Reverse

This project is a simple warm-up to get you used to how this whole project thing will go. It also serves to get you into the mindset of a C programmer. You will write a simple program called `reverse`. This program should be invoked in one of the following ways:

```
prompt> ./reverse
prompt> ./reverse input.txt
prompt> ./reverse input.txt output.txt
```

The above line means the users typed in the name of the reversing program `reverse` (the `./` in front of it simply refers to the current working directory (called dot, referred to as `.`) and the slash (`/`) is a separator; thus, in this directory, look for a program named `reverse`) and gave it either no command-line arguments, one command-line argument (an input file, `input.txt`), or two command-line arguments (an input file and an output file `output.txt`).

An input file might look like this:

```
hello
this
is
a file
```

The goal of the reversing program is to read in the data from the specified input file and reverse it; thus, the lines should be printed out in the reverse order of the input stream. Thus, for the aforementioned example, the output should be:

```
a file
is
this
hello
```

The different ways to invoke the file (as above) all correspond to slightly different ways of using this simple new Unix utility. For example, when invoked with two command-line arguments, the program should read from the input file the user supplies and write the reversed version of said file to the output file the user supplies.

When invoked with just one command-line argument, the user supplies the input file, but the file should be printed to the screen. In Unix-based systems, printing to the screen is the same as writing to a special file known as **standard output**, or `stdout` for short.

Finally, when invoked without any arguments, your reversing program should read from **standard input** (`stdin`), which is the input that a user types in, and write to standard output (i.e., the screen).

Sounds easy, right? It should. But there are a few details...

Details

Assumptions and Errors

- **Input is the same as output:** If the input file and output file are the same file, you should print out an error message "Input and output file must differ" and exit with return code 1.
- **String length:** You may not assume anything about how long a line should be. Thus, you may have to read in a very long input line...
- **File length:** You may not assume anything about the length of the file, i.e., it may be **VERY** long.
- **Invalid files:** If the user specifies an input file or output file, and for some reason, when you try to open said file (e.g., `input.txt`) and fail, you should print out the following exact error message: `error: cannot open file 'input.txt'` and then exit with return code 1 (i.e., call `exit(1);`).

- **Malloc fails:** If you call `malloc()` to allocate some memory, and malloc fails, you should print the error message `malloc failed` and exit with return code 1.
- **Too many arguments passed to program:** If the user runs `reverse` with too many arguments, print `usage: reverse <input> <output>` and exit with return code 1.
- **How to print error messages:** On any error, you should print the error to the screen using `fprintf()`, and send the error message to `stderr` (standard error) and not `stdout` (standard output). This is accomplished in your C code as follows: `fprintf(stderr, "whatever the error message is\n");`

Useful Routines

To exit, call `exit(1)`. The number you pass to `exit()`, in this case 1, is then available to the user to see if the program returned an error (i.e., return a non-zero) or exited cleanly (i.e., returned 0).

For reading in the input file, the following routines will make your life easy: `fopen()`, `getline()`, and `fclose()`.

For printing (to screen, or to a file), use `fprintf()`. Note that it is easy to write to standard output by passing `stdout` to `fprintf()`; it is also easy to write to a file by passing in the `FILE *` returned by `fopen`, e.g., `fp=fopen(...); fprintf(fp, ...);`.

The routine `malloc()` is useful for memory allocation. Perhaps for adding elements to a list?

If you don't know how to use these functions, use the man pages. For example, typing `man malloc` at the command line will give you a lot of information on malloc.

Tips

Start small, and get things working incrementally. For example, first get a program that simply reads in the input file, one line at a time, and prints out what it reads in. Then, slowly add features and test them as you go.

For example, the way we wrote this code was first to write some code that used `fopen()`, `getline()`, and `fclose()` to read an input file and print it out. Then, we wrote code to store each input line into a linked list and made sure that worked. Then, we printed out the list in reverse order. Then we made sure to handle error cases. And so forth...

Testing is critical. A great programmer we once knew said you have to write five to ten lines of test code for every line of code you produce; testing your code to make sure it works is crucial. Write tests to see if your code handles all the cases you think it should. Be as comprehensive as you can be. Of course, when grading your projects, we will be. Thus, it is better if you find your bugs first, before we do.

Keep old versions around. Keep copies of older versions of your program around, as you may introduce bugs and not be able to easily undo them. A simple way to do this is to keep copies around, by explicitly making copies of the file at various points during development. For example, let's say you get a simple version of `reverse.c` working (say, that just reads in the file); type `cp reverse.c reverse.v1.c` to make a copy into the file `reverse.v1.c`. More sophisticated developers use version control systems git (perhaps through github); such a tool is well worth learning, so do it!