1 Files

1.1 File functions

- **Opening Files** : The function `fopen` opens a file and returns a FILE pointer.

  ```
  FILE *fopen( const char * filename, const char * mode );
  ```

  The allowed modes for `fopen` are as follows

r	–	Reading only. Start at beginning of file.
$r+$	–	Reading and writing. Note that all writing is overwriting, not inserting.
w	–	Writing only. In other words, deletes contents of file (sets to 0 length). Cannot read previously written contents of file.
$w+$	–	open for reading and writing (overwrite file)
a	–	Appending. Cannot read contents of file.
$a+$	–	Reading and Appending. Can read, but all writes are done at the end of the file regardless of calls to fseek or similar.

  To open a file in a binary mode you must add a b to the end of the mode string; for example, "rb" (for the reading and writing modes, you can add the b either after the plus sign - "r+b" - or before - "rb+")

- **Closing a File** : The function `fclose` returns zero on success, or EOF if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file stdio.h.

  ```
  int fclose( FILE *fp );
  ```

- **Writing a File** :

  – `int fputc( int c, FILE *fp );`
  The function `fputc` writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error. You can use the following functions to write a null-terminated string to a stream:

  – `int fputs( const char *s, FILE *fp );`
  The function `fputs` writes the string `s` to the output stream referenced by `fp`. It returns a non-negative value on success, otherwise EOF is returned in case of any error.

- **Reading a File** :

  – `int fgetc( FILE * fp );`
  The `fgetc` function reads a character from the input file referenced by `fp`. The return value is the character read, or in case of any error it returns EOF. The following functions allow you to read a string from a stream:

  – `char *fgets( char *buf, int n, FILE *fp );`
  The functions `fgets` reads up to $n-1$ characters from the input stream referenced by `fp`. It copies the read string into the buffer buf, appending a null character to terminate the string.
  If this function encounters a newline character or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character.

- **Testing the End of a File** : `int feof(FILE *stream)` returns a non-zero value when End-of-File indicator associated with the stream is set, else zero is returned.
- **Flushing the Output Buffer of a Stream** : `int fflush(FILE *stream)`
  The function `fflush` is used to flush an output stream or an update stream in which the most recent operation was not input. If strean is an input stream, the behavior is undefined.

1.2 Standard Streams

In the header file `stdio.h`, `stderr`, `stdin`, and `stdout` are defined as macros. They are pointers to FILE types which correspond to the standard error, standard input, and standard output streams respectively.
All file pointer arguments to the functions in Section 1.1 are considered streams.

2 Piping

In Unix, one can use the standard output of a command as the input stream of another command. Such a feature is called piping. The symbol | is the Unix pipe symbol used on the command line.

- **Using a Pipe**
  Recall that we can display the content of a file using the command `cat`:

```
$ cat tobe.txt
To be
or not to be
that is the question
```

  Instead of displaying the content of the file, the following commands send it through a pipe to the wc (word count) command.

```
$ cat tobe.txt | wc
       3      10      42
$ cat tobe.txt | wc -l
       3
$ cat tobe.txt | wc -w
      10
```

  Note that `cat` displays a file to `stdout` without pause. To view text page by page for a large file or several files, we can use pipe:

```
$ cat tobe.txt EliotLoveSong.txt | more
```

  In the above command, `cat` concatenate files `tobe.txt` and `EliotLoveSong.txt`, and through pipe, the stdout stream of cat becomes the input stream of the program `more` which is used for paging through text one screen at a time.

- **Using sed to manipulate the text of a stream.**

```
$cat tobe.txt
to be or not to be
that is the question
to be or not to be
that is the question

$ cat tobe.txt | sed 's/to/not to/'
not to be or not to be
that is the question
not to be or not to be
that is the question

$ cat tobe.txt
to be or not to be
that is the question
to be or not to be
that is the question
```

In the above example, at the first shell promt, the content of the file tobe.txt was shown.
At the second promt, the `cat` command to display the contents of the tobe.txt file, and send that display through a pipe to the sed command. The `sed` command took the input that it got through the pipe, changed the *first occurrence* of the word "to" on each line to "not to", and then displayed its output to the screen.
Note that, from another `cat` command, the original file was not changed.

To change all occurrences of "to" to "not to", we use the flag "g" as follows:

```
$ cat tobe.txt | sed 's/to/not to/g'
not to be or not not to be
that is the question
not to be or not not to be
that is the question
```

- **Using egrep to search for strings in a file.** To find the lines that contains the string "question" and display those lines to the standard output:

```
$cat tobe.txt
to be or not to be
that is the question
to be or not to be
that is the question

$ egrep question tobe.txt
that is the question
that is the question
```

3 Redirection

The shell and many Unix commands take input from `stdin`, write output to `stdout`, and write error output to `stderr` where standard input is normally connected to the terminal keyboard and standard

output and error to the terminal screen. Redirection allows you to use a file in place of any of these three streams locations.

The form of a command with standard input and output redirection is:

```
$ command -[options] [arguments] < input file  > output file
```

In the following forms of redirection, file descriptor 0 is normally standard input (`stdin`), 1 is standard output (`stdout`), and 2 is standard error output (`stderr`).

Command	**Description**
pgm > file	Output of pgm is redirected to file
pgm < file	Program pgm reads its input from file.
pgm ≫ file	Output of pgm is appended to file.
$n >$ file	Output from stream with descriptor n redirected to file.
$n \gg$ file	Output from stream with descriptor n appended to file.
$n >\& m$	Merge output from stream n with stream m.
$n <\& m$	Merge input from stream n with stream m.
≪ tag	Standard input comes from here through next tag at start of line.
|	Pipe: takes output from one program, or process, and sends it to another.

Examples:

- `$ wc tobe.c`
- `$ wc < tobe.c`
- `$ cat tobe.c | wc`
- `$ ls -l >ls.txt`
- `$ program-name 2> error.log`

  redirects the standard error stream to a file.
- Redirecting the standard error (stderr) and stdout to file

```
$ cat tobe.txt blah >error.txt 2>&1
$ cat error.txt
to be or not to be
that is the question
to be or not to be
that is the question

cat: blah: No such file or directory
```

- Redirecting the standard error to stdout

```
$ ls -l blah 2>&1
ls: blah: No such file or directory
$ ls -l blah | wc -l
ls: blah: No such file or directory
       0
```

- Discarding the output:
  Sometimes you will need to execute a command, but you don't want the output displayed to the screen. In such cases you can discard the output by redirecting it to the file `/dev/null`:

  ```
  $ command > /dev/null
  ```

  discards the standard output where `command` is the name of the command you want to execute and the file `/dev/null` is a special file that automatically discards all its input. For example,

  ```
  $ cat tobe.txt blah
  to be or not to be
  that is the question
  to be or not to be
  that is the question

  cat: blah: No such file or directory
  $ cat tobe.txt blah >/dev/null
  cat: blah: No such file or directory

  $ cat tobe.txt blah >/dev/null 2>&1
  ```

  At the last shell prompt, both output of a command and its error output are discarded.

  ```
  $ cat tobe.txt blah 2> /dev/null
  to be or not to be
  that is the question
  to be or not to be
  that is the question
  ```

  Here, only its error output is discarded.

4 Exercise

- Write a C program that prints something to stdout and stderr.
- Get familiar with pipe.
- Redirect program out:
  - `stdout` pipe to `wc -l`
  - `stdout` only to file
  - `stderr` only to file
  - `stderr` to `stdout` then pipe to `wc -l`
  - all output to file