ECS 36A, May 3, 2023

Announcements

Because of the stabbings in Davis:

1. I will switch to remote classes until at least Monday
2. I will also hold office hours over Zoom
   - Same Zoom link for both; see the Announcement from this morning; link is:
     https://ucdavis.zoom.us/j/95840281592?pwd=a1NhNmpLNFp2VVVrYkpGY3pDcWdlQT09
3. I am postponing the midterm until *next* Friday, May 12
   1. Depending on what happens between now and then, I may delay it further; *don't count on it so be ready to take it in person on Friday, May 12!*

C and Files

- Files represented by a *file pointer*
  - Note: the actual representation in Linux is a *file descriptor*, which is a non-negative integer, but that is non-portable; the file pointer is
  - 3 predefined file pointers: stdin, stdout, stderr
- File pointer contains information:
  - Which file is being referenced (ie, the file descriptor)
  - Whether opened for reading, writing, or appending
  - Where in the file the next access is to occur
  - And lots of other information not relevant here

Predefined File Pointers

- *stdin*: reads from the standard input
  - Usually a terminal or terminal emulator
  - Sometimes from a file, if input is redirected (use “<“ in the shell)
  - Sometimes it is the output of another program (use “|”, like “ps | more”)
- *stdout*: writes to the standard output
  - Usually to a terminal or terminal emulator
  - Sometimes to a file, if output is redirected (use “>” in the shell)
  - Sometimes it is the input to another program (use “|”, like “ps | more”)
- *stderr*: writes to the error output
  - Usually to a terminal or terminal emulator, *even if output is redirected*
  - Sometimes to a file, if output is redirected (use “2>” in the shell)

Accessing a file

- To open a file:

```
FILE *fp;
if ((fp = fopen(filename, mode)) == NULL)
        error handling
```

- Mode is one of the following:
  - "r" read
  - "w" write – erases file if it exists, creates it if it doesn't
  - "a" append – adds to end of file if it exists, creates it if it doesn't
- All references to the contents and the file itself (usually) use the file descriptor

When You’re Done With the File

- Closing the file releases all resources associated with the process and the file
- To close a file:

```
result = fclose(fp)
```

- result is 0 if closed successfully, and EOF (–1) if not
- In either case, do *not* refer to that file pointer again!
  - If you do, the results are undefined
- Usual call:

```
(void) fclose(fp)
```

Error Handling

- Error codes are stored in a global variable *int errno*
  - Include the header file “errno.h”
- To interpret the error code:
- First way: print corresponding system error message

```
perror(str)
```

- Second way: get a pointer to the corresponding system error message

```
char *strerror(errno)
```

- Third way: access the system error message array

```
char *sys_errlist[errno]
```

Reading Text Files

- Read a line:

```
if (fgets(buf, n, fp) == NULL)
        EOF and error handling
```

  - On success, returns line (or maximum n-1 characters) in buf
  - On EOF or error, returns NULL
- Read a character:

```
if ((ch = fgetc(fp)) == EOF)
        EOF and error handling
```

  - On EOF or error, returns NULL
  - On success, returns ch as an unsigned char cast to an int

Reading Text Files

- Other ways to read characters:
  - `getc(fp)` same as `fgetc(fp)`
  - `getchar()` same as `getc(stdin)`
- Unread characters (really, push them back into the input stream):

```
if (ungetc(ch, fp) == EOF)
        handle error
```

  - Important: only 1 character of pushback is guaranteed
  - If more than that are pushed back (bad idea), they will be read in reverse order of pushback

Writing Text Files

- Write a line:

```
if (fputs(buf, fp) == EOF)
        handle error
```

  - On success, returns a non-negative integer (it does not append a newline)
  - On error, returns EOF
- Write a character:

```
if (fputc(ch, fp) == EOF)
        error handling
```

  - On error, returns EOF
  - On success, returns ch as an unsigned char cast to an int

Writing Text Files

- Other ways to write lines:

```
if (puts(buf) == EOF)
        handle error
```

  - On success, writes buf’s contents followed by a newline and returns a non-negative integer
  - On error, returns EOF
- Other ways to write characters:
  - `putc(ch, fp)` same as `fputc(ch, fp)`
  - `putchar(ch)` same as `putc(ch, stdout)`

Formatted Read

These read input or a string, and attempt to match the input with the format. If any match the desired input. the appropriate variable is set. It stops when it reads the first character that does not match the format.

- scanf (*format*, *variables*)
  - Same as fscanf(stdin, *format*, *variables*)
- sscanf(*string*, *format*, *variables*)
  - Reads from the given *string*
- fscanf(*fp*, *format*, *variables*)
  - Reads from the file with file pointer *fp*

The Formats

A string with the following format indicators:

- One or more whitespace characters: matches 0 or more whitespaces in the input
- Ordinary character (except whitespace or %): matches the next character
- Conversion specification: begins with % (to match a % in the input, say %%)

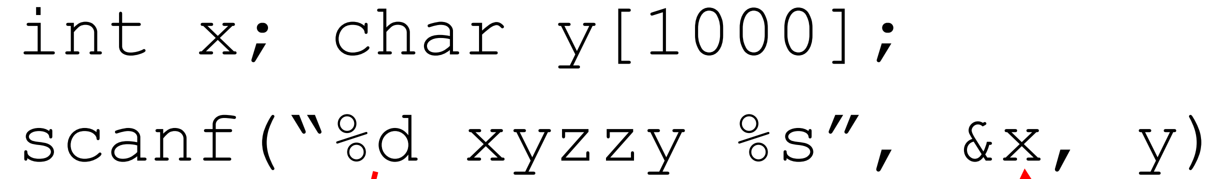
Conversion Specifiers

- Integer specification
  - ”%d”: next comes a decimal integer
  - “%o”: next comes an octal integer
  - “%x”: next comes a hexadecimal integer
  - ”%f”, “%e”, “%g”: next comes a floating point number
  - “%c”: next comes a single character
  - “%s”: next comes a string (a sequence of characters without whitespace)
- Examples:
  - “(%d,%d)” matches “(5,6)” but not “(5 6)”
  - “%s %s” matches “hello<tab>there” but not “hellothere”

Variables

- These are listed in the order of the conversion specifiers
- All must be pointers (addresses)

Examples:



```
int x; char y[1000];
scanf(“%d xyzzy %s”, &x, y)
```

Formatted Write

These write to a file or the standard output as indicated by the format string.

- printf (*format*, *variables*)
  - Same as fprintf(stdout, *format*, *variables*)
- sprintf(*string*, *format*, *variables*)
  - Writes its output into *string*
- fprintf(*fp*, *format*, *variables*)
  - Writes to the file with file pointer *fp*

The Formats

A string with the following format indicators:

- Anything except the conversion specifiers is copied
- Conversion specification: begins with %; “%%” outputs a single “%”
    - ”%d”: print next argument as a decimal integer
    - “%o”: print next argument as an octal integer
    - “%x”: print next argument as a hexadecimal integer
    - “%c”: print next argument as a single character
    - “%s”: print next argument as a string (sequence of characters)
    - “%f”: print next argument as a floating point number, like 12.345678
    - “%e”: print next argument as a floating point number, line 1.2345678e1

Format Modifiers

- %#x, %#o: put “0x” in front of hexadecimal output, 0 in front of octal output
- %+d: always print a sign, either “+” or “-”
- %*n*d: print integer in a field of *n* characters
- %0*n*d: as above, but pad with leading 0s
- %-*n*d: left align the number in the field

If the number is bigger than the field, the field size is ignored

More Format Modifiers

For floating-point numbers:

- %*n*.*m*f: print float in a field of size *n*, with *m* decimal digits
  - If *m* is 0, then no decimal point or decimal fraction is printed
  - Printed value is in normal floating point form, like 123.4567
- %*n*.*m*e: print float in a field of size *n*, with *m* decimal digits
  - If *m* is 0, then no decimal point or decimal fraction is printed
  - Printed value is in scientific notation, like 1.234567e2

Still More Format Modifiers

For characters:

- %12.4c: this ignores the .4 and prints the character in a field of 12 wide
- %012c: as above, but ignores the 0

For strings:

- %12.4s: prints first 4 characters of string in a field 12 wide
- %-12.4s: left justifies the above
- %012s: this ignores the 0