File Input / Output

ENEE 140

Prof. Tudor Dumitraș
Assistant Professor, ECE
University of Maryland, College Park

http://ter.ps/enee140

Today's Lecture

- Where we've been
  - Scalar data types
  - Arrays and strings
  - Functions
  - Random number generation
  - Control flow
  - Structuring complex programs
- Where we're going today
  - 2D arrays
  - File Input/Output
  - Project 3
- Where we're going next
  - More file I/O (low-level functions)

2

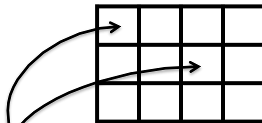
Two-Dimensional Arrays

Needed for Project 3

- Two-dimensional arrays

  `int a[3][4];` int array with 3 rows and 4 columns (12 elements)

  – Think of this as 3 arrays with 4 elements each



- Working with 2D arrays

```
a[0][0] = 0;     access element on first row and first column
a[1][2] = 0;     access element on row 1 and column 2
a[0][4] = 0;     error: index out of bounds
a[3][0] = 0;     error: index out of bounds
```

  – Use 2D arrays to represent matrices

Text File I/O

- Declaring and manipulating file variables

```
#include <stdio.h>
FILE *file;
```
declare the `file` variable

  – Opening

```
file = fopen("filename.txt", "r");
```
mode — open file for reading

    - Mode "r": open existing file for reading
    - Mode "w": open file for writing and erase existing content
    - Mode "a": open file for writing and append after existing content
    - Opening a file in modes "a" or "w" will create the file if it doesn't already exist
    - The `fopen()` function returns NULL if there is an error

  – Closing

```
fclose(file);
```
close file

- **Frequent mistake: Not closing all the files you have opened**

Text File I/O – continued

- Declaring and manipulating file variables

```
#include <stdio.h>
FILE *file;
int i;
char line[256];
```

declare the file variable

- Reading

```
fscanf(file, "%d", &i);
i = getc(file);
fgets(line, 256, file);
```

fscanf: like scanf()
getc: like getchar()
fgets: read an entire line

- Writing

```
fprintf(file, "%d", i);
putc(i, file);
fputs(line, file);
```

fprintf: like printf()
putc: like putchar()
fputs: write an entire line

- The file must be open in order to read or write

A Common Pattern: Reading a File Line-by-Line

```
#include <stdio.h>

char line[MAX_LINE];
int a, b;
FILE *file;                                  variable representing the file

file = fopen("myfile.txt", "r");             open file for reading

if (file == NULL) {                          fopen() failed
 printf ("Could not open the myfile.txt file.\n");
    exit (-1);
}

...
fgets(line, MAX_LINE, file);                 read a line of text from the file
sscanf(line, "%d %d", &a, &b);               parse line with sscanf()
...

fclose(file);                                close file
```

Position in the File

- When operating on a file, you read/write data sequentially
- You can change the current position in the file

```
rewind(file);
```
go back to the beginning

offset → 0, whence → SEEK_END

```
fseek(file, 0, SEEK_END);
```
go to the end of the file

  - `whence==SEEK_SET`: move `offset` bytes after the beginning of the file
  - `whence==SEEK_CUR`: move `offset` bytes after the current position
  - `whence==SEEK_END`: move `offset` bytes after the end of the file (`offset` may be negative)

7

Special Files

- stdio, stdout, stderr

`fscanf(stdin, "%d", &i);`	read from standard input
`fprintf(stdout, "%d", i);`	write to standard output
`fprintf(stderr, "%d", i);`	write to standard error stream

- You don't have to open or close these special files
- By default, they are associated with the console
  - You can redirect them from the command line

`prog <infile.txt`	stdin redirected to infile.txt
`prog >outfile.txt`	stdout redirected to outfile.txt
`prog 2>errfile.txt`	stderr redirected to errfile.txt
`prog1 | prog2`	pipe stdout of prog1 into stdin of prog2

8

Review: Formatted Input

- You can read from stdin, from a file or from a string

```
FILE *file;
int  read;
char string[256];
read = scanf(format, vars);            read from standard input
read = fscanf(file, format, vars);     read from file
read = sscanf(string, format, vars);   read from string
```

- These functions allow you to read primitive data types (format specifiers (`%d, %u, %f`, etc.) and strings (format specifier `%s`)
  - Remember to put an & before each variable you are reading, e.g. `scanf(“%d”, &a);`
- The `Xscanf()` functions return the number of variables read
  - Return is 0: the input did not match the format provided
  - Return is EOF: the end-of-file was reached

9

Aside: Pointer Notation in C

- The `&` and `*` operators corresponds to the pointer notation in C
  - A pointer is the memory address of a variable
  - `&` and `*` are unary operators (they have a single operand)
  - `*` is used for declaring pointer variables:
    - `*file` is a pointer to a `FILE` data structure
  - `&` is used for getting a pointer to an existing variable
    - `&a` is the address of variable `a`
- Internally, C arrays are pointers
  - You may see strings declared as `char s[]` or `char *s`
  - Declaring an array of strings:

```
char *array_of_strings[];
```

- Pointer operations will be covered in ENEE 150

10

Review: Formatted Output

- You can write to stdout, to a file, or to a string

```
FILE *file;
int  read;
char s[MAX_S];
```

```
printf(format, vars);            print to standard output
fprintf(file, format, vars);     print to file
sprintf(s, format, vars);        print to string
```

- `format` uses the same specifiers as the `Xscanf` functions
  - Additionally, may specify the width and precision, e.g. `“%4.2f”`
  - Width or precision may be specified as `*`: read it from next argument
    `printf(“%.*s”, MAX_S, s);` print at most `MAX_S` chars from `s`
  - For `Xscanf`, there is no modifier like `*` for `printf`
  - For all specifiers and modifiers, see Chapter 7.2 or type `man printf`

- **With sprintf, you must be careful not to exceed the size of the string!**

Pushing Back Characters

- We’ve seen: character I/O

```
c = getc(file);          read a character from file
putc(c, file);           write a character to file
```

- Can also push a character back to the input stream

```
ungetc(c, file);         c will be returned by the next read operation
```

- The formatted I/O functions (fscanf, fprintf) are implemented using the character I/O functions
  - Ability to push back characters is needed when reading formatted numbers
  - You know that you have all the digits of the number when you read a non-digit character
  - But that character may be part of the next formatted input requested (you’ve read one character too far) => push it back to the stream

12

Status of File Streams

- File operations interact with hardware devices
  - These operations may fail
  - You must be able to distinguish between these errors and reaching EOF during normal file operations

- You can check the status of your `FILE*` stream

`FILE *file;`	
`if (ferror(file)) {…}`	check if an error occurred
`if (feof(file)) {…}`	check if you reached EOF
`rewind(file);`	rewind clears the EOF and error flags

13

Error Checking

- If you receive an error, you can print an error-specific message

```
#include <stdio.h>
FILE *file;
if ( (file=fopen("my_file.txt","r")) == NULL) {
     perror("Cannot open file");    prints a message describing the error
     exit(-1);
}
```

- `perror()` appends an error-specific message to the text provided and prints it to stderr
  - You may also print additional error messages to stderr with `fprintf(stderr, …)`

- **Good programming practice: check the return values of all the functions you invoke – an error may have occurred!**

14

Error Checking: Examples

```
#include <stdio.h>

FILE *file;
unsigned options;

if ( (file=fopen("my_file.txt","r")) == NULL) {
      perror("Cannot open file for reading");
      exit(-1);                         cannot proceed: file is not opened
}

if ( fscanf(file, "%u", &options) < 1 ) {
      fprintf(stderr, "File must start with an unsigned int");
}

printf("Read %u from the file\n", options);

if ( ferror(stdout) ) {
      perror ("Error writing to stdout");
}
```

15

Review of Lecture

- What did we learn?
  - 2D arrays
  - Opening and closing files
  - Changing position in file: `rewind`, `fseek`
  - `stdin`, `stdout`, `stderr` and redirecting program input or output
  - Review of formatted I/O
  - Error checking
- Next lecture
  - Low level file I/O
- Assignments for this week
  - Read **K&R Chapters 6.2, 6.3, 6.7, 6.8, 8.1, 8.2, 8.3, 8.4**
  - Homework: **`lab10.pdf`** (on http://ter.ps/enee140), due on Friday at 11:59 pm
  - **Quiz 9**, due on Monday at 11:59 pm
  - Project 3: **`enee140_s15_p3.pdf`** (on http://ter.ps/enee140), due on May 10 at 11:59 pm