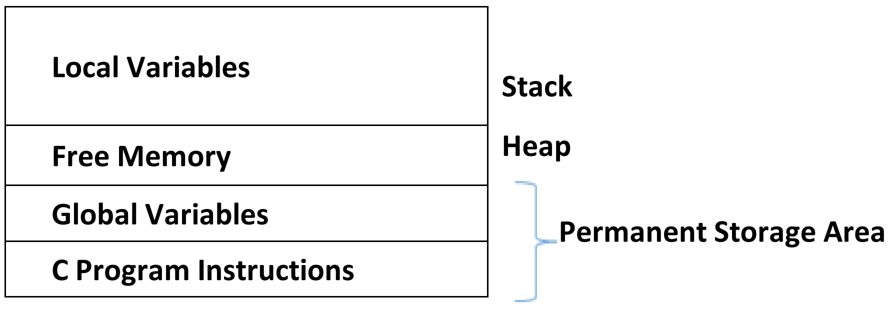
Dynamic Memory Allocation

The process of allocating memory at run time is known as **dynamic memory allocation.** The C programming does not have this feature. But it offers through memory management functions. The memory management functions are used to allocate and de-allocate storage during program execution. The functions are:

1. malloc 2. calloc 3. free 4. realloc

Memory Allocation Process

The storage for a C program is shown below:



The program instructions and the global variables as well as the static variables are stored in the region known as **permanent storage area**. The local variables are stored in the in the area called **stack**. The memory space between these two is the free area called **heap**.

The allocation or de-allocation of the space takes place from the heap. Therefore the size of the heap increases or decreases depending on the allocation and de-allocation of variables. If there is no space for allocation then the functions return **NULL**.

malloc()

This function is used to allocate one block of storage. It reserves the size specified by the pointer and returns void. The syntax for malloc is as follows:

ptr = (cast-type *) malloc (byte-size);

Example: **x = (int *) malloc (100 * sizeof(int));**

It is also used to allocate storage for complex data such as structures.

calloc()

This function is used to allocate multiple blocks of memory. It is used to store run time data types such as arrays and structures. It allocates storage for multiple blocks and initializes them all to zero and the pointer will point to the first byte of the first block. Here all blocks are same size. The syntax is as follows:

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

Example: **x = (int *) calloc(n, sizeof(int));**

free()

Whenever we are not in need of the variable for which the storage is allocated dynamically then such variable storage is removed. It is done by using **free** function. The syntax is as follows: **free (ptr);**

realloc()

Reallocation is done when:

1. Storage allocated to a variable is not sufficient so as to modify the variable's storage.
2. If the storage is allocated more than the required storage then also we need to alter the storage.

The process that is carried in the above situations is called **reallocation**.

ptr = realloc (ptr, newsize);

ARRAYS OF POINTERS

- ✓ Another useful structure that uses arrays and pointers is array of pointers.
- ✓ This is useful when the number of elements in the array is variable.
- ✓ For example,

32	18	12	24			
13	11	16	12	42	19	14
22						
13	13	14				
11	18					

- ✓ The table contains 5 rows and 7 columns. It is a 2-D array.
- ✓ It is called ragged array because it has different sized elements in each row
- ✓ If we use 2-D array we waste lot of memory
- ✓ The solution is to create a five 1-D arrays that are joined through array of pointers.
- ✓ For example the implementation is

  int **table;

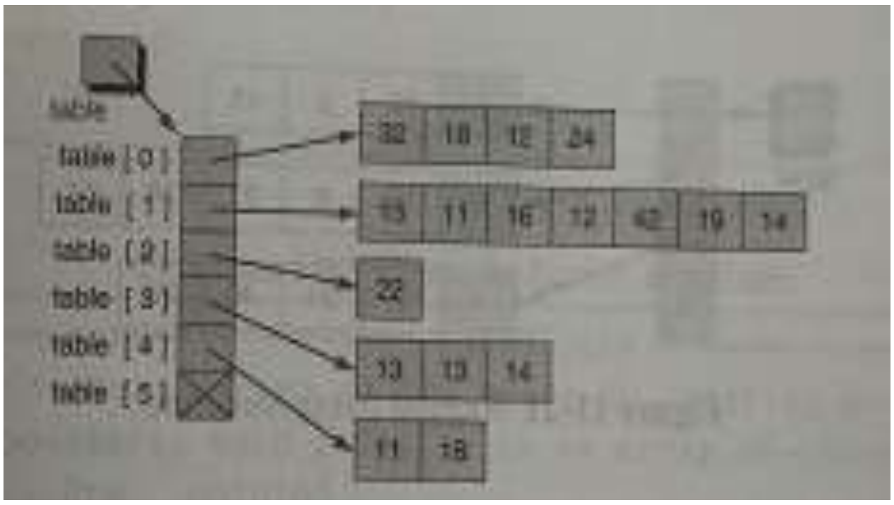


table = (int **) calloc(rownum+1, sizeof(int *));

table[0] = (int *) calloc(4, sizeof(int));

table[1] = (int *) calloc(7, sizeof(int));

table[2] = (int *) calloc(1, sizeof(int));

table[3] = (int *) calloc(3, sizeof(int));

table[4] = (int *) calloc(2, sizeof(int));

table[5] = NULL;

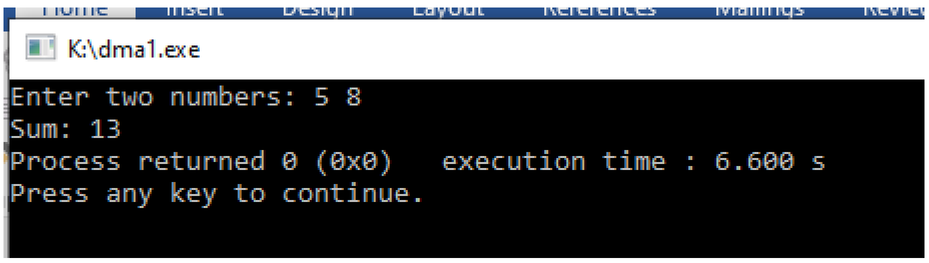
PROGRAMMING APPLICATION

1) To find sum of two numbers using dynamic memory allocation

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

```
int main()
{
   int *p,*q,*r;
   p=(int *)malloc(sizeof(int));
   q=(int *)malloc(sizeof(int));
   r=(int *)malloc(sizeof(int));
   printf("Enter two numbers: ");
   scanf("%d%d",p,q);
   *r=*p+*q;
   printf("Sum: %d",*r);
   free(p);
   free(q);
   free(r);
   return 0;
}
```

OUTPUT:



K:\dma1.exe

```
Enter two numbers: 5 8
Sum: 13
Process returned 0 (0x0)   execution time : 6.600 s
Press any key to continue.
```

2) To find the sum of n elements entered by user using dynamic memory allocation function calloc().

```
#include <stdio.h>
#include <stdlib.h>
#include<malloc.h>
int main()
{
        int n, i, *a, sum = 0;
        printf("Enter number of elements: ");
        scanf("%d", &n);
        a = (int*) calloc(n, sizeof(int));
        if(a == NULL)
        {
                printf("memory not allocated.");
                exit(0);
        }
        printf("Enter %d elements of array", n);
        for(i = 0; i < n; ++i)
        {
                scanf("%d", &a[i]);
```

```
            sum =sum+a[i];
        }
        printf("sum is %d\n", sum);
        free(a);
        return 0;
}
```

PREPROCESSOR COMMANDS

"A C preprocessor is a program that processes our source program before it is passed to the compiler".

- ✓ The compiler is made up of two components: a preprocessor and a translator.
- ✓ The preprocessor uses commands to prepare the source program for compilation.
- ✓ The translator converts the C statements into machine code.
- ✓ Both the preprocessor and translator work together.
- ✓ All preprocessors start with a pound sign(#).
- ✓ They can be placed anywhere but normally before main().
- ✓ There are three major tasks of preprocessor:
    - o File Inclusion,
    - o Macro Definition and
    - o Conditional Compilation

File Inclusion

- ✓ The file inclusion directive causes one file to be included into another file.
- ✓ The preprocessor command for the file inclusion looks as shown below:

  #include "filename"
- ✓ From the above syntax we say that the entire contents of the filename to be inserted into the source code at point in the program.
- ✓ When and why this feature is used?
- ✓ If we have very large program, the best way is to divide the code into different files each containing a set of related functions. These functions are **#included** at the beginning of the main program.
- ✓ There are some functions and macro definitions that are needed in all programs. They are stored in a file and that file is included in every program.
- ✓ The files that are included should have an extension **.h**. The Extension stands for header files.

- ✓ It is written at the beginning of the program. We can have two ways of writing the #include statements:

#include<filename>

#include"filename"

The meaning of the above two is as follows:

- ✓ #include<filename> means it searches only the list of the specified/standard directories
- ✓ #include"filename" means it searches not only in current directory but also in the specified list of directories.

Macro Definition

- ✓ A macro definition command associates a name with a sequence of tokens.
- ✓ The name is the macro name and the tokens are referred as macro body.
- ✓ It is of the form **#define name body**
- ✓ Consider the following program to explain about macro expansion

```
#include<stdio.h>
#define UPPER 25
main()
{
        int i;
        for(i=0;i<UPPER;i++)
                printf("%d",i);
}
```

In the above program instead of writing 25 in the for loop we have written UPPER which is already defined before main through the statement **#define UPPER 25**

This statement is called **macro definition** and simply we can call it as **macro** also. What is its purpose? During preprocessing the preprocessor replaces every occurrence of UPPER in the program with 25. In the above program **UPPER** is called the **macro template** and **25** is the **macro expansion**.

When we compile a program, before it is passed to the compiler it is examined by the preprocessor for any macro definitions. When the preprocessor sees the **#define** directive it goes through the entire program in search of macro template and replaces the macro template with appropriate macro expansion. Then it is handed to the compiler. There is a space between macro template and macro expansion. The space between # and define is optional.

Points to remember about #define directive

- ✓ A #define directive is many a times used to define operators.
- ✓ A #define directive could be used even to replace a condition.
- ✓ A #define directive could be used to replace even an entire C statement.

Macro with Arguments

Macros can have arguments just like functions have. To explain let us consider an example.

```
#include<stdio.h>
#define AREA(x) (3.14*x*x)
main()
{
        float r1=6.25,a;
        a=AREA(r1)
        printf("Area of the circle%f", a);
}
```

In the above program, when the preprocessor finds the AREA(x) it expands it into the statement (3.14*x*x). The variable r1 is substituted for x. Therefore the assignment would be **(3.14*r1*r1)**

Macros versus Functions

In macro call the preprocessor replaces the macro template with its macro definition. But in case of function call the control is transferred to the called function and perform some calculations and return some value back to the function.

Macros run the program faster but it uses bulk memory. But in case of functions, they run slowly and need only small memory.

If we use macro hundred times then macro expansion goes hundred times into the source program in search of the macro definition. But in functions, the function is written only once and but how many times you call it we refer the same function.

Nested Macros

- ✓ It is possible to nest macros.
- ✓ For example,

```
#define product(a, b)       (a) * (b)
#define square(a)           product(a, a)
```

The expansion of x = square(5); might result as x = product(5, 5);

then x = (5) * (5);

Undefining Macros

- ✓ Once a macro defined it cannot be redefined.
- ✓ Any attempt will result in error.
- ✓ It is possible to redefine a macro by first undefining it using #undef and defining it again.
- ✓ For example,

```
#define x 10
#undef x
#define x 20
```

Conditional Compilation

The conditional compilation preprocessor commands are:

#ifdef and #endif directives

The general format is

```
#ifdef macroname
        statement1;
        statement2;
#endif
```

#if and #elif directives

The #if is used to test an expression to evaluate zero or nonzero value. If the result of the expression is nonzero then the lines upto #else, #elif or #endif are compiled otherwise they are skipped.

```
main()
{
#if TEST<=5
        statement1;
        statement2;
#else
        statement3;
        statement4;
#endif
}
```

Other commands

Error command

It is used to print the message detected by the preprocessor. It is of the form:

#error message

#Pragma Directive

This directive is used to turn off or turn on certain features. Pragma varies from compiler to compiler. Some of these pragma directives are

#pragma startup and #pargma exit

These directives allow us to specify functions that are called upon program start up and program exit.

#pargma warn

On compilation the compiler reports errors or warnings in the program. Errors provide programmers with no options so that we have to correct them. Warnings on the other hand they offer the programmer a message.

#pragma warn directive tells the compiler whether or not we want to suppress a specific warning.

COMMAND LINE ARGUMENTS (argc and argv— Arguments to main( ))

Sometimes it is useful to pass information into a program when you run it. Generally, you pass information into the **main( )** function via command line arguments. A *command line argument* is the information that follows the program's name on the command line of the operating system. For example, when you compile a program, you might type something like the following after the command prompt,

C:\> ***program_name***

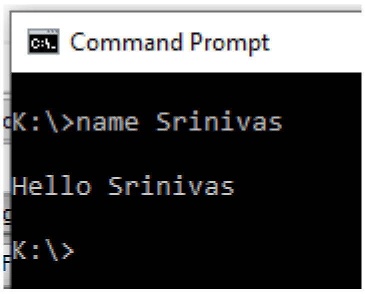
where *program_name* is a command line argument that specifies the name of the program you wish to compile.

Two special built-in arguments, **argc** and **argv**, are used to receive command line arguments. The **argc** parameter holds the number of arguments on the command line and is an integer. It is always at least 1 because the name of the program qualifies as the first argument. The **argv** parameter is a pointer to an array of character pointers. Each element in this array points to a command line argument. All command line arguments are strings— any numbers will have to be converted by the program into the proper binary format, manually.

Here is a simple example that uses a command line argument. It prints **Hello** and your name on the screen, if you specify your name as a command line argument.

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

int main(int argc, char *argv[])
{
    if(argc!=2)
    {
    printf("You forgot to type your name.\n");
    exit(1);
    }
    printf("\nHello %s\n", argv[1]);
    return 0;
}
```



```
K:\>name Srinivas

Hello Srinivas

K:\>
```

If you called this program **name** and your name was Srinivas, you would type **name Srinivas** to run the program. The output from the program would be **Hello Srinivas**.

In many environments, each command line argument must be separated by a space or a tab. Commas, semicolons, and the like are not considered separators.

Usually, you use **argc** and **argv** to get initial commands into your program that are needed at startup. For example, command line arguments often specify such things as a filename, an option, or an alternate behavior. Using command line arguments gives your program a professional appearance and facilitates its use in batch files.

The names **argc** and **argv** are traditional but arbitrary. You may name these two parameters to **main( )** anything you like.

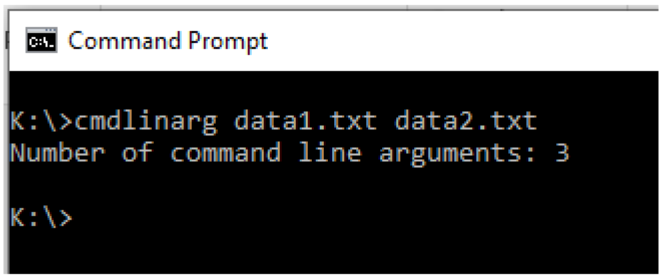
When a program does not require command line parameters, it is common practice to explicitly declare **main( )** as having no parameters. This is accomplished by using the **void** keyword in its parameter list.

/***Program to copy the contents of one file to another file using Command Line Arguments***/

```
#include<stdio.h>
int main(int argc,char *argv[])
{
  FILE *fp1,*fp2;
  char ch;
  printf("Number of command line arguments: %d\n",argc);
  fp1=fopen(argv[1],"r");
```

```
   fp2=fopen(argv[2],"w");
   while((ch=getc(fp1))!=EOF)
      putc(ch,fp2);
   fclose(fp1);
   fclose(fp2);
   return 0;
}
```

OUTPUT:



```
K:\>cmdlinarg data1.txt data2.txt
Number of command line arguments: 3

K:\>
```