Lecture 10 - Standard I/O (last updated: 10/20/22)

Programs rely on the operating system to communicate with their environment (i.e., other programs, your terminal, the file system etc.). These channels of communication are known broadly as I/O (input/output). In this lecture, we will discuss the mechanism by which programs communicate with you in the command line.

The C standard provides every running program with 3 I/O streams: standard input (stdin), standard output (stdout), and standard error (stderr). C programs use these communication channels to print output to the command line, and to receive typed input while they are running.

stdin, stdout, and stderr are sometimes referred to numerically as 0, 1, and 2, respectively. These are their file descriptor numbers, which we will revisit in a later lecture.

Standard output (stdout)

Most of the time, when you see a program's output on the command line, that is the text it has printed to stdout:

```
$ echo "hello"                  # writes argument "hello" to stdout
hello
```

C programs can write formatted text to stdout using the printf() function:

```
$ cat printf-test.c
#include <stdio.h>

int main(void) {
    printf("This is printed to stdout.\n");
    return 0;
}

$ gcc -o printf-test printf-test.c

$ ./printf-test
This is printed to stdout.
```

There are also other ways to write to stdout, including putchar() ("put character") and puts() ("put string"), though printf() is usually used in their place.

printf(), putchar(), and puts(), are all functions provided by the C standard library, and tell the operating system to write to stdout.

Standard input (stdin)

The "opposite" of printf() is scanf(), which reads input from stdin according to some input format:

```
$ cat scanf-test.c
#include <stdio.h>

int main(void) {
    int x;
    printf("Enter a number on the next line.\n");
    scanf("%d", &x);
    printf("Your number was %d.\n", x);
    return 0;
}

$ gcc -o scanf-test scanf-test.c

$ ./scanf-test
Enter a number on the next line.
56
Your number was 56.
```

The shell session shown here is actually a little misleading; the lines saying "Enter a number on the next line." and "Your number was 56." represent the program's stdout, printed using printf(). The line consisting of "56" is what I typed into the terminal; it is only shown here because the terminal leaves what I typed on the screen. The only characters that appeared in the program's stdin were '5', '6', and '\n' (when I hit the return key).

scanf() is not used nearly as often as printf(), since its interface can be awkward to use (the fixed format does not handle arbitrary user input well). The stdin counterparts to putchar() and puts() are getchar() and gets(); while getchar() is sometimes useful for reading in a single character from stdin, gets() should never be used (see its man page). For this course, we prefer to use the functions fgets() and fread() (we will talk about these next lecture).

By the way, do not confuse stdin with a program's arguments: the arguments are typed in BEFORE the program is executed, and are given to the program via argv; stdin is typed in WHILE the program is running, and is retrieved using I/O functions such as scanf(), getchar(), fputs(), and fread() (all provided by the C standard library).

Redirection

You can use the shell to redirect I/O streams. For example, you can redirect the stdout of echo to a file, using the ">" shell operator:

```
$ echo hello > myfile           # redirect "hello" to myfile

$ cat myfile                    # show contents of myfile
hello
```

You can do the same with printf-test from before:

```
$ ./printf-test > myfile        # redirect program's stdout to myfile

$ cat myfile                    # old contents were overwritten
This is printed to stdout.
```

If you would like to append to a file instead of overwriting its contents, you can use the ">>" shell operator:

```
$ echo "more text" >> myfile    # append program's stdout to myfile

$ cat myfile
This is printed to stdout.
more text
```

You can redirect from a file to a program's stdin, using the "<" shell operator:

```
$ echo "42" > anotherfile       # write "42" to anotherfile

$ ./scanf-test < anotherfile    # read "42" from another file
Enter a number on the next line.
Your number was 42.
```

Note that "42" does not appear in the shell session, since we didn't type it in this time.

Pipes

You can also use the shell to "pipe" (i.e., connect) the stdout of one program to the stdin of another, using the "|" operator. For example:

```
$ echo "36" | ./scanf-test      # send "36" to the stdin of scanf-test
Enter a number on the next line.
Your number was 36.
```

Pipes are really useful for automating text-processing tasks. For example, here's how I count the number of lines in all of my lecture notes (so far):

```
$ ls *.txt                      # list all the lecture notes in pwd
01-cli-basics.txt       04-bytes.txt            07-arrays.txt
02-git.txt              05-pointers.txt         08-structs.txt
03-compile.txt          06-heap.txt             09-libraries.txt

$ cat *.txt | wc -l             # count the number of lines
2483
```

Here's what's going on:

- cat *.txt: concatenates the contents of all .txt files in the pwd, and outputs those contents to stdout
- wc -l: counts the number of lines given to stdin, and outputs the final count in its stdout

You can even chain pipes together, and use them with redirection. For example, here I write each unique word in my notes to a text file named "lecture-words":

```
$ cat *.txt | tr ' ' '\n' | sort | uniq > lecture-words
```

Breaking it down (though you don't need to understand every command):

- cat *.txt: concatenates the contents of all .txt files in the pwd, and outputs those contents to stdout
- tr ' ' '\n': reads from stdin, translates each space to a newline, and writes it to stdout (i.e., puts every space-separated word on its own line)
- sort: sorts lines from stdin in alphabetical order, and outputs to stdout
- uniq > lecture-words: omit repeated lines, i.e., the repeated words from my lecture notes; stdout is redirected to a file named lecture-words

Now I can count how many unique words I have using wc -l:

```
$ wc -l < lecture-words
3189
```

Standard error (stderr)

Since stdout is often redirected or piped to another file or process, printing errors on stdout can be problematic. Thus, UNIX also provides a second output stream, stderr, separate from stdout. You can write to stderr using the standard library function fprintf(), which works exactly like printf(), except it takes an additional first argument indicating which I/O stream to print to:

```
$ cat fprintf-test.c
#include <stdio.h>

int main(int argc, char **argv) {
    if (argc < 2)
        fprintf(stderr, "Warning: no arguments given.\n");
    else
        fprintf(stderr, "%d arguments given.\n", argc - 1);

    for (int i = 1; i < argc; i++)
        fprintf(stdout, "%s\n", argv[i]);
        // ^same as writing printf("%s\n", argv[i]);

    return 0;
}

$ gcc -o fprintf-test fprintf-test.c

$ ./fprintf-test
Warning: no arguments given.

$ ./fprintf-test hello world
2 arguments given.
hello
world
```

It looks as if "Warning: no arguments given." and "2 arguments given." are printed to stdout, but they are actually printed to stderr. This distinction becomes clear when we redirect stdout (but not stderr):

```
$ ./fprintf-test hello world > myfile
2 arguments given.
```

If we would like to redirect stderr, we can use "2>" (2 is the file descriptor number for stderr):

```
$ ./fprintf-test 2> myfile

$ cat myfile
Warning: no arguments given.
```

We can also redirect stderr to stdout by writing "2>&1" (redirect stderr, AKA file descriptor 2, to wherever stdout is going, AKA file descriptor 1):

```
$ ./fprintf-test hello world 2>&1
2 arguments given.
hello
world
```

A quirk of shell redirection syntax: when redirecting both stderr and stdout to a file, order matters. For example, this works as expected:

```
$ ./fprintf-test hello world > myfile 2>&1

$ cat myfile
2 arguments given.
hello
world
```

But not this:

```
$ ./fprintf-test hello world 2>&1 > myfile
2 arguments given.

$ cat myfile
hello
world
```

By the way, you may recall that when we ask you to append your Valgrind output to your README, we instruct you to run:

```
$ valgrind --leak-check=yes ./myprogram args >> README.txt 2>&1
```

Hopefully, the ">> README.txt" and "2>&1" parts make a little more sense now. We need to specify "2>&1" because Valgrind writes its memory leak/error-checking output to stderr rather than stdout.