**System Calls** – Open, Read, Write, Lseek, Close **Unbuffered**

File descriptors – non negative int, returned by open / create, passed to read / write

0/1/2 – input, output, error – stdin / stdout / stderr_fileno <unistd.h>

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );

int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Both return: file descriptor if OK, –1 on error

mode optional permission bits

0400 – read | 0200 – write | 0100 – exec

0040 – read by group | 0777 – read, write, exec all

<fcntl.h> O_RDONLY | O_WRONLY | O_RDWR

O_APPEND – end of file | O_CREAT – create if doesn't exist, set permissions

O_TRUNC – if file exists and its opened for write/read-write truncate its length to 0 | O_EXCL - err for creation exists

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

Returns: file descriptor opened for write-only if OK, –1 on error

equivalent = Open(path, wronly | o_creat | o_trunc, mode)

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns: new file offset if OK, –1 on error

SEEK_SET – offset by offset bytes

SEEK_CUR – cur value + offset

SEEK_END – size of file + offset

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);
```

Returns: number of bytes read, 0 if end of file, –1 on error

reads bytes up to nbytes,

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbytes);
```

Returns: number of bytes written if OK, –1 on error

offset is incremented by bytes actually written

```
int main(int argc, char *argv[]){
 char buffer[512];
 int fd1, fd2;
 long int n1;

 if(((fd1 = open(argv[1], O_RDONLY)) == -1) ||  ((fd2 = open(argv[2],
      O_CREAT|O_WRONLY|O_TRUNC,0700)) == -1)){
    perror("file open/create problem ");
    exit(1);
 }
 while( (  n1 = read(fd1, buffer, 512)  > 0 )  )

     if(write(fd2, buffer, n1) != n1){
           perror("writing problem ");
           exit(3);
      }

// in Case of an error exit from the loop
 if(n1 == -1){
   perror("Reading problem ");
   exit(2);
  }
  close(fd1);
  close(fd2);
  exit(0);
}
```

```
int main(int argc, char *argv[]){
        int fd1, fd2;
        char buffer; // 1 character buffer

        long int i=0, fileSize=0;

        fd1=open(argv[1], O_RDONLY);
        fd2=open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0755);

        while(read(fd1, &buffer, 1)>0)
                fileSize++;
        while(++i <= fileSize){  // differnce between ++i and i++
                lseek(fd1, -i, SEEK_END);
                read(fd1, &buffer, 1);
                write(fd2, &buffer, 1);
        }
        close(fd1);
        close(fd2);
}
```

[No Title]

**Standard Calls** – <stdio.h> stdin, stdout, stderr - equiavalent file pointers to the fd in system calls **Buffered**

Buffering - stderr is unbuffered, stream line buffered for terminal. Otherwise fully buffered

Full – i/o takes place when standard io buffer is full / Line – io takes place when a new line is encountered

Unbuffered – each character initiates a io call, 15 chars = 15 calls

```
#include <stdio.h>

void setbuf(FILE *restrict fp, char *restrict buf);

int setvbuf(FILE *restrict fp, char *restrict buf, int mode,
            size_t size);
```

Returns: 0 if OK, nonzero on error

buf points to static BUFSIZ

_IOFBF, _IOBLF, _IOBNF

Fully buffered – optional buf of size

line buff - optional buf and size optional

unbuf ignores buf, size

Buff null, uses buf of BUFSIZ

| Function | *mode* | *buf* | Buffer and length | Type of buffering |
|---|---|---|---|---|
| `setbuf` | | non-null | user *buf* of length `BUFSIZ` | fully buffered or line buffered |
| | | `NULL` | (no buffer) | unbuffered |
| `setvbuf` | `_IOFBF` | non-null | user *buf* of length *size* | fully buffered |
| | | `NULL` | system buffer of appropriate length | |
| | `_IOLBF` | non-null | user *buf* of length *size* | line buffered |
| | | `NULL` | system buffer of appropriate length | |
| | `_IONBF` | (ignored) | (no buffer) | unbuffered |

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);

FILE *freopen(const char *restrict pathname, const char *restrict type,
              FILE *restrict fp);

FILE *fdopen(int fd, const char *type);
```

All three return: file pointer if OK, `NULL` on error

Freopen - opens a file on a specified stream

Fdopen – associates fd with stream

Input to output using + symbol or vice versa requires rewind, fseek fflush

| *type* | Description | open(2) Flags |
|---|---|---|
| `r` or `rb` | open for reading | `O_RDONLY` |
| `w` or `wb` | truncate to 0 length or create for writing | `O_WRONLY\|O_CREAT\|O_TRUNC` |
| `a` or `ab` | append; open for writing at end of file, or create for writing | `O_WRONLY\|O_CREAT\|O_APPEND` |
| `r+` or `r+b` or `rb+` | open for reading and writing | `O_RDWR` |
| `w+` or `w+b` or `wb+` | truncate to 0 length or create for reading and writing | `O_RDWR\|O_CREAT\|O_TRUNC` |
| `a+` or `a+b` or `ab+` | open or create for reading and writing at end of file | `O_RDWR\|O_CREAT\|O_APPEND` |

| Restriction | r | w | a | r+ | w+ | a+ |
|---|---|---|---|---|---|---|
| file must already exist | • | | | • | | |
| previous contents of file discarded | | • | | | • | |
| stream can be read | • | | | • | • | • |
| stream can be written | | • | • | • | • | • |
| stream can be written only at end | | | • | | | • |

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

return next char if ok
eof on err / eof

```
#include <stdio.h>
long ftell(FILE *fp);
```

Returns: current file position indicator if OK, –1L on error

```
int fseek(FILE *fp, long offset, int whence);
```

Returns: 0 if OK, –1 on error

```
void rewind(FILE *fp);
```

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nobj,
             FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
```

Both return: number of objects read or written

**unformatted io**

char at a time – standard io handles buffering
line at a time - fgets fputs
direct IO – supported by fread, fwrite – binary file

cannot distinguish between err and eof can use
int ferror(file * fp) / int feof(file *fp) – return non-zero if true, 0 false
clear error and eof flag using, void clearer(file *fp);

int ungetc(int c , file *fp) push back char to stream that is read

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);
```

can be implemented as a macro
cannot^
equivalent to putc(c, stdout)

**line at a time**

char * fgets(char * buf, int sizeOfBuffer, file *fp) – returns buf if ok, null if err / eof

```
#include <stdio.h>

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

int puts(const char *str);
```

Both return: non-negative value if OK, EOF on error

Puts - appends new line to standard output

```
#include <stdio.h>

int printf(const char *restrict format, ...);

int fprintf(FILE *restrict fp, const char *restrict format, ...);

int dprintf(int fd, const char *restrict format, ...);
```

All three return: number of characters output if OK, negative value if output error

```
int sprintf(char *restrict buf, const char *restrict format, ...);
```

Returns: number of characters stored in array if OK, negative value if encoding error

```
int snprintf(char *restrict buf, size_t n,
             const char *restrict format, ...);
```

Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

Printf – to stdout
Fprintf – specified stream
Dprintf – file descriptor
Sprintf – to char array

```
#include <stdio.h>

int scanf(const char *restrict format, ...);

int fscanf(FILE *restrict fp, const char *restrict format, ...);

int sscanf(const char *restrict buf, const char *restrict format, ...);
```

All three return: number of input items assigned, EOF if input error or end of file before any conversion

```
int main(int argc, char *argv[]){
        FILE *fp;

        //unsigned char ch;  //what will happen if you uncomment this line and comment the
        char ch;

        int fileSize=-1;

        fp = fopen(argv[1], "r");

        do{
                ch = getc(fp);  //     0X  FF ,   ch =0X 00 00 00 FF

                fileSize++;
                // printf("fileSize=%d\n", fileSize);
                // printf("Char read is ch=%c, in hex ch=%#hhx EOF is %#x\n\n", ch, ch, EOF);
                //sleep(1);
        } while( ch != EOF);  //ch =0x FF, EOF=0x FF FF FF FF
         //   while(!feof(fd));

//      printf(" \nout of do while loop now.\n\n");
//      printf("ch=%d EOF=%#x\n", ch, EOF);
//      printf("size of char =%ld size of EOF=%ld\n", sizeof(char), sizeof(EOF));

        printf("Size of %s is %d\n", argv[1], fileSize);
}
```

```
int main (int argc, char * argv[]) {

        FILE *f1;
        int data[10], i;
        int data1[10];

        for(i=0; i<10; i++)
                data[i] = 10*i*i;

        if(!(f1=fopen(argv[1], "w"))){
                printf("could not create file");  exit(1);
        }

        if(fwrite(data, sizeof(int), 10, f1) != 10){
                printf("Error on writing into file");
                exit(2);
        }
        fclose(f1);

        if(!(f1=fopen(argv[1], "r"))) {
               printf("could open file");
               exit(1);
        }
        if(fread(data1, sizeof(int), 10, f1) != 10)
        {  printf("Could not read data");
                exit(2);
       }

       for(i=0; i<10; i++)
         printf("reading data %d \n", data1[i]);
       fclose(f1);
}
```

Process has a unique non negative integer
Process id 0, scheduler process (swapper) / Process id 1, init process

```
#include <unistd.h>

pid_t getpid(void);
```
Returns: process ID of calling process

```
pid_t getppid(void);
```
Returns: parent process ID of calling process

```
uid_t getuid(void);
```
Returns: real user ID of calling process

```
uid_t geteuid(void);
```
Returns: effective user ID of calling process

```
gid_t getgid(void);
```
Returns: real group ID of calling process

```
gid_t getegid(void);
```
Returns: effective group ID of calling process

```
#include <unistd.h>

pid_t fork(void);
```
Returns: 0 in child, process ID of child in parent, –1 on error

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```
Both return: process ID if OK, 0 (see later), or –1 on error

The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking. • The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

`fork`: allows an existing process to create a new process which is a copy of the caller
`exec`: allows an existing process to be replaced with a new one.
`wait`: allows a process to wait for one of its child processes to finish and also to get the termination status value.

```c
int i;
FILE *f;
 char buffer[100];

f=fopen(argv[1], "w");

printf("before fork, my pid is %d\n" , getpid());

for (i=0; i<3; i++){
    if ( fork() == 0){
        printf("Hi, I am child. My pid is %d, myppid=%d\n", (int)getpid(), (int)getppid());
        //uses sleep to add buffer time in order to handle orphans
        //might need to increase time in order to work effectivly
        sleep(2000);
        //exits program such that child does not get forked
        exit(0);
    }
    //write it to a file
    sprintf(buffer, "Hi, I am child. My pid is %d, myppid=%d\n", (int)getpid(), (int)getppid());
    fputs(buffer, f);
}
fclose(f);
```

Writing to file sys call
print file size
counting lines standard calls

```c
char buffer[512];
int file_descriptor;
int bytes_read;
char *filename = argv[1];
//open file for read and write
file_descriptor = open(filename, O_RDWR);
//check for error
if (file_descriptor == -1) {
    perror("Error opening the file");
    return 1;
}
//while not at the end of the file
while ((bytes_read = read(file_descriptor, buffer, 512)) > 0) {
    // write the content to the terminal (STDOUT_FILENO)
    write(STDOUT_FILENO, buffer, bytes_read);
}
char string[100];
printf("Enter input: ");
//gets and input string
fgets(string, sizeof(string), stdin);
//use lseek to get to the end of the file
lseek(file_descriptor, 0, SEEK_END);
//wrote to the file the nre string
write(file_descriptor, string, strlen(string));
//rewind to the start of the file
lseek(file_descriptor, 0, SEEK_SET);
//repeat above steps
while ((bytes_read = read(file_descriptor, buffer, 512)) > 0) {
    write(STDOUT_FILENO, buffer, bytes_read);
}
close(file_descriptor);
```

```c
int main(int argc, char *argv[]){
    //get and open file
    char *filename = argv[1];
    int fd = open(filename, O_RDONLY);
    //check for error
    if (fd  == -1){
        printf("error in opening file\n");
        exit(1);
    }

    off_t size = lseek(fd, 0, SEEK_END);
    //use lseek to set the file size

    if (size == -1) {
    //check for errors
        perror("Error");
        close(fd);
        return 1;
    }

    printf("size = %ld\n", (long)size); //
```

```c
char *filename = argv[1];
//open function
FILE *fp = fopen(filename,"r");
int ch;
int lines = 0;
//error handeling
if (fp == NULL) {
    printf("error in opening file\n");
    return 0;
}
lines++;
//added too lines every time new line c
while ((ch = fgetc(fp)) != EOF){
    if (ch == '\n')
        lines++;
}
fclose(fp);
printf("lines: %d\n", lines);
```

Standard calls

```c
char buffer[512];
FILE *file;
char *filename = argv[1];
//open the file
file = fopen(filename, "r+");
//check for errors
if (file == NULL) {
    perror("Error opening the file");
    return 1;
}
// Read until file is empty
while (fgets(buffer, sizeof(buffer), file) != NULL) {
    // print to the terminal
    fputs(buffer, stdout);
}
//take in a string
char string[100];
printf("Enter input: ");
scanf("%s", string);
fputs(string, file);
// Rewind to the beginning of the file
rewind(file);
// Print the entire file again
while (fgets(buffer, sizeof(buffer), file) != NULL) {
    fputs(buffer, stdout);
}
fclose(file);
```

Prints to terminal, input string to file, file to terminal

```c
char prev_char = '\0';
char current_char;
char *filename = argv[1];
//open the file
int fd = open(filename, O_RDONLY);
if (fd == -1) {
//check for error
    perror("error in opening file\n");
    return 1;
}
//read until end of file
while (read(fd, &current_char, 1) > 0) {
    //if current char is new line add 1
    if (current_char == '\n') {
```

```c
FILE *inputFile, *outputFile;
char buffer[256];
//open output file
outputFile = fopen(argv[1], "w");
if (outputFile == NULL) {
    perror("Error opening output file");
    return 1;
}
//itterate through the files
for (int i = 2; i < argc; i++) {
    //open file
    inputFile = fopen(argv[i], "r");
    //check for errors in the opened file
    if (inputFile == NULL) {
        fprintf(stderr, "Error");
        continue;
    }
    //write contents of file too output file
    while (fgets(buffer, 256, inputFile)) {
        fputs(buffer, outputFile);
    }
    fclose(inputFile);
}
fclose(outputFile);
```

Creating a hole

```c
FILE *file;
char buffer1[] = "This is the beginning";
char buffer2[] = "This is the end";
//open file and check for errors
char *filename = argv[1];
if((file = fopen(filename, "w+")) == NULL){
    perror("Error opening the file");
    exit(1);
}
// Write the contents of buffer1 to the beginning of the file
if (fwrite(buffer1, 1, sizeof(buffer1), file) != sizeof(buffer1)) {
    exit(2);
}
// Move the file pointer to position 16384 from the beginning of the
if (fseek(file, 16384, SEEK_SET) != 0) {
```

Counting lines In sys call

```c
char *array[MAX_LINES];
char buffer[MAX_LINE_LENGTH];
FILE *inputFile, *outputFile;
int count = 0;
//open file
inputFile = fopen("input.txt", "r");
//check for errors opening file
if (inputFile == NULL) {
    perror("Error");
    return -1;
}
//appends lines to array
while (fgets(buffer, MAX_LINE_LENGTH, inputFile)) {
    array[count] = strdup(buffer);
    count++;
}
//close file
fclose(inputFile);
//open file
outputFile = fopen("output.txt", "w");
//itterate through the array backwards
for (int i = count - 1; i >= 0; i--) {
    //if the last line doesnt end in a \n make one
    if (i == count - 1 && array[i][strlen(array[i]) - 1] != '\n') {
        fprintf(outputFile, "%s\n", array[i]);
        //else print normally
    } else {
        fprintf(outputFile, "%s", array[i]);
    }
```

Reverses lines

Concat x # of files

```c
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

All seven return: –1 on error, no return on success

- **System Calls and File Descriptors: These are the basic methods by which a program interacts with the operating system to perform operations like opening, reading, writing, seeking, and closing files. File descriptors are simply numbers that uniquely identify an open file within a process.**
- **Standard I/O and Buffering: Explains how standard input, output, and error streams work in Unix, along with the concept of buffering to optimize reading from and writing to these streams.**
- **Mode and Permission Bits: Details on how files have different modes (read, write, execute) and permissions that determine who can do what with a file.**
- **Open, Read, Write, Seek, Close Functions: These functions are used to manipulate files at a low level, providing precise control over file operations.**
- **Error Handling with `errno`, `perror`: Methods for identifying and responding to errors that occur during file operations.**
- **Buffer Sizes and Types: Discussion on how data is buffered (stored temporarily) during input and output operations to enhance performance.**
- **Function Descriptions: Brief explanations of various functions like `freopen`, `fdopen`, `fgets`, `fputs`, `printf`, `fprintf`, `sprintf`, and how they are used for file and data manipulation.**
- **Process Identifiers: Information on how each process in Unix is identified by a unique process ID, with special IDs for the scheduler and init process.**
- **Waiting for Processes: The difference between `wait` and `waitpid` functions in terms of blocking behavior and specificity.**

**Each point covered in the document provides foundational knowledge necessary for understanding how programs interact with the Unix operating system at a low level, particularly concerning file and process management.**

- `char buffer[512];` declares a buffer array of 512 characters to store data read from a file.
- `int file_descriptor;` declares an integer to store the file descriptor returned by open.
- `int bytes_read;` will hold the number of bytes read by `read`.
- `char *filename = argv[1];` sets a pointer to the first command-line argument, which is the file name.
- `file_descriptor = open(filename, O_RDWR);` opens the file with read and write permissions.
- The `if` statement checks if the file opened successfully. If `file_descriptor` is -1, an error occurred.
- `perror("Error opening the file");` prints the error message.
- `return 1;` exits the program with a status code indicating an error.
- The `while` loop continues as long as `read` returns more than 0, which means there's data to read.
- `write(STDOUT_FILENO, buffer, bytes_read);` writes the data read into the buffer to the standard output (usually the terminal).
- `char string[100];` declares an array to store user input.
- `printf("Enter input: ");` prompts the user.
- `fgets(string, sizeof(string), stdin);` reads input from the user and stores it in `string`.

- `lseek(file_descriptor, 0, SEEK_END);` moves the file offset to the end of the file.
- `write(file_descriptor, string, strlen(string));` writes the user input to the end of the file.
- `lseek(file_descriptor, 0, SEEK_SET);` moves the file offset back to the beginning of the file.
- The second `while` loop reads the file again after the new input is added and writes it to the standard output.
- `close(file_descriptor);` closes the file.

- `int main(int argc, char *argv[])`: This is the starting point of a C program. `argc` is the argument count, and `argv` is an array of arguments passed to the program.
- `char *filename = argv[1];`: This line assigns the second argument passed to the program to a pointer named `filename`. The first argument (`argv[0]`) is the program name.
- `int fd = open(filename, O_RDONLY);`: Here, the `open` system call tries to open the file specified by `filename` in read-only mode. The file descriptor returned is stored in `fd`.
- `if (fd == -1)`: This conditional checks if `open` failed (which would return `-1`).
- `printf("error in opening file\n");`: If `open` failed, an error message is printed.
- `exit(1);`: The program exits with a return code of 1, indicating an error.

Next, it moves to determine the size of the file:

- `off_t size = lseek(fd, 0, SEEK_END);`: The `lseek` function sets the file offset to the end of the file, effectively measuring the file size.
- `if (size == -1)`: This checks if `lseek` failed.
- `perror("Error");`: If there's an error, `perror` prints it along with a descriptive message.
- `close(fd);`: Before exiting due to the error, the file is closed.
- `return 1;`: The program exits with an error status.

Finally, it prints the file size:

- `printf("size = %ld\n", (long)size);`: This prints out the size of the file as a long integer, which is the appropriate format for `off_t` on most systems.

This code snippet opens a file, checks for errors, determines the file size, and prints that size to the screen. If any errors occur during opening or determining the size of the file, the program prints an error message and exits with a status code of 1.

`char *filename = argv[1];`: This line gets the filename from the command line arguments.
`FILE *fp = fopen(filename, "r");`: Opens the file with read-only access and assigns the file pointer `fp`.
`int ch;`: Declares an integer to store characters read from the file.
`int lines = 0;`: Initializes a line counter to zero.
`if (fp == NULL) {`: Checks if the file was successfully opened.
`printf("error in opening file\n");`: If not, prints an error message.
`return 0;`: And exits the function returning 0.

Assuming there's an error in the provided snippet since the `lines++` should be inside the while loop to count the lines correctly. Here's what should follow:

`while ((ch = fgetc(fp)) != EOF) {`: Reads the file character by character until the end of the file.
`if (ch == '\n')`: Checks if the character is a newline character.
`lines++;`: If so, increments the line count.
`}`: Closes the while loop.
`fclose(fp);`: Closes the file.
`printf("Lines: %d\n", lines);`: Prints the total number of lines.

This snippet is intended to count the number of lines in a text file. It reads through the file character by character, incrementing the line count every time it encounters a newline character. After reaching the end of the file, it prints out the total line count and then closes the file.

- `char buffer[512];` declares a buffer to hold the data read from the file.
- `FILE *file;` declares a pointer to a `FILE` object.
- `char *filename = argv[1];` gets the filename from the command-line arguments.
- `file = fopen(filename, "r+");` opens the file for reading and writing.
- The `if` statement checks if the file was successfully opened.
- `perror("Error opening the file");` prints an error message if the file couldn't be opened.
- `while (fgets(buffer, sizeof(buffer), file) != NULL) {` reads from the file until there's nothing left.
- `fputs(buffer, stdout);` writes the contents of `buffer` to the standard output.
- `char string[100];` declares a string to store input from the user.
- `printf("Enter input: ");` prompts the user for input.
- `scanf("%s", string);` reads a string from the standard input.
- `fputs(string, file);` writes the input string to the file.
- `rewind(file);` moves the file pointer back to the beginning of the file.
- The second `while` loop reads the updated file and outputs it to the terminal.
- `fclose(file);` closes the file.

The program is designed to display the contents of a file, then get input from the user, add that input to the file, and display the updated contents.

- `char prev_char = '\0';` initializes a character variable to store the previous character read from the file, starting with the null character.
- `char current_char;` declares a character variable to store the current character read from the file.
- `char *filename = argv[1];` sets a character pointer to the first command-line argument, which should be the file name.

- `int fd = open(filename, O_RDONLY);` opens the file in read-only mode and returns a file descriptor.
- `if (fd == -1) {` checks if the file descriptor is -1, which indicates an error in opening the file.
- `perror("error in opening file\n");` prints the system error message if an error occurred.
- `return 1;` exits the program with a return value of 1 to indicate an error.
- `while (read(fd, &current_char, 1) > 0) {` reads the file one character at a time in a loop.
- `if (current_char == '\n') { lines++; }` increments the line count each time a newline character is encountered.
- `prev_char = current_char;` after each read, the current character becomes the previous character for the next iteration.
- `if (prev_char != '\n') { lines++; }` after the loop, if the last character wasn't a newline, it adds one to the line count to account for the last line.
- `close(fd);` closes the file descriptor.

**Second Code Snippet:**

- `FILE *file;` declares a file pointer.
- `char buffer1[] = "This is the beginning";` initializes a buffer with a string.
- `char buffer2[] = "This is the end";` initializes a second buffer with a string.
- `char *filename = argv[1];` sets a character pointer to the first command-line argument, which should be the file name.
- `if((file = fopen(filename, "w+")) == NULL){` opens the file for reading and writing (creating it if it doesn't exist) and checks if the file pointer is NULL, which indicates an error.
- `perror("Error opening the file");` prints the system error message if there's an error opening the file.
- `exit(1);` exits the program with a status code of 1, indicating an error.
- `if (fwrite(buffer1, 1, sizeof(buffer1), file) != sizeof(buffer1)) { exit(2); }` writes the contents of `buffer1` to the file and checks if the write operation was successful.

- `if (fseek(file, 16384, SEEK_SET) != 0) { exit(1); }` moves the file pointer to a specific location in the file, and exits if this operation fails.
- `if (fwrite(buffer2, 1, sizeof(buffer2), file) != sizeof(buffer2)) { exit(1); }` writes the contents of `buffer2` to the new position in the file and checks if the write operation was successful.
- `fclose(file);` closes the file.
- `exit(0);` exits the program with a status code of 0, indicating success.

**Third Code Snippet:**

- `FILE *inputFile, *outputFile;` declares pointers for the input and output files.
- `char buffer[256];` declares a buffer to hold file content temporarily.
- `outputFile = fopen(argv[1], "w");` opens the first command-line argument as an output file to write to.
- `if (outputFile == NULL) {` checks if the file was successfully opened.
- `perror("Error opening output file");` prints an error message if the output file could not be opened.
- `return 1;` exits with a return code of 1 to indicate an error.
- `for (int i = 2; i < argc; ++i) {` starts a loop over the remaining command-line arguments.
- `inputFile = fopen(argv[i], "r");` opens the next file in the list for reading.
- `if (inputFile == NULL) { fprintf(stderr, "Error"); continue; }` checks for an error opening the input file and continues to the next if there's an error.
- `while (fgets(buffer, 256, inputFile)) { fputs(buffer, outputFile); }` reads from the input file and writes to the output file.
- `fclose(inputFile);` closes the current input file before moving on to the next.
- `fclose(outputFile);` closes the output file after all input files have been processed.

**Fourth Code Snippet:**

- `char *array[MAX_LINES];` declares an array of string pointers to hold lines from the file.
- `FILE *

## Page 5 and 6 codes^^^^

System I/O

System I/O in UNIX is designed around file descriptors, which are non-negative integers acting as references to open files. The core functions for working with file descriptors include open(), read(), write(), and close():

open() is used to open a file and obtain a file descriptor.

read() and write() are used to read data from and write data to these file descriptors, respectively.

close() is used to close an open file descriptor, signaling that you're done using it.

File descriptors are a low-level mechanism for I/O operations, providing direct access to the underlying files or devices.

Standard I/O Library

The Standard I/O library provides a higher-level interface compared to System I/O, abstracting file descriptors into file streams. This library is easy to use and handles details like buffering, making I/O operations more efficient without needing to manage block sizes. It's part of the ISO C standard, making it portable across different systems.

Streams and FILE Objects: Unlike file descriptors, streams are used in the Standard I/O library, represented by FILE objects. Opening a file with this library associates a stream with that file, which can be used for buffered I/O operations.

Buffering: The Standard I/O library automatically buffers data, reducing the number of system calls and improving performance. It supports three types of buffering: fully buffered, line buffered, and unbuffered. The behavior of these buffers can be customized with functions like setbuf and setvbuf.

Formatted I/O: The library provides functions like printf and scanf for formatted input and output, making it easier to work with different data types.

Process Control

Process control involves managing and manipulating processes within the operating system. This includes creating new processes, executing different programs within processes, and performing operations like waiting for processes to change state.

Creating Processes: UNIX systems use the fork() system call to create a new process. The new process, called the child, is an exact copy of the calling process, called the parent.

Executing Programs: After creating a new process, you might want to run a different program within it. This is achieved with one of the exec() family of functions, which replaces the current process image with a new program image.

Process Termination: Processes can terminate normally or due to a signal. The exit() function terminates a process and returns an exit status to the parent process.

Waiting for Process Termination: Parent processes can wait for their children to terminate using the wait() or waitpid() functions, which also allow the parent to retrieve the child's exit status.

1. Finding Paths: The Maze Adventure

Imagine you're in a maze looking for treasure. There are many paths, but some are longer, and some are shorter. We need to find the best path to the treasure without getting too tired or lost.

2. Types of Paths: Friends Helping You Search

BFS (Breadth-First Search) is like having a friend who checks every path close to them before moving on. They make sure not to miss anything nearby, but it might take them a long time to reach faraway places.

DFS (Depth-First Search) is like a friend who picks one direction and keeps going as far as they can until they can't go anymore, then tries a new direction. They might find deep, hidden spots quickly but could miss easier paths close by.

UCS (Uniform Cost Search) is a friend who always takes the cheapest path, even if it's not directly towards the treasure. They save energy but might take a long route.

A (A-Star)* is the smartest friend, who uses a magic map that shows both how far away the treasure is and how hard the path is. They find the quickest, easiest way to the treasure.

3. Playing Fair: Rules of the Game

Heuristic Functions: For A*, imagine you have a magic telescope that can guess how far you are from the treasure. It's not always right, but it helps you decide which way to go next.

Complexity: Think of this as how big your backpack is and how fast you can run. Some friends might fill their backpacks with too many things (taking up a lot of space) or take a long time to decide which way to go.

4. Special Tricks: Secret Powers

Advanced Techniques: Sometimes, our friends learn new tricks. For example, one might learn to use a bouncy rope to go back quickly (Iterative Deepening) or send messages to each other to find the best path together (Bidirectional Search).

5. Choosing the Right Path: Picking the Best Friend for the Adventure

Not every friend is perfect for every adventure. If the treasure is close, your friend who checks nearby paths (BFS) might find it quickly. If you think the treasure is hidden deep, the friend who dives deep (DFS) could be your best bet. If you want to save energy, the friend who always looks for cheap paths (UCS) will help. But if you want the smartest, quickest way, the friend with the magic map (A*) is the way to go.

6. The Big Picture: Why We Explore Mazes

Finding treasure in mazes is like solving problems. Each friend has their own way of solving problems, and learning about them helps us think about the best way to tackle our own challenges, whether it's doing homework, cleaning our room quickly, or finding the fastest way to the playground.

7. Making It Fun: Games and Puzzles

Just like in games and puzzles, exploring different paths teaches us there are many ways to find a solution. Sometimes, we need to try a few before finding the best one.

And there you have it! Each search method is like a friend with their own special way of finding treasure in the maze. By learning about them, we get better at picking which friend to bring along on our adventures.

---

# Chapter one

Introduction and Overview: It sets the stage for understanding Unix, its significance in the realm of operating systems, and system programming.

Operating Systems Basics: Highlights the role of an operating system, including its essential services like program execution, resource sharing, and communication.

History of Unix: Traces Unix's evolution from its inception at Bell Labs to its various versions and distributions, emphasizing its efficiency and the introduction of C programming language for system development.

Unix Features and Philosophy: Describes Unix's availability across platforms, multi-user capabilities, and its design principles of simplicity and modularity, achieved through utilities that perform specific tasks well.

Unix Layers: Explains the architecture of Unix, detailing the kernel, application programs, the shell (command interpreter), and the windowing system, and how users interact with these components.

Login Process: Details how users log into Unix systems, the role of the /bin/login command, and the structure of user credentials in /etc/passwd and /etc/shadow.

User and Group Identification: Discusses how Unix differentiates users and groups through unique IDs and the importance of these IDs in system security and resource management.

Shells: Introduces different types of Unix shells, their case sensitivity, and their role as command-line interpreters.

Shell Commands: Lists useful shell commands for tasks like finding manual pages, listing processes, changing passwords, and file manipulation.

Pipes and Data Flow: Illustrates how pipes allow chaining commands together, enabling the output of one command to serve as input to another.

File System and Directories: Describes the hierarchical structure of the Unix file system, including paths, directories, and file operations.

Standard I/O and Redirection: Explains the concepts of standard input, output, and error streams, and how they can be redirected in Unix.

Processes and Process Control: Delves into Unix processes, how they are created, managed, and communicated with, highlighting system calls like fork, exec, and wait.

Signals: Covers Unix signals, which are notifications sent to a process to indicate the occurrence of specific events.

Time Measurement and System Calls: Discusses how Unix measures process execution time and the distinction between system calls and library functions.

## Chapter 2:

Unix I/O System Calls: Imagine telling a robot (Unix) to do tasks like opening a box (file), looking inside (reading), putting things inside (writing), moving to a different part of the box (seeking), and closing the box when done (closing). These tasks are done using special commands.

File Descriptors: Think of these as labels or numbers the robot uses to remember which box it opened, so it doesn't get confused if there are many boxes.

Opening and Closing Files: When you want to open a box, you tell the robot exactly how (just to look, add things, or both), and it gives you a number (file descriptor) so you can refer to this box. Closing a box means you're done with it, so the robot can forget the number and use it for a new box later.

Reading and Writing: Reading is like looking into the box and counting how many toys (bytes) you can see up to a certain number. Writing is putting toys into the box. The robot tells you how many toys you saw or put in, so you know everything went well.

Seeking (lseek): This is when you tell the robot to move to a specific spot in the box before doing anything else, like skipping straight to your favorite part of a storybook.

Errors and Problems: If the robot has trouble, it will tell you what went wrong, like if it can't open a box or put things in. This way, you can try to fix the problem.

Think of Unix handling files like a very obedient robot that does exactly what you tell it to, but you need to use special commands it understands.

## Chapter 3

Opening Files: It's like telling your toy, "Hey, let's start recording" or "Let's listen to what we recorded yesterday." The computer needs to know we're ready to either look at what's in a file or add something new to it.

Reading and Writing: Once the file is open, it's like reading what you wrote in your diary or recording a new message on your toy. The computer can take information from the file to show us or add new information we tell it to.

Closing Files: When we're done, just like closing your diary or turning off your toy, we tell the computer to close the file. This way, everything is kept safe and ready for next time.

Special Commands for Problems: Sometimes, things don't go as planned—maybe the diary is locked, or the toy's batteries are low. The computer has special ways to tell us something went wrong, so we can fix it.

Playing Backwards or Skipping Ahead: Just like you can skip to your favorite part of a storybook, the computer can move around in a file, not just go from the beginning to the end.

The presentation teaches us that computers use a system to open, read, write, and close files—kind of like how you might interact with your favorite toys or books. It's a way to make sure the computer can safely and efficiently keep our stories and information for us to come back to anytime.

## Chapter 4

- **Binary I/O**: It's like using a more advanced toy that can handle lots of tiny pieces (like LEGO blocks) all at once, instead of one block at a time. This is much faster and more efficient when dealing with a lot of data.
- **Random Access**: Imagine a book where you can quickly flip to any page you want, instead of reading it in order. Computers can do this with files, jumping straight to the part they need.
- **Formatted I/O**: This is about making sure the data looks right, whether it's numbers or letters. It's like drawing within the lines, making sure everything is neat and understandable.
- **Error Handling (`errno` and `perror`)**: Sometimes, things go wrong, like when a toy breaks or doesn't work as expected. Computers have a way to understand what went wrong (using `errno`) and tell you about it (using `perror`), so you can try to fix it.

## Chapter 5

- **Introduction to Unix Processes**: Think of a process like a worker in a factory. Each worker has a unique badge number (process ID) and a specific job they do using tools (data and code).
- **Creating a New Process with `fork()`**: This is like a worker cloning themselves to do a task twice as fast. The original worker is the parent, and the clone is the child. They're almost identical but have different badge numbers.
- **Terminating a Process with `exit()`**: When a worker finishes their job, they sign out. This tells the boss (the parent process) they're done, so the boss can continue with other work.
- **Waiting for a Process with `wait()` and `waitpid()`**: Sometimes, a boss waits for a worker to finish a task before starting something new. `wait()` is like the boss waiting at the door for any worker, while `waitpid()` waits for a specific worker.

## Chapter 6

Orphan and Zombie Processes

Orphan Processes: These occur when a parent process exits (e.g., is terminated) while its children processes are still running. The operating system reassigns these orphaned processes to the init process, which then becomes their new parent. This ensures that the orphaned processes still have a parent process to manage them.

Zombie Processes: A process becomes a zombie if it has terminated, but its parent process has not yet retrieved its termination status. The system keeps the terminated process in a zombie state to allow the parent process to read its exit status. If the parent never calls wait(), the child remains in this undead state, occupying system resources.

Advanced Process Management

Exec Functions: These are used to replace the current process image with a new one. This is akin to a worker deciding to do a completely different job, requiring a total change in tools and instructions.

Changing Working Directories (chdir()): Allows a process to change its current working directory. It's like a worker moving to a different part of the factory to work on another task.

Priority Adjustment (nice() and setpriority()): These functions adjust the priority level of a process, determining how much CPU time it gets compared to others. It's like scheduling which tasks or workers should be prioritized based on urgency.

Process Groups and Sessions: Processes can be organized into groups and sessions for better management, signaling, and communication. This helps in coordinating tasks that need to work closely together.

These segments explain how Unix systems manage processes throughout their lifecycle, from creation to termination, and how they handle special scenarios like orphaned or zombified processes. Additionally, it covers how processes can change their behavior or state to adapt to new tasks or improve their execution environment.

## Chapter 7

- **Definition and Role**: Signals are a form of software interrupt that provide a way for the operating system or a program to notify a process about an event.
- **Generation and Delivery**: Explains how signals are generated (by the system, users, or errors) and delivered to processes, detailing synchronous (direct result of process's actions) and asynchronous signals (external to the process's current activities).
- **Handling Signals**: Discusses the various ways processes can manage signals, including default handling, ignoring signals, or defining custom handlers using system calls like `signal()` or `sigaction()`.
- **Specific Signal Functions**: Detailed exploration of functions involved in signal handling, such as:
  - `signal()`: Sets a function to handle a signal.
  - `sigaction()`: Offers more control over signal handling than `signal()`.
  - `kill()`: Sends a signal to a process or a group of processes.
  - `raise()`: Allows a process to send a signal to itself.
  - `sigpending()`: Checks which signals are blocked and pending.
  - `sigprocmask()`: Blocks or unblocks signals.
  - `pause()`: Suspends the process until a signal is received.
- **Examples and Code Snippets**: Provides practical coding examples showing how to implement signal handling in applications, such as custom handlers for `SIGINT` (interrupt signal) or using `sigprocmask()` to block signals during a critical section of code.

Each topic is meticulously explained with the intention of making the concept of Unix signals accessible, from their theoretical foundation to practical application in system programming.

Here's the explanation for the program found in `count_lines_sys.c`:

- `#include <stdio.h>`: Includes the Standard Input and Output Library for basic I/O operations.
- `#include <unistd.h>`: Includes the POSIX operating system API for various system calls.
- `#include <fcntl.h>`: Includes the File Control options for handling file operations.
- `int main() {`: Defines the main function where the program execution starts.
- `int fd = open("list1.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);`: Opens `list1.txt` for writing only, creating it if it doesn't exist, or truncating it to zero length if it does, with permissions set to 0644.
- `if (fd == -1) {`: Checks if the file descriptor `fd` is -1, indicating an error in opening the file.
- `perror("Error opening file");`: Prints the error message "Error opening file" if the file could not be opened.
- `return 1;`: Exits the program with a return code of 1, indicating an error.
- `char *data = "1011 GM\tBuick\t2014\n1024 Ford\tLincoln\t2025\n";`: Initializes a character pointer `data` with a string to write to the file.
- `if (write(fd, data, 45) != 45) {`: Writes the data to the file and checks if the write operation was not exactly 45 characters, indicating an error.
- `perror("Error writing to file");`: Prints the error message "Error writing to file" if the write operation failed.
- `close(fd);`: Closes the file descriptor to release resources.
- `return 1;`: Exits the program with a return code of 1, indicating an error during the write operation.
- `close(fd);`: Closes the file descriptor, releasing the resources associated with it.
- `return 0;`: Returns 0 from `main`, indicating successful program execution.

This program demonstrates how to open, write to, and close a file using system calls in C.

Here's the breakdown of `count_lines_std.c`:

- `#include <stdio.h>`: Includes the Standard Input and Output Library for basic I/O operations.
- `int main(int argc, char *argv[]) {`: Starts the main function, taking command-line arguments for the program execution.
- `if (argc != 2) {`: Checks if the number of command-line arguments is not equal to 2 (program name and filename).
- `printf("Usage: %s filename\n", argv[0]);`: Informs the user of the correct way to run the program if the argument check fails.
- `return 1;`: Exits the program with a status code of 1, indicating an error.
- `FILE *file = fopen(argv[1], "r");`: Attempts to open the specified file in read mode.
- `if (file == NULL) {`: Checks if the file failed to open.

- `perror("Error opening file");`: Prints an error message if the file cannot be opened.
- `return 1;`: Exits the program with a status code of 1 due to the failure in opening the file.
- `int lines = 0;`: Initializes a counter for the number of lines in the file.
- `char ch;`: Declares a character variable to read the file, character by character.
- `while ((ch = fgetc(file)) != EOF) {`: Reads the file character by character until the end of file (EOF).
- `if (ch == '\n') lines++;`: Increments the line counter each time a newline character is encountered.
- `printf("Number of lines: %d\n", lines);`: Prints the total number of lines found in the file.
- `fclose(file);`: Closes the file to free up system resources.
- `return 0;`: Successfully exits the program with a status code of 0.

This program counts the number of lines in a text file by reading through it character by character and incrementing a counter every time it finds a newline character.

Here's the breakdown of `file_manip_sys.c`:

- `#include <stdio.h>`: Includes the Standard Input and Output Library.
- `#include <unistd.h>`: Includes POSIX operating system API.
- `#include <fcntl.h>`: Includes the File Control library for file operations.
- `#include <string.h>`: Includes the String handling library.
- `int main(int argc, char *argv[]) {`: Begins the main function with command-line arguments.
- `if (argc != 2) {`: Checks if the program was not run with exactly one additional argument.
- `printf("Usage: %s filename\n", argv[0]);`: Instructs the user on how to properly use the program if the argument check fails.
- `return 1;`: Exits the program due to incorrect usage.
- `int fd = open(argv[1], O_RDWR | O_APPEND);`: Opens the specified file in read-write mode, appending to the end of the file.
- `if (fd == -1) {`: Checks if there was an error opening the file.
- `perror("Error opening file");`: Prints an error message if the file could not be opened.
- `return 1;`: Exits the program because the file couldn't be opened.
- `char buffer[1024];`: Allocates a buffer for reading the file's contents.
- `ssize_t bytes_read;`: Declares a variable to store the number of bytes read.
- `while ((bytes_read = read(fd, buffer, sizeof(buffer) - 1)) > 0) {`: Reads the file's content into the buffer.
- `buffer[bytes_read] = '\0';`: Null-terminates the string read from the file.
- `printf("%s", buffer);`: Prints the contents of the file read into the buffer.
- `printf("\nEnter data to append: ");`: Prompts the user to enter data to append to the file.
- `char input[256];`: Allocates a buffer for the user's input.
- `scanf("%255s", input);`: Reads the user's input.
- `write(fd, input, strlen(input));`: Writes the user's input to the file.
- `write(fd, "\n", 1);`: Appends a newline character after the user's input.
- `lseek(fd, 0, SEEK_SET);`: Seeks back to the beginning of the file.

- `while ((bytes_read = read(fd, buffer, sizeof(buffer) - 1)) > 0) {`: Reads the updated file content.
- `buffer[bytes_read] = '\0';`: Ensures the buffer is null-terminated.
- `printf("%s", buffer);`: Prints the updated file content.
- `close(fd);`: Closes the file descriptor.
- `return 0;`: Exits the program successfully.

This program opens a specified file, reads its content, allows the user to append text, and then reads the updated content back to the user.

Here's the breakdown of `file_manip_std.c`:

- `#include <stdio.h>`: Includes the Standard Input and Output Library for I/O operations.
- `#include <string.h>`: Includes the String handling library for manipulating arrays of characters.
- `int main(int argc, char *argv[]) {`: Starts the main function with command-line arguments for program execution.
- `if (argc != 2) {`: Checks if the program was not executed with exactly one additional argument (the filename).
- `printf("Usage: %s filename\n", argv[0]);`: Informs the user how to correctly run the program if the argument count is incorrect.
- `return 1;`: Exits the program with a return code of 1, indicating an error.
- `FILE *file = fopen(argv[1], "a+");`: Opens or creates the specified file for reading and appending.
- `if (file == NULL) {`: Checks if the file failed to open.
- `perror("Error opening file");`: Prints an error message if the file cannot be opened.
- `return 1;`: Exits the program with a failure status due to file opening error.
- `fseek(file, 0, SEEK_SET);`: Moves the file pointer to the beginning of the file for reading.
- `char buffer[1024];`: Declares a buffer for storing file contents during reading.
- `while (fgets(buffer, sizeof(buffer), file) != NULL) {`: Reads the file line by line into the buffer.
- `printf("%s", buffer);`: Prints each line stored in the buffer to the console.
- `printf("\nEnter data to append: ");`: Prompts the user for input to append to the file.
- `char input[256];`: Declares a buffer to store the user input.
- `scanf("%255s", input);`: Reads a string from the user, with a maximum length of 255 characters.
- `fprintf(file, "%s\n", input);`: Appends the user input to the file, followed by a newline character.
- `fseek(file, 0, SEEK_SET);`: Moves the file pointer back to the beginning of the file to read the updated content.
- `while (fgets(buffer, sizeof(buffer), file) != NULL) {`: Reads the updated file content line by line.
- `printf("%s", buffer);`: Prints the updated file contents to the console.
- `fclose(file);`: Closes the file to free system resources.
- `return 0;`: Successfully exits the program with a status code of 0.

This program demonstrates how to open a file for appending, read its content, append new data entered by the user, and then read and display the updated content.

Here's the explanation for `get_file_size.c`:

- `#include <stdio.h>`: Includes the Standard Input and Output Library.
- `#include <unistd.h>`: Includes POSIX operating system API for miscellaneous symbolic constants and types.
- `#include <fcntl.h>`: Includes the File Control Options library for file descriptor manipulations.
- `int main() {`: Begins the main function where the program execution starts.
- `int fd = open("test.txt", O_RDONLY);`: Opens `test.txt` in read-only mode and stores the file descriptor in `fd`.
- `if (fd == -1) {`: Checks if there was an error opening the file.
- `perror("Error opening file");`: Prints an error message if the file couldn't be opened.
- `return 1;`: Exits the program with a return code of 1, indicating an error.
- `off_t size = lseek(fd, 0, SEEK_END);`: Uses `lseek` to seek to the end of the file, which returns the size of the file in bytes.
- `if (size == -1) {`: Checks if there was an error getting the file size.
- `perror("Error getting file size");`: Prints an error message if the file size couldn't be determined.
- `return 1;`: Exits the program with a return code of 1 due to the error.
- `printf("Size of the file is %ld bytes\n", size);`: Prints the size of the file in bytes.
- `close(fd);`: Closes the file descriptor to release resources.
- `return 0;`: Exits the program successfully with a return code of 0.

This program demonstrates how to open a file, determine its size by seeking to the end, and then close the file, handling any errors that may occur during the process.

Here's the explanation for `create_hole_std.c`:

- `#include <stdio.h>`: Includes the Standard Input and Output Library for I/O operations.
- `int main() {`: Begins the main function where the program execution starts.
- `FILE *file = fopen("holey_file.txt", "w+");`: Opens (or creates if it doesn't exist) a file named `holey_file.txt` for reading and writing, positioning the stream at the beginning of the file.
- `if (file == NULL) {`: Checks if the file failed to open.
- `perror("Error opening file");`: Prints an error message if the file couldn't be opened.
- `return 1;`: Exits the program with a return code of 1, indicating an error.
- `fputs("Start of file", file);`: Writes the string "Start of file" to the file.
- `fseek(file, 1024, SEEK_CUR);`: Moves the file pointer 1024 bytes forward from the current position, creating a "hole" in the file.
- `fputs("End of file", file);`: Writes the string "End of file" at the end of the "hole".
- `fclose(file);`: Closes the file, which also flushes the stream for any buffered output.
- `return 0;`: Returns 0 from `main`, indicating successful program completion.

This program demonstrates how to create a file with a "hole" in it by seeking beyond the current end of the file before writing additional data.

Here's the breakdown of `writer.c`:

- `#include <stdio.h>`: Includes the Standard Input and Output Library.
- `#include <unistd.h>`: Includes POSIX operating system API.
- `#include <fcntl.h>`: Includes the File Control library for file operations.
- `int main() {`: Starts the main function where program execution begins.
- `int fd = open("list1.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);`: Opens `list1.txt` for writing, creates it if it doesn't exist, or truncates it if it does, with permissions set to 0644.
- `if (fd == -1) {`: Checks if there was an error opening the file.
- `perror("Error opening file");`: Prints an error message if the file couldn't be opened.
- `return 1;`: Exits the program with a return code of 1, indicating an error.
- `char *data = "1011 GM\tBuick\t2014\n1024 Ford\tLincoln\t2025\n";`: Initializes a character pointer `data` with a string to write to the file.
- `if (write(fd, data, 45) != 45) {`: Writes the data to the file and checks if the write operation was not exactly 45 characters, indicating an error.
- `perror("Error writing to file");`: Prints an error message if the write operation failed.
- `close(fd);`: Closes the file descriptor to release resources.
- `return 1;`: Exits the program with a return code of 1, indicating an error during the write operation.
- `close(fd);`: Closes the file descriptor.
- `return 0;`: Returns 0 from `main`, indicating successful program execution.

This program demonstrates opening a file, writing specified data to it, and handling errors during file operations.

Assignment 6:

```
#include <stdio.h>          // Include standard input/output library for using printf and other functions.
#include <stdlib.h>         // Include standard library for using functions like exit.
#include <string.h>         // Include string library for functions like strlen and bzero.
#include <unistd.h>         // Provide access to the POSIX operating system API, including read and write functions.
#include <sys/types.h>      // Include types used in system calls.
#include <sys/socket.h>     // Include definitions for socket operations.
#include <netinet/in.h>     // Include constants and structures for internet domain addresses.
#include <arpa/inet.h>      // Definitions for internet operations.

void error(const char *msg) {   // Function to handle error messages; it prints the error and exits the program.
```

```
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[]) {  // Main function starting point of the server program.
    int sockfd, newsockfd, portno;  // Declare socket file descriptor, new socket for client, and port number.
    socklen_t clilen;               // Declare a variable to store the size of the client address.
    char buffer[256];               // Buffer to store messages up to 255 characters.
    struct sockaddr_in serv_addr, cli_addr;  // Internet address structures for the server and client.
    int n;                          // Variable to store the number of characters read or written.

    if (argc < 2) {                 // Check if the port number is provided as an argument.
        fprintf(stderr, "ERROR, no port provided\n");
        exit(1);
    }

    sockfd = socket(AF_INET, SOCK_STREAM, 0);  // Create a new socket of type TCP (SOCK_STREAM).
    if (sockfd < 0)
        error("ERROR opening socket");

    bzero((char *) &serv_addr, sizeof(serv_addr));  // Clear the server address structure.
    portno = atoi(argv[1]);                     // Convert port number from string to integer.
    serv_addr.sin_family = AF_INET;             // Set the family of the address to AF_INET (IPv4).
    serv_addr.sin_addr.s_addr = INADDR_ANY;     // Set the server address to any incoming interfaces.
    serv_addr.sin_port = htons(portno);         // Convert the port number to network byte order.

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)  // Bind the socket to the address.
        error("ERROR on binding");

    listen(sockfd, 5);  // Listen on the socket for connections, with a backlog limit of 5.
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);  // Accept a connection from a client.
    if (newsockfd < 0)
        error("ERROR on accept");
```

```
    while (1) {  // Infinite loop to keep processing incoming messages.
        bzero(buffer, 256);  // Clear the buffer.
        n = read(newsockfd, buffer, 255);  // Read data from the client into the buffer.
        if (n < 0) error("ERROR reading from socket");
        printf("Here is the message: %s\n", buffer);  // Print the received message.

        n = write(newsockfd, buffer, strlen(buffer));  // Echo the message back to the client.
        if (n < 0) error("ERROR writing to socket");
    }

    close(newsockfd);  // Close the client socket.
    close(sockfd);     // Close the server socket.
    return 0;          // End the program.
}
```

Part 2:

```
#include <stdio.h>            // Include standard input/output library for using printf and other functions.
#include <stdlib.h>           // Include standard library for functions like exit.
#include <unistd.h>           // Provide access to the POSIX operating system API, including read and write functions.
#include <string.h>           // Include string library for functions like strlen and bzero.
#include <sys/types.h>        // Include types used in system calls.
#include <sys/socket.h>       // Include definitions for socket operations.
#include <netinet/in.h>       // Include constants and structures for internet domain addresses.
#include <netdb.h>            // Definitions for network database operations.

void error(const char *msg) {   // Function to handle error messages; it prints the error and exits the program.
    perror(msg);
    exit(0);
}
```

```
int main(int argc, char *argv[]) {  // Main function starting point of the client program.
   int sockfd, portno, n;          // Declare socket file descriptor, port number, and number of characters read or written.
   struct sockaddr_in serv_addr;   // Internet address structure for the server.
   struct hostent *server;         // Pointer to a structure that defines a host computer on the internet.

   char buffer[256];               // Buffer to store messages up to 255 characters.
   if (argc < 3) {                 // Check if hostname and port number are provided as arguments.
     fprintf(stderr,"usage %s hostname port\n", argv[0]);
     exit(0);
   }
   portno = atoi(argv[2]);         // Convert port number from string to integer.
   sockfd = socket(AF_INET, SOCK_STREAM, 0);  // Create a new socket of type TCP (SOCK_STREAM).
   if (sockfd < 0)
      error("ERROR opening socket");
   server = gethostbyname(argv[1]);  // Get the server's DNS details.
   if (server == NULL) {
      fprintf(stderr,"ERROR, no such host\n");
      exit(0);
   }

   bzero((char *) &serv_addr, sizeof(serv_addr));  // Clear the server address structure.
   serv_addr.sin_family = AF_INET;                 // Set the family of the address to AF_INET (IPv4).
   bcopy((char *)server->h_addr,
       (char *)&serv_addr.sin_addr.s_addr,
       server->h_length);  // Copy the server's address from DNS info to sockaddr structure.
   serv_addr.sin_port = htons(portno);  // Convert the port number to network byte order.

   if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)  // Connect to the server.
      error("ERROR connecting");

   printf("Please enter the message: ");  // Prompt the user to enter a message.
   bzero(buffer,256);                     // Clear the buffer.
   fgets(buffer,255,stdin);               // Get input from the user and store it in the buffer.
```

```
    n = write(sockfd,buffer,strlen(buffer));  // Send the message to the server.
    if (n < 0)
        error("ERROR writing to socket");

    bzero(buffer,256);  // Clear the buffer again.
    n = read(sockfd,buffer,255);  // Read the server's response into the buffer.
    if (n < 0)
        error("ERROR reading from socket");
    printf("%s\n",buffer);  // Print the received message from the server.

    close(sockfd);  // Close the socket.
    return 0;       // End the program.
}
```

Assignment 5

```
#include <stdio.h>   // Include the standard input/output library for functions like printf.
#include <stdlib.h>  // Include the standard library for miscellaneous functions like exit.
#include <string.h>  // Include the string library for memory and string functions like memset.
#include <unistd.h>  // Include POSIX operating system API for various constants.
#include <sys/types.h>// Include data types used in system calls.
#include <sys/socket.h>// Include main sockets definitions.
#include <netinet/in.h>// Include constants and structures needed for internet domain addresses.
#include <arpa/inet.h> // Include definitions for internet operations.

void error(const char *msg) {  // Define a function to handle errors.
    perror(msg);              // Print the error message to stderr.
    exit(1);                  // Terminate the program with an error status.
}

int main(int argc, char *argv[]) {  // Start of main function.
    int sockfd, newsockfd, portno;  // Socket descriptors and port number variables.
    socklen_t clilen;               // Variable to store the size of the client address.
    char buffer[256];               // Buffer to store the data received from the client.
    struct sockaddr_in serv_addr, cli_addr;  // Structures for handling internet addresses.
    int n;                          // Variable to store the number of bytes read or written.

    if (argc < 2) {                 // Check if the port number was provided.
        fprintf(stderr, "ERROR, no port provided\n"); // If not, print an error message.
        exit(1);                    // Terminate the program.
    }
```

```
   sockfd = socket(AF_INET, SOCK_STREAM, 0);  // Create a socket using IPv4 and TCP.
   if (sockfd < 0)
      error("ERROR opening socket"); // If socket creation fails, handle the error.

   bzero((char *) &serv_addr, sizeof(serv_addr)); // Clear the structure by setting all values to zero.
   portno = atoi(argv[1]);                  // Convert the port number from string to integer.
   serv_addr.sin_family = AF_INET;            // Set the family to IPv4.
   serv_addr.sin_addr.s_addr = INADDR_ANY;     // Allow connections to any IP address of the machine.
   serv_addr.sin_port = htons(portno);        // Convert port number to network byte order.

   if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)  // Bind the address to the socket.
      error("ERROR on binding"); // If binding fails, handle the error.

   listen(sockfd, 5);  // Listen on the socket for connections, with a maximum of 5 pending connections.
   clilen = sizeof(cli_addr);  // Get the size of the client address structure.
   newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);  // Accept a connection.
   if (newsockfd < 0)
      error("ERROR on accept");  // If accepting fails, handle the error.

   while (1) {  // Infinite loop to continuously handle messages from the client.
      bzero(buffer, 256);  // Clear the buffer.
      n = read(newsockfd, buffer, 255);  // Read data from the socket into the buffer.
      if (n < 0) error("ERROR reading from socket");  // If reading fails, handle the error.
      printf("Here is the message: %s\n", buffer);  // Print the received message.

      n = write(newsockfd, buffer, strlen(buffer));  // Send the same data back to the client.
      if (n < 0) error("ERROR writing to socket");  // If writing fails, handle the error.
   }

   close(newsockfd);  // Close the client socket.
   close(sockfd);     // Close the server socket.
   return 0;         // Exit the program.
}
```

Part 2:

```
#include <stdio.h>   // Include the standard input/output library for functions like printf.
#include <stdlib.h>  // Include the standard library for miscellaneous functions like exit.
#include <unistd.h>  // Include POSIX operating system API for various constants.
#include <string.h>  // Include the string library for memory and string functions.
#include <sys/types.h>// Include data types used in system calls.
#include <sys/socket.h>// Include main sockets definitions.
#include <netinet/in.h>// Include constants and structures needed for internet domain addresses.
#include <netdb.h>     // Definitions for network database operations, like DNS lookups.

void error(const char *msg) {  // Define a function to handle errors.
```

```
    perror(msg);            // Print the error message to stderr.
    exit(0);                // Terminate the program with an error status.
}

int main(int argc, char *argv[]) {  // Start of main function.
    int sockfd, portno, n;          // Socket descriptor, port number, and number of bytes read or written.
    struct sockaddr_in serv_addr;   // Structure to hold server's address.
    struct hostent *server;         // Structure to hold server's data.

    char buffer[256];               // Buffer to store the message to be sent.
    if (argc < 3) {                 // Check if hostname and port number are provided.
       fprintf(stderr,"usage %s hostname port\n", argv[0]);  // Print usage message if not.
       exit(0);                     // Exit the program.
    }
    portno = atoi(argv[2]);         // Convert port number from string to integer.
    sockfd = socket(AF_INET, SOCK_STREAM, 0);  // Create a socket using IPv4 and TCP.
    if (sockfd < 0)
       error("ERROR opening socket");  // If socket creation fails, handle the error.

    server = gethostbyname(argv[1]);  // Get server details from its domain name.
    if (server == NULL) {
       fprintf(stderr,"ERROR, no such host\n");  // If the server is not found, handle the error.
       exit(0);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr)); // Clear the structure by setting all values to zero.
    serv_addr.sin_family = AF_INET;             // Set the family to IPv4.
    bcopy((char *)server->h_addr,
        (char *)&serv_addr.sin_addr.s_addr,
        server->h_length);                  // Copy the server's IP address from the DNS result to serv_addr.
    serv_addr.sin_port = htons(portno);         // Convert port number to network byte order.

    if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)  // Connect to the server.
       error("ERROR connecting");  // If connection fails, handle the error.

    printf("Please enter the message: ");  // Ask user to input a message.
    bzero(buffer,256);                  // Clear the buffer.
    fgets(buffer,255,stdin);            // Read input from the user into the buffer.
    n = write(sockfd,buffer,strlen(buffer));  // Send the message to the server.
    if (n < 0)
        error("ERROR writing to socket");   // If writing fails, handle the error.

    bzero(buffer,256);  // Clear the buffer again.
    n = read(sockfd,buffer,255);  // Read the server's response into the buffer.
    if (n < 0)
        error("ERROR reading from socket");  // If reading fails, handle the error.
    printf("%s\n",buffer);  // Print the received message from the server.
```

```
    close(sockfd);  // Close the socket.
    return 0;       // Exit the program.
}
```

Part 3

```
#include <stdio.h>   // Include standard input/output library for using printf, etc.
#include <stdlib.h>  // Include standard library for miscellaneous functions like exit.
#include <string.h>  // Include string library for memory manipulation functions like memset.
#include <unistd.h>  // Include POSIX API for various constants.
#include <sys/types.h>// Include data types used in system calls.
#include <sys/socket.h>// Include main sockets definitions.
#include <netinet/in.h>// Include constants and structures needed for internet domain addresses.
#include <arpa/inet.h> // Include definitions for internet operations.

void error(const char *msg) {  // Define a function to display error messages and exit.
    perror(msg);              // Print the error message and terminate the program.
    exit(1);                  // Exit with status 1 indicating an error.
}

int main(int argc, char *argv[]) {  // Main function, where execution starts.
    int sockfd, newsockfd, portno;  // Variables to store socket file descriptors and port number.
    socklen_t clilen;               // Variable to store the size of the client address.
    char buffer[256];               // Buffer to store the data received from the client.
    struct sockaddr_in serv_addr, cli_addr;  // Structures to store internet addresses.
    int n;                          // Variable to store the number of bytes read or written.

    if (argc < 2) {                 // Check if the port number is provided as an argument.
        fprintf(stderr, "ERROR, no port provided\n"); // Print error message if not provided.
        exit(1);                    // Exit the program if no port number is provided.
    }

    sockfd = socket(AF_INET, SOCK_STREAM, 0);  // Create a TCP socket.
    if (sockfd < 0)
        error("ERROR opening socket"); // Handle error in socket creation.

    bzero((char *) &serv_addr, sizeof(serv_addr)); // Zero out the structure serv_addr.
    portno = atoi(argv[1]);                    // Convert port number from string to integer.
    serv_addr.sin_family = AF_INET;            // Set the family of the address to IPv4.
    serv_addr.sin_addr.s_addr = INADDR_ANY;    // Allow connections to any IP address of the host.
    serv_addr.sin_port = htons(portno);        // Convert port number to network byte order.

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)  // Bind the address to the socket.
        error("ERROR on binding"); // Handle error in binding.

    listen(sockfd, 5);  // Listen on the socket for connections, with a queue limit of 5.
    clilen = sizeof(cli_addr);  // Get the size of the client address structure.
```

```
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);  // Accept a connection from a client.
    if (newsockfd < 0)
        error("ERROR on accept");  // Handle error in accepting a connection.

    while (1) {  // Infinite loop to handle messages from the client continuously.
        bzero(buffer, 256);  // Clear the buffer.
        n = read(newsockfd, buffer, 255);  // Read data from the client into the buffer.
        if (n < 0) error("ERROR reading from socket");  // Handle error in reading from socket.
        printf("Here is the message: %s\n", buffer);  // Print the received message.

        n = write(newsockfd, buffer, strlen(buffer));  // Send the same data back to the client (echo).
        if (n < 0) error("ERROR writing to socket");  // Handle error in writing to socket.
    }

    close(newsockfd);  // Close the client socket after finishing communication.
    close(sockfd);     // Close the server socket.
    return 0;          // Return 0 indicating successful termination.
}
```

Assignment 4:

```
#include <stdio.h>   // Include standard input/output library for using functions like printf and scanf.
#include <stdlib.h>  // Include standard library for general purpose functions, including memory allocation and process control.

int main() {  // Main function where the program execution starts.
    int number;  // Declare an integer variable to store user input.

    printf("Enter an integer: ");  // Prompt the user to enter an integer.
    scanf("%d", &number);  // Read the integer input from the user and store it in the variable 'number'.

    if (number % 2 == 0) {  // Check if the number is even by using modulus operator. If remainder is 0, number is even.
        printf("%d is even.\n", number);  // If the condition is true, print that the number is even.
    } else {
        printf("%d is odd.\n", number);  // If the condition is not true, print that the number is odd.
    }

    return 0;  // Return 0 to the operating system, indicating successful termination of the program.
}
```

Part 2

```
#include <stdio.h>   // Include standard input/output library for functions like printf.
#include <math.h>    // Include math library for mathematical functions.

int main() {  // Main function, the starting point of the program.
```

```
    double area, side;  // Declare double precision floating-point variables for the area of a square and its side length.

    printf("Enter the side of the square: ");  // Ask the user to input the side length of the square.
    scanf("%lf", &side);  // Read the side length from user input.

    area = side * side;  // Calculate the area of the square as side squared.

    printf("The area of the square is: %.2f\n", area);  // Print the calculated area, formatted to two decimal places.

    return 0;  // Return 0 to indicate that the program finished successfully.
}
```

Part 3

```
#include <stdio.h>   // Standard library for input/output.
#include <string.h>  // Library for string manipulation functions.

int main() {  // Main function, where the program execution begins.
    char str[100];  // Declare a character array (string) that can hold up to 99 characters plus a null terminator.

    printf("Enter a string: ");  // Prompt the user to enter a string.
    fgets(str, sizeof(str), stdin);  // Read a line of text from standard input (keyboard) into the string 'str'.

    printf("String output: ");  // Print "String output: " to indicate the output follows.
    puts(str);  // Display the string that was entered by the user.

    return 0;  // End of program, return 0 indicating successful completion.
}
```

Assignment 3

```
#include <stdio.h>   // Include standard input/output library for using functions like printf.
#include <stdlib.h>  // Include standard library for miscellaneous functions like abort.

int main() {  // Main function where the program execution starts.
    printf("This program will now abort.\n");  // Print a message to inform the user that the program will abort.
    abort();  // Immediately terminate the program abnormally.

    return 0;  // This line will never be executed because abort() terminates the program.
}
```

Part 2

```
#include <stdio.h>       // Include standard input/output library.
```

```
#include <omp.h>        // Include OpenMP library for parallel programming.

int main() {            // Main function, starting point of the program.
   #pragma omp parallel // Directive to OpenMP to execute the following block in parallel.
   {
      printf("Hello, world.\n");  // Each parallel thread executes this line.
   }
   return 0;           // Return 0 to indicate that the program finished successfully.
}
```

Part 3

```
#include <stdio.h>   // Standard library for input/output.

int main() {        // Main function where the program execution begins.
   int a = 10, b = 5;  // Declare two integer variables and initialize them.
   int sum = a + b;    // Declare an integer to store the sum and calculate it.
   int diff = a - b;   // Declare an integer to store the difference and calculate it.
   int prod = a * b;   // Declare an integer to store the product and calculate it.
   int quot = a / b;   // Declare an integer to store the quotient and calculate it.

   printf("Sum: %d\n", sum);      // Print the sum of a and b.
   printf("Difference: %d\n", diff);  // Print the difference of a and b.
   printf("Product: %d\n", prod);     // Print the product of a and b.
   printf("Quotient: %d\n", quot);    // Print the quotient of a and b.

   return 0;  // Return 0 indicating successful completion of the program.
}
```

Lab 9

```
#include <stdio.h>   // Include standard input/output library for functions like printf and scanf.
#include <stdlib.h>  // Include standard library for general utilities like rand and srand.
#include <time.h>    // Include the time library for time functions.

int main() {  // Main function where program execution starts.
   int number, guess, attempts = 0;  // Declare variables for the random number, user's guess, and attempt
count.
   srand(time(NULL));  // Seed the random number generator with current time.
   number = rand() % 100 + 1;  // Generate a random number between 1 and 100.

   printf("Guess the number (1 to 100): ");  // Prompt the user to guess the number.
   do {
      scanf("%d", &guess);  // Read user's guess from standard input.
      attempts++;  // Increment the attempt count each time the user guesses.

      if (guess > number) {  // Check if the guess is higher than the random number.
```

```
            printf("Lower! Try again: ");  // If so, prompt to guess a lower number.
        } else if (guess < number) {  // Check if the guess is lower than the random number.
            printf("Higher! Try again: ");  // If so, prompt to guess a higher number.
        }
    } while (guess != number);  // Repeat until the user guesses the correct number.

    printf("Congratulations! You guessed the number in %d attempts.\n", attempts);  // Congratulate the user and show the number of attempts.

    return 0;  // Return 0 indicating successful termination.
}
```

Part 2

```
#include <stdio.h>   // Include standard library for input/output functions.
#include <stdlib.h>  // Include standard library for miscellaneous functions such as rand.

int main() {  // Main function where the program execution starts.
    int secretNumber = 5;  // Declare and initialize the secret number to be guessed.
    int guess;             // Declare a variable to store the user's guess.

    printf("Enter your guess (1 to 10): ");  // Ask the user to enter their guess.
    scanf("%d", &guess);  // Read the user's guess from the input.

    if (guess == secretNumber) {  // Check if the guess is equal to the secret number.
        printf("You won!\n");  // If true, print a winning message.
    } else {
        printf("You lost! The correct number was %d\n", secretNumber);  // Otherwise, indicate the user lost and reveal the secret number.
    }

    return 0;  // Return 0 indicating the program ended successfully.
}
```

Lab 8

```
#include <stdio.h>   // Include standard input/output library for functions like printf.
#include <string.h>  // Include string library for functions related to string manipulation.

int main() {  // Main function where program execution starts.
    char str[100];  // Declare a character array (string) that can hold up to 99 characters plus a null terminator.
    printf("Enter a string: ");  // Prompt the user to enter a string.
    fgets(str, sizeof(str), stdin);  // Read a string from user input, including spaces, up to 99 characters.

    printf("Length of the string is: %lu\n", strlen(str) - 1);  // Print the length of the string minus the newline character.
    return 0;  // Return 0 indicating successful termination.
```

Part 2

```
#include <stdio.h>   // Include standard input/output library for functions like printf and scanf.
#include <stdbool.h> // Include the standard boolean library for using boolean types.

int main() {  // Main function where the program execution starts.
   bool check = true;  // Declare and initialize a boolean variable to true.
   int age;            // Declare an integer variable to store age.

   printf("Enter your age: ");  // Prompt the user to enter their age.
   scanf("%d", &age);  // Read the age value entered by the user.

   if (age >= 18) {  // Check if the entered age is 18 or more.
      printf("You are eligible to vote.\n");  // If true, print that the user is eligible to vote.
   } else {
      printf("You are not eligible to vote.\n");  // Otherwise, print that the user is not eligible to vote.
   }

   if (check) {  // Check if the boolean variable 'check' is true.
      printf("Boolean condition is true.\n");  // If true, print a confirmation message.
   }

   return 0;  // Return 0 indicating that the program ended successfully.
}
```

Lab 7

```
#include <stdio.h>   // Include standard input/output library for functions like printf and scanf.
#include <math.h>    // Include math library for mathematical functions.

int main() {  // Main function where the program execution starts.
   double radius, area;  // Declare two double precision floating-point variables for radius and area.

   printf("Enter the radius of the circle: ");  // Prompt the user to input the radius of a circle.
   scanf("%lf", &radius);  // Read the double input for the radius from the user.

   area = M_PI * pow(radius, 2);  // Calculate the area using the formula πr^2. M_PI is from math.h and pow is
power function.

   printf("The area of the circle is: %.2f\n", area);  // Print the calculated area, formatted to two decimal places.

   return 0;  // Return 0 to indicate that the program finished successfully.
}
```

Part 2

```
#include <stdio.h>   // Standard library for input/output functions.
```

```
int main() {  // Main function where the program execution starts.
   int num1, num2, sum;  // Declare three integer variables for two numbers and their sum.

   printf("Enter two integers: ");  // Ask the user to input two integers.
   scanf("%d %d", &num1, &num2);  // Read two integers from user input.

   sum = num1 + num2;  // Calculate the sum of the two integers.

   printf("Sum of %d and %d is %d\n", num1, num2, sum);  // Print the sum of the two integers.

   return 0;  // Return 0 indicating successful completion of the program.
}
```

Part 3

```
#include <stdio.h>      // Include standard library for input/output functions.
#include <string.h>     // Include string library for string manipulation functions.
#include <signal.h>     // Include signal handling library.

int main() {           // Main function where the program execution begins.
   int sig;           // Declare an integer variable to store signal numbers.

   for (sig = 1; sig < NSIG; sig++) {  // Loop through all signal numbers up to NSIG (number of signals defined in
signal.h).
      printf("Signal %d: %s\n", sig, strsignal(sig));  // Print each signal number and its string description.
   }

   return 0;  // Return 0 indicating successful completion.
}
```

Lab 6

```
#include <stdio.h>   // Include standard input/output library for functions like printf and scanf.
#include <math.h>    // Include math library for accessing mathematical functions.

int main() {  // Main function where the program execution starts.
   double base, exponent, result;  // Declare double precision variables for the base, exponent, and result.

   printf("Enter a base: ");  // Ask the user to enter the base for exponentiation.
   scanf("%lf", &base);  // Read the base from user input.

   printf("Enter an exponent: ");  // Ask the user to enter the exponent for exponentiation.
   scanf("%lf", &exponent);  // Read the exponent from user input.

   result = pow(base, exponent);  // Calculate the power of the base to the exponent using the pow function.

   printf("%.2lf^%.2lf = %.2lf\n", base, exponent, result);  // Print the result of the exponentiation.
```

```
   return 0;  // Return 0 indicating successful termination of the program.
}
```

Part 2

```
#include <stdio.h>   // Include standard input/output library for functions like printf.

int main() {  // Main function where the program execution starts.
   int i, n, t1 = 0, t2 = 1, nextTerm;  // Declare integers for loop counter, number of terms, and Fibonacci
sequence elements.

   printf("Enter the number of terms: ");  // Prompt user to enter the number of terms in the Fibonacci
sequence.
   scanf("%d", &n);  // Read the number of terms from user input.

   printf("Fibonacci Series: ");  // Print label for the Fibonacci series output.

   for (i = 1; i <= n; ++i) {  // Loop from 1 to n.
      printf("%d, ", t1);  // Print the current term in the Fibonacci series.
      nextTerm = t1 + t2;  // Calculate the next term in the Fibonacci series.
      t1 = t2;  // Update the second last term.
      t2 = nextTerm;  // Update the last term.
   }

   printf("\n");  // Print a newline character after the series output.

   return 0;  // Return 0 indicating successful completion of the program.
}
```

Lab 5

```
#include <stdio.h>   // Include standard input/output library for functions like printf and scanf.

int main() {  // Main function where the program execution starts.
   int number;  // Declare an integer variable to store the user input.

   printf("Enter an integer: ");  // Prompt the user to enter an integer.
   scanf("%d", &number);  // Read the integer value entered by the user.

   if (number % 2 == 0) {  // Check if the entered integer is even.
      printf("The number %d is even.\n", number);  // If true, print that the number is even.
   } else {
      printf("The number %d is odd.\n", number);  // If false, print that the number is odd.
   }
```

```
    return 0;  // Return 0 indicating that the program ended successfully.
}
```

Lab 4

```
#include <stdio.h>    // Include standard input/output library for file handling and printf.
#include <stdlib.h>   // Include standard library for exit function.

int main() {
    FILE *fptr1, *fptr2;  // Declare pointers to FILE for both source and destination files.
    char filename[100], c;  // Declare a filename array and a character variable for reading.

    printf("Enter the filename to open for reading: \n");  // Prompt user to enter the filename.
    scanf("%s", filename);  // Read the filename.

    fptr1 = fopen(filename, "r");  // Open the source file in read mode.
    if (fptr1 == NULL) {  // Check if the file is opened successfully.
        printf("Cannot open file %s \n", filename);
        exit(0);  // Exit if the file is not opened.
    }

    printf("Enter the filename to open for writing: \n");  // Prompt user for the destination file name.
    scanf("%s", filename);  // Read the filename.
    fptr2 = fopen(filename, "w");  // Open the destination file in write mode.
    if (fptr2 == NULL) {  // Check if the file is opened successfully.
        printf("Cannot open file %s \n", filename);
        exit(0);  // Exit if the file is not opened.
    }

    c = fgetc(fptr1);  // Read a character from the source file.
    while (c != EOF) {  // Loop until the end of file is reached.
        fputc(c, fptr2);  // Write the character to the destination file.
        c = fgetc(fptr1);  // Read the next character.
    }

    printf("Contents copied to %s", filename);  // Inform the user that the contents have been copied.

    fclose(fptr1);  // Close the source file.
    fclose(fptr2);  // Close the destination file.
    return 0;  // Return 0 to indicate successful completion.
}
```

Part 2

```
#include <stdio.h>  // Include standard input/output library.
```

```
int main() {
   int numbers[5] = {1, 2, 3, 4, 5};  // Declare an array of integers.
   int sum = 0;  // Initialize sum variable to store the sum of the array elements.
   for (int i = 0; i < 5; i++) {  // Loop over the array.
      sum += numbers[i];  // Add each element to sum.
   }
   printf("The sum of the array elements is: %d\n", sum);  // Print the total sum of the array.

   return 0;  // Return 0 to indicate successful completion.
}
```

Lab 3

```
#include <stdio.h>   // Include standard input/output library for functions like printf and scanf.

int main() {  // Main function where the program execution starts.
   int n, factorial = 1;  // Declare integer variables for input number and factorial result, initializing factorial to 1.

   printf("Enter a positive integer: ");  // Prompt the user to enter a positive integer.
   scanf("%d", &n);  // Read the integer from user input.

   for (int i = 1; i <= n; ++i) {  // Loop from 1 to n to calculate factorial.
      factorial *= i;  // Multiply factorial by i for each iteration to get factorial value.
   }

   printf("Factorial of %d = %d\n", n, factorial);  // Print the factorial of the entered number.

   return 0;  // Return 0 to indicate that the program ended successfully.
}
```