1. myfopen()

```
myFILE *myfopen(const char *pathname, const char *mode){
    int flag = 0;
    int mode_flag = 0;
    if (strcmp(mode, "r")==0){
        flag = O_RDONLY;
        mode_flag = O_RDONLY;
    } else if (strcmp(mode,"r+")==0) {
        flag = O_RDWR;
        mode_flag = O_RDWR;
    } else if (strcmp(mode,"w")==0) {
        flag = O_WRONLY| O_CREAT | O_TRUNC;
        mode_flag = O_WRONLY;
    } else if (strcmp(mode,"w+")==0) {
        flag = O_RDWR|O_CREAT|O_TRUNC;
        mode_flag = O_RDWR;
    } else if (strcmp(mode, "a")==0) {
        flag = O_WRONLY|O_CREAT|O_APPEND;
        mode_flag = O_WRONLY;
    } else if (strcmp(mode, "a+")==0) {
        flag = O_RDWR|O_CREAT|O_APPEND;
        mode_flag = O_RDWR;
    } else
        return NULL;
```

this part I tried to parse (open file for read_only, write_only, create file if needed, append, truncate, …)int fd = open(pathname, flag, 0644);

```
    if (fd == -1) {
        struct stat sb;
        if (errno == EACCES) {
            const char *error = "Permission Denied\n";
            write(STDERR_FD, error, strlen(error));
            return NULL;
        } else { //including false positive
            const char *error = "File Format Error\n";
            write(STDERR_FD, error, strlen(error));
            return NULL;
        }
    }
    myFILE *file = malloc(sizeof(myFILE));
    if (!file) {
        close(fd);
        return NULL;
    }
    file->fd = fd;
    file->mode_flag = mode_flag;
    file->offset = 0;
    file->last_operation = OP_NONE;
    file->buffer_length = 0;
    return file;
```

Next, open the file, if it doesn't exist then create with mode 0644 (rw-r—r--) + handle error.
Allocate and initialize file and its fields.
*Note: for error handling, in case I want to detect permission error, I attempt to use stat() to detect the file's mode and ownership, and compare it to the process's owner's user and group ID, which are obtained from getuid(2) and getgid(2). Even with all that, it's still error prone because of the existence of SUID and ACL. Furthermore, if the file is in the inaccessible folder, stat(2) would fail in the first place. In short, without checking errno, it will be more cumbersome while less correct. I feel including errno.h to be justified given the problem requirements.

2. myfclose()

```
int myfclose(myFILE *stream) {
    if (stream == NULL)
        return EOF;
    if (myfflush(stream) == EOF) {
        return EOF;
    }
    if (close(stream->fd)==-1) {
        free(stream);
        return EOF;
    }
    free(stream);
    return 0;
}
```

if stream pointer is null return EOF
flush buffer data, if myfflush() fails return EOF
if close() fails, return EOF
lastly, free fynamicall allocated memory for the file

3. myfseek()

```
int myfseek(myFILE *stream, int offset, int whence){
    if (stream == NULL)
        return -1;
    if (stream->last_operation == FILE_WRITE) {
        if(myfflush(stream) == EOF)
            return -1;
    }
    off_t seek = lseek(stream->fd, offset, whence);
    if (seek == -1)
        return -1;
    stream->last_operation= OP_NONE;
    return 0;
}
```

If last operation is write, then flush the buffer before seeking
then used lseek() to move to file descriptor position
after seek, indicate last operation is cleared

4. myfread()

```
int myfread(void *ptr, int size, int nmemb, myFILE *stream){
    if (stream->mode_flag==O_WRONLY)    return 0;
    if (stream==NULL||ptr==NULL||size<=0||nmemb<=0) return 0;
```

```
    int total = nmemb*size;
    if (stream->last_operation == FILE_WRITE) {
        if (myfflush(stream) == EOF)
            return 0;
    }
    ssize_t read_byte = read(stream->fd, ptr, total);
    if (read_byte < 0) { // error
        return 0;
    }
    stream->last_operation = FILE_READ;
    return read_byte/size;
}
```

calculate the total byte size (total). Flush the buffer if the last operation is a write. Raise error if the number of byte read is 0. Set the last operation to be a read.

5. myfwrite()

```
int myfwrite(const void *ptr, int size, int nmemb, myFILE *stream){
    if (stream->mode_flag == O_RDONLY)
        return 0;
    if (stream == NULL || ptr == NULL || size <= 0 || nmemb <= 0)
        return 0;
    int total = nmemb * size; //total write bytes
    const char *start = (const char *)ptr; //pointer to the start of buffer
    const char *stop = start + total; //pointer to the end of buffer
    while (stop > start) {
        size_t remaining = BUFSIZE – stream→buffer_length; //empty portion of the buffer
            //write until the end of buffer if sufficient input
        size_t copy_size = (remaining > stop - start) ? (stop - start) : remaining;
        memcpy(stream->wrbuffer + stream->buffer_length, start, copy_size);
        stream->buffer_length += copy_size;
        start += copy_size;
        stream->last_operation = FILE_WRITE;
        if (remaining == copy_size && myfflush(stream) == EOF)
            return 0;
    }
    return nmemb;
}
```

6. myfflush()

```
int myfflush(myFILE *stream){
    if (stream==NULL)
        return EOF;
    switch (stream->last_operation) {
    case OP_NONE:
        return 0;
    case FILE_READ: //discard buffered data fetch from file
        stream->buffer_length = 0;
        return 0;
    case FILE_WRITE:
        if (stream->buffer_length > 0) {
```

```
            ssize_t w = write(stream->fd, stream->wrbuffer, stream→buffer_length); //write buffered
data
            if (w != stream->buffer_length)
                return EOF;
            stream->buffer_length = 0;
        }
        return 0;
    }
    /*
     * unreachable, which is standardized in C23
     * but that is too new for Ubuntu 22.10
     */
    return EOF;
}
```

6. myfputs()

```
int myfputs(const char* str, myFILE* stream){
    int length = strlen(str);
    return (myfwrite(str, sizeof(char), length, stream) == length) ? 0 : EOF;
} //write until null. If number of bytes to be write is not equal to string length, return EOF
```

7. myfgets()

```
char* myfgets(char *str, int num, myFILE* stream){
    if (stream == NULL || str == NULL || num < 1)
        return NULL;
    if (stream->mode_flag == O_WRONLY)
        return NULL;
    char c;
    int count = 0;

    while (count < num-1) {
        ssize_t r = myfread(str, sizeof(char), 1, stream);
        if(r == 0) //EOF
            break;
        else if (r < 0)
            return NULL;
        str[count++] = c; //write char to buffer
        if (c = '\n')
            break;
    }

    if (count == 0 && num>1) //EOF cannot be reached with num=1
        return NULL;

    str[count] = '\0'; write ‘\0’ when reaching the last character
    stream->last_operation = FILE_READ;
    return str;
}
```