

Unit #: 4

Unit Name: File handling and Portable programming

Topic: File Handling - Functions

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine.

CObj2: Map algorithmic solutions to relevant features of C programming language constructs.

CObj3: Gain knowledge about C constructs and its associated eco-system.

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors.

CObj5: Get insights about testing and debugging C Programs.

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs.

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions.

CO3: Understand prioritized scheduling and implement the same using C structures.

CO4: Understand and apply sorting techniques using advanced C constructs.

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs.

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

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.

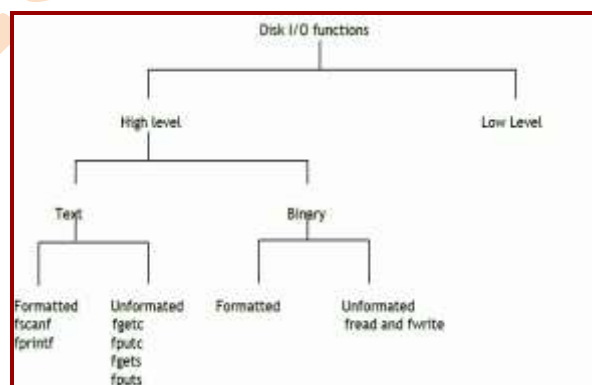
Need of Files

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.

Major File handling Functions are listed as below:

- Creation of a new file
- Opening an existing file
- Reading data from a file
- Writing data in a file
- Closing a file

Let us discuss each one in a detailed section.

Opening a file

- fopen is a C library function used to open an existing file or create a new file.
- The basic format of fopen is: `FILE *fopen(const char * filePath, const char * mode);`
- filePath: The first argument is a pointer to a string containing the name of the file to be opened.
- mode: The second argument is an access mode.
- fopen function returns NULL in case of a failure and returns a FILE stream pointer on success.
- The mode can be read, write, append.

Closing a file

- fclose() function is a C library function and it's used to release the memory stream, opened by fopen() function.
- The basic format of fclose() function is: `int fclose(FILE * stream);`
- fclose returns EOF in case of failure and returns 0 on success.

Read /Write operations on Files

File I/O operations are categorized into following types:

- Character I/O
- String I/O
- Formatted I/O
- Block read / write

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 putc() 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.

- On success, it reads n items from the file and returns n. On error or end of the file, it returns a number less than n.

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 from 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(fp);Where fp 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)

Let us write few example codes to understand the above functions.

Cdocs.txt contains the below data in it.

Good morning all

Welcome to files and file handling section in C Programming

Have a nice day

Stay well and safe

Coding Example_1: Illustrate fopen()

// Program to open an existing file.

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

```
int main()
```

```
{
```

```
    FILE *fp;
```

```
    fp = fopen("Cdocs.txt","w");
```

```
    return 0;
```

```
}
```



The contents of the file is overwritten since it is write mode. Open Cdocs.txt and check

Coding Example_2: Illustrate fclose()

```
#include<stdio.h>

int main()
{
    FILE *fp;
    fp = fopen("Cdocs.txt","w");
    fprintf(fp, "%s", "Sample C File");
    fclose(fp);
    return 0;
}
```

Coding Example_3: Illustrate getc() and putc()

```
#include <stdio.h>

int main()
{
    char ch;
    FILE *fp;
    if (fp = fopen("test.c", "r"))
    {
        ch = getc(fp);
        while (ch != EOF)
        {
            putc(ch, stdout); // ch on the terminal
            ch = getc(fp);
        }
        fclose(fp);
    }
    return 0;
}
```

```
\Users\cse>gcc l35p3.c
\Users\cse>a
#include<stdio.h>
int main()

    int a =5;
    int b = -7;
    int c = 0,d;
    d=++a&&++b|++c;
    printf("%d%d%d\n",a,b,c,d);
    return 0;
```

Coding Example_4: Illustrate fgetc() and fputc()

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

```
int main()
```

```
{
```

```
    FILE *fp;
```

```
    char ch;
```

```
    fp = fopen("Cdocs.txt","w");           //Statement 1
```

```
    if(fp == NULL)
```

```
    {
```

```
        printf("\nCan't open file or file doesn't exist.");
```

```
    }
```

```
    else
```

```
    {
```

```
        while((ch=getchar())!=EOF)         //Statement 2
```

```
            fputc(ch,fp);
```

```
        printf("\nData written successfully...");
```

```
        fclose(fp);    }
```

```
    return 0;
```

```
}
```

```
C:\Users\cse>gcc l35p4.c
C:\Users\cse>a
Data written successfully...
C:\Users\cse>
```

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);
```

Think about using fread with different types of data as above.

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
    }
}
```

Problem Solving with C – UE23CS151B

```
//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;
}
```




```
C:\Users\cse>gcc fseek.c
C:\Users\cse>a
0
5
10
20
58
8
C:\Users\cse>
```

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;  
}
```



```
C:\Users\cse>gcc rewind.c  
C:\Users\cse>a  
040  
C:\Users\cse>.
```

Coding Example_11: Illustrate fprintf() and fscanf()

Version1:

```
#include<stdio.h>  
  
void read(int *a, int n, FILE *fp);  
int find_sum(int *a, int n);  
int main()  
{  
    FILE *fp1 = fopen("formatted_data.txt", "r");  
    FILE *fp2 = fopen("formatted_out.txt", "w");  
    if(fp1 == NULL || fp2 == NULL)  
        printf("issue in opening the file\n");  
    else  
    {  
        int a[500];  
        int n = 5;  
        read(a, n, fp1);  
        int sum = find_sum(a, n);    //fprintf(stdout, "sum is %d", sum);  
        fprintf(fp2, "sum is %d", sum);  
    }  
}
```



```
        return 0;
    }
    void read(int *a, int n, FILE *fp)
    {
        int i;
        for(i = 0; i < n; i++)
        {
            fscanf(fp, "%d", &a[i]);
        }
    }
    int find_sum(int *a, int n)
    {
        int i;
        int sum = 0;
        for(i = 0; i < n; i++)
        {
            sum = sum + *(&a[i]);
        }
        return sum;
    }
```

Version 2:

```
//      if we do not specify n, above code fails.
//      So use the return value of fscanf to find n
#include <stdio.h>
int read(int *a, FILE *fp);
int find_sum(int *a, int n);
int main()
{
    FILE *fp1 = fopen("formatted_data.txt", "r");
    FILE *fp2 = fopen("formatted_out.txt", "w");
    if(fp1 == NULL || fp2 == NULL)
        printf("issue in opening the file\n");
    else
```

Problem Solving with C – UE23CS151B

```
{
    int a[500];
    int n = read(a,fp1);
    int sum = find_sum(a,n);
    //fprintf(stdout,"sum is %d",sum);
    fprintf(stdout,"sum is %d",sum);
}
return 0;
}

int read(int *a, FILE *fp)
{
    int i = 0;
    while (fscanf(fp,"%d",a+i) != EOF)
        i++;
    return i;
}

int find_sum(int *a, int n)
{
    int i;
    int sum = 0;
    for(i = 0;i<n;i++)
    {
        sum = sum+*(a+i);
    }
    return sum;
}
```

Happy Coding using File handling constructs!!

Unit #: 4

Unit Name: File handling and Portable programming

Topic: Error handling in File handling

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine.

CObj2: Map algorithmic solutions to relevant features of C programming language constructs.

CObj3: Gain knowledge about C constructs and its associated eco-system.

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors.

CObj5: Get insights about testing and debugging C Programs.

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs.

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions.

CO3: Understand prioritized scheduling and implement the same using C structures.

CO4: Understand and apply sorting techniques using advanced C constructs.

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs.

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

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.

Error handling helps during the processing of Input-Output statements by catching severe errors that might not otherwise be noticed.

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

Problem Solving with C – UE23CS151B

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 errno.

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.)

Coding Example_1: This Program illustrates the use of perror function.

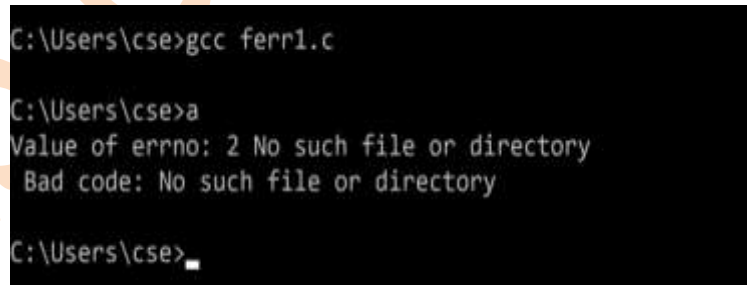
```
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main ()
{
    FILE *fp;
    fp = fopen ("File.txt", "r"); // File with this name doesn't exist
    printf("Value of errno: %d %s\n ", errno, strerror(errno));
    perror("Bad code");
    return 0;
}
```

Note that the function 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).

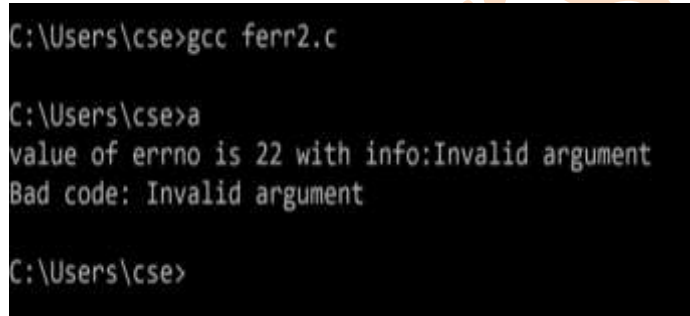


```
C:\Users\cse>gcc ferr1.c
C:\Users\cse>a
Value of errno: 2 No such file or directory
Bad code: No such file or directory
C:\Users\cse>_
```

Coding Example_2: This program illustrates how perror() and strerror() functions are used to display error messages in textual format

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

```
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);`

Coding Example_3:

Consider the following C program to print contents of file test.c on screen. In the program, returned value of `getc()` is compared with `EOF` first, then there is another check using `feof()`. By putting this check, we make sure that the program prints “End of file reached” only if end of file is reached. And if `getc()` returns `EOF` due to any other reason, then the program prints “BAD Code”

```
#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" ) ;
    }
    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); // doesnt show any error. so better to use feof function
        //perror("BAD Code\n");
        fclose ( fp ) ; // Closing the file
    }
    return 0;
}
```



```
C:\Users\cse>a
Reading the file
#include<stdio.h>
int main()
{
    FILE *fp=fopen("data141.txt","r");
    char ch;
    fputc(ch,stdout);
    //printf("%d\n",fp);
    printf("File has been read successfully");
    return 0;
} BAD Code
: No error
```

feof():

The feof() function checks for any error in the stream. **It returns a value zero if no error has occurred and a non-zero value if there is an error.** The error indication will last until the file is closed unless it is cleared by the clearerr() function.

Syntax: int feof (FILE *file_pointer);

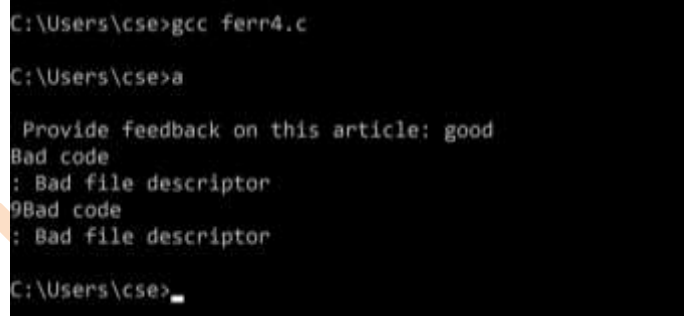
The following two example programs(4 and 5) illustrate the errors generated while trying to open a file with incorrect permissions.

Coding Example_4:

```
#include <stdio.h>
int main()
{
    FILE* fp;
    char feedback[100];
    fp = fopen("feedback.txt", "r"); // opened in r mode. File must be existing
    if (fp == NULL)
        printf("\n The file could not be opened");
    else
    {
        printf("\n Provide feedback on this article: ");
        fgets(feedback, 100, stdin);
        for (i = 0; i < feedback[i]; i++)
            fputc(feedback[i], fp); // writing to a file character by character
```

Problem Solving with C – UE23CS151B

```
//printf("%d",errno);
//perror("Bad code\n");
// writing to a file is not allowed as the mode used is r while opening
if (ferror(fp))
{
    printf("\n Error writing in file");
    printf("%d",errno);
    perror("Bad code\n");
}
fclose(fp);
}
return 0;
}
```



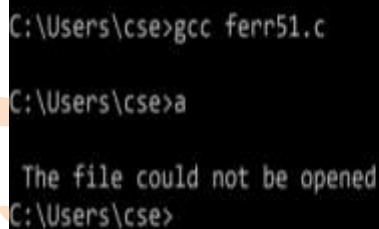
```
C:\Users\cse>gcc ferr4.c
C:\Users\cse>a
Provide feedback on this article: good
Bad code
: Bad file descriptor
Bad code
: Bad file descriptor
C:\Users\cse>
```

Coding Example_5:

```
#include <stdio.h>
int main()
{
    FILE* fp;
    char feedback[100];
    fp = fopen("feedback.txt", "r"); // opened in r mode. File must be existing
    if (fp == NULL)
        printf("\n The file could not be opened");
    else
```

Problem Solving with C – UE23CS151B

```
{  
    printf("\n Provide feedback on this article: ");  
    fgets(feedback, 100, stdin);  
    int i;  
    for (i = 0; i < feedback[i]; i++)  
        fputc(feedback[i], fp); // writing to a file character by character  
    //printf("%d",errno);  
    //perror("Bad code\n");  
    // writing to a file is not allowed as the mode used is r while opening  
    if (ferror(fp))  
        printf("\n Error writing in file");  
    clearerr(fp); // error indication is stopped . next ferror not executed  
    printf("after error thrown\n");  
    if (ferror(fp)) {  
        printf("Error in reading from file : file.txt\n");  
    }  
    return 0;  
}
```



```
C:\Users\cse>gcc ferr51.c  
C:\Users\cse>a  
The file could not be opened  
C:\Users\cse>
```

Best Practices in File handling

1. Given a file pointer check whether it is NULL before proceeding with further operations
2. Use errno.h and global variable errno to know the type of error that occurred. Usage of strerror() and perror() helps in providing textual representation of the current errno value
3. Good to check whether EOF is reached or not before performing any operation on the file

Happy Coding using Error handling in File handling!!

Unit #: 4

Unit Name: File Handling and Portable programming

Topic: Problem Solving – File Handling

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine.

CObj2: Map algorithmic solutions to relevant features of C programming language constructs.

CObj3: Gain knowledge about C constructs and its associated eco-system.

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors.

CObj5: Get insights about testing and debugging C Programs.

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs.

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions.

CO3: Understand prioritized scheduling and implement the same using C structures.

CO4: Understand and apply sorting techniques using advanced C constructs.

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs.

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Before solving the questions listed in this lecture file, please make sure you have two datasets – matches.csv and train_dataset.csv

A. Let us consider few problem statements to solve.

1. Count the number of matches played in the year 2008 – Solution: 58
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

Optional: The output of all of the above can be written to a file to store the record of outputs in one file

Data file Description

File to be used is **matches.csv**: Contains the information of cricket matches held between 2008 and 2016. First row represents the column headers such as id, season, city, date, team1, team2, toss_winner, toss_decission, result, dl_applied, winner, win_by_runs, win_by_wickets, player of match, venue, umpire1, umpire2 and umpire3. Contains around 577 rows. Every data is stored with a comma in between.

Coding Example_1: Count the number of matches played in the year 2008

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
    FILE *fp=fopen("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;
}
```

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.

Can you pass the FILE pointer to a user defined function? Modified code by separating the interface and implementation is as below.

Coding Example_2:

//Header file: p2.h

```
#include<stdio.h> // To use FILE in p2.h, stdio.h must be used
```

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

//Client code: p2_client.c

```
#include"p2.h"
```

```
#include<stdio.h> // To use NULL and FILE, need to add stdio.h
```

```
int main()
```

```
{
```

```
    FILE *fp=fopen("matches.csv","r");
```

```
    char line[500];
```

```
    if(fp==NULL)
```

```
    {
```

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

```
    }
```

```
    else
```

```
    {
```

```
        int count = get_count(fp);
```

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

```
        fclose(fp);
```

```
    }
```

```
    return 0;
```

```
}
```

//Source File: p2.c

```
#include<stdio.h> // To use FILE in p2.c, stdio.h must be used
```

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

```
int get_count(FILE* fp)
```

```
{
```



```
char line[500];
int count=0;
while(fgets(line,500,fp) != NULL)
{
    char *val = strtok(line,"");
    val = strtok(NULL,"");
    if(strcmp(val,"2008") == 0)
    {
        count++;
    }
}
return count;
}
```

B. A person called Hrithik is travelling from Bangalore to Mumbai and he is new to the city. He wants to travel back to Mumbai by train. Help Hrithik to reach Mumbai by giving train details that are available for Mumbai from bangalore where the cost is less than 1000 per head. Write a C program to display the details of all trains meeting this requirement.

Coding Example_3:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
    FILE *fp = fopen("Dataset_train.csv","r");
    if(fp == NULL)
    {
        printf("cannot open file\n");
    }
}
```

```
}  
else{  
  
    char line[1000];  
    //gets(line);  
    int count = 0;  
    while(fgets(line,1000,fp) != NULL)  
    {  
  
        char *pnr = strtok(line,"," );  
        char *name = strtok(NULL,"," );  
        char *src = strtok(NULL,"," );  
        char *dest = strtok(NULL,"," );  
        char *cost = strtok(NULL,"," );  
        if(strcmp(src,"Bangalore") == 0 && strcmp(dest,"Mumbai")== 0 &&  
atoi(cost) < 1000) // Logical not ! can also be used  
        // to consider capital case and small case strings, use appropriate functions here  
        {  
            printf("%s: %s - %s\n",pnr,name,cost);  
        }  
    }  
}  
fclose(fp);  
return 0;  
}
```

Note: The data from the file can be stored in array of structures as well if we want to use it again and again.

Other questions listed in A, can be considered as practice questions for you to solve.

Happy Coding!!

Unit #: 4

Unit Name: File Handling and Portable programming

Topic: Searching

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Introduction

Even a single day does not pass without we searching for something in our day to day life. Might be car keys, books, pen, mobile charger and what not. Same is the life of a computer. There is so much data stored in it and whenever user asks for some data, computer has to search it's memory to look for the data and make it available to the user. And the computer has it's own techniques to search through it's memory in a faster way.

Why Searching?

We often need to find one particular item of data amongst many hundreds, thousands, millions or more. For example, if one wants to find someone's phone number in your phone, or a particular business's address from address book.

A searching algorithm can be used to find the data without spending much time. What if you want to write a program to search a given integer in an array of integers? Two popular algorithms available:

Linear Search and Binary Search

Linear Search

A 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.

Coding Example_1: Implementation of Linear Search (Covered in unit -2)

```
int key = 100;
int a[100] = {22,55,77,99,25,90,100};
int n = 7;
int linear_search(int *a, int n, int key)
{
    int i; int found = 0; int pos = -1;
    for(i = 0; i < n; i++)
```

```
{  
    if(a[i]==key && found == 0)  
    {  
        found = 1;  
        pos = i;  
    }  
}  
return pos;  
}
```

Binary Search (Covered in unit-2)

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.

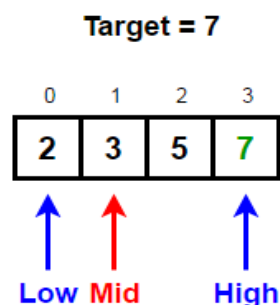
Binary Search Representation

Example: Take an input array by name arr = [2, 3, 5, 7, 8, 10, 12, 15, 18, 20] and **target or key to be searched is 7.**



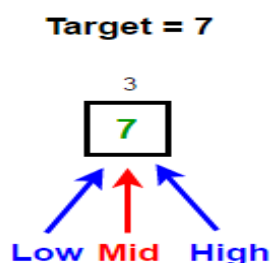
Since 8 (Mid) > 7 (target),
we discard the right half and go LEFT

New High = Mid - 1



Since 3 (Mid) < 7 (target),
We discard the left half and go RIGHT

New low = mid + 1



Now our search space consists of only one element 7. Since 7 (Mid) = 7 (target), we return index of 7 i.e. 3 and terminate our search

Note: Between 2,3,5,7 and 7, write diagrams for missing two iterations. Drawing these two iterations might help you to understand better.

Binary Search Algorithm(Iterative solution)

Algorithm Binary_Search(list, item)

1. Set L to 0 and R to n: 1
2. if $L > R$, then Binary_Search terminates as unsuccessful
3. else, Set m (the position in the mid element) to the floor of $(L + R) / 2$
4. if $A_m < T$, set L to m + 1 and go to step 2
5. if $A_m > T$, set R to m: 1 and go to step 2
6. Now, $A_m == T$, the search is done and return (m).

Explanation

The iterative binary search algorithm works on the principle of "Divide and Conquer". First it divides the large array into smaller sub-arrays and then finding the solution for one sub-array. It finds the position of a key or target value within an array and compares the target value to the middle element of the array, if they are unequal, the half in which the target cannot lie is eliminated. The search continues on the remaining half until it is successful.

Binary Search Algorithm(Recursive solution)

BinarySearch(x:target; {a₁,a₂,...,a_n}:sorted list of items;

1. begin:pos;end:pos
2. mid:=(begin+end)/2
3. If begin>end:return -1
4. else if x==amid:return mid
5. else if x<amid:return BinarySearch(x;{a₁,a₂,...,a_n};begin;mid-1)
6. else return BinarySearch(x;{a₁,a₂,...,a_n};mid+1,end);

Explanation

In recursive approach, the function BinarySearch is recursively called to search the element in the array. It accepts the input from the user and store it in **m**. The array elements are declared and initialized. In the recursive call to function **the array, the lower index, higher index and the number are passed as arguments** to be searched again and again until element is found. If not found, then it returns -1.

Coding Example_2: Implementation of Binary Search (iterative & recursive) on an array of integers

```
// Binary search on files:
```

```
/*      works on sorted collection of elements.
```

```
Time required to access each element in the collection must be same - constant time
```

```
*/
```

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

```
int mysearch(int a[],int low,int high,int key)
```

```
{  
    /* binary search - iterative solution  
    int pos = -1;  
    int found = 0;  
    // if found variable is not created, what is the problem. Think about it? Is there any other way?  
    while(low<=high && found ==0)  
    //while(low<=high && pos != mid)  
    // while(low<=high && pos == -1)  
    {  
        int mid = (low+high)/2;  
        if(a[mid]==key)  
        {  
            pos = mid;found = 1;  
        }  
        else if(key<a[mid])  
            high = mid-1;  
        else  
            low = mid+1;  
    }  
    return pos;  
    */  
    // Recursive solution  
    if(low > high) // base condition  
        return -1;  
    else  
    {  
        int mid = (low+high)/2;  
        if(a[mid]==key)  
        {  
            return mid;  
        }  
    }  
}
```


Problem Solving With C – UE23CS151B

```
    }  
    else if(key<a[mid])  
        return mysearch(a,low,mid-1,key);  
    else  
        return mysearch(a,mid+1,high,key);  
}  
}
```

// client code is as below

```
int main()  
{  
    int a[100];  
    int key; int n; int i;  
    FILE *fp = fopen("numbers.txt","r"); // file exists with 5 integers in it  
    printf("How many numbers you want to read from the file?");  
    scanf("%d",&n);  
    for(i = 0;i<n;i++)  
        fscanf(fp,"%d",&a[i]);  
    fclose(fp);  
    printf("enter the element to be searched\n");  
    scanf("%d",&key);  
    int res = mysearch(a,0,5,key);  
    if(res == -1)        printf("not found");  
    else                printf("found at %d\n",res);  
    return 0;  
}
```

PES University

Unit #: 4

Unit Name: File Handling and Portable programming

Topic: Problem Solving – Sorting the file using Array of Pointers

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine.

CObj2: Map algorithmic solutions to relevant features of C programming language constructs.

CObj3: Gain knowledge about C constructs and its associated eco-system.

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors.

CObj5: Get insights about testing and debugging C Programs.

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs.

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions.

CO3: Understand prioritized scheduling and implement the same using C structures.

CO4: Understand and apply sorting techniques using advanced C constructs.

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs.

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Problem Solving with C – UE23CS151B

Given a data file student.csv, create a menu driven program to perform bubble sort based on roll_number and name. Write the sorted record to a new file.

Data file Description:

- Contains the information of students in two columns only
- First row represents the column headers such as roll_no and name
- Contains around 55 rows. Every data is stored with a comma in between

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct student
{
    int roll;
    char name[20];
};
void init_ptr(struct student s[], struct student *p[], int n);
void swap( struct student** lhs, struct student** rhs);
void Bubble_sort_roll_no(struct student* s[],int n);
void Bubble_sort_names(struct student* s[],int n);

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,"");
```

Problem Solving with C – UE23CS151B

```
s[i].roll=atoi(item);
item= strtok(NULL,"");
strcpy(s[i].name,item);
i++;

}

int n = i;
int ch;
fclose(fr);
struct student *p[100];
init_ptr(s, p, n);

printf("1.Sort on Roll Number\n\n");
printf("2.Sort on name\n\n");
printf("Enter the choice(1 or 2):");
scanf("%d",&ch);
switch(ch)
{
    case 1:Bubble_sort_roll_no(p,n);

        break;
    case 2:Bubble_sort_names(p,n);

        break;

    default: printf("Exiting from the program");
        exit(0);

}

FILE *fw=fopen("student_sorted.csv","w");
fprintf(fw,"Roll_number,Name\n");
```

Problem Solving with C – UE23CS151B

```
i=0;
while(i<n)
{
    fprintf(fw,"%d,%s",p[i]->roll,p[i]->name);
    i++;
}
fclose(fw);
printf("Sorted data is written to a file student_sorted.csv\n");

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 Bubble_sort_roll_no(struct student *s[],int n)
{
```

Problem Solving with C – UE23CS151B

```
int i,j;
for(i = 0;i<n-1;i++)
    for(j = 0;j<n-i-1;j++)
        if((s[j]->roll) > (s[j+1]->roll))

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

void Bubble_sort_names(struct student* s[],int n)
{
    int i,j;
    for(i = 0;i<n-1;i++)
        for(j =0;j<n-i-1;j++)

            if(strcmp(s[j]->name,s[j+1]->name)>0)

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

Few points to think about the above code:

- Could you modify the code to sort in descending order?
- What happens when you sort on names when two students have the same name?
- Rather than having two functions to sort on roll_no and sort on names, can we use one function which does sorting based on some condition?
- Can we pass the function name as an argument to a function and call appropriate function inside the sort function? This is done using the concept of callback. We will be discussing this in the next lecture

Coding Example_2: Create an array to contain the details of n books. Sort the details of books based on ISBN using an array of pointers. Do not change the original array. Use this

sorted array of pointers to perform binary search to get the Book name provided ISBN.

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

```
struct bookdetail
{
    char name[20];
    int ISBN;
};
void display(struct bookdetail **,int );
int isbn_search(struct bookdetail **p, int low, int high,char *key);
void sort(struct bookdetail **,int n);
```

```
int main()
{
    struct bookdetail b[200];
    struct bookdetail *p[200];
    int n;
    char book[200];
    printf("Enter the Numbers of Books:");
    scanf("%d",&n);
    printf("\n");
    for(int i=0;i<n;i++)
    {
        p[i]=&b[i];
        printf("----Book %d Detail-----\n",i+1);
        printf("\nBook Name:\n");
        scanf("%s",b[i].name);
```



```
printf("ISBN of the book:\n");
scanf("%d",&b[i].ISBN);
}
printf("Before sorting\n");
display(p,n);
printf("\n");
sort(p,n);
printf("sorted data based on name is\n");
display(p,n);

char key[1000];
printf("\nEnter the name for searching book: ");
scanf("%s",key);
int book_id=isbn_search(p,0,n-1,key);
    if(book_id == -1)
        printf("not found\n");
    else
        printf("Book name: %s has the id %d\n",key,book_id);
printf("\n");
return 0;
}

void display(struct bookdetail **p,int n)
{
    int i,t=1;
    for(i=0;i<n;i++,t++)
    {
        printf("\n");
        printf("Book No.%d\n",t);
        printf("\t\tBook %d Name is=%s \n",t,p[i]->name);
    }
}
```

```
printf("\t\tBook %d ISBN is=%d \n",t,p[i]->ISBN);
    printf("\n");
}
}

int isbn_search(struct bookdetail **p, int low, int high,char *key)
{
    int found=0;
    int mid;
    // iterative code
    while(low <= high)
    {
        int mid=(low+high)/2;
        if(strcmp(p[mid]->name,key)==0)
        {
            return p[mid]->ISBN;
        }
        else if (strcmp(p[mid]->name,key)>0)
            high=mid-1;
        else
            low=mid+1;
    }
    return -1;
}

void sort(struct bookdetail *p[],int n)
{
    int i,j;
    struct bookdetail s;
    for(i=0;i<n-1;i++)
        for(j=0;j<n-i-1;j++)
            if(strcmp(p[j]->name,p[j+1]->name)>0)
```

```
{  
    struct bookdetail *temp = p[j];  
    p[j] = p[j+1];  
    p[j+1] = temp;  
}
```

}Display appropriate message.

Note: In the above code, array of pointers are sent to searching function as the array of structures are not sorted. Only the pointers in the array of pointers are modified to point to sorted list of books.

Unit #: 4

Unit Name: File Handling and Portable programming

Topic: Callback

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and its associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

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.

Consider the below cases:

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.

We can assign a function name to p as long as the function has the same signature which means the function takes three integer parameters and returns an integer. The function name acts like a pointer to a function.

```
int a2(int,int,int);  
p = a2;  
int y = p(2,3,4); // This statement is the same as calling a2.
```

Why is the callback function required?

Let us answer a few questions to understand the importance of callback function.

How to write a generic library for sorting algorithms such as bubble sort, shell sort, shake sort, quick sort?

How to create a notification event to set a timer in your application?

How to have one common method to develop libraries and event handlers for many programming languages?

How to redirect page action is performed for the user based on one click action?

How to extend the features of one function using another one?

A callback is any executable code that is passed as an argument to other code, which is used to call (execute) the argument at a given time. In simple language, if a function name is passed to another function as an argument to call it, then it will be called as a Callback function.

Coding Example_1: Consider the below simple example to understand callback

```
int what(int x, int y, int z, int (*op)(int, int,int))  
{  
    return op(x, y, z);  
}
```

Observe the third argument op. It is a pointer to a function which takes three int arguments and returns an int. The return statement in turn calls the function with three arguments. This function does not know which function is getting called. It depends on the value of op. Compiler knows the type and not the value. The value of a variable is a runtime mechanism.

```
int add(int x, int y, int z)
{
    return x + y+z;
}
int mul(int x, int y, int z)
{
    return x * y * z;
}
int main()
{
    int (*p)(int, int, int);
    p = add; // p = &add; Both are equivalent
    int res = p(2,3,4); // (*p)(2, 3, 4); //Both are equivalent
    printf("result is %d\n", res);
    p = mul;
    res = p(2,3,4);
    printf("result is %d\n", res);
    // function name can be directly passed as argument to function with callback
    printf("sum is %d\n", what(1, 3, 4, add));
    printf("product is %d\n", what(5, 3, 4, mul));
    return 0;
}
```

Coding Example_2: If you are aware of Python's Functional programming constructs such as map, reduce and filter, we will be implementing those here in C using callback functions.

// Mimic map, filter and reduce function of python

//client_callback.c contains the below code

#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 c = myfilter(a,b,n,is_even); // write appropriate implementation in fun.c for
myfilter function returning integer
    printf("\\nb is ----- \\n");
    disp(b,c);*/

    int m;
    myfilter(a,b,n,&m,is_even); // this myfilter function doesn't return any value
```



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

```
int result = myreduce(a,n,add);  
printf("Result is %d\n",result);  
return 0;
```

```
}
```

// fun.c contains the below code

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

```
void mymap(int a[],int b[],int n,int (*p)(int))
```

```
{
```

```
    //int n = (sizeof(a)/sizeof(*a)); // array degenerates to a pointer at runtime. Think about size  
of pointer
```

```
    //printf("n is %d\n",n);
```

```
    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]);
```

```
    }
```

```
}
```

```
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++)
    //for(int i = 0;i<n-1;i++)
    {
        //res = op(a[i],a[i+1]) // logical error
        //res = op(res,a[i+1]) // logical error
        res = op(res,a[i]);
    }
    return res;
}
```

//fun.h contains below 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 (*)(int));  
int is_greater_than_22(int x);  
int myreduce(int a[],int n,int (*)(int,int));
```

Happy Calling Back!!

Unit #: 4

Unit Name: File Handling and Portable Programing

Topic: Problem Solving: Sorting and Searching using Callback

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and its associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Let us solve the sorting problem using a call back mechanism. Rather than calling two different functions based on the user's choice, call the same function with two different arguments.

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,"");
```

Problem Solving with C – UE23CS151B

```
        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)
    {
```

Problem Solving with C – UE23CS151B

```
        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]);  
  
}
```

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)
```


Problem Solving with C – UE23CS151B

```
        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;
}
```

Unit #: 4

Unit Name: File Handling and Portable Programming

Topic: Qualifiers in C

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and its associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Qualifiers in C

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.

Coding Example_1:

```
#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;
}
```

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.

Note: In a t-bit signed representation of n, the most significant (leftmost) bit is reserved for the sign, “0” means positive, “1” means negative.

Coding Example_2:

```
#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;
}
```

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

Coding Example_3:

```
#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

Coding Example_4: Demo of const

```

#include<stdio.h>
int main()
{
    /*const int i = 5;
    printf("%d\n",i);
    i = 6;    // error
    printf("%d",i);
    */

    /*const int i;

```

```

i = 10; //error
*/

/*
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);
*/

/*const int i = 5;
int *p = &i;
printf("%d\n",*p);
*p = 6; // only warning. But code works
printf("%d\n",*p);
*/

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

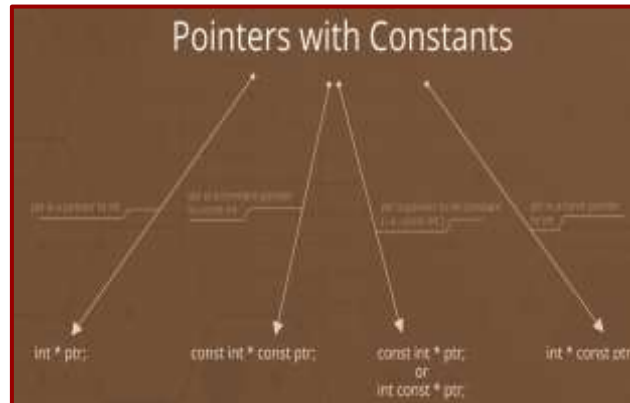
```

```

    //const const int a; // illegal
    return 0;
}

```

Summary:



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

Coding Example_5:

Version1: No volatile keyword added while declaring the variable a. Run this code using `-save-temps` and observe the size of .s(assembly file). Optimization is done.

```

#include<stdio.h>
int main()
{
    int a = 0;
    if(a == 0)
        printf("a is 0\n");
}

```



```

else
    printf("a is not zero\n");
return 0;
}

```

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, .o, .s)
dir program5.i program5.o program5.s(press enter) //windows OS. Check the size of
each
ls -l program6.i program6.o program6.s(press enter) // Ubuntu OS

```

If interested – Check the usage of gcc -O3 program5.c -save-temps. With and without volatile keyword in the code. → Does optimization

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

PES University

Unit #: 4

Unit Name: File Handling and Portable Programming

Topic: Pre-processor Directives

Course Objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course Outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Pre-Processor Directives

Introduction

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

We will discuss each of these in detail with coding examples.

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

For all the below code snippets, use **gcc -E filename.c** to check the output of preprocessing stage on the terminal.

Coding Example_1: Macros doesn't judge anything

```
#include<stdio.h>

#define PI 3.14           // symbolic constant, macro, plain text substitution

// It doesn't judge anything
// No memory is allocated for this

int main()
{
    printf("%f",PI);
    return 0;
}
```

Coding Example_2: No Memory allocation for Macro

```
// name given to 10 is MAX. No memory is allocated for MAX. So &MAX is an error.

#define MAX 10

int main()
```

```
{  
    int a[MAX];  
    scanf("%d", &MAX);           // Error  
    return 0;  
}
```

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 with 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;
}
```

In all the above examples, we created the user defined macro. There are few pre-defined macros in C which are very helpful in performing conditional compilation and portable programming

Coding Example_10: Below program demonstrates few pre-defined constants in C. Comments in the program helps you to understand.

```
#include<stdio.h>

int main()
{
    printf("file name is %s",__FILE__);           // current file name
    printf("\nDate is %s",__DATE__);              // current date
    printf("\nTime is %s",__TIME__);              // time during reprocessing
    printf("\nC version is %d",__STDC_VERSION__);
    //This macro expands to the C Standard version number, a long integer
    constant of the form yyyyymmL where yyyy and mm are the year and month
    of the Standard version
    //Example: 199901L signifies the 1999 revision of the C standard
    printf("\nC version is %d",__STDC__);         //In normal operation, this macro expands to the
    constant 1, to signify that this compiler conforms to ISO Standard C
    printf("\nLine number is %d",__LINE__);
    if( __LINE__ == 12)
        printf("hello");
    else
        printf("%d",__LINE__);
    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

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"`

Note:

- It is not compulsory to have standard file names using angular brackets and user defined files using double quotes. If we change the environment variable `PATH` to have current directory also, then we can use angular brackets for user defined files too. Demo is available in the live video.
- Double quote can be used for both if the standard file is available in the current path[cut paste `stdio.h`]. To avoid these confusions, we follow the convention.

Coding Example_11: Demo of file inclusion directives

```
#include "stdio.h" // possible if it is present in current foldery
```

```
#include <1_file.h> // possible if it is present in PATH
```

```
int main()
```

```
{
```

```
    printf("Hello. lets discuss file inclusion directives\n");
```

```
    return 0;
```

```
}
```

Conditional Compilation Directives

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 #ifndef
    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

Coding Example_17: This program demonstrates #elif

If the result of #if or #ifdef or #ifndef is not true, then the control moves to #elif Preprocessor Directive where it has another constant expression to test. If the constant expression is a non - zero value, then the statements within the scope of #elif will be included for compilation.

```
#include<stdio.h>  
int main()  
{  
    #if (12%5 == 0)
```

```
        printf("Multiple of 5\n");  
#elif (12 % 3 == 0)  
        printf("Multiple of 3\n");  
#endif  
return 0;  
}
```

Output: Multiple of 3

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.

#pragma pack(n) where **n** 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

Note: If the compiler doesn't support pragma, then it ignores that statement. No error

Unit #: 4

File Handling and Portable Programming

Topic: Portable Program Development

Course Objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course Outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Portable Program Development

A portable application is a software **product designed to be easily moved from one computing environment to another**. To make the part of the code to be compiled and executed, we check whether the below macros are set or not.

If mingw in windows system, the value of `__MINGW32__` will be 1

If unix/Linux system is used, the value of `__unix__` is 1

If mac system is used, the value of `__APPLE` is 1

Coding Example_1: The following program demonstrates the real use of conditional compilation and pre-defined macros that makes the code portable across different platforms.

// Use gcc -E option with this code initially to check which part of the code is provided for compilation

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

```
int main()
```

```
{
```

```
    #ifdef __MINGW32__        // No ( and ) // if mingw in windows system is used, the  
    value of this macro is 1
```

```
        printf("windows system");
```

```
        char c,d;
```

```
        printf("enter the character");
```

```
        scanf("%c",&c);
```

```
        fflush(stdin);
```

```
        printf("enter one more character");
```

```
        scanf("%c",&d);
```

```
        printf("entered characters are %c %c",c,d);
```

```
    #elif __unix__ // if unix/Linux system is used, the value of this macro is 1
```

```
        char c;
```

```
        printf("unix system");
```

```
        #include<stdio_ext.h>
```

```
        printf("enter the character");
```

```
        scanf("%c",&c);
```

```
        __fpurge(stdin);
```

```
        printf("enter one more character");
```

```
scanf("%c",&d);
printf("entered characters are %c %c",c,d);
#elif __APPLE__ // if MAC system is used, the value of this macro is 1
printf("mac system");
#else
printf("other system");
#endif
return 0;
}
```

Coding Example_2: Perform the Arithmetic operation using C(demo purpose + and -). User would like to provide inputs as 2+8 and 7-3 OR using enter key between every input key

```
#include<stdio.h>
int main()
{
    int a,b,res;
    char c;
    /*printf("enter the number, sign and enter another number\n");
    scanf("%d%c%d",&a,&c,&b);
    if(c == '+')    res = a+b; // can use switch if considering all operators
    else            res = a-b;
    */
    #if __MINGW32__ // predefined macros. Not defined explicitly in the code
        printf("enter the number");
        scanf("%d",&a);
        fflush(stdin);
        printf("enter the sign");
        scanf("%c",&c);
        printf("enter another number\n");
        scanf("%d",&b);
        if(c == '+')    res = a+b;
        else            res = a-b;
    #elif __unix__
```

```
#include<stdio_ext.h>
printf("enter the number");
scanf("%d",&a);
__fpurge(stdin);
printf("enter the sign");
scanf("%c",&c);
printf("entert another number\n");
scanf("%d",&b);
if(c == '+')          res = a+b;
else                  res = a-b;
#elif __Apple
printf("enter the number");
scanf("%d",&a);
fpurge(stdin);
printf("enter the sign");
scanf("%c",&c);
printf("entert another number\n");
scanf("%d",&b);
if(c == '+')          res = a+b;
else                  res = a-b;
#else
printf("OS is not specific");
#endif
printf("result is %d\n", res);
return 0;
}
```