Chapter 3

Input Output

```
foobar < inputfromthisfile.blah --stdout
foobar > outputtothisfile.blah --stdin
foobar 2> error.blah --stderr
```

```
printf("Hello World") == fprintf(stdout, "Hello World");
scanf(...) == fscanf(stdin,...);
```

If we have set our output to a file and we use `printf("Error, invalid blah: %f\n", blah)` for an error message it will go to our file too. Instead, we can use `fprintf(stderr, "Error, invalid blah: %f\n", blah)`. `stderr` can also be output to a file.

`stderr` and `stdout` are just data streams, there is nothing preventing you from using them for anything.

Small Tools

Small tools all follow these design principles:

- Read data from stdin
- Display data on stdout
- Deal with text rather than binary formats
- Perform **one** *simple* task

Pipes (|)

```
(./foo | ... | ./bar) < foo.blah > bar.blah
```

More data streams

`FILE *blah = fopen("file.boo", "mode")` where mode =

w = write to a file

r = read a file

a = append to the *end* of a file

```
FILE *in_file = fopen("input.blah", "r");
FILE *out_file = fopen("output.blah", "w");
```

You can then use your data stream, por examplar:

```
fprintf(out_file, "Foo the bar");
```

or

```
fscanf(in_file, "%79[^\n]\n", sentence);
```

When you are finished with the data stream, you need to close it (they are automatically closed when the program ends, but it is good practice to close it yourself) with `fclose(file)`.

It is also good practice to check for errors with a data stream. For example:

```
FILE *in = fopen("i_dont_exist.txt", "r");
```

should be written as

```
FILE *in;
if (!(in = fopen("dont_exist.txt", "r"))) {
fprintf(stderr, "Can't open the file.\n");
return 1; }
```

main()

There are two versions of main:

1. `main()`
2. `int main(int argc, char *argv[])`

In the second version, the `main()` function can read the command line as an array of strings (technically, an array of character pointers to strings).

For example:

`./foo`	`monkey`	`monkey.blah`	`frog`	`frog.blah`	`the_rest.blah`
argv[0]	argv[1]	argv[2]	argv[3]	argv[4]	argv[5]

`argc` is a count of the number of elements in the array.

Command-line options

`getopt()` returns the next option it find on the command line

```
coffee_maker -n 4 -m Java
```

Takes a value: (`-n` = quantity)

Is on or off (`-m` = milk)

These options are handled in a loop:

```
#include <unistd.h> // required (not part of standard C library)
...
while ((ch = getopt(argc, argv, "mn:")) != EOF)
switch(ch) {
...
case 'n':
cup_count = optarg;
...
}
// optind stores the number of strings read from the command line to get past the options
argc -= optind;
argv += optind;
```

A `switch` statement handles each of the valid options. `mn:` tells `getopt()` that `m` and `n` are valid options. `n` is followed by a colon (`:`) to let `getopt()` know that `m` needs to be followed by an extra argument. `getopt()` will point to that argument with the *optarg* variable.

When the loop finishes, `argv` and `argc` are tweaked to skip past the options, getting to the main command-line arguments. Therefore `argv` will look as such:

`Java`
argv[0]

Note: `getopt()` will stop reading options when it sees `--`, this can prevent ambiguity