CSE 303
Lecture 15

C File Input/Output (I/O)

reading: *Programming in C* Ch. 16;
Appendix B pp. 473-478

slides created by Marty Stepp
http://www.cs.washington.edu/303/

Console I/O review

- `#include <stdio.h>`

function	description
`int getchar()`	reads/returns a `char` from console
`int putchar(int c)`	writes a char from console
`char* gets(char* buf)`	reads a line from console into given buffer; returns buffer or NULL on failure
`int puts(char* s)`	writes a string to console, followed by `\n` ; returns >= 0 on success, < 0 on failure
`int printf(char* format, ...)`	prints formatted console output
`int scanf(char* format, ...)`	reads formatted console input; returns number of tokens successfully read

man page sections

- some commands occur in multiple places in man pages

```
$ man printf
PRINTF(1)                        User Commands                        PRINTF(1)
NAME
       printf - format and print data
SYNOPSIS
       printf FORMAT [ARGUMENT]...
DESCRIPTION
       Print ARGUMENT(s) according to FORMAT, or execute according to OPTION:
...
```

- search for a command in man using -k; specify section with -s

```
$ man -k printf
Printf []            (3)  - Formatted output functions
Tcl_AppendPrintfToObj [] (3)  - manipulate Tcl objects as strings
asprintf []          (3)  - print to allocated string
...

$ man -s 3 printf
NAME
       printf,   fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion
SYNOPSIS
       #include <stdio.h>
       int printf(const char *format, ...);
...
```

File I/O functions

- `#include <stdio.h>`

function	description
`FILE* fopen(char* filename, char* mode)`	mode is `"r"`, `"w"`, `"a"`; returns pointer to file or `NULL` on failure
`int fgetc(FILE* file)` `int fgets(char* buf, int size, FILE* file)`	read a `char` from a file; read a line from a file
`int fputc(char c, FILE* file)` `int fputs(char* s, FILE* file)`	write a `char` to a file; write a string to a file
`int feof(FILE* file)`	returns non-zero if at EOF
`int fclose(FILE* file)`	returns 0 on success

`FILE* stdin`
`FILE* stdout`
`FILE* stderr`
	streams representing console input, output, and error

  - most return EOF on any error (which is -1, but don't rely on that)

More file functions

function	description
`int fprintf(FILE* file, char* format, ...)`	prints formatted output to file, a la `printf`
`int fscanf(FILE* file, char* format, ...)`	reads formatted input from file, a la `scanf`
`FILE* freopen(char* filename, char* mode, FILE* stream)`	re-opens the file represented by given name and stream
`flockfile`, `ftrylockfile`, `funlockfile`	functions for lock/unlocking a file for outside modification
`fseek`, `ftell`, `rewind`, `fgetpos`, `fsetpos`	functions for get/setting the offset within the input
`setbuf`, `setbuffer`, `setlinebuf`, `fflush`	functions for performing *buffered I/O* (much faster)
`int ungetc(int c, FILE* file)`	un-reads a single character, so `fgetc` will later return it (limit 1 time in a row)

Exercise

- Write a program that reads a file of state-by-state 2008 presidential election polls, where each line contains a state code, percent votes for Obama/McCain, and number of electoral votes for that state:

```
AL 34 54 9
AK 42 53 3
AZ 41 49 10
...
```

- The program outputs the electoral votes of each candidate:

```
Obama ???, McCain ???
```

Exercise solution

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

int main(void) {
    int obama_total = 0;
    int mccain_total = 0;
    FILE* f = fopen("polls.txt", "r");
    while (!feof(f)) {
        char state[4];
        int obama, mccain, evotes;
        fscanf(f, "%s %d %d %d", state, &obama, &mccain, &evotes);
        if (obama > mccain) {
            obama_total += evotes;
        } else if (mccain > obama) {
            mccain_total += evotes;
        }
    }

    printf("Obama = %d, McCain = %d\n", obama_total, mccain_total);
    return 0;
}
```

Exercise

- Write a program hours that reads a file of worker hours such as:

```
123 Kim 12.5 8.1 7.6 3.2
456 Brad 4 11.6 6.5 2.7 12
789 Stef 8 7.5
```

- The program outputs each employee's total hours and hours/day:

```
Kim   (id #123) worked 31.4 hours (7.85 / day)
Brad  (id #456) worked 36.8 hours (7.36 / day)
Stef  (id #789) worked 15.5 hours (7.75 / day)
```

Exercise solution

```
int main(void) {
    char buf[1024] = {'\0'};
    FILE* f = fopen("hours.txt", "r");
    while (!feof(f)) {
        double hours = 0.0;
        int days = 0;
        int id;
        char name[80] = {'\0'};
        char* token;
        fgets(buf, sizeof(buf), f);       // read line from file
        token = strtok(buf, " ");
        id = atoi(token);                 // read id

        token = strtok(NULL, " ");
        strcpy(name, token);              // read name
        token = strtok(NULL, " ");
        while (token) {                   // read each day's hours
            days++;
            hours += atof(token);
            token = strtok(NULL, " ");
        }

        printf("%-6s (id #%d) worked %4.1f hours (%.2f / day)\n", name,
               id, hours, (hours / days));
    }
    return 0;
}
```

File ops; temp files

function	description
`int remove(char* filepath)`	deletes the given file
`int rename(char* oldfile, char* newfile)`	renames/moves a file; if newfile exists, will be replaced
`int mkdir(char* path, int mode)`	creates a directory

- functions return 0 on success, -1 on failure

- **temporary files**: data that need not persist after program exits
  - are put in a specific folder (`/tmp` on Linux)

function	description
`char* tmpnam(char* buffer)`	returns a full path that can be used as a temporary file name
`FILE* tmpfile(void)`	returns a pointer for writing to a temp file

Error handling

- `#include <errno.h>`

function	description
`int errno`	an integer containing the last system I/O error code that has occurred (`E_OK` if none)
`void perror(char* msg)`	prints a description of the last error that occurred, preceded by `msg` (if not `NULL`)
`int ferror(FILE* file)`	returns error status of the given file stream (`E_OK` if no error has occurred)
`char* sys_errlist[]`	array of error messages, indexed by error code
`int sys_nerr`	size of `sys_errlist` array

```
FILE* infile = fopen();
if (fputs(infile, "testing 1 2 3\n") < 0) {
    perror("Error writing test string");
}
```

Exceptions vs. error codes

- Java uses exceptions for most error handling:

```
try {
    Scanner in = new Scanner(new File("in.txt"));
    String line = in.nextLine();
} catch (IOException ioe) {
    System.out.println("I/O error: " + ioe);
}
```

- C uses an error return code paradigm:

```
char buf[80];
FILE* in = fopen("in.txt", "r");
if (!in) {
    perror("Error opening file");
}
if (fgets(buf, 80, in) < 0) {
    perror("Error reading file");
}
```

Command-line arguments

- you can declare your `main` with two optional parameters:
  - `int argc` - number of command-line arguments
  - `char* argv[]` - command-line arguments as an array of strings

```
int main(int argc, char* argv[]) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("arg %d is %s\n", i, argv[i]);
    }
    return 0;
}
```

- Output:

```
$ ./example testing 42
arg 0 is ./example
arg 1 is testing
arg 2 is 42
```

* getopt***

Buffer overrun

- What's wrong with this code?

```
char str1[8] = {'\0'};     // empty strings
char str2[8] = {'\0'};
char str3[8] = {'\0'};
...
strcpy(str2, "Hello there");
scanf("%s", str3);
printf("str1 is \"%s\"\n", str1);
printf("str2 is \"%s\"\n", str2);
```

- Output:

```
str1 is "ere"
str2 is "Hello there"
```

Preventing overruns

- `gets` and `scanf` (with `%s`) are considered inherently unsafe
  - there is no way to constrain them to a buffer's size
  - the user can always supply an input that is too large and overrun it
  - advice: never use scanf or gets in "production" code

- instead, use `fgets` with `stdin`, which has a length limit

```
char buffer[80];
fgets(buffer, sizeof(buffer) - 1, stdin);
```

- do not use `strcat`, `strcmp` with unknown input
  - safer to use `strncat`, `strncmp` and pass the buffer length as *n*

```
char buffer[80] = {'\0'};
strncpy(buffer, "Hello there", 12);
```

Binary data

function	description
`size_t fwrite(void* ptr, size_t size, size_t count, FILE* file)`	writes given number of elements from given array/buffer to file *(size_t means unsigned int)*
`size_t fread(void* ptr, size_t size, size_t count, FILE* file)`	reads given number of elements to given array/buffer from file

```
// writing binary data to a file
int values[5] = {10, 20, 30, 40, 50};
FILE* f = fopen("saved.dat", "w");
fwrite(values, sizeof(int), 5, f);

// reading binary data from a file
int values[5];
FILE* f = fopen("saved.dat", "r");
fread(values, sizeof(int), 5, f);
```

Processes and pipes

- A C program can execute external commands/processes
  - you can open a stream for reading input/output from the process

function	description
`int system(char* command)`	executes an external program; returns that program's exit code or -1 on failure
`FILE* popen(char* command, char* type)`	`type` is `"r"` or `"w"`; starts a program and returns a FILE* to read or write the process's stdin/out
`int pclose(FILE* process)`	waits for external process to complete; returns its exit code

Interacting with the OS

- `#include <unistd.h>`

function	description
`int chdir(char* path)`	changes working directory
`int mkdir(char* path, int mode)`	creates a directory
`int rmdir(char* path)`	removes a directory
`char* getcwd(char* buf, size_t len)`	gets working directory
`chown, fork, getgid, getgroups, gethostname, getlogin, getgid, getsid, getuid, link, unlink, nice, pause, setgid, setsid, setuid, sleep, unlink, usleep`	other misc. functions

- `#include <sys/stat.h>`

function	description
`int stat(char* filepath, struct stat* buf)`	get information about a file (put it into the given struct)

  - most functions return 0 on success, -1 on failure