unit 4 notes

FILE HANDLING

Introduction

Variables used in programs will die at the end of program execution. There may be instances where there is a need to display large data on the console. As the CPU memory is volatile, it is impossible to recover the programmatically generated data again and again. We use files to persist data even after the program execution is completed. **A file represents a sequence of bytes.** File handling process enables us to create, update, read, and delete the files stored on the local file system.

Example: User Login Credentials on Web page. The user ID and password entry is compared with information stored at the time of Registration.

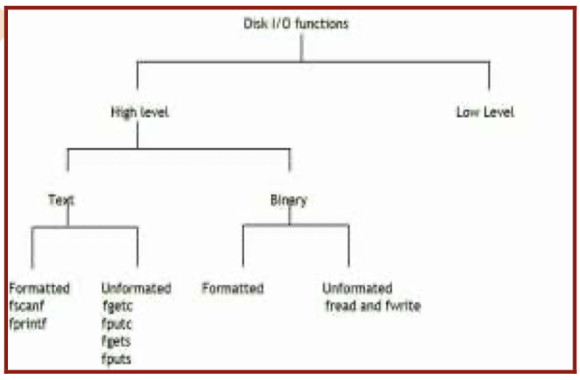
Need of Files

When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates. If you have to enter a large number of data, it will take a lot of time to enter them every time you run the code. However, if you have a file containing all the data, you can easily access the contents of file using few functions in C.

To check the output of the program several times is accomplished by running the same program multiple times. Usage of file addresses the problem of storing data in bulk. There are certain programs that require a lot of inputs from the user and can easily access any part of the code with the help of certain commands.

Classification of Files

Files can be classified as **Stream-oriented (High Level) data files** and **System oriented(Low Level) data files**



Stream oriented data files are of two types:

In the first category, the **data file** comprises consecutive characters. These characters can be interpreted as individual data items or as components of strings or numbers, which are called **text files**. In the second category, often called as unformatted data files, organizes data into blocks containing contiguous bytes of information. These blocks represent more complex data structures, such as arrays and structures, these files are hence called **binary files**

System-oriented data files: They are more closely related to the computer's OS than are stream-oriented data files. To perform the I/O from and to files, an extensive set of library functions are available in C. They are more closely related to the computer's OS than are stream-oriented data files

Text File

It comprises consecutive characters, which can be interpreted as individual data items or as components of strings or numbers. A text file contains only textual information like alphabets, digits and special symbols. In text files, the ASCII codes of alphabets, digits, special symbols are stored. A new-line character is converted into the carriage return-linefeed combination before being written to the disk, Likewise, the carriage return-linefeed combination on the disk is converted back into a new-line when the file is read by a C program. A special character, whose ASCII value is 26, is inserted after the last character in the file to mark the end of file(EOF), upon detecting this character at any point in the file, the read function would return the EOF signal to the program. The only function that is available for storing numbers in a disk file is the fprintf() function.

Binary File

Often called as unformatted data files, organizes data into blocks containing contiguous bytes of information, these blocks represent more complex data structures, such as arrays and structures. A binary file is merely a collection of bytes. If a file is opened in binary mode, as opposed to text mode, carriage return-linefeed conversions will not take place. There is no such special character present in the binary mode files to mark the end of file. The binary mode files keep track of the end of the file from the number of characters present in the directory entry of the file. The functions that are available for storing and retrieving numbers in a binary file are the fwrite() and fread() function.

Operations on Files

The data in a file can be **structured** – stored in the form of rows and columns, file name and values. The data can also be **unstructured** like social media posts. A program in C is itself data for the 'C' compiler. The keyboard and output screen are also considered as files. A file is maintained by the operating system. The operating system decides the naming convention of a file. This is referred to as the physical filename. In a program in a programming language like 'C', we use an identifier to refer to a file. This is called the logical name or the file handle. In a programming language like 'C', we use an **identifier to refer to a file.** This is called the **logical name or the file handle.** There is an opaque type (structure declared in stdio.h) called **FILE – which is a typedef**. This type varies from one implementation to another. We use **FILE* in our program so that our program will not depend on the layout of the type FILE.**

To perform any operation on a file, we connect the physical filename, the logical filename and the mode by using a function **fopen()**

Physical Name: A file is maintained by the OS. The OS decides the naming convention of a file

Logical Name: In a C Program, identifier is used to refer to a file. Also called as File Handle

Mode: Can be read only, write only, append or a combination of these.

Character I/O operations on File

fputc()

- The function is used to write at the current file position and then increments the file pointer.
- On success it returns an integer representing the character written, and on error it returns EOF.
- **Syntax:**int fputc(int c,FILE *fp);

fgetc()

- This function reads a character from the file and increments the file pointer position. On success it returns an integer representing the character written, and on error it returns EOF.
- **Syntax:**int fgetc(FILE *fp);

getc() and putc()

- The operation performed by getc() and put() functions is the same as that performed by fgetc() and fputc() functions.
- The difference is fgetc() and fputc() are the functions while getc() and putc() are macros.

String I/O operation on File

fputs()

- This function writes the null terminated string pointed to by str to a file.
- This null character that marks the end of string is not written to the file.
- On success it returns the last character written and EOF on error.
- **Syntax:**int fputs(const char *str, FILE *fp);

fgets()

- This function is used to read characters from the file and these characters are stored in the string pointed to by s.
- It reads n-1 characters from the file where n is the second argument.
- fp is a file pointer which points to the file from which character is read.
- This function returns of string pointed to by s on success, and NULL on EOF.
- **Syntax:**char *fgets(char *s,int n, FILE *fp);

fprintf()

- This function is the same as printf() function and writes formatted data into the file instead of the standard output.
- This function has the same parameter as printf() but has one additional parameter which is a pointer of FILE type.
- It returns the number of characters output to the file on success and EOF on error.
- **Syntax:** fprintf (FILE *fp, const char *format [, argument….]);

fscanf()

- This function is the same as scanf () function but it reads data from the file instead of the standard output.
- This function has a parameter which is a pointer of FILE type.
- It returns a number of arguments that were assigned some values on success and EOF on error.
- **Syntax:** fscanf(FILE *fp,const char *format[,address,…..]);

Block read / write operation on File

fwrite()

- The fwrite() function writes the data specified by the void pointer ptr to the file.
- On success, it returns the count of the number of items successfully written to the file,on error, it returns a number less than n.
- **Syntax:**size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp);

fread()

- fread() function is commonly used to read binary data.
- It accepts the same arguments as the fwrite() function does.
- The syntax of fread() function is as follows:
- **Syntax:** size_t fread(void *ptr, size_t size, size_t n, FILE *fp);
- The function reads n items from the file where each item occupies the number of bytes specified in the second argument.

Random access to file

Random access to a file means permitting non-sequential or random access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file. In order to access a particular file or record, C supports below functions.

fseek()

ftell()

rewind()

fseek()

- This function is used for seeking the pointer position in the file at the specified byte.
- **Syntax:** fseek(file pointer, displacement, pointer position);

  Where,

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

  displacement ---- It is positive or negative based on the number of bytes, skipped forward or backward position from the current position

  Pointer position -----This sets the pointer position in the file rom where displacement must happen

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

ftell()

- This function returns the value of the current pointer position in the file.
- The value is counted from the beginning of the file.
- **Syntax:** ftell(fptr);Where fptr is a file pointer.

rewind()

- The function is used to move the file pointer to the beginning of the given file.
- The function sets the file position to the beginning of the file for the stream pointed to by stream.
- The rewind function does not return anything.
- **Syntax:** rewind(fptr);where fptr is a file pointer.(pending)

code:to open a file

```
#include<stdio.h>

#include<stdlib.h>

int main(){

FILE *fp=NULL;

fp=fopen("my_file.txt","w");

if(fp==NULL){

printf("file doesnt exist");

exit(1);

}

else{

char s[]="preksha reddy";

fprintf(fp,"%s",s);

}

fclose(fp);

}
```

code 2: getc,putc

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

int main() {
    FILE *fp = NULL;
    char ch;
```

```
    fp = fopen("my_file.txt", "r");
    if (fp == NULL) {
        printf("File doesn't exist.\n");
        exit(1);
    } else {
        ch = getc(fp);
        while (ch != EOF) {
            putchar(ch); // or putc(ch, stdout);
            ch = getc(fp);
        }
        fclose(fp);
    }
    return 0;
}
```

code 3: fputc() and fgetc()

```
#include <stdio.h>

#include <stdlib.h>



int main() {

FILE *fp = NULL;

char ch;



fp = fopen("my_file.txt", "w");

if (fp == NULL) {

printf("File doesn't exist.\n");
```

```
exit(1);

} else {

while ((ch = getchar()) != EOF) {

fputc(ch, fp);

}

fclose(fp);

printf("Data is written successfully.\n");

}



return 0;

}
```

code 4: imppp to copy content from one file to another

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

int main() {
    FILE *source, *target;
    char ch;

    source = fopen("source.txt", "r");
    if (source == NULL) {
        printf("Error: Source file cannot be opened.\n");
        exit(1);
    }

    target = fopen("target.txt", "w");
    if (target == NULL) {
```

```
        fclose(source);
        printf("Error: Target file cannot be opened.\n");
        exit(1);
    }

    while ((ch = fgetc(source)) != EOF) {
        fputc(ch, target);
    }

    printf("File copied successfully.\n");

    fclose(source);
    fclose(target);

    return 0;
}
```

Coding Example_5: Illustrate fputs()

```
#include <stdio.h>
int main ()
{
        FILE *fp;
        fp = fopen("Cdocs.txt", "w+");
        fputs("This is c programming.", fp);
        fputs("This is a system programming language.", fp);
        fclose(fp);
        return(0);
}
```

Coding Example_6: Illustrate fgets()

```
#include <stdio.h>
int main()
{
        char buf[15];
        fgets(buf, 15, stdin); // read from keyboard – standard input
        printf("string is: %s\n", buf);
        return 0;
}
```

Coding Example_7: Illustrate fwrite()

Version1: Writing a variable

```
        float f = 100.13;
        fwrite(&p, sizeof(f), 1, p);
```

Version2: Writing an array

```
        int arr[3] = {101, 203, 303};
        fwrite(arr, sizeof(arr), 1, fp);
```

Version 3: Writing some elements of array

```
int arr[3] = {101, 203, 303};
fwrite(arr, sizeof(int), 2, fp);
```

Version 4: Writing structure

```
struct student
{
        char name[10];
        int roll;
         float marks;
};
struct student student_1 = {"Tina", 12, 88.123};
fwrite(&student_1, sizeof(student_1), 1, p);
```

Version 5: Writing array of structure

```
struct student {
        char name[10];
        int roll;
        float marks;
};
```

```
struct student students[3] = {
        {"Tina", 12, 88.123},
        {"Jack", 34, 71.182},
        {"May", 12, 93.713}
    };
fwrite(students, sizeof(students), 1, fp);
```

Coding Example_8: Illustrate ftell()

```
#include<stdio.h>
int main()
{
   /* Opening file in read mode */
   FILE *fp = fopen("Cdocs.txt","r");
   printf("%ld", ftell(fp));   // Printing position of file pointer
    /* Reading first string */
   char string[20];
   fscanf(fp,"%s",string);
    /* Printing position of file pointer again */
   printf("%ld", ftell(fp));
   return 0;
}
```

Coding Example_9: Illustrate fseek()

```
#include<stdio.h>
int main()
{
        FILE *fp = fopen("data_out.txt","r");
        if(fp == NULL)
                printf("cannot open the file");
        else
        {
                printf("%d\n",ftell(fp));
                fseek(fp,5,SEEK_SET);   // start from the beginning
                printf("%d\n",ftell(fp)); //5
                //fseek(fp,2,SEEK_SET);
                //printf("%d",ftell(fp)); // 2
                //fseek(fp,2,SEEK_CUR);  //start from the current position of the pointer

                  //printf("%d",ftell(fp));  //7
                  fseek(fp,-2,SEEK_END);    //start from the end of the file
                  printf("%d",ftell(fp));
                  putchar(fgetc(fp));
                  fclose(fp);
          }
          return 0;
  }
```

Coding Example_10: Illustrate rewind()

```
#include <stdio.h>
int main ()
{
  char str[] = "Hello All";
  FILE *fp;
  char ch;
  /* First let's write some content in the file */
  fp = fopen( "Cdocs.txt" , "w" );
  fputs(str,fp);
  fclose(fp);
  fp = fopen( "Cdocs.txt" , "r" );
  printf("%d", ftell(fp));
  ch = fgetc(fp);
  printf("%c", ch);
    rewind(fp);
    printf("%d", ftell(fp));
    fclose(fp);
    return 0;
}
```

one more problem is there😭
based on fprintf and fscanf

ERROR HANDLING

Introduction to Error Handling

Errors are the problems or the faults that occur in the program, which make the behaviour of the program abnormal, and experienced programmers and developers can also make these faults. **Programming errors** are also known as the **bugs or faults.**

While dealing with files, it is possible that an error may occur. The most common errors that occur are:

1. **Reading beyond the end-of- file:** EOF is a condition in a computer operating system where no more data can be read from a data source. The data source is usually a file or stream.
2. **Performing operations on the file that has not still been opened**: Before performing any operation on a file, you must first open it. An attempt to perform operations on a file that has not been opened yet will lead to error.
3. **Writing to a file that is opened in the read mode:** A mode is used to specify whether you want to open a file to perform operations like read, write append etc.
4. **Opening a file with invalid filename:** If you attempt to upload a file that has an invalid file name or file type you will receive error.
5. **Write to a write-protected file:** Write-protected file is the ability to prevent new information from being written or old information being changed. In other words, information can be read, but nothing can be added or modified.

Types of Error Handling in C

1. Global Variable errno:

C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values from the function.

Most of the C function calls return -1 or NULL in case of any error and set an error code global variable errno. When a function is called in C, a variable named as errno is automatically assigned a code (value) which can be used to identify the type of error that has

been encountered. Its a global variable **indicating the error occurred during any function call** and defined in the header file **errno.h.**

Different codes (values) for errno mean different types of errors.

errno value	Type of Error
1	Operation not permitted
2	No such file or directory
5	I/O error
7	Argument list too long
9	Bad file number
11	Try again
12	Out of memory
13	Permission denied
0	No Error

2. **perror() and strerror():**

The errno value indicate the types of error encountered. If you want to show the error description, then there are two functions that can be used to display a text message that is associated with errorno.

The functions are: **perror() and strerror()**:

i. **perror():** The function perror() stands for print error. In case of an error, the programmer can determine the type of error that has occurred using the perror() function. It displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

Syntax: **void perror (const char *str)** // (str: is a string containing a custom message to be printed before the error message itself.)

```
#include <stdio.h>

#include <errno.h>

#include<string.h>
```

```
int main ()

{

FILE *fp;

fp = fopen ("File.txt", "r"); // File with this name doesn't exist

printf("Value of errno: %d %s\n ", errno, strerror(errno));

perror("Bad code");

return 0;

}
```

```
Value of errno: 2 No such file or directory
Bad code: No such file or directory
```

perror() displays a string passed to it, followed by a colon and the textual message of the current errno value.

ii. **strerror()**: returns a pointer to the textual representation of the current errno value. Function is used from string.h

Syntax: char *strerror (int errnum) //errnum is the error number (errno).

```
C:\Users\cse>gcc ferr1.c

C:\Users\cse>a
Value of errno: 2 No such file or directory
 Bad code: No such file or directory

C:\Users\cse>
```

```
int main ()
{
   FILE *fp;
   fp = fopen ("file1.txt", "r");  // File with this name exist
   fputc('A',fp);   // writing to a file which is opened for read
   printf("value of errno is %d with info:%s\n",errno,strerror(errno));
   perror("Bad code");
   fclose(fp);
   return 0;
}
```

```
C:\Users\cse>gcc ferr2.c

C:\Users\cse>a
value of errno is 22 with info:Invalid argument
Bad code: Invalid argument

C:\Users\cse>
```

3. **Two-status library functions are used to prevent performing any operation beyond EOF.**

1. **feof():** Used to test for an end of file condition
2. **ferror():** Used to check for the error in the stream

feof():

In C, getc() returns EOF when end of file is reached. getc() also returns EOF when it fails. So, only comparing the value returned by getc() with EOF is not sufficient to check for actual end of file. To solve this problem, C provides **feof() which returns non-zero value only if end of file has reached, otherwise it returns 0.**

Syntax: feof(FILE *file_pointer);

```
#include <stdio.h>

int main() {
    FILE *fp;
    char c;
    fp = fopen("test.c", "r"); // opening an existing file

    if (fp == NULL) {
        printf("Could not open file test.c\n");
    } else {
        printf("Reading the file\n");
        while (!feof(fp)) { // returns nonzero if EOF encountered
            c = fgetc(fp); // reading the file
            printf("%c", c);
        }
        // c = getc(fp); // doesn't show any error. so better to use feof
function
        // perror("BAD Code\n");
        fclose(fp); // Closing the file
    }

    return 0;
}
```

refer slides

CSV FILES (IMP)

matches.csv

1. Count the number of matches played in the year 2008
2. Count the number of times the Toss winner is same as the Winner of the match
3. Display the count of matches played between KKR and RCB
4. Display the Winner of each match played in 2016
5. Display the list of Player of the match when there was a match between RCB and CSK in the year 2010

Count the number of matches played in the year 2008

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>



int main()

{

FILE *fp = fopen("/Users/prekshakreddy/Desktop/pesu/c/matches.csv", "r");

char line[500];



if(fp == NULL)

{

printf("error in opening the file\n");

}
```

```
else

{

int count = 0;

while(fgets(line, 500, fp) != NULL)

{

char *val = strtok(line, ",");

val = strtok(NULL, ",");

if(strcmp(val, "2008") == 0)

{

count++;

}

}

fclose(fp);

printf("Number of matches in 2008 are %d\n", count);

}

return 0;

}
```

output:

```
Number of matches in 2008 are 58
```

Strtok: Available in string.h

This function returns a pointer to the first token found in the string. A null pointer is returned if there are no tokens left to retrieve. Function takes two arguments – a source string or NULL and the delimiter string. The first time strtok is called, the string to be splitted is passed as the first argument. In subsequent calls, NULL is passed to indicate that strtok should keep splitting the same string for the next token.

cars.csv

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

int main(){

char line[500];

FILE *fp=NULL;

fp=fopen("/Users/prekshakreddy/Desktop/pesu/c/cars.csv","r");

int count=0;

if(fp==NULL){

printf("file not present");

}

else{

while(fgets(line,500,fp)!=NULL){

char * val=strtok(line,",");

val=strtok(NULL,",");
```

```
val=strtok(NULL,",");

val=strtok(NULL,",");

val=strtok(NULL,",");

val=strtok(NULL,",");

if(strcmp(val,"Petrol")==0){

count++;

}

}

fclose(fp);

printf("the no of cars having petrol has it fuel type %d",count);

}

return 0;

}
```

output

```
the no of cars having petrol has it fuel type 60
```

SEARCHING

linear search and binary search

linear search

linear search, also known as a sequential search, is a method of finding an element within a collection. It checks each element of the list sequentially until a match is found or the whole list has been searched. It returns the position of the element in the array, else it return -1. Linear Search is applied on unsorted or unordered collection of elements.

binary search

Binary Search is used to search an element in a sorted array. The following steps are applied to search an element.

1. Start by comparing the element to be searched with the element in the middle of the array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or reater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1. 5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

bubble sort using file handling

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

int binary_search_in_file(const char *filename, int target) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return -1;
    }

    int low = 0, high = 0, mid;
    int number;

    // Determine the number of lines in the file
    while (fscanf(file, "%d", &number) != EOF) {
        high++;
```

```
    }
    high--; // Adjust high to the last index

    // Binary search
    while (low <= high) {
        mid = (low + high) / 2;

        // Seek to the mid line in the file
        fseek(file, 0, SEEK_SET);
        for (int i = 0; i < mid; i++) {
            fscanf(file, "%d", &number);
        }
        fscanf(file, "%d", &number);

        if (number == target) {
            fclose(file);
            return mid; // Return the index where the target is found
        } else if (number < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    fclose(file);
    return -1; // Target not found
}

int main() {
    const char *filename = "sorted_numbers.txt";
    int target;

    printf("Enter the number to search: ");
    scanf("%d", &target);

    int result = binary_search_in_file(filename, target);
    if (result != -1) {
        printf("Number %d found at index %d.\n", target, result);
    } else {
        printf("Number %d not found.\n", target);
```

```
    }

    return 0;
}
```

CALL BACK

Introduction

In computer programming, a **callback** is also known as a **"call-after function"**. The callback execution may be immediate or it might happen at a later point in time. Programming languages support call backs in different ways, often implementing them with subroutines, lambda expressions, blocks, or function pointers. In C, a **callback function is a function that is called through a function pointer/pointer to a function.** A callback function has a specific action which is bound to a specific circumstance. It is an important element of GUI in C programming.

Let us understand the use of **pointer to a function or a function pointer.**

Function Pointer points to code, not data. The function pointer stores the start of executable code. The function pointers points to functions and store its address.

Syntax:int (*ptrFunc) ();

The **ptrFunc** is a pointer to a function that takes no arguments and returns an integer. If parentheses are not given around a function pointer then the compiler will assume that the ptrFunc is a normal function name, which takes nothing and returns a pointer to an integer. Function pointers are not used for allocating or deallocating memory, instead used in reducing code redundancy.

Case 1: int *a1(int, int, int); // a1 is a function which takes three int arguments and returns a pointer to int.

Case 2: int (*p)(int, int, int); //p is a pointer to a function which takes three int as parameters and returns an int.

addition and subtraction

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

int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}

int display(int (*fp)(int, int), int a, int b) {
    return fp(a, b);
}

int main() {
    int a = 3, b = 1;
    char select[100];

    printf("Enter 'add' for addition or 'sub' for subtraction: ");
    scanf("%s", select);

    int (*fp)(int, int) = NULL;

    if (strcmp(select, "add") == 0) {
        fp = add;
    } else if (strcmp(select, "sub") == 0) {
        fp = sub;
    } else {
        printf("Invalid selection\n");
        return 1;
    }

    int y = display(fp, a, b);
    printf("Result: %d\n", y);

    return 0;
```

```
}
```

imp Mimic map, filter and reduce function of python

client_callback.c

```
#include <stdio.h>
#include "fun.h"

int incr(int x) {
    return x + 1;
}

int is_even(int x) {
    return x % 2 == 0;
}

int add(int x, int y) {
    return x + y;
}

int main() {
    int a[] = {11, 22, 33, 44, 55};
    int n = sizeof(a) / sizeof(*a);
    int b[n];

    printf("a is --------\n");
    disp(a, n);

    mymap(a, b, n, incr);
    printf("\nb is ------- \n");
    disp(b, n);

    int m;
    myfilter(a, b, n, &m, is_even);
    printf("\nb (even numbers) is ------- \n");
    disp(b, m);
```

```
    myfilter(a, b, n, &m, is_greater_than_22);
    printf("\nb (greater than 22) is ------- \n");
    disp(b, m);

    int result = myreduce(a, n, add);
    printf("Result is %d\n", result);

    return 0;
}
```

fun.c

```
#include <stdio.h>

void mymap(int a[], int b[], int n, int (*p)(int)) {
    for (int i = 0; i < n; i++) {
        b[i] = (*p)(a[i]);
    }
}

void disp(const int a[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

void myfilter(const int a[], int b[], int n, int *m, int (*op)(int)) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (op(a[i])) {
            b[count] = a[i];
            count++;
        }
    }
    *m = count;
}

int is_greater_than_22(int x) {
```

```
    return x > 22;
}

int myreduce(int a[], int n, int (*op)(int, int)) {
    int res = 0;
    for (int i = 0; i < n; i++) {
        res = op(res, a[i]);
    }
    return res;
}
```

fun.h

```
#ifndef FUN_H
#define FUN_H

void mymap(int a[], int b[], int n, int (*p)(int));
void disp(const int a[], int n);
void myfilter(const int a[], int b[], int n, int *m, int (*op)(int));
int is_greater_than_22(int x);
int myreduce(int a[], int n, int (*op)(int, int));

#endif // FUN_H
```

code

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

// Function declarations
void mymap(int a[], int b[], int n, int (*p)(int));
void disp(const int a[], int n);
void myfilter(const int a[], int b[], int n, int *m, int (*op)(int));
int is_greater_than_22(int x);
int myreduce(int a[], int n, int (*op)(int, int));

// Function definitions
```

```
int incr(int x) {
    return x + 1;
}

int is_even(int x) {
    return x % 2 == 0;
}

int add(int x, int y) {
    return x + y;
}

void mymap(int a[], int b[], int n, int (*p)(int)) {
    for (int i = 0; i < n; i++) {
        b[i] = (*p)(a[i]);
    }
}

void disp(const int a[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

void myfilter(const int a[], int b[], int n, int *m, int (*op)(int)) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (op(a[i])) {
            b[count] = a[i];
            count++;
        }
    }
    *m = count;
}

int is_greater_than_22(int x) {
    return x > 22;
}
```

```
int myreduce(int a[], int n, int (*op)(int, int)) {
    int res = 0;
    for (int i = 0; i < n; i++) {
        res = op(res, a[i]);
    }
    return res;
}

int main() {
    int a[] = {11, 22, 33, 44, 55};
    int n = sizeof(a) / sizeof(*a);
    int b[n];

    printf("a is --------\n");
    disp(a, n);

    mymap(a, b, n, incr);
    printf("\nb is ------- \n");
    disp(b, n);

    int m;
    myfilter(a, b, n, &m, is_even);
    printf("\nb (even numbers) is ------- \n");
    disp(b, m);

    myfilter(a, b, n, &m, is_greater_than_22);
    printf("\nb (greater than 22) is ------- \n");
    disp(b, m);

    int result = myreduce(a, n, add);
    printf("Result is %d\n", result);

    return 0;
}
```

Coding Example_1:

```
 struct student
{
        int roll;
        char name[20];
};
typedef struct student student_t;
void init_ptr(struct student s[], struct student *p[], int n);
void swap( struct student** lhs,  struct student** rhs);
void disp(struct student* p[], int n) ;
int sort_on_names(student_t*,student_t*);
int sort_on_roll_number(student_t*,student_t*);
void Bubble_sort(student_t *s[],int n,int (*comp)(student_t*,student_t*));
int main()
{
        FILE *fr=fopen("student.csv","r");
        char a[200];
        fgets(a,200,fr);
        char *item;
        struct student s[100];
        int i=0;
        while(fgets(a,200,fr))
        {
                item=strtok(a,",");
                s[i].roll=atoi(item);
                item=strtok(NULL,",");
```

```
        strcpy(s[i].name,item);
        i++;


}
int n = i;
//printf("n is %d\n", n);
int ch;
fclose(fr);
struct student *p[100];
init_ptr(s, p, n);
printf("Enter your choice:\n\n1.Sort on Roll Numbers\n");
printf("2.Sort on names\n");
scanf("%d",&ch);
```

```
        switch(ch)
        {

                case 1: Bubble_sort(p,n,sort_on_roll_number); disp(p,n); break;
                case 2: Bubble_sort(p,n,sort_on_names); disp(p,n); break;
                default: printf("exiting from the program"); exit(0);
        }

return 0;
}

void init_ptr(struct student s[], struct student *p[], int n)

{
                int i;
                for( i = 0; i < n; ++i)
                {
```

```
                    p[i] = &s[i];
                }

}
void swap( struct student** lhs,  struct student** rhs)
{
        struct student* temp = *lhs;
        *lhs = *rhs;
        *rhs = temp;
}

void disp(struct student* p[], int n)
{
        int i;
        for(i = 0; i < n; ++i)
        {
                printf("%d %s", p[i]->roll, p[i]->name);
        }
}
```

```
int sort_on_roll_number(student_t* s1,student_t* s2)
{
        return s1->roll > s2->roll;
}
int sort_on_names(student_t *s1,student_t *s2)
{
        return strcmp(s1->name,s2->name)>0;
}
void Bubble_sort(student_t *s[],int n,int (*comp)(student_t*,student_t*))
{
        int i,j;

            for(i = 0;i < n-1;i++)
                                for(j = 0;j < n-i-1;j++)

                                                if(comp( s[j],s[j+1])>0)

                                swap(&s[j],&s[j+1]);
  }
```

code2:

Coding Example_2:

Now solve the implementation of Binary Search on a sorted array of integers using a call back when there is a constraint to check for before searching for the element.

A. Search for a number if the number is even

B. Search for a number if the number is less than 22.

Client.c

```
#include<stdio.h>
#include "search.h"
// search for a key with a new search constraint every time.

int main()
{
        int a[] = {11,13,18,19,22,33,44,53,56,101};
        printf("enter the element to be searched\n");
        int n = sizeof(a)/sizeof(*a);
        int key;
        scanf("%d",&key);
// perform search on a collection of elements and print only if it is even
        int pos = my_search(a,0,n-1,key,is_even);
        //printf("result is %d",pos);
        if(pos !=-1)
```

```
            printf("It is even and found at %d position\n",pos);
        else
            printf("not found\n");

// perform search on a collection of elements and print only if it is less than 22
        pos = my_search(a,0,n-1,key,is_less_than_22);
        //printf("result is %d",pos);
        if(pos !=-1)
            printf("It is less than 22 and found at %d position\n",pos);
        else
            printf("not found\n");
        return 0;
}
```

Search.h

```
int my_search(int a[],int low, int high,int key,int(*p)(int));
int is_even(int x);
int is_less_than_22(int x);
```

Search.c

```
#include<stdio.h>
int is_even(int x)
{       return x%2 == 0;    }
int is_less_than_22(int x)
{       return x<22;        }
int my_search(int a[],int low,int high,int key,int(*p)(int))
{
        // recursive solution
        int pos = -1;
        int mid;
```

```
        if (low>high)
                return pos;
        else
        {
                mid = (low+high)/2;
                if(a[mid]==key && p(key))
                {               pos = mid;      }
                else if(a[mid]>key)
                {               return my_search(a,low,mid-1,key,p);        }
                else
                {               return my_search(a,mid+1,high,key,p);       }
        }
        return pos;
}
```

QUALIFIERS

Introduction

Qualifiers are keywords which are applied to the data types resulting in Qualified type. Applied to basic data types to alter or modify its sign or size.

Types of Qualifiers are as follows.

- **Size Qualifiers**
- **Sign Qualifiers**
- **Type qualifiers**

Size Qualifiers

Qualifiers are prefixed with data types to **modify the size of a data type** allocated to a variable. Supports two size qualifiers, **short and long.** The Size qualifier is generally used with an integer type. In addition, double type supports long qualifier.

Rules regarding size qualifier as per ANSI C standard:

short int <= int <=long int

float <= double <= long double

Note: short int may also be abbreviated as short and long int as long. But, there is no abbreviation for long double.

```
#include<stdio.h>

int main()

{

short int i = 100000; // cannot be stored this info with 2 bytes. So warning

int j = 100000; // add more zeros and check when compiler results in warning

long int k = 100000;

printf("%d %d %ld\n",i,j,k);

printf("%d %d %d",sizeof(i),sizeof(j),sizeof(k)); return 0;

}
```

output

```
-31072 100000 100000
2 4 8
```

Sign Qualifiers

Sign Qualifiers are used to specify the signed nature of integer types. It specifies whether a variable can hold a negative value or not. It can be used with int and char types

There are two types of Sign Qualifiers in C: **signed and unsigned**

A signed qualifier specifies a variable which can hold both positive and negative integers

An unsigned qualifier specifies a variable with only positive integers.

```
#include<stdio.h>
int main()
{
unsigned int a = 10;
unsigned int b = -10;   // observe this
int c = 10;  // change this to -10 and check
signed int d = -10;
printf("%u %u %d %d\n",a,b,c,d);
printf("%d %d %d %d",a,b,c,d);
return 0;
}
```

output

```
10 4294967286 10 -10
10 -10 10 -10
```

Type Qualifiers

A way of expressing additional information about a value through the type system and ensuring correctness in the use of the data.

Type Qualifiers consists of two keywords i.e., **const** and **volatile.**

const

The **const** keyword is like a normal keyword but the only difference is that once they are defined, their values can't be changed. They are also called as literals and their values are fixed.

Syntax: const data_type variable_name

```
#include <stdio.h>
int main()
{
        const int height = 100; /*int constant*/
        const float number = 3.14; /*Real constant*/
        const char letter = 'A'; /*char constant*/
```



```
        const char letter_sequence[10] = "ABC"; /*string constant*/
        const char backslash_char = '\?'; /*special char cnst*/
        //height++; //error
        printf("value of height :%d \n", height );
        printf("value of number : %f \n", number );
        printf("value of letter : %c \n", letter );
        printf("value of letter_sequence : %s \n", letter_sequence);
        printf("value of backslash_char : %c \n", backslash_char);
}
```

Output:

value of height : 100

value of number : 3.140000

value of letter : A

value of letter_sequence : ABC

value of backslash_char : ?

Pointer to a constant: **A pointer through which one cannot change the value of variable it points. These types of pointers can change the address they point to but cannot change the value kept at those address.**

Constant Pointer: **Cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. We can as well say that if a constant pointer is pointing to some variable, then it cannot point to any other variable**

code 1:

```
#include<stdio.h>

int main()

{

int i = 5;

int j = 6;

const int *p = &i; // p is a pointer to constant integer printf("%d\n",*p);

p = &j; // no error

//*p = j; // error

printf("%d\n",*p);

return 0;

}
```

```
6
```

code 2:

```
#include<stdio.h>

int main()

{

const int i = 5;

int *p = &i;

printf("%d\n",*p);

*p = 6;// only warning. But code works

printf("%d\n",*p);

return 0;

}
```

```
5
6
```

```
/*int i = 5;
int j = 6;
int* const p = &i;                // p is a constant pointer to integer
printf("%d\n",*p);
//p = &j; // eror
*p = j;
printf("%d\n",*p);
*/

int i = 5;
int j = 6;
const int* const p = &i;        //p is a constant pointer to constant integer
 //p = &j; // eror
//*p = j;  // error

            //const const int a; // illegal

            return 0;

}
```

volatile

The volatile keyword is intended to prevent the compiler from applying any optimizations. Their values can be changed by the code outside the scope of current code at any time. A type declared as volatile can't be optimized because its value can be easily changed by the code. The declaration of a variable as volatile tells the compiler that the variable can be modified at any time by another entity that is external to the implementation, for example: By the operating system or by hardware.

Syntax: volatile data_type variable_name

Version2: volatile keyword added while declaring the variable a. Run this code using –save-temps and observe the size of .s(assembly file). Optimization is not done.

```
#include<stdio.h>
int main()
{
        volatile int a = 0; // observe this
        if(a == 0)
                printf("a is 0\n");
        else
                printf("a is not zero\n");
        return 0;
}
```

Execution steps:

```
gcc program5.c --save-temps(press enter)          // creates 3 files( i.e., o, s)
```

Applicability of Qualifiers to Basic Types

The below table helps us to understand which Qualifier can be applied to which basic type of data.

No.	Data Type	Qualifier
1.	char	signed, unsigned
2.	int	short, long, signed, unsigned
3.	float	No qualifier
4.	double	long
5.	void	No qualifier

PREPROCESSORS DIRECTIVES

The execution of a 'C' program takes place in multi stages. The stages are: **Pre-processing, Compilation, Loading, Linking and Execution.** The first stage is Pre - processing, which is performed by the 'C' pre-processor. It is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just **a text substitution tool** and it instructs the compiler to do **required pre-processing before the actual compilation**. We refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. Possible to see the effect of the pre-processor on source files directly by using the -E option of gcc.

Types of Preprocessor Directives

- Macros
- File Inclusion
- Conditional Compilation
- Other directives

macro

Macro in C

Macro is a piece of code in a program which has been given a name. During preprocessing, when the 'C' preprocessor encounters the name, it **substitutes the name with the piece of code**. **#define directive** is used to define a macro.

Few points to know about macros are as below:

- Macro **does not** judge anything
- **No memory Allocation** for Macros
- Can define string using macros
- Can define macro with expression
- Can define macro with parameter
- Macro can be used in another macro
- Constants defined using #define cannot be changed using the assignment operator
- **Redefining the macro with #define is allowed.** But not advisable

Coding Example_3: Defining a string using #define

```
#include<stdio.h>
#define STR "Hello All"                    //#define can be used for strings as well
int main()
{
   printf("The string is %s\n",STR);        // The string is Hello All
   return 0;
}
```

Coding Example_4: Macro with expression

```
#include<stdio.h>
#define STR 2+5*1                          //macro wit expression
int main()
{
        printf("Value is %d\n",STR);  //STR replaced with 2+5*1 in preprocessing stage.
                                            During execution, expression is evaluated
        return 0;
}
```

Coding Example_5: Macro with parameter

```
#include<stdio.h>
#define SUM(a,b) a+b                    // SUM is not a function. It is a macro
int main()
{
        int a,b;
        printf("Enter two numbers:\n");
```



```
        scanf("%d%d",&a,&b);
        printf("The sum of two numbers is %d\n",SUM(a,b));
        return 0;
}
```

Coding Example_6: Demo of plain text substitution

```
#include<stdio.h>
#define sqr(x) (x*x)                    //change this to (x)*(x)
int main()
{
        int y=8;
        printf("%d",sqr(2+3));          // 11(2+3*2+3)        // Not (2+3)*(2+3)
        return 0;
}
```

What changes to be done in above code to perform (2+3)*(2+3) ?

#define sqr(x) (x)*(x) – Think about it

Coding Example_7: Macro in another macro

```
#include<stdio.h>
#define sqr(x) x*x
#define cube(x) sqr(x)*x                        // using sqr in cube
int main()
{
            printf("The cube of 9 is d\n",cube(9));
            return 0;
}
```

Coding Example_8: #define is used to define constants. These constants cannot be changed using assignment operator(=). Code results in error

```
#include<stdio.h>
#define MAX 200
int main()
```

```
{
        if(MAX==20)
                printf("hello");
        else
        {
                printf("U here??");
                MAX=20;// Error because no memory allocated for MAX
                // To avoid this, use #define and check
         }
        printf("MAX is %d",MAX);
        return 0;
 }
```

Coding Example_9: Redefining the macro with #define is allowed. But not advisable

//Results in warning

```
#include<stdio.h>
#define MAX 200
int main()
        {

                if(MAX==20)
                        printf("hello");
                else
                        {
                        printf("U here??");
                        #define MAX 20
                        }
                        printf("MAX is %d",MAX);//20
                         return 0;
        }
```

Enum v/s Macros

- ✓ Macro doesn't have a type and enum constants have a type int.
- ✓ Macro is substituted at pre-processing stage and enum constants are not.
- ✓ Macro can be redefined using #define but enum constants cannot be redefined. However assignment operator on a macro results in error.

file inclusion directives

File Inclusion Directives

This type of preprocessor directive tells the compiler **to include a file in the source code program**. There are two types of files which can be included by the user in the program.

- **Header File or Standard files:** These files contain definitions of pre-defined functions like printf(), scanf(), malloc(), atoi, strtok() etc. To work with these functions, respective header files must be included. Different functions are declared in different header files. These files can be added using **#include< *file_name* >** where file_name is the name of the header file to be included. The angular brackets **'<' and '>'** tells the compiler to look for the file in standard directory(PATH).

  Example: Standard I/O functions are in 'stdio.h' file whereas functions which perform string operations are in 'string.h'

- **User- defined files:** When a program becomes very large, it is a good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as **#include"*filename*"**

  Example: #include"sort.h" and include "sort.c"

conditional compilations

Implies that in a particular program, all blocks of code will not be compiled. Rather, a few **blocks of code will be compiled based on the result of some condition**. Conditional Compilation in 'C' is performed with the help of the following Preprocessor Directives -> **#ifdef, #ifndef, #if, #else, #elif, #endif.** During Conditional Compilation, **Preprocessor depends on the conditional block name to be passed from the compilation process or not**, which is decided at the time of pre-processing. If the condition is True, then the block will be passed from the compilation process, if the condition is false then the complete block of the statements will be removed from the source at the time of the Preprocessor . The **advantage of this Preprocessor is reducing .exe OR .out file size because when source code is reduced then automatically object code is reduced.**

#ifdef & #ifndef are called **Macro Testing Conditional Compilation PreProcessor**. When we are working with this PreProcessor, depends on the condition only, code will pass for the compilation process (depends on macro status). By using this PreProcessor, **we can avoid multiple substitutions of the header file code.**

Coding Example_12: This program demonstrates #ifdef

If the identifier checked by #ifdef is defined, only then, statements followed by #ifdef will be included in the file to be compiled.

```
#define MAX 20
#include<stdio.h>
int main()
{
        #ifdef MAX                              //No parentheses for #ifdef
            printf("MAX  defined\n");           // If #define above is removed, this line is not
                                                 // included for compilation.
        endif                                    // compulsory for #ifdef
        printf("Outside the scope of #ifdef\n");
        return 0;
}
```

Output: MAX defined

Outside the scope of #ifdef

Coding Example_13: This program demonstrates #ifdef

If the identifier checked by #ifdef is defined, only then, statements followed by #ifdef will be included for compilation. But here #define MAX is missing

```
#include<stdio.h>
int main()
        {
    #ifdef MAX                          //No parentheses for #ifdef
    printf("MAX defined\n");            // Not included for compilation
    #endif                              // Compulsory for #ifdef
    printf("Outside the scope of #ifdef\n");
    return 0;
        }
```

Output: **Outside the scope of #ifdef**

Coding Example_14: This program demonstrates #ifndef

If the identifier checked by #ifndef is not defined, only then, statements followed by #ifndef will be included for compilation.

```
#include<stdio.h>
 int main()
 {
      #ifndef MAX                          //No parentheses for #ifdef
           printf("MAX  not  defined\n");  // If #define above is added, this line is not
                                              included for compilation.
       #endif                 // compulsory for #ifndef
       printf("Outside the scope of #ifndef\n");
       return 0;
 }
```

Output: MAX not defined

Outside the scope of #ifndef

Coding Example_15: This program demonstrates #if

```
#include<stdio.h>
int main()
```



```
{
        #if (2>1)                   //Syntax:    #if   Constant_Expression
                                    // #if 2<1. True will not be printed
            printf("True\n");
        #endif
        printf("Outside if\n");
        return 0;
 }
```

Output: True

Outside if

Coding Example_16: This program demonstrates #if and #else

```
#include<stdio.h>
int main()
{
        #if ((2+3) < 2))
                printf("Hello\n");
        #else
                printf("Hi\n");
        #endif
        return 0;
}
```

Output: // printf("Hi\n"); will be sent to compiler for compilation

Hi

other directives

#undef:

Used to undefine an existing macro. This directive works as: #undef LIMIT. Using this statement will undefine the existing macro LIMIT. After this statement every "#ifdef LIMIT" statement will evaluate to false.

Coding Example_18: This program demonstrates #undef

```
#define LIMIT 1
#include<stdio.h>
#undef LIMIT
int main()
{
        #ifdef LIMIT
                //#undef LIMIT    // uncomment this and comment the above.observe the output
                printf("Defined");
                printf("\n%d",LIMIT);
        #else
                printf("not defined");
        #endif
        return 0;
}
```

Output: not defined

Pragma directives:

This directive is a special purpose directive and is used to **turn on or off some features**. This type of directives are **compiler(implementation)-specific** i.e., they vary from compiler to compiler. The pragma directive is used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. The effect of pragma will be applied from the point where it

is included to the end of the compilation unit or until another pragma changes its status. A #pragma directive is an instruction to the compiler and is usually ignored during preprocessing.

Sr.No.	#pragma Directives & Description
1	**#pragma startup** Before the execution of main(), the function specified in pragma is needed to run.
2	**#pragma exit** Before the end of program, the function specified in pragma is needed to run.
3	**#pragma warn** Used to hide the warning messages.
4	**#pragma GCC dependency** Checks the dates of current and other file. If other file is recent, it shows a warning message.
5	**#pragma GCC system_header** It treats the code of current file as if it came from system header.
6	**#pragma GCC poison** Used to block an identifier from the program.

hashtag(symbol🥲) pragma pack(n) where is the alignment in bytes, valid alignment values being 1, 2, 4 and 8. This pragma aligns members of a structure to the minimum of n and their natural alignment. Packed objects are read and written using unaligned accesses.

Coding Example_21:

```
#include<stdio.h>
//#pragma pack(n) //n=1 2 or 4
#pragma pack(1)
struct sample
{
        int a; //4 bytes
        char b; //1 byte + 3 padding
        int x;//4 byte
};
int main()

{       printf("%lu\n",sizeof(struct sample));
        return 0;
}
```

Output: 9

learn portable program