Computer Science I

File Input/Output

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

Part I: Introduction & File Output

Overview

- A *file* is a unit of stored memory, usually on disk
- The following are also files:
  - Directories
  - Buffers (standard input/output)
  - Programs, stored and running
  - Network sockets
- Files may be plaintext (ASCII) or binary
- Or: plaintext data not intended for human consumption
- CSV, XML, JSON, base-64 encoding

Overview

- You read from a file (input) or write to a file (output)
- Three basic steps for file I/O:
  - Open the file
  - Process the file
  - Close the file

Opening Files

- Files are supported in C using a *file pointer*
- `FILE *` points to a location in a file
- The `fopen()` function opens a file and returns a file pointer
- Initially: points to the start of the file
- As you read through the file, the pointer is updated
- Returns `NULL` if unsuccessful
- Also called a *handle*

Opening Files

- `FILE * fopen(const char *filename, const char *mode);`
- First argument: path and name of the file to open
- Second argument: *mode* to open it up in
- File input: `"r"` for *reading*
- File output: `"w"` for *writing*
- Demonstration

Demonstration

```c
//open a file data.txt in the current directory for reading:
FILE *f = fopen("data.txt", "r");

//open a file data.txt for writing:
FILE *f = fopen("data.txt", "w");

//you can also use relative paths
FILE *f = fopen("../../data.txt", "r");

//absolute path:
FILE *f = fopen("/etc/shadow", "r");

//error checking:
if(f == NULL) {
  printf("Unable to open file!\n");
}
```

Pitfalls

- Your program must have proper permissions to read/write a file
- Opening a file for writing will create it if it does not already exist
- Opening an existing file for writing will *clobber* it
- Failure to *close* a file *may* lead to data corruption
- Closing a file:
  `fclose(f);`

Paths

- Current Working Directory: `.`
- File System Root: `/`
- One directory "up" the hierarchy `..`

File Output

- Many ways to output to a file
- Easiest and most simple: `fprintf()`
- Same functionality as `printf()` and `sprintf()`
- Takes a `FILE*` as its first argument and prints to it
- Demonstration

File Output

```c
int a = 42;

FILE *f = fopen("data.txt", "w");

fprintf(f, "Hello World!\n");
fprintf(f, "a = %d\n");
fprintf(f, "pi is %.4f\n", M_PI);

fclose(f);
```

Part II: File Input

File Input

- There are many (dangerous) ways of reading from a file
- Best to limit the amount of data read so it is predictable
- Avoid "buffer overflows" (strings where we store the data)
- Focus on two useful functions:
- `fgetc()` gets a single character from a file
- `fgets()` gets (up to) an entire line from a file

fgetc

- `int fgetc(FILE *f);`
- Reads a single `char` from the file `f`
- Returns the ASCII value of the character
- Automatically advances the file pointer to the next character
- Returns a special flag, `EOF` when it gets to the end-of-file
- Demonstration

fgetc

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

int main(int argc, char **argv) {

  FILE *f = fopen("./data/students.csv", "r");
  char c = fgetc(f);
  while(c != EOF) {
    printf("c = %c\n", c);
    c = fgetc(f);
  }
  fclose(f);

  return 0;
}
```

fgets

- `char * fgets(char *str, int size, FILE *f);`
- Reads at most `size-1` characters from `f`
- Places the result into `str` (a "buffer")
- Automatically null-terminates the string
- Stops early if the end-of-line character `\n` is encountered
- *Retains* the end-of-line character in `str`
- Returns `NULL` when it reaches the end of file
- Alternatively (for both): `int feof(FILE *f);`
- Demonstration

fgets

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

int main(int argc, char **argv) {

  FILE *f = fopen("./data/students.csv", "r");

  char buffer[100];
  char *line = fgets(buffer, 100, f);
  while(line != NULL) {
    //chomp out the endline character:
    int n = strlen(buffer);
    if(buffer[n-1] == '\n') {
      buffer[n-1] = '\0';
    }
    printf("line = %s\n", line);
    line = fgets(buffer, 100, f);
  }
  fclose(f);
  return 0;
}
```

Part III: Binary Files

Binary Files

- Most data formats are *binary*: raw bits and bytes
- May have specific format and magic number identifiers
- GIF format: https://en.wikipedia.org/wiki/GIF
- Often more efficient: less space, easier to read/write
- C: `fread` (reading) and `fwrite` (writing) binary data
- Reads/writes multiple pieces of data at once

Binary Files

- `size_t fread(void *ptr, size_t size, size_t n, FILE *f);`
- `size_t fwrite(const void *ptr, size_t size, size_t n, FILE *f);`
- Both take the same arguments
  - `ptr` – pointer to the data to be read/written
  - `size` – number of bytes for each item (use `sizeof`)
  - `n` – number of items to be read/written
  - `f` – file pointer
- Demonstration

Part IV: Exercises

File I/O

Exercise

Write a program to process a CSV file with student information including last name, first name, NUID, and GPA to prepare a Dean's list. Output to a separate file all student names whose GPA is greater than 3.5, but only include their first name and last name (one to a line).

File I/O

Exercise

- Passwords are stored using cryptographic hash functions
- $hash(\text{password}) \rightarrow$ `0x5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8`
- Common for users to use dictionary words as passwords
- They can easily be broken using a *dictionary attack*
- Exercise: dictionary attack a SHA-256 hashed password: `0xaa97302150fce811425cd84537028a5afbe37e3f1362ad45a51d467e17afdc9c`