CSC / CPE 357

Systems Programming

Topics

- File I/O

These slides are based on materials from Professors Aaron Keen and Andrew Migler

Standard I/O: File Access

- Examples thus far have relied on `stdin/stdout` which are made available automatically

- To read or write to a named file using standard I/O functions, a file must first be opened by the library function `fopen()`
  - `FILE *fopen(char *name, char *mode)`

- `FILE *` (file pointer) represents a structure that contains information about the file, allowing other functions to read/write

- `mode` a character string indicating the intended use of the file:
  - `"r"` read
  - `"w"` write
  - `"a"` append
  - (additional options described here: `man 3 fopen`)

Standard I/O: File Access Modes

Read Modes	Write Modes	Append Modes
• `r`: open for reading • `rb`: open for reading in binary mode • `r+`: open for reading or writing • `rb+`: binary mode read or write `fopen()` returns `NULL` if file does not exist	• `w`: write • `wb`: write, binary mode • `w+`: read or write • `wb+`: read or write, binary mode If the file exists, contents are overwritten. File is created if it does not exist.	• `a`: open for appending • `ab`: append, binary mode • `a+`: append or read • `ab+`: append or read binary mode New data will be added to the end of the file. File is created if it does not exist.

Standard I/O: File Access

- `fopen()` will fail (returning `NULL`) if
  - In read mode, the file does not exist
  - The current user does not have permission to access the file

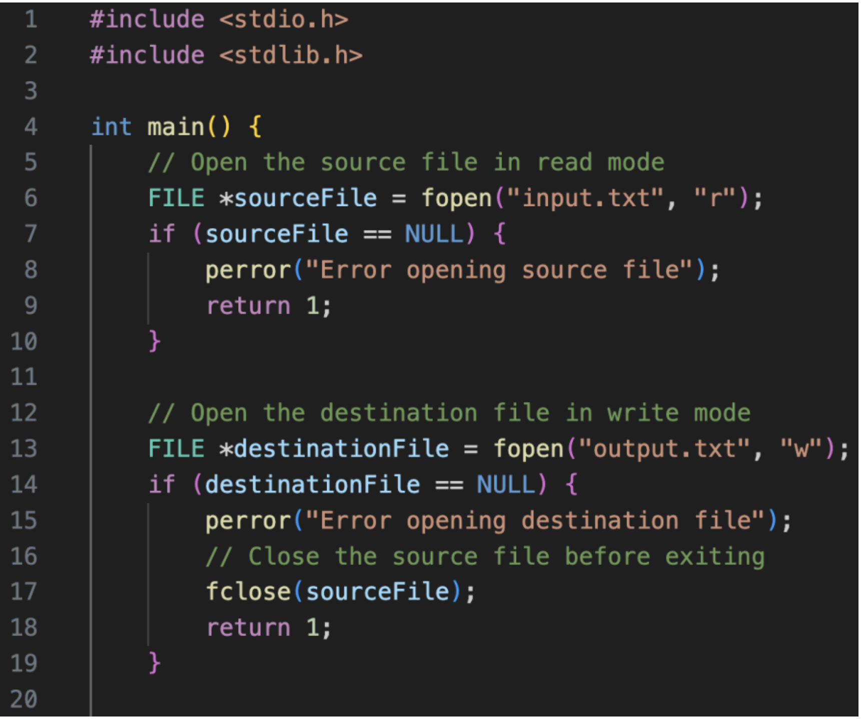
- If `fopen()` succeeds, characters can be read/written using:
  - `int getc(FILE *fp)`
  - `int putc(int c, FILE *fp)`

- Formatted read/write based on a file pointer:
  - `int fprintf(FILE *fp, char *format, …)`
  - `int fscanf(FILE *fp, char *format, …)`

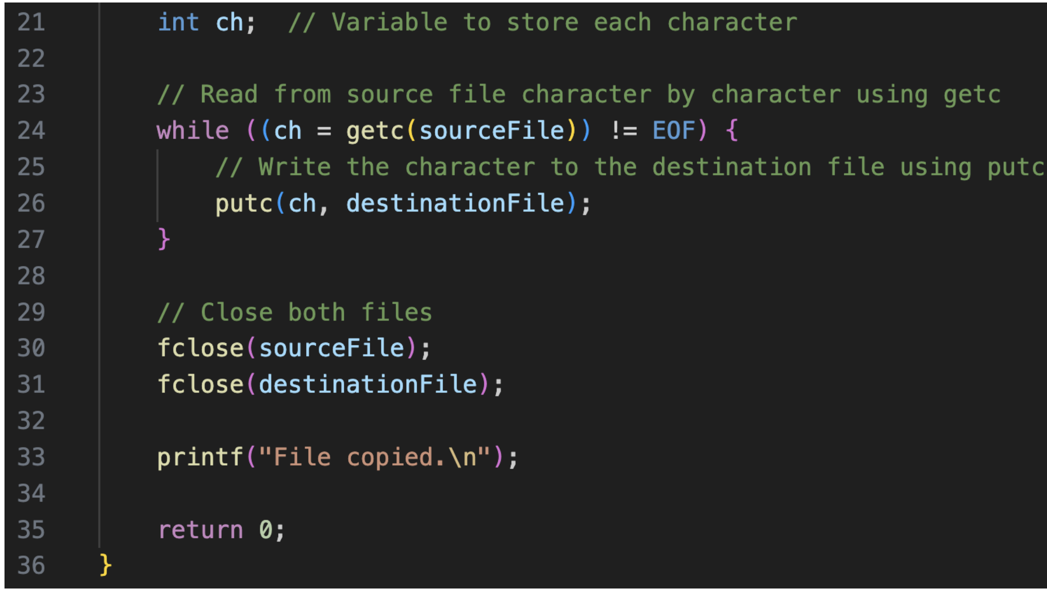
- When finished reading/writing:
  - `int fclose(FILE *fp)`

Example Code

In class example: Copying contents from one file to another



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

int main() {
    // Open the source file in read mode
    FILE *sourceFile = fopen("input.txt", "r");
    if (sourceFile == NULL) {
        perror("Error opening source file");
        return 1;
    }

    // Open the destination file in write mode
    FILE *destinationFile = fopen("output.txt", "w");
    if (destinationFile == NULL) {
        perror("Error opening destination file");
        // Close the source file before exiting
        fclose(sourceFile);
        return 1;
    }

```



```c
    int ch;  // Variable to store each character

    // Read from source file character by character using getc
    while ((ch = getc(sourceFile)) != EOF) {
        // Write the character to the destination file using putc
        putc(ch, destinationFile);
    }

    // Close both files
    fclose(sourceFile);
    fclose(destinationFile);

    printf("File copied.\n");

    return 0;
}
```

Use of *getline()*

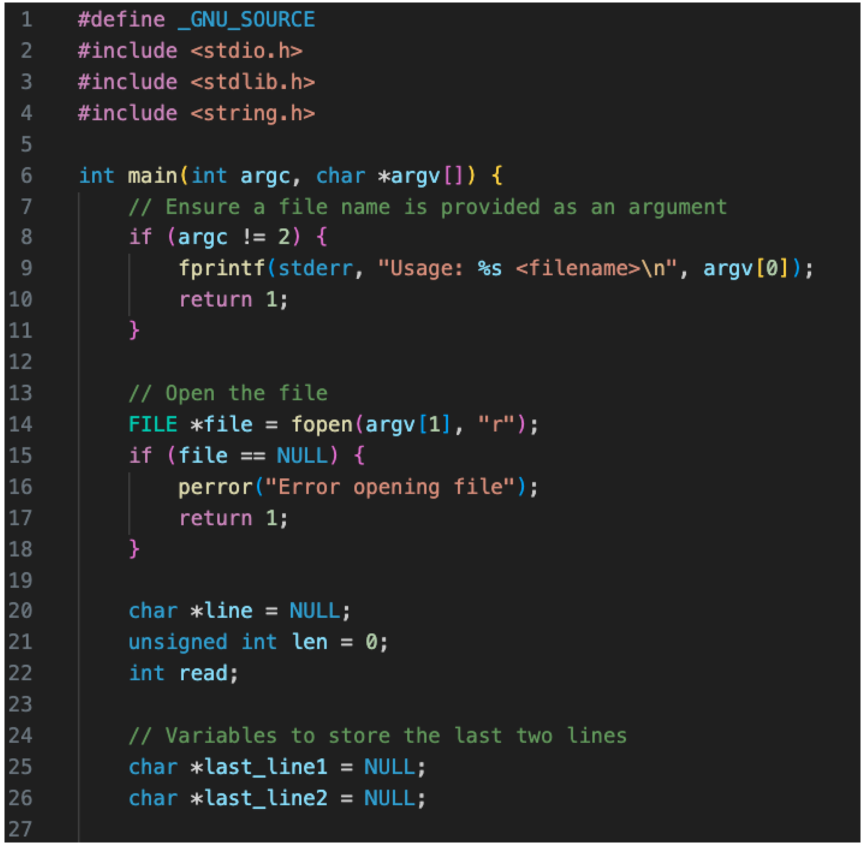
- The getline() function reads a line from stream, delimited by the character newline.

- The getline() functions return the number of characters written, excluding the terminating NUL character. The value -1 is returned if an error occurs, or if end-of-file is reached.

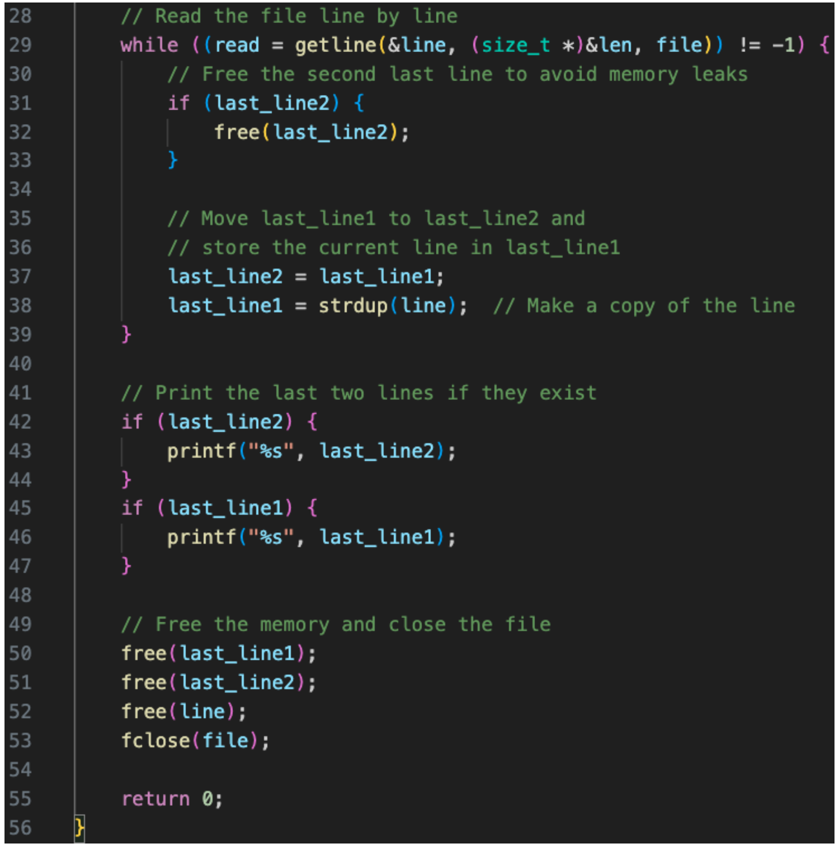
```
ssize_t getline(char ** restrict linep, size_t * restrict linecapp, FILE *
restrict stream);
```

Example Code

In class example: Lab 2 task 6



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

int main(int argc, char *argv[]) {
    // Ensure a file name is provided as an argument
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // Open the file
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    char *line = NULL;
    unsigned int len = 0;
    int read;

    // Variables to store the last two lines
    char *last_line1 = NULL;
    char *last_line2 = NULL;

```



```c
    // Read the file line by line
    while ((read = getline(&line, (size_t *)&len, file)) != -1) {
        // Free the second last line to avoid memory leaks
        if (last_line2) {
            free(last_line2);
        }

        // Move last_line1 to last_line2 and
        // store the current line in last_line1
        last_line2 = last_line1;
        last_line1 = strdup(line);  // Make a copy of the line
    }

    // Print the last two lines if they exist
    if (last_line2) {
        printf("%s", last_line2);
    }
    if (last_line1) {
        printf("%s", last_line1);
    }

    // Free the memory and close the file
    free(last_line1);
    free(last_line2);
    free(line);
    fclose(file);

    return 0;
}
```

Standard I/O: Line Input and Output

- To read an entire line from a file (including the newline):
  - `char *fgets(char *line, int maxline, FILE *fp)`
  - Reads the next input line into the character array `line`
  - Returns the line read or `NULL` on end of file or error
- Write a string to a file (with or without a newline):
  - `int fputs(char *line, FILE *fp)`

Unbuffered I/O

Open via system calls:

```
int open(const char *path, int oflag, …);
```

Arguments:

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_SEARCH	open directory for searching
O_EXEC	open for execute only

Documentation: `man 2 open`

(recall: manual page section 2 contains information about system calls)

Unbuffered I/O

Input and output via system calls:

```
    int n_read = read(int fd, char *buf, int n);

    int n_written = write(int fd, char *buf, int n);
```

Arguments:

- `fd`: file descriptor
- `buf`: buffer where data is read/writen
- `n`: number of bytes to be read/written

Documentation: `man 2 read`
(recall: manual page section 2 contains information about system calls)

File Descriptors

- A **file descriptor** is a non-negative integer.
  - Kernel returns a file descriptor on `open()` of an existing file or `creat()` a new file
- UNIX Conventions
  - File descriptor `0` represents `STDIN`
  - File descriptor `1` represents `STDOUT`
  - File descriptor `2` represents `STDERR`
- `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` constants are defined in `<unistd.h>`
- Contrast with *file pointers* `stdin`, `stdout`, `stderr`, declared in `<stdio.h>`
  - `int fileno(FILE *stream)`
  - `FILE *fdopen(int fd, const char *mode);`