Standard streams

We've seen I/O functions that work *only* with `stdin`/`stdout`

- `printf`, `scanf`

Others work with any file, including named files

- `fprintf`, `fscanf`

C file I/O

`fopen("output.txt", "w")`

- Open file `"output.txt"` in writing mode (`"w"`)

Possible modes:

- `"r"`: reading
- `"w"`: open file for writing
- `"r+"`: open for reading & writing
- `"w+"`: open file for reading & writing

`"r"` or `"w"` are common

Note: `"w"` and `"w+"` cause the named file to be overwritten if it already exists

C file I/O

`fopen` returns a `FILE*`, a pointer to a `FILE struct`

- We'll return to `struct`s and pointers later

Equals `NULL` if `fopen` failed

- Always check, since reading or writing `NULL` causes a crash
- `NULL` is a special pointer value, usually equal to 0; common way to indicate failure for functions with pointer return type

C file I/O

- `feof(fileptr)` returns non-zero if we've already read past the end of the file
- `ferror(fileptr)` returns non-zero if file is in an error state, e.g. if we've opened file for writing but then attempt a read
- `rewind(fileptr)` returns `fileptr` to beginning of file

C file I/O

```
numbers.txt:
10 20
3 50
100 -100
400 -800
```

C file I/O

making sure there was no malformed line in the file (e.g., a line with one in value only)

```
file_io_loop_eg.c:
#include <stdio.h>
int main() {
    FILE* input = fopen("numbers.txt", "r");
    if (input == NULL) {
        printf("Error: could not open input file\n");
        return 1; // indicate error
    }

    int a = 0, b = 0;
    int numCollected = fscanf(input, "%d%d", &a, &b);
    while (numCollected == 2) {
        printf("%d\n", a+b);
        numCollected = fscanf(input, "%d%d", &a, &b); }
    }
```

```
    if (ferror(input)) {
        printf("Error: error indicator ");
        printf("was set for input file\n");
        return 2; // indicate error
    } else if (numCollected != EOF) {
         printf("Error: could not parse line\n");
         return 3; // indicate error
    }

    fclose(input); // Close input file
    return 0; // no error
```

C file I/O

```
$ gcc file_io_loop_eg.c -std=c99 -pedantic -Wall -Wextra
$ cat numbers.txt
10 20
3 50
100 -100
400 -800
$ ./a.out
30
53
0
-400
```

C file I/O

We saw that `printf` and `scanf` use the *standard streams*

You can refer to them by these names, defined in `stdio.h`

- `stdin`
- `stdout`
- `stderr`

You don't have to open or close them; C handles that

For example, `fprintf` can write to `stdout` like `printf`:

$$\underbrace{\texttt{fprintf(stdout, "Hello, World\textbackslash n");}}_{\text{same as printf("Hello, World\textbackslash n");}}$$

Assertions

```
assert(boolean expression);
```

- Assertion statements help catch bugs as close to the source as possible
  - Require `#include <assert.h>`
  - `boolean expression` is an expression that *should be true* if everything is OK
  - If it's false, program immediately exits with an error message indicating the assertion failed
- You will create *test cases* using `assert`

Assertions

Assertions can help to make your assumptions clear

```c
int sum = a*a + b*b;
assert(sum >= 0);
```

```c
if(isalpha(c)) {
    assert(c >= 'A');
    printf("%d\n", c - 'A');
}
```

Assertions

`assert` is not for typical error checking

```
FILE* input = fopen("numbers.txt", "r");
if(input == NULL) {
    printf("Error: could not open input file\n");
    return 1; // indicate error
}
```

making sure the file was opened successfully

If checking for bad user input, or another strange but not impossible situation, use `if` and print a meaningful message. If you must exit, return non-zero to indicate failure.

If you're checking for something that implies that your program is incorrect, use `assert`

Assertions

```
assert_eg.c:
#include <stdio.h>
#include <assert.h>

int main() {
    int n = 0;
    scanf("%d", &n);
    if(n == 0) {
        printf("n must not be 0\n");
        return 1;
    }

    int n_sq = n * n;
    assert(n_sq >= n); // if false, something's wrong

    float n_inv = 1.0 / n;
    printf("squared=%d, inverse=%0.2f\n", n_sq, n_inv);
    return 0;
}
```

Assertions

```
$ gcc assert_eg.c -std=c99 -pedantic -Wall -Wextra
$ echo 4 | ./a.out
squared=16, inverse=0.25

$ echo -2 | ./a.out
squared=4, inverse=-0.50

$ echo 0 | ./a.out
n must not be 0

$ echo 200000000 | ./a.out
Assertion failed: (n_sq >= n), function main,
    file assert_eg.c, line 12.
```

The last run fails due to overflow of `int`!

Math library

`#include math.h` **and** compile with `-lm` option

- `sqrt(x)`: square root
- `pow(x, y)`: x^y
- `exp(x)`: e^x
- `log(x)`: natural log
- `log10(x)`: log base 10
- `ceil(x) / floor(x)`: round up / down to nearest integer
- `sin(x)`: sine (other trigonometric functions available)

Math library

x and *y* arguments have type `double`

It's also OK to pass another numeric type, like `int`

- Argument type promotion: `int` -> `float` -> `double`

`-lm` includes the math library when *linking*; more on this later.