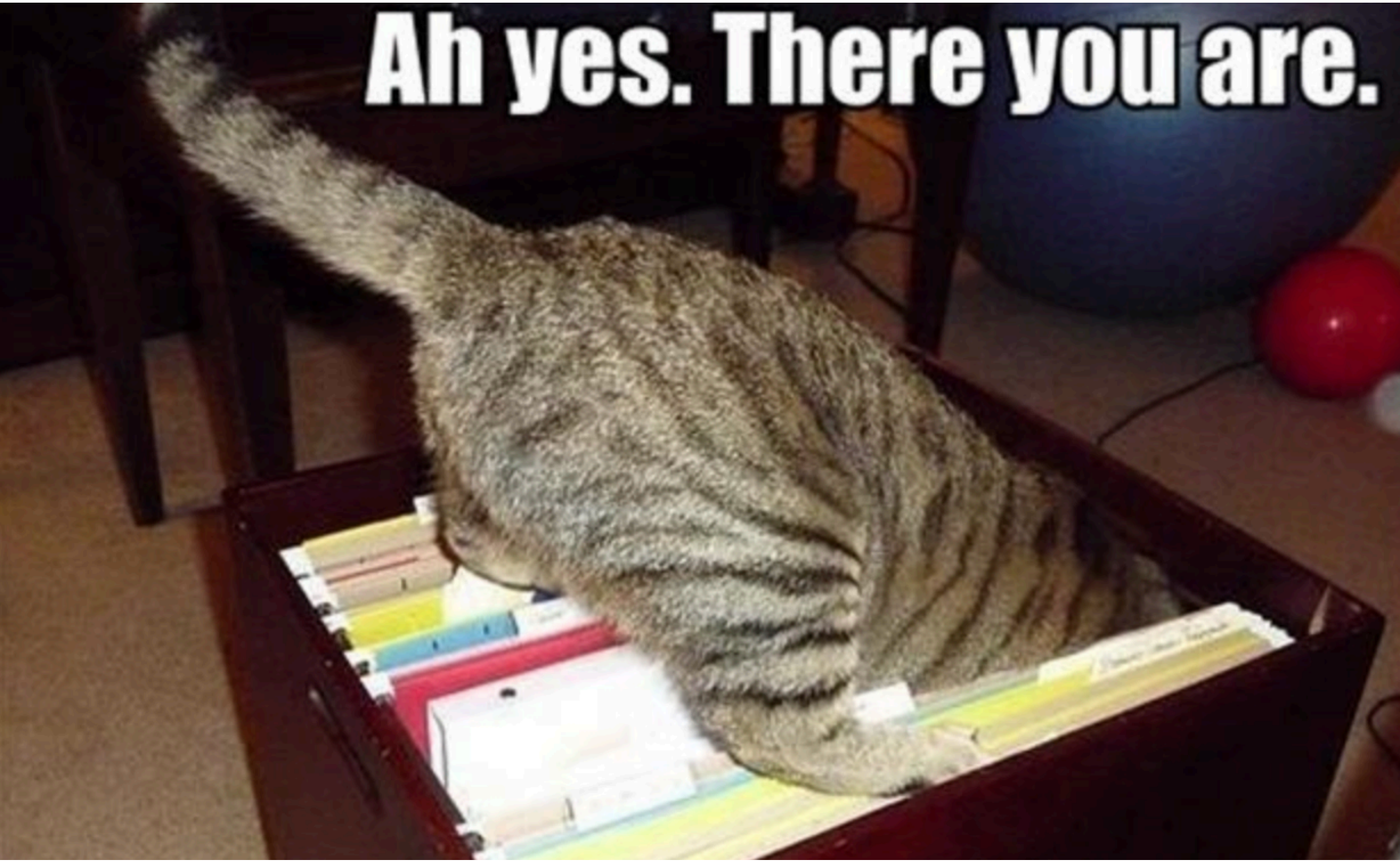
# Week 7 Lecture 2: Files!



# Review and Preview

- Libraries and Archive
- Interacting with the User (scanf, fgets)

- Files and File I/O
- Files and the Operating System

**K&R: Chapters 1-5**
**Sections 4.5, 4.11**
**Sections 5.1-5.5**

# Files

The Last Way to Interact with the User

# File Types - Text Files

- There are two types of files - text files and binary files
  - A **text file** is a sequence of ASCII characters stored on a disk (secondary storage).
  - Each character is stored as one byte.
  - Equivalent to "**strings**" in C.

# File Types - Text Files

- Text **editors** are applications that read/display/write text files.
- There are two types of files (text and binary) and the only difference between them is the type of data stored in them.
- Text files contain only bytes that can be interpreted as ASCII characters.
- Both types are read from and written to using the same techniques.

# File Types - Binary Files

- Binary files consist of a sequence of bytes that are not necessarily interpretable as ASCII characters.
  - Text is still stored using ASCII.
  - Numbers are stored in their binary format (integer, float, double).

# Interpreting Data in Files

- Text and binary data can be mixed in the same file.
- UNIX does not care what is in the file, it is all dependant on how it is interpreted by a program.
- The system considers the file to be a **collection of bytes**.
    - Programs interpret these bytes.

# File Operations

- Files in C are treated as a **stream**:
  - an ordered **sequence** of **bytes** (similar to an array)
  - can be read, written, and moved through
  - separate streams to several files can be opened at one time

# File Operations

- There are five major file operations:
  - **Creating** a new file or **opening** an existing file
  - **Closing** a file
  - **Reading** from a file
  - **Writing** to a file
  - **Moving** around a file (called **seeking**) - this is most important with binary files

# Opening a File

- A file can be **opened** for reading and/or writing.
- The **filename** is associated with a **file pointer**.
- The pointer points to a structure that contains **file information** which is managed by the operating system.
  - Bookkeeping information that the operating system has to share with the program so that it can access the file.

# Declaring a File Pointer

- A file pointer is declared using the `FILE` type:

```
FILE *fp;
```

- The file is created/opened using `fopen()`.

```
fp = fopen ( "filename", "r" );
```

*file pointer* *name of file to be opened* *mode*

- `fopen()` returns a file pointer. If this is `NULL` there was a problem. Always check this.

# File Modes

- **Mode** indicates what you can do with the file.

| Mode | Meaning |
| --- | --- |
| r | **Open** an **existing** file for **reading**; start at the *beginning* |
| w | **Create** a **new** file for **writing** |
| a | **Append** - open an **existing** file; **write** at the *end* of the file |

# File Modes

| Mode | Meaning |
| --- | --- |
| r+ | **Open existing** file for **reading** & **writing**; start at the *beginning* |
| w+ | **Create new** file for **reading** and **writing**; start at the *beginning;* **truncates existing** file |
| a+ | **Open existing** file for **reading** & **writing**; write to the *end* of the file |

# Reading from and Writing to a File

- Reading from a file containing text can be done using the same commands that are used to read from the keyboard (*i.e.* the `stdin` stream).
- You need to use the "**file**" versions: `fscanf`, `fgets`.
- Both commands are exactly the same as reading from the keyboard/`stdin` with the addition of a file pointer.
- Writing text to a file is performed using `fprintf` and `fputs`.

exactly the same as `scanf, printf`

```
int fscanf ( fp, “format”, &vars);
```

- Read individual values from a specified stream (`fp`).
- On success, the number of input items successfully matched and assigned is returned
  - This can be fewer than requested or even zero.
- The value EOF is returned if the end of input has been reached.

# `char *fgets ( str, size, fp );`

- Read a string from an input stream (`fp`).
- Reads in at most one less than `size` characters from `fp` and stores them into the memory pointed to by `str`.
- Reading stops after an `EOF` or a newline. If a newline is read, it is stored in the string.
- A terminating `'\0'` is stored after the last character in the string.
- Returns `str` on success, and `NULL` on error or when end of file occurs while no characters have been read.

same as `printf`

```
int fprintf ( fp, "format", vars );
```

- Similar to output using **`printf`** but writes to a file instead of the screen (**`stdout`**).
- Returns the number of characters printed (excluding the `'\0'` used to end output to strings).

# `int fputs( strPtr, fp );`

- Writes the string **`strPtr`** to the file **`fp`**, without its terminating byte `'\0'`.
- Return a nonnegative number on success, or **`EOF`** on error.

# Closing a File

- When you have finished reading or writing to a file it must be closed using `fclose`.

```
fclose ( fp );
```

- This forces any **buffered** data to be written to the disk.
- If the file is not closed then data may be lost (*i.e.* not written to the file).
- It also frees the memory used to store file information.

# Closing a File - What Does It Do?

Security Alert: Always close a file after you have finished using it.

1. Removes it from the file descriptor table.
   - This table is of limited size - can become full (**attack**).
2. Flushes the **buffer**.
   - There is not a direct path to the disk.
   - It would be inefficient to actually write to the disk (hardware) after each program write.

```c
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  FILE *fp;
  char line[100];
  int i = 0;

  /* just read the file */
  if ((fp = fopen("testfile1.txt", "r")) == NULL) {
    printf ( "ERROR - cannot open testfile1.txt
              for reading\n");
    exit (-1);
  }

  while ( fgets ( line, 100, fp ) != NULL ) {
    printf ( "%s", line );
  }
  fclose ( fp );
```

```
/* create a new file and write to it */
if ((fp = fopen("testfile2.txt", "w")) == NULL) {
   printf ( "ERROR - cannot create testfile2.txt
             for writing\n");
   exit ( -1 );
}
fprintf(fp, "This is a test file to demonstrate
             \"w\" mode.\n" );
for ( i=0; i<10; i++ ) {
   fprintf ( fp, "i = %d\n", i );
}
fclose ( fp );
```

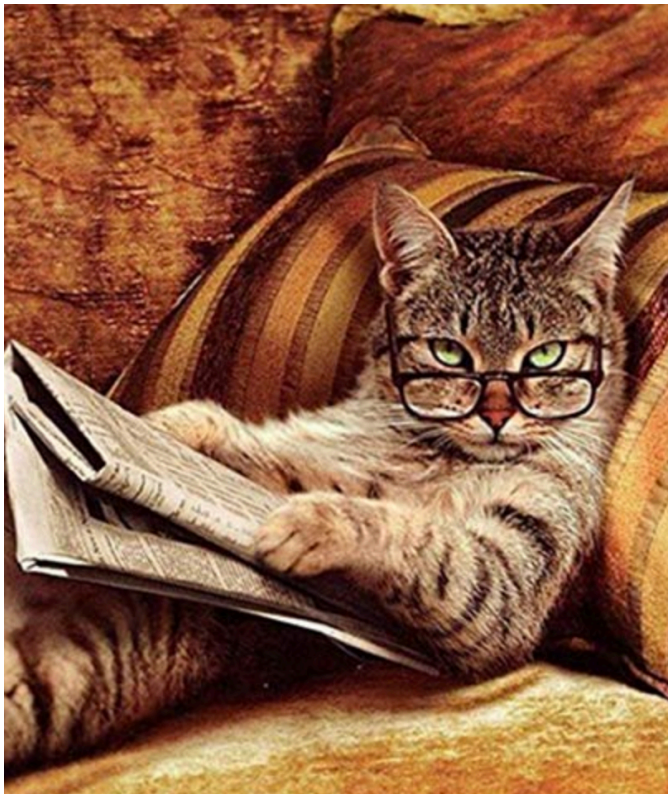
```
/* show the append mode */
if ((fp = fopen("testfile2.txt", "a")) == NULL) {
   printf ( "ERROR - cannot open testfile2.txt
             for appending\n");
   exit ( -1 );
}
for ( i=10; i>0; i-- ) {
   fprintf ( fp, "i = %d\n", i );
}
fclose ( fp );
```


# Files in UNIX

How does the OS see files?

# Information about a File

- The operating system is responsible for the file system and has to maintain information about each file.
- What types of information does it need to know?



# File Information

- **File**: Absolute path name of the file
- **Size**: File size in bytes
- **Blocks**: Total number of blocks used by this file
- **File Type**

How big is a block?
$ sudo blockdev --getbsz /dev/sda1
4096 *or* 4K

# Available File Types

- **Regular** file (all normal files both text and binary)
- **Directory**
- **Socket**
- **Symbolic link**
- **Block special file** (provide buffered access to hardware devices like hard disks)
- **Character special file** (provide unbuffered, direct access to the hardware devices like the terminal device file)

A *socket* is one endpoint of a two-way communication link between two programs running on a network.

A *symbolic link* is a file that points to another file. It does not contain the data in the target file. It simply points to the target file.

# Available File Types

- **Device**: Device number in hex and decimal
- **Inode**: Inode number is a unique number for each file which is used for the internal maintenance by the file system.

```
$ ls -il fileBasics.c
254 -rw-r--r-- 1 debs debs 699 Oct 23 14:35 fileBasics.c
```

- **Links**: Number of links to the file
- **Uid**: File owner's user id and user name are displayed.
- **Gid**: File owner's group id and group name are displayed.

# Available File Types

- **Access**: Access specifier displayed in both octal and character format.
- **Access**: Last access time of the file.
- **Modify**: Last modification time of the file.
- **Change**: Last change time of the inode data of that file.
- **Birth**: Time of file birth

# File Permissions

- **Read**
  - *File*: permission to open and read a file
  - *Directory*: permission to lists its content
- **Write**
  - *File*: permission to modify the contents of a file
  - *Directory*: permission to add, remove and rename files in the directory
- **Execute**
  - *File*: permission to run (execute) an executable file / program

# File Permissions: Octal Format

- Values for read, write and execute permissions in octal are:
  - 4 = Read permission
  - 2 = Write permission
  - 1 = Execute permission

| Read | Write | Execute | |
|---|---|---|---|
| 1 | 0 | 0 | *4* |
| 0 | 1 | 0 | *2* |
| 0 | 0 | 1 | *1* |
| 1 | 1 | 0 | *6* |
| 1 | 1 | 1 | *7* |

`0644 = Read & Write / Read / Read`

# File Permissions: Character Format

- File Type: First character is the type of the file.
- User Permission: 2nd - 4th characters specify the read (r), write (w), and execute (x) permission of the **user**.
- Group Permission: 5th - 7th characters specify the read (r), write (w), and execute (x) permission of the **group**.
- World Permission: 8th - 10th characters specify the read (r), write (w), and execute (x) permission of the world.

Regular file `-rw-r--r--`

Directory `drw-r--r--`

`t|uuu|ggg|www`

# Files, Users, and Permissions

- **User**: a user is the owner of the file.
- **Group**: a group can contain multiple users and are identified by the system with a specific gid (group id).
- **World/Other**: any other user who has access to the file.

```
$ ls -l days*
-rwxr-xr-x 1 debs debs 16720 Sep 24 14:49 daysCalculatorA
-rw-r--r--  1 debs debs  2318  Sep 24 14:48 daysCalculatorA.c
```

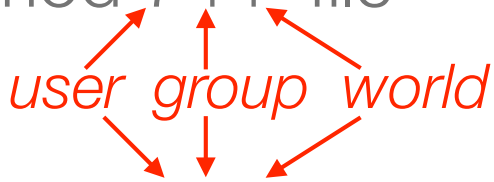
# UNIX Command of the Day: `stat`

```
$ stat fileBasics.c
File: fileBasics.c
Size: 699         Blocks: 8
                IO Block: 1048576 regular file
Device: 2bh/43d  Inode: 254           Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/
debs)   Gid: ( 1001/    debs)
Access: 2019-10-23 14:35:45.000000000 -0400
Modify: 2019-10-23 14:35:45.000000000 -0400
Change: 2019-10-23 14:35:45.000000000 -0400
 Birth: -
```

# `chmod` - Octal Mode

- `chmod ooo file`

$ chmod 744 file `-rwx-r—r—`




$ chmod 664 file `-rw-rw-r—`

$ chmod 777 file `-rwx-rwx-rwx`

# `chmod` - Symbolic Mode

- `chmod ugoa +- rwx`
- **u** – user , **g** – group, **o** – others , **a** – all
- **+** to add permission , **–** to remove permission
- **r w x** is used for read , write, execute

```
$ chmod u+rwx,g+rw,o+r  file

User permissions are read, write, execute

Group permissions are read and write

Other/World permissions are read
```

```
debs@deb-socs:~/Programs$ ls -l days*
-rwxr-xr-x 1 debs debs 16720 Sep 24 14:49 daysCalculatorA
-rw-r--r-- 1 debs debs  2318 Sep 24 14:48 daysCalculatorA.c
debs@deb-socs:~/Programs$ chmod g+w,o+w daysCalculatorA.c
debs@deb-socs:~/Programs$ ls -l days*
-rwxr-xr-x 1 debs debs 16720 Sep 24 14:49 daysCalculatorA
-rw-rw-rw- 1 debs debs  2318 Sep 24 14:48 daysCalculatorA.c
debs@deb-socs:~/Programs$ chmod 777 daysCalculatorA
debs@deb-socs:~/Programs$ ls -l days*
-rwxrwxrwx 1 debs debs 16720 Sep 24 14:49 daysCalculatorA
-rw-rw-rw- 1 debs debs  2318 Sep 24 14:48 daysCalculatorA.c
```