

# File Access in C

Benjamin Brewster

Except as noted, all images copyrighted with Creative Commons licenses,  
with attributions given whenever available

# Unix Paradigm

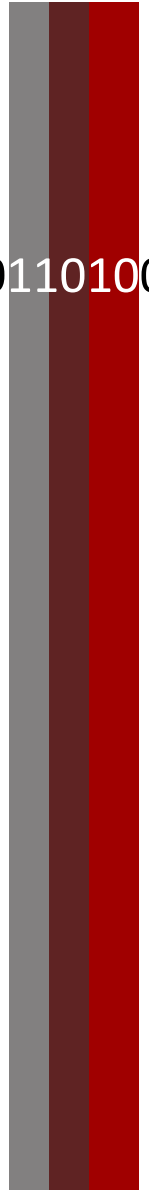
- Everything is a file!
  - Except processes
- Directory contents could include:
  - Hard links
  - Symbolic links
  - Named pipes
  - Device character special file
  - Device block special file
  - Named socket



# What is a File in UNIX?

10101010100101010111010010101

- System Programmer View:
  - A stream of bytes
    - Could be accessed as an array
    - Newlines/carriage returns & tabs are all just bytes, too!
  - Persistent
- How do we access files for reading and writing?



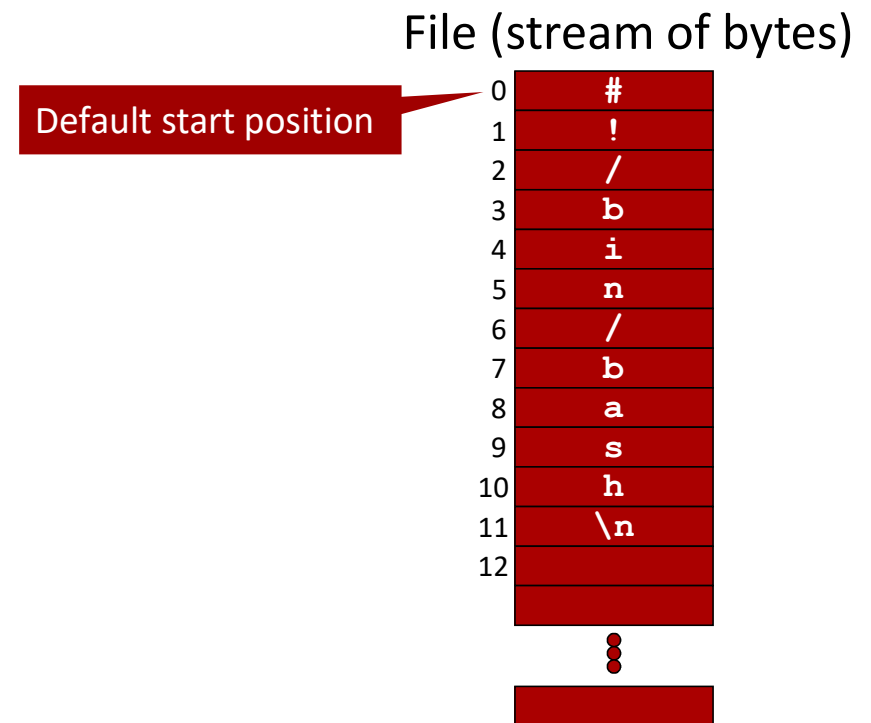
# Opening a File

- Files can be open for:
  - read only :: `O_RDONLY`
  - write only :: `O_WRONLY`
  - read and write :: `O_RDWR`
- When you open a file for writing...
  - Should you delete an existing file with the same name?
  - If not, where do you want to start writing
    - Beginning? End? Somewhere else?
  - If the file doesn't exist, should you create it?
  - If you create it, what should the initial access permissions be?

Reminder: We're talking about C programming now, not shell scripts

# The File Pointer

- Tracks where the next file operation occurs in an open file
- A separate file pointer is maintained for each open file
- All of the operations we're talking about:
  - Directly impact which byte in a file is pointed to by the file pointer when the file is opened
  - Move the file pointer



# Open for Read

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

int main(void)
{
    char file[] = "cs344/grades.txt";
    int file_descriptor;

    file_descriptor = open(file, O_RDONLY);

    if (file_descriptor < 0)
    {
        fprintf(stderr, "Could not open %s\n", file);
        exit(1);
    }

    close(file_descriptor);
    return 0;
}
```

Using `open()` and `close()` allows us to represent file descriptors as **ints**.

The more modern `fopen()` and `fclose()` require a special file descriptor type.

# Open for Write

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

int main(void)
{
    char file[] = "cs344/grades.txt";
    int file_descriptor;

    file_descriptor = open(file, O_WRONLY);

    if (file_descriptor < 0)
    {
        fprintf(stderr, "Could not open %s\n", file);
        perror("Error in main()");
        exit(1);
    }

    close(file_descriptor);
    return 0;
}
```



# Truncating an Existing File

- When you open a file for writing, should you delete all contents of an existing file with the same name, or write over existing contents?

- To delete it and start fresh: `O_TRUNC`

- Example:

```
file_descriptor = open(file, O_WRONLY | O_TRUNC);
```

- Opens an existing file for writing only, then deletes all the data in it
  - Sets the file pointer to position 0

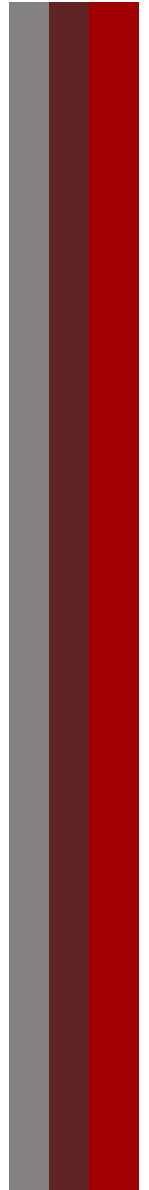


# Appending to an Existing File

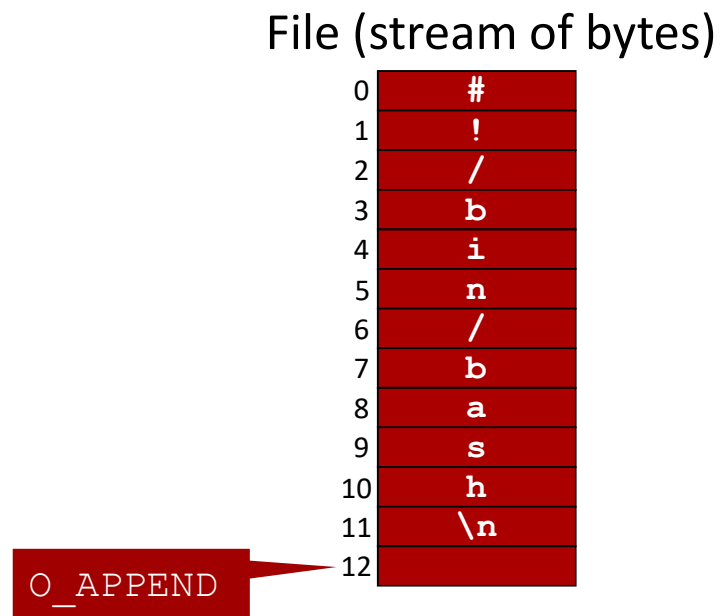
- Open the file in append mode with flag: `O_APPEND`
- Before *every* write, the file pointer will be automatically set to the end of the file
- Example

```
file_descriptor = open(filepath, O_WRONLY | O_APPEND);
```

  - Opens an existing file for writing only in append mode



# O\_APPEND and the File Pointer



# Creating a New File

- To open (or create) a file that doesn't exist, use flag: `O_CREAT`
- Example: open a file for writing only, creating it if it doesn't exist:  

```
file_descriptor = open(filepath, O_WRONLY | O_CREAT, 0600);
```
- The third parameter of `open()` *must* be used when the creation of a new file is requested (i.e. using `O_CREAT` or `O_TMPFILE`)

Even though the `open()` call will probably fail in bizarre ways if you don't include the third argument here, it still compiles! Thanks, C!

# Creating a New File - Access Permissions

- Again, the third parameter of `open()` *must* be used when the creation of a new file is requested (i.e. using `O_CREAT` or `O_TMPFILE`)
- Third parameter contains octal number permissions bits:
  - Specify directly as with `chmod`: `0600`
  - Or you can bit-wise OR flags together: `S_IRUSR | S_IWUSR`

- Example:

```
file_descriptor = open(file, O_WRONLY | O_CREAT, 0600);  
file_descriptor = open(file, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
```

User has read and write permission

User has read permission

User has write permission

# lseek ( )

- Manipulates a file pointer in a file
- Used to control where you're messing with da bitz

- Examples:

- Move to byte #16

```
newpos = lseek(file_descriptor, 16, SEEK_SET);
```

- Move forward 4 bytes

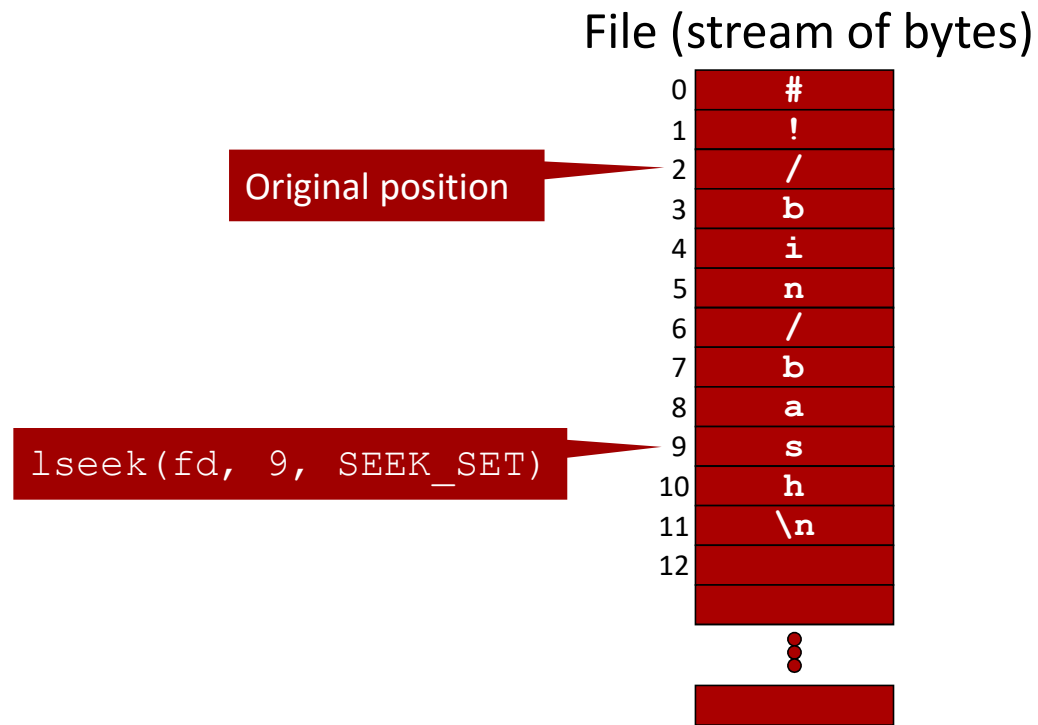
```
newpos = lseek(file_descriptor, 4, SEEK_CUR);
```

- Move to 8 bytes from the end

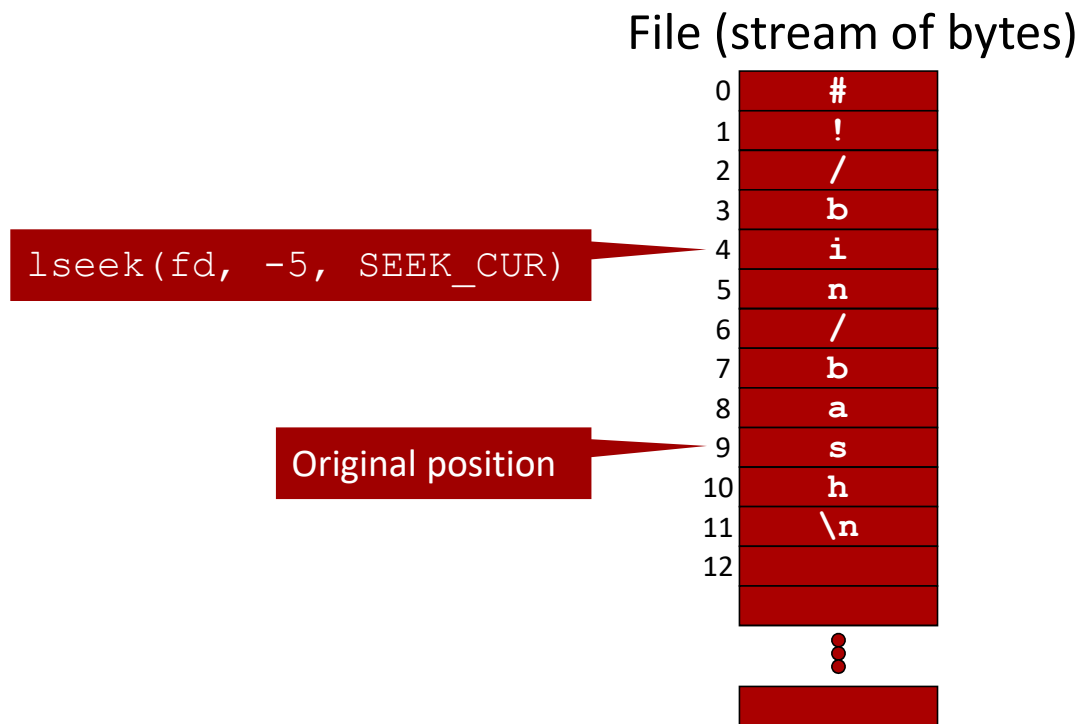
```
newpos = lseek(file_descriptor, -8, SEEK_END);
```



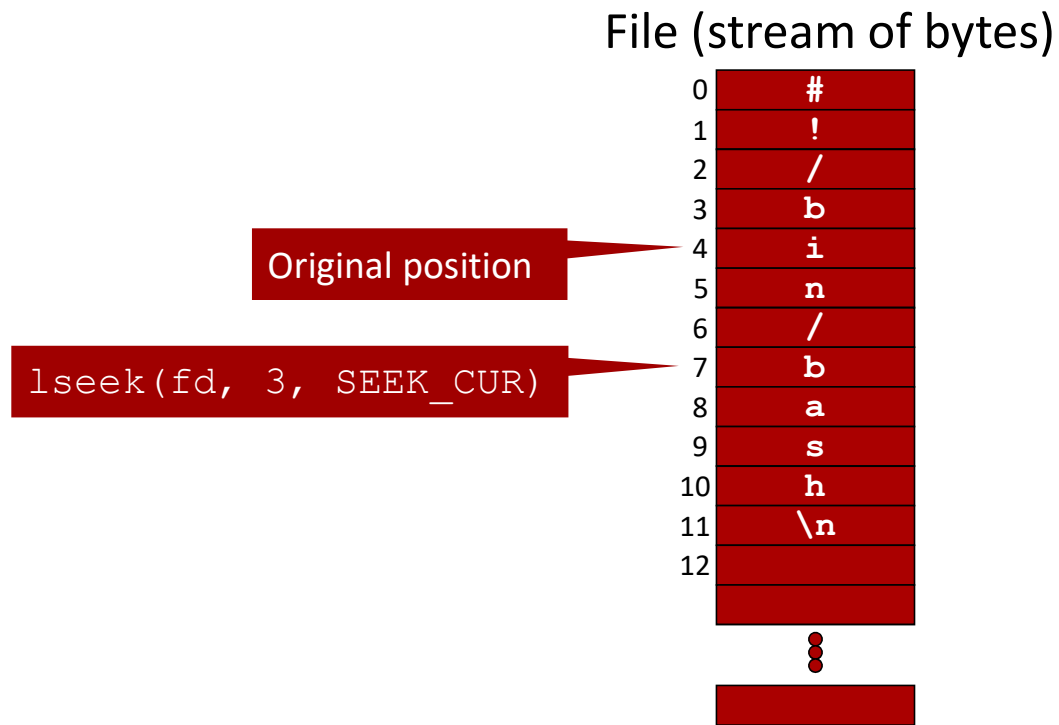
# `lseek()` :: `SEEK_SET` :: Setting Position



```
lseek() :: SEEK_CUR :: Moving backwards
```

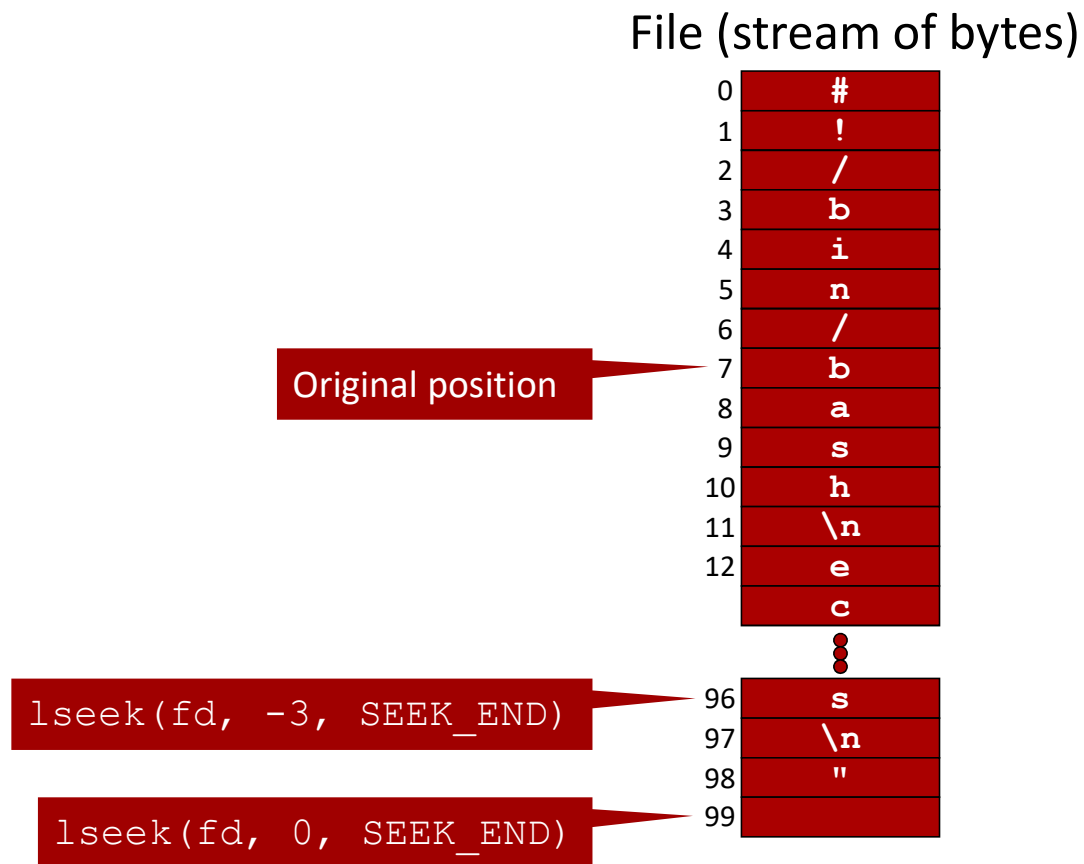


```
lseek () :: SEEK_CUR :: Moving Forwards
```





## lseek() :: SEEK\_END :: Moving Relative to the End

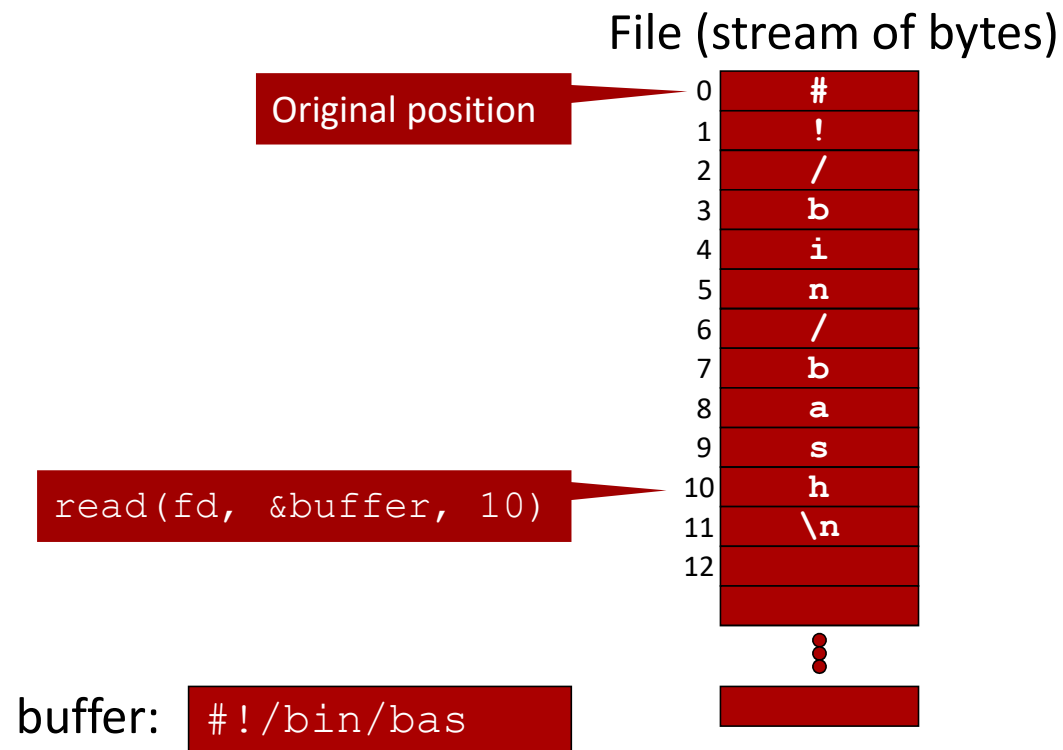


## Read/Write and the File Pointer

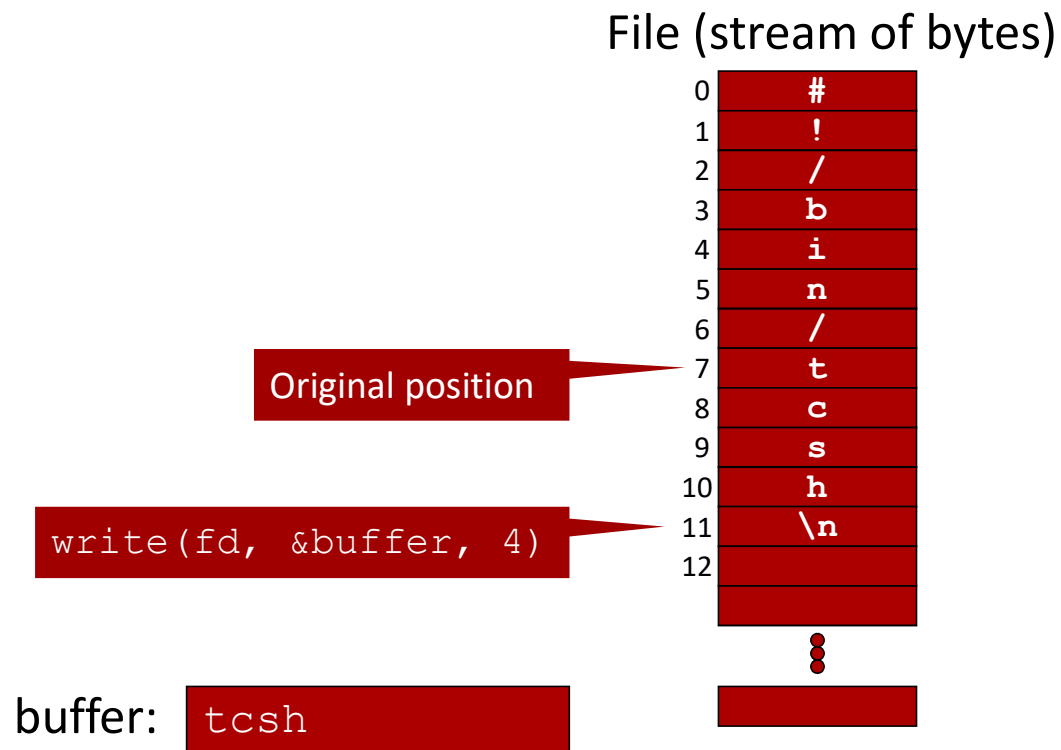
- If you've opened a file for reading and/or writing, be aware that *both* of these operations will change the file pointer location!
- The pointer will be incremented by exactly the number of bytes read or written



## read () and the File Pointer



# `write()` and the File Pointer



# Complete Read/Write Example

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

```
int main(void)
{
```

```
    int file_descriptor;
    char* newFilePath = "./newFile.txt";
    char* giveEm = "THE BUSINESS\n";
    ssize_t nread, nwritten;
    char readBuffer[32];
```

Mode bits

Permissions

```
    file_descriptor = open(newFilePath, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
```

```
    if (file_descriptor == -1)
```

```
    {
        printf("Hull breach - open() failed on \"%s\"\n", newFilePath);
        perror("In main()");
        exit(1);
    }
```

```
    nwritten = write(file_descriptor, giveEm, strlen(giveEm) * sizeof(char));
```

```
    memset(readBuffer, '\0', sizeof(readBuffer)); // Clear out the array before using it
    lseek(file_descriptor, 0, SEEK_SET); // Reset the file pointer to the beginning of the file
    nread = read(file_descriptor, readBuffer, sizeof(readBuffer));
```

```
    printf("File contents:\n%s", readBuffer);
    return 0;
```

```
}
```

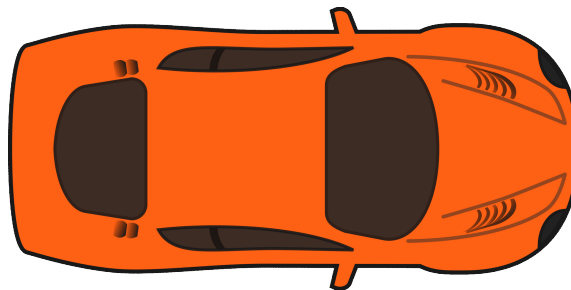
These two steps are really important to avoid nasty bugs

# The Standard IO Library in C

- `fopen`, `fclose`, `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `getc`, `putc`, `gets`, `fgets`, `fseek`, **etc.**
- Automatically buffers input and output intelligently
- Easy to work in line mode
  - i.e., read one line at a time
  - write one line at a time
- Powerful string and number formatting
- To use them:  
`#include <stdio.h>`

# Why Teach and Use `read()` & `write()`?

- Maximum performance
  - IF you know exactly what you are doing
  - No additional hidden overhead from `stdio`, which is much slower!
  - No hidden system calls behind `stdio` functions which may be non-reentrant
- Control exactly what is written/read and at what times



# Some stdio Functions

- `fclose` Close a stream
- `feof` Check if End Of File has been reached
- `fgetc` Get next character from a stream
- `fgetpos` Get position in a stream
- `fopen` Open a file
- `fprintf` Print formatted data to a stream
- `fputc` Write character to a stream
- `fread` Read block of data from a stream
- `fseek` Reposition stream's position indicator (stdio version of `lseek`)
- `getc` Get the next character
- `getchar` Get the next character from stdin





# Some More stdio Functions

- `gets`            Get a string from stdin
- `printf`          Print formatted data to stdout
- `putc`            Write character to a stream
- `putw`            Write an integer to a stream
- `remove`          Delete a file
- `rename`          Rename a file or directory
- `rewind`          Reposition file pointer to the beginning of a stream
- `scanf`           Read formatted data from stdin
- `sprintf`          Format data to a string
- `sscanf`          Read formatted data from a string
- `ungetc`          Push a character back into stream



# Files or Streams?



- **stdin**, **stdout**, and **stderr** are actually *file streams*, not file system files
- File streams wrap around, and provide buffering to, the underlying file descriptor among other features
- The stdio library streams are connected with the `fopen()` call to a variable of type `FILE*` :

```
FILE* myFile = fopen("datafile103", "r");
```

- Streams are closed when a process terminates
- Raw file descriptors with open files are passed on to child processes:
  - A process spawns a new child process with `fork()` , all open files are shared between parent and child processes

# Getting Input From the User: userInput.c

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

void main()
{
    int numCharsEntered = -5; // How many chars we entered
    int currChar = -5; // Tracks where we are when we print out every char
    size_t bufferSize = 0; // Holds how large the allocated buffer is
    char* lineEntered = NULL; // Points to a buffer allocated by getline() that holds our entered string + \n + \0

    while(1)
    {
        // Get input from the user
        printf("Enter in a line of text (CTRL-C to exit):");
        numCharsEntered = getline(&lineEntered, &bufferSize, stdin); // Get a line from the user
        printf("Allocated %zu bytes for the %d chars you entered.\n", bufferSize, numCharsEntered);
        printf("Here is the raw entered line: \"%s\"\n", lineEntered);

        // Print out the actual contents of the string that was entered
        printf("Here are the contents of the entire buffer:\n");
        printf(" # CHAR INT\n");
        for (currChar = 0; currChar < bufferSize; currChar++) // Display every character in both dec and ASCII
            printf("%3d `c` ' %3d\n", currChar, lineEntered[currChar], lineEntered[currChar]);
        free(lineEntered); // Free the memory allocated by getline() or else memory leak
    }
}
```

When bufferSize = 0 and lineEntered = NULL, getline() allocates a buffer for you with malloc()

Could also be a regular file opened as a stream with fopen()

getline() is my preferred tool to get user input

# Results - Getting Input From the User: userInput.c

```
$ gcc -o userInput userInput.c
```

```
$ userInput
```

```
Enter in a line of text (CTRL-C to exit):abc o00lI1
```

```
Allocated 120 bytes for the 11 chars you entered.
```

```
Here is the raw entered line: "abc o00lI1
```

```
"
```

```
Here are the contents of the entire buffer:
```

```
# CHAR INT
```

```
0 `a' 97
```

```
1 `b' 98
```

```
2 `c' 99
```

```
3 ` ' 32
```

```
4 `o' 111
```

```
5 `0' 48
```

```
6 `O' 79
```

```
7 `l' 108
```

```
8 `I' 73
```

```
9 `1' 49
```

```
10 `
```

```
' 10
```

```
11 ` ' 0
```

```
12 ` ' 0
```

```
... (cut) ...
```

```
118 ` ' 0
```

```
119 ` ' 0
```

```
Enter in a line of text (CTRL-C to exit):^C
```

Newline; if you don't want this, just add:

```
lineEntered[numCharsEntered - 1] = '\\0';
```

after calling `getline()`

Null terminator

# Obtaining File Information

- `stat()` and `fstat()`
- Retrieve all sorts of information about a file or directory
  - Which device it is stored on
  - Ownership/permissions of that file
  - Number of hard links pointing to it
  - Size of the file
  - Timestamps of last modification and access
  - Ideal block size for I/O to this file

