Input/Output:
Advanced Concepts

CSE 130: Introduction to Programming in C
Stony Brook University

Related reading: Kelley/Pohl 1.9, 11.1–11.7
Recall that `printf()` employs a control string that may contain  *conversion specifications* (AKA formats).

- Formats are replaced by specific values when the output is ultimately generated at run-time.
- Formats begin with the prefix character `%`.
- Formats end with a *conversion character* that indicates the type of value being substituted into the output.
Between the `%` and the conversion character, a format may contain (in order):

- Zero or more flags
- An optional minimum field width (a positive integer)
- Precede the field width with 0 to zero-pad the output
- An optional precision (a `. followed by a nonnegative integer)
- An optional “h” (short) or “l” (long) modifier for integral types
- An optional “L” (long) modifier for float/double types
Flag Options

- Minus sign ("-"'): the argument should be left-aligned in its field
- Plus sign ("+"'): non-negative signed values should begin with a +
- Space (" "): non-negative signed values should begin with a space
- Hash ("#"'): prints the result in an alternate form based on the conversion character
  - "%#o" prepends a 0 to octal values
  - "%#x" prepends 0X to hexadecimal values
- Zero ("0"): pads the field with leading zeros
```c
int i = 123;
double x = 0.123456789;
```

<table>
<thead>
<tr>
<th>Format</th>
<th>Argument</th>
<th>Actual Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>i</td>
<td>“123”</td>
<td>(default) width 3</td>
</tr>
<tr>
<td>%05d</td>
<td>i</td>
<td>“00123”</td>
<td>zero-padded</td>
</tr>
<tr>
<td>%7o</td>
<td>i</td>
<td>“173”</td>
<td>right adjusted octal</td>
</tr>
<tr>
<td>%-9x</td>
<td>i</td>
<td>“7b”</td>
<td>left adjusted hex</td>
</tr>
<tr>
<td>%-#9x</td>
<td>i</td>
<td>“0x7b”</td>
<td>left adjusted alt. hex</td>
</tr>
<tr>
<td>%10.5f</td>
<td>x</td>
<td>“0.12346”</td>
<td>width 10, precision 5</td>
</tr>
<tr>
<td>%-12.5e</td>
<td>x</td>
<td>“1.23457e-01”</td>
<td>left adjusted e-format</td>
</tr>
</tbody>
</table>
Special Strings and `scanf()`

- A `scanf()` conversion specification of the form `%[ ... ]` means that a special string is to be read in.
- If the first character inside the brackets is `^`, the string may not contain any of the other bracketed characters.
- If the first bracketed character is NOT `^`, the string may only contain the other bracketed characters.
- e.g., `scanf("%[AB \n\t ]", s);` will read in a string that only contains As, Bs, spaces, newlines, and tabs.
Files provide stable storage for a program

- They can be used to hold data between invocations, so that it does not need to be re-entered the next time the program runs

File processing (reading and writing data) is similar to console I/O in C

- Use fprintf() and fscanf(), two variants of the I/O functions we already know
Start by creating a pointer to a FILE structure (defined in stdio.h):

```c
FILE *infile;
```

The `fopen()` function opens the specified file and returns a pointer to FILE:

```c
infile = fopen("my_file.txt", "r");
```
The `fopen()` Command

- `fopen()` takes two string arguments: the name of the file (including its path) and the opening mode

- There are three opening modes:
  - "r" opens a file to read from it
  - "w" opens a file to (destructively) write to it
    - If the file does not exist, "w" mode creates it
  - "a" opens a file to append to its contents
    - Use "r+" or "w+" to read and write to the same file

- If `fopen()` fails to open the file, it returns `NULL`
Reading From Files

- `getc()` reads one character at a time (like `getchar()`)
- `getc()` takes a file pointer as its argument
- `getc()` returns `EOF` (end-of-file) when there are no more characters to read

- `fscanf()` works like `scanf()` for more elaborate input
  - It takes the file pointer as its first argument
  - e.g., `fscanf(infile, "%c %5d", &letter, &code);`
Writing To Files

❖ `putc()` writes one character to a file stream (like `put()`)
❖ `putc()` takes a `char` and a file pointer as its arguments
❖ `putc()` returns `EOF` (end-of-file) on failure

❖ `fprintf()` works like `printf()` for more elaborate output
❖ It takes the file pointer as its first argument
❖ e.g., `fprintf(outfile, "%d %s\n", n, message);`
When You’re Done...

- When a C program completes, all open files are closed automatically.
- C limits the number of files that a program can have open at one time (usually to 20 or 64 files).
- If you’re working with a lot of files, you may need to close some of them manually.
- Do this with the `fclose()` function.
Random File Access

- Files are normally read from (or written to) sequentially.
- We can move the file position indicator as we wish, though.

- `ftell(file_ptr)` returns the current value of the file position indicator.
- This value is the number of bytes from the beginning of the file, counting from 0.
Moving The File Position Indicator

- Use `fseek()` to relocate the file position indicator

Syntax: `fseek(file_ptr, offset, place);`

- This moves the file position indicator `offset` bytes from `place`
  - `place` can be 0 (file beginning), 1 (current location), or 2 (file end)

- Note that this is only guaranteed to work correctly with binary files (so add “b” to the mode, e.g., “rb”)
Example: Printing File Contents in Reverse Order

```c
FILE *ifp = fopen("data.txt", "rb");
fseek(ifp, 0, 2);    /* go to end of file */
fseek(ifp, -1, 1);   /* back up 1 position */

while (ftell(ifp) > 0)
{
    int c = getc(ifp); /* moves ahead 1 space */
    putchar(c);
    fseek(ifp, -2, 1); /* back up 2 positions */
}

fclose(ifp);
```
**`sprintf()` and `sscanf()`**

- These functions write to, and read from, strings (variables of type `char *`) rather than the console or a file.
- Their first argument must be of type `char *`.

```c
sscanf("1 2 3 go", "%d%d%d%s", &a, &b, &c, tmp);
```
- Note that repeated calls to `sscanf()` restart at the beginning of the source string.