

Computer Science & Engineering 150A

Problem Solving Using Computers

Lecture 12 - File Processing

Christopher M. Bourke

Spring 2009

- Up to now, all input/output has been via the standard interface
- Standard Input: `stdin`, keyboard
- Standard Output: `stdout`, console
- Useful to save, load, process data to/from a *file*
- Actually: `stdout`, `stdin` are files!

- *Files* are collections of data stored on secondary media (disks)
- Files can either be text files or data (binary) files
- Text files contain printable ASCII text characters
- Binary files contain pure binary data: 0s and 1s
- Every file has an **EOF** (end of file) marker
- Files sometimes referred to as *streams*

- C has a built-in file type: `FILE`
- All functions dealing with `FILE` types are by *reference*
- Best to only deal with file *pointers*
- Declaration:

```
FILE *myInputFile = NULL;  
FILE *myOutputFile = NULL;
```
- Best practice: assign a `NULL` value in declaration

- Open file streams using `fopen`:
`FILE *fopen(const char *path, const char *mode);`
- First argument: string of the *path* of the file
- Second argument: open mode
- Return type: a pointer to the file stream

- Default: current directory (the directory the program is executed in)
- Path may be *absolute*:
`"/home/grad/cbourke/cse150a/program01/file.txt"`
- or *relative*:
`"../file.txt"`
`"file.txt"`

Second argument: string indicating the *mode*

- r** Open text file for reading. The stream is positioned at the beginning of the file.
- r+** Open for reading and writing. The stream is positioned at the beginning of the file.
- w** Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- w+** Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- a** Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- a+** Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

- Some systems recognize the `b` flag in the mode argument
- Used to indicate the file will be opened for *binary* read/write
- Ignored in POSIX systems (used for C89 compatibility)
- Unnecessary on CSE

```
1 FILE *myInputFile = NULL;  
2 FILE *myOutputFile = NULL;  
3 myInputFile = fopen("data.txt", "r");  
4 myOutputFile = fopen("results.txt", "w");
```

Opens two files: one for read, one for write, in the current working directory

- Opening files can fail
- File not found, program doesn't have permissions, etc.
- In the event of failure, `fopen` returns `NULL`
- Best practice: check for file open failure

```
1  if(myInputFile == NULL)
2  {
3      printf("Opening file, data.txt failed, quitting\n");
4      exit(-1);
5  }
```

- The standard input/output uses `scanf` and `printf`
- Processing input/output with files uses `fscanf` and `fprintf`
- Both process from any given file stream
- Full function definitions:

```
int fprintf(FILE *stream, const char *format, ...);
```

```
int fscanf(FILE *stream, const char *format, ...);
```

- First argument: file stream
- Second: string representing format
- Same placeholders and formatting rules as `printf`, `scanf`
- Followed by list of variables

```
1 fscanf(myInputFile, "%d", &aNumber);  
2 fprintf(myOutputFile, "Hey, I read %d! Neato!\n", aNumber);
```

Processing Text Files

End of File

CSCE150A

Introduction

Text Files

Processing

Binary Files

Examples &
Exercises

- The special `EOF` (end-of-file) character lets you know when the end of the file has been reached
- `fscanf` either returns the number of successfully matched placeholders or `EOF`
- Useful when processing to the end of a file

```
1  /* read in the first integer: */
2  int nums[100], i = 0;
3  int status = fscanf(myInputFile, "%d", &nums[i]);
4  while(status != EOF) {
5      i++;
6      /* status is number of successful items read */
7      status = fscanf(myInputFile, "%d", &nums[i]);
8  }
```

Close a file using `fclose` (**file close**)

```
1  fclose(myInputFile);  
2  fclose(myOutputfile);
```

Failure to close may result in corrupted files.

- Reading files essentially requires you to know in advance what format they are
- Assumption: the file actually follows the format
- If formats are not followed, undefined behavior?
- No program can handle *every* possible contingency
- Other, more complex solutions exist

- You can think of a file stream as simply another output/input device
- In fact, `stdin`, `stdout` are also technically file streams
- Everything in a unix system is a file!
- The following have the same effect:

```
printf("Standard output via printf!\n");  
fprintf(stdout,"Standard output via fprintf!\n");
```

- *Binary* files are files that have no encoding: strictly 0s and 1s
- Less resource intensive
- No delimiters necessary
- Input/output streams are declared exactly the same:

```
FILE *binaryOutputFile = fopen("file.dat", "w");
```

- For binary files, we use `fread` and `fwrite` instead of `fscanf/fprintf`
- Full function definitions:

```
1  size_t fread(void *ptr,  
2           size_t size,  
3           size_t nmemb,  
4           FILE *stream);  
5  size_t fwrite(const void *ptr,  
6           size_t size,  
7           size_t nmemb,  
8           FILE *stream);
```

- Arguments:
 - ① `*ptr` is a pointer to the item that is being read/written
 - ② `size` is the number of bytes that the item `ptr` is pointing to requires.
 - ③ `nmemb` is the number of items to be written/read
 - ④ `stream` is the file stream to be read from/written to
- Return value: number of successfully read/written items

Binary File Processing

Example

CSCE150A

Introduction

Text Files

Processing

Binary Files

Examples &
Exercises

```
1  int aNumber;  
2  fread(&aNumber, sizeof(int), 1, binaryInputFile);  
3  fwrite(&aNumber, sizeof(int), 1, binaryOutputFile);
```

The formatting of a file must be consistent if it is ever to be read properly again.

```
1  #include <limits.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main(void) {
6      /* 2^31 = 2147483647 is the maximum value a signed
7         integer can take; confirm this using limits.h:
8         it defines a macro, INT_MAX, that holds the
9         maximum value a signed integer can take
10     */
11     printf("int's maximum value = %d\n", INT_MAX);
12     printf("int's take %d bytes to store\n", sizeof(int));
13     int a = 2147483647;
14     printf("a = %d\n", a);
15
16     /* */
```

```
17  /* now output this to a regular file */
18  FILE *myStringInteger = fopen("aNumber.txt", "w");
19  fprintf(myStringInteger, "%d", a);
20  fclose(myStringInteger);
21
22  /* now output this to a *binary* file */
23  FILE *myBinaryInteger = fopen("aNumber.bin", "wb");
24  fwrite(&a, sizeof(int), 1, myBinaryInteger);
25  fclose(myBinaryInteger);
26
27  /* How big is each file and why? */
28  }
```

Write a program that opens a plain-text file (read as an argument from the command line) containing integers delimited by white space. Echo the sum to the standard output and also store it in a file called `sum.txt`

CSCE150A

Introduction

Text Files

Processing

Binary Files

Examples &
Exercises

Do Programming project 6 from Chapter 12