Chapter 22

Input/Output

C's Input/Output Library

- `C`'s input/output library is the biggest and most important part of the standard library
- So far, we gain some experience with such functions, e.g.,
  - `printf`, and `scanf`,
  - `putchar`, and `getchar`,
  - `puts`, and `gets`
- This chapter:
  - provides more information about these functions
  - introduces new functions, most of which deal with files
- Many of the new functions are closely related to functions with which we are already familiar with, for instance, `fprintf` function is the "*file version*" of the `printf` function

File Pointers and Streams

- In `C`, the term *stream* means:
  - any *source of input*, or
  - any *destination for output*

- Many small programs:
  - obtain all their input from one stream (usually associated with the keyboard) and
  - write all their output to another stream (usually associated with the screen)

- Larger programs may need additional streams

File Pointers and Streams

- Accessing a stream in a `C` program is done through a *file pointer*
- A file pointer is a pointer to a `FILE` *structure*
- A program may declare as many file pointers as needed:

```
FILE *fp1, *fp2;
```

- `<stdio.h>` header file contains:
  - A definition of `FILE` structure and
  - Declarations of all functions that perform file input/output

File Pointers and Streams

- There are three standard streams
  - `stdin` standard input
  - `stdout` standard output
  - `stderr` standard error

  These streams need ***not*** to be opened or closed by the program
- By default,
  - `stdin` represents the keyboard
  - `stdout` and `stderr` represent the screen
- Many operating systems allow these *default meanings* to be *changed* via a mechanism known as ***redirection***

Standard Input, Output, and Error Redirection

- In Unix, standard input, output, and error redirection method vary from one shell to another
  - In **sh** shell:
    - To redirect standard input, use "<0" or simply "<"
    - To redirect standard output, use "1>" or simply ">"
    - To redirect standard error, use "2>"
    - To redirect and append standard output, use "1>>" or simply ">>"
    - To redirect and append standard error, use "2>>"
    - To redirect standard input from the text to be written immediately after the command, use "<<"

    Example: Standard input, output, and error are redirected

    ```
    prog < data > result 2> error
    ```

    Example: Standard error and output are redirected to the same file

    ```
    prog < data > result_and_error 2>&1
    ```

Standard Input, Output, and Error Redirection

– In **csh** shell:
  - To redirect standard input, use "<"
  - To redirect standard output, use ">"
  - To redirect both standard output and error, use ">&"
  - To redirect and append standard output, use ">>"
  - To redirect and append both standard output and error, use ">>&"
  - To redirect standard input from the text to be written immediately after the command, use "<<"

Examples:
```
prog < data > result
```
In this case, standard error is displayed on screen
```
prog < data >& result_and_error
```
In this case, nothing will be displayed on screen
```
(prog < data > result) >& error
```
This is a trick to separate standard output from stander error

Standard Input, Output, and Error Redirection

- Simplicity is one of the attractions of input and output redirection
- Unfortunately, *redirection is too limited* for many applications
  – When a program relies on redirection, it has no control over its files; it doesn't even know their names
  – Redirection doesn't help if the program needs to
    - read from two files, or more, at the same time
    - write to two files, or more, at the same time
- When redirection isn't enough, we'll use the file operations that `<stdio.h>` provides

File Buffering

- Transferring data to, or from, a disk drive is a relatively slow operation
- Hence, it is not feasible for a program to access a disk file directly each time it wants to read or write a byte
- The secret to achieve acceptable performance is *buffering*
  - Data written to a stream is actually stored in a buffer area in memory
  - When it is *full*, or the *stream is closed*, the buffer is *flushed*
- Input streams can be buffered in a similar way
  - The buffer contains data from the input device
  - Input is read from this buffer instead of the device itself

File Buffering

- The buffering takes place behind the scenes, and we usually do not worry about it
- Yet, in some occasion, we may need to take a more active role

- By calling `fflush(FILE *fp)`, a program can *flush* a file's buffer as often as it wishes
- The call

```
fflush(fp)      /* flushes buffer for fp */
fflush(NULL)    /* flushes all buffers */
```

- `fflush` returns `zero` if it is successful and `EOF` otherwise

File Buffering

- If you will direct both standard output and standard error to the *same file*, it is very important to
  - `fflush(stderr)` and `fflush(stdout)` after each `printf`, otherwise the order of both output will be unpredictable

```
fflush(stderr) /* flushes buffer for stderr */
fflush(stdout) /* flushes buffer for stdout */
```


File Buffering

- Example

```
#include <stdio.h>
main(void)
{ fprintf(stdout, "1 ");    fflush(stdout);
  fprintf(stderr, "2 ");    fflush(stderr);
  fprintf(stdout, "3 ");    fflush(stdout);
  fprintf(stderr, "4 ");    fflush(stderr);
}
```

Output:

```
1 2 3 4
```

File Buffering

- Example

```
#include <stdio.h>
main(void)
{ fprintf(stdout, "1 "); // fflush(stdout);
  fprintf(stderr, "2 "); // fflush(stderr);
  fprintf(stdout, "3 "); // fflush(stdout);
  fprintf(stderr, "4 "); // fflush(stderr);
}
```

Output:

```
2 4 1 3
```