CMSC 216
Introduction to Computer Systems
Lecture 7
Input/Output

Administrivia

- Read Reek, Chapter 15: Input/Output

Hashing

- Let’s discuss the hashing
- Using open addressing scheme with linear probing (what you need to implement for Project #2)

Chapter 15, Reek

INPUT/OUTPUT

Standard I/O Library

- **`stdio.h`** contains prototypes and constants for I/O routines
- Most I/O is stream-based and buffered to make things more efficient
  - line buffered
  - fully buffered
  - unbuffered
- Can use the function **`fflush()`** to flush buffers immediately

Types of streams

- Text streams are composed of lines of text, each terminated by a newline
  - there are library functions to deal with text streams in three ways:
    - character-oriented I/O
    - line-oriented I/O
    - formatted I/O
- Binary streams are composed of just plain data

The type `FILE *`

- Variables of type **`FILE *`** are used to represent open streams
- Three predefined streams for every program:
  - **`stdin`**: standard input (redirect with **`<`**)
  - **`stdout`**: standard output (redirect with **`>`**)
  - **`stderr`**: standard error (redirect both stdout and stderr with **`>&`** -- in tcsh only!)

Stream I/O overview

1. Declare a **`FILE *`** variable for the file
2. Use **`fopen()`** to open the file
3. Read or write (or both!)
4. Use **`fclose()`** when done

Opening files

```
FILE *fopen(char *filename, char *mode);
```

- arguments are strings
- **`mode`** specifies access mode:
  - **`"r"`** - read; file must already exist
  - **`"w"`** - write; if file exists, it is overwritten
  - **`"a"`** - append; if file does not exist, it's created
  - there are other modes …
- returns **`NULL`** on failure; can use **`perror()`** to see why it failed

Opening files, example

```
#include <stdio.h>

int main() {
  FILE *fp = fopen("infile.txt", "r");
  if (fp == NULL) {
    perror("can't open infile.txt");
    exit(EXIT_FAILURE);
  }
  /* read the file and close when done */
  return EXIT_SUCCESS;
}
```

Closing files

```
int fclose(FILE *fp);
```

- closes a file, flushing output if necessary
- returns 0 on success
- Failure does occur - the closing operation can be interrupted by the OS, or other, worse things

Line-oriented I/O

```
char *fgets(char *buf, int size,
            FILE *stream);
```

- reads chars from **`stream`**, storing in **`buf`**, stopping when either
  - a newline is read; or
  - **`size`** - 1 characters are read
- null byte is always appended to string
- Unlike gets, fgets includes the newline in buf
- **`NULL`** returned on error or **`EOF`**
- on success (no error and non-**`EOF`**), returns **`buf`**

Reading line by line

```
#include <stdio.h>

#define MAX_LEN 80

int main(int argc, char *argv[]) {
  FILE *input;
  char buffer[MAX_LEN + 1];
  input = stdin;
  if (argc > 1)
    if ((input = fopen(argv[1], "r")) == NULL) {
      perror("error opening file");
      exit(EXIT_FAILURE);
    }
  while (fgets(buffer, MAX_LEN + 1, input) != NULL) {
    /* process a line of the file */
  }
  fclose(input);
  return 0;
}
```

Working with long lines

- A line can be any length from 1 to ∞
- But working with lines of infinite length can be quite difficult …
- Some programs deal with this by repeatedly reading into a buffer until the end of a line
- Other programs pick a maximum line length and **enforce** it
- Bad C programs **assume** input lines are no longer than an arbitrary length

Formatted I/O

- Uses the **`scanf()`** and **`printf()`** family of functions
- Can operate on:
  - standard input/output
  - other file streams
  - strings

`scanf()` family

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

- reads formatted input from **`stream`**

```
int scanf(char *fmt, ...);
```

- reads formatted input from **`stdin`**

```
int sscanf(char str[], char *fmt, ...);
```

- reads formatted input from the string **`str`**

`scanf()` family

- Format strings can contain:
  - whitespace, meaning any whitespace at that point will be skipped
    - many (but not all) skip leading whitespace anyway
  - format specifiers, which cause something to be read
  - any other characters, which are then required in input
- **`scanf()`** does not check for type agreement between the format specifier and the associated variable
- If a given format specifier doesn't match, **`scanf()`** stops processing there

Reading data from strings

- Used in combination with **`fgets()`**, you can ensure that you read data line-by-line, instead of simply treating newlines as whitespace
- For what inputs would this loop stop?

```
char buf[1025];
int a, b;
while (fgets(buf, 1024, stdin) != NULL) {
  if (sscanf(buf, "%d %d", &a, &b) == 2) {
    printf("%d %d\n", a, b);
    break;
  }
}
```

The **`printf()`** family

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

- sends formatted output to **`stream`**

```
int printf(char *fmt, ...);
```

- sends formatted output to **`stdout`**

```
int sprintf(char *buf, char *fmt, ...);
```

- formats output and places in **`buf`**, adding null byte

```
int snprintf(char *buf, size_t limit,
             char *fmt, ...);
```

- writes at most **`limit`** - 1 characters to **`buf`**, then null byte

Some common format specifiers

- **`%c`** print the corresponding argument to **`printf`** as an unsigned character
- **`%d`** print as a decimal integer
- **`%u`** print as an unsigned integer
- **`%x`** print in hexadecimal (use **`%X`** for capital A-F)
- **`%f`** print in floating point format
- **`%e`** print in exponential form (e.g., 6.02300e3)
- **`%s`** print as a string (null-terminated character array)
- **`%%`** print a `%` (so `%%` prints as `%`)

Controlling formatting

- We can supply a field width, precision, and other flags to format our output exactly as we want
  - **`%04x`** : format as unsigned hex number, with 4 spaces and zero padding
  - **`%-10s`** : format as string, allot 10 spaces, left justify (default is right justify)
  - **`%6.4f`** : format as floating point, allot 6 spaces, 4 digits after the decimal point
- Much more detail available in §15.10.3