Notes 14.0: File input/output

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2013 session 1

Introduction

- A file is a named section of storage.
  - It might be stored in several scattered fragments in memory.
  - It might contain additional information that allows the operating system to determine what kind of data it stores.
- What matters to us is that a file appears to a C program as a continuous sequence of bytes, each of which can be read individually.
- This corresponds to the file structure in the Unix environment.
- To be able to deal with other operating systems where the file structure is different, C provides two ways of viewing a file:
  - the text view or text mode;
  - the binary view or binary mode.
- Because Unix uses just one file structure, both views are the same, even with files that have been created on a Windows machine.

Views

- In the *binary* view, every byte of the file is accessible to a C program.
- In the *text* view, what a C program sees can be different from what is actually stored in the file.
- For instance:
  - On a Windows machine, an end of line is represented by `\r\n`; in the binary view of such a file, `\r\n` is seen as `\r\n` whereas in the text view of such a file, `\r\n` is seen as `\n`.
  - When Mac machines were running OS 9, an end of line used to be represented by `\r`; in the binary view of such a file, `\r` was seen as `\r` whereas in the text view of such a file, `\r` was seen as `\n`.
- When a program writes to a file in text view, the text view of this file is translated into the binary view that is appropriate for the operating system on which the program is run.

This is illustrated in open.c

Standard files

- C programs automatically open three files:
  - the standard input—the keyboard by default; the file pointer `stdin`, of type pointer to `FILE`, is defined in `stdio.h` points to a data package containing information about this file;
  - the standard output—the screen by default; the file pointer `stdout`, of type pointer to `FILE`, is defined in `stdio.h` points to a data package containing information about this file;
  - the standard error—the screen by default; the file pointer `stderr`, of type pointer to `FILE`, is defined in `stdio.h` points to a data package containing information about this file.
- Redirection causes other files to be recognized as the standard input or the standard output.
- It is useful to have logically distinct places for standard output and standard error: when output is redirected to a file, error messages are still displayed on the screen and can be seen without having to open that file.

The fopen() and fclose() functions (1)

- The `fopen()` function, declared in `stdio.h`, takes as first argument the address of a string containing the name of the file to be opened, and as second argument a string identifying the mode in which the file should be opened:
  - `"r"` to open a file for reading;
  - `"w"` to open a file for writing, removing the content of the file if it already exists, or creating the file otherwise;
  - `"a"` to open a file for writing, appending what will be written to the content of the file if it already exists, or creating the file otherwise;
  - `"a+"` to open a file for reading and writing, appending what will be written to the content of the file if it already exists, or creating the file otherwise;
  - ...
- Appending `b` to the end of the string that identifies the mode, yielding `"rb"`, `"wb"`, `"ab"`, `"a+b"`, ..., performs the same operations but using binary view instead of text view.

The fopen() and fclose() functions (2)

- `fopen()` returns a pointer of type pointer to `FILE`, that can be used by other IO functions.
- The data package pointed to by the pointer returned by `fopen()` indicates where the file is located, where the buffer that will be used to process the file is located, how big and how full that buffer is, etc.
- `fopen()` returns the null pointer `NULL` in case it cannot open the file.
- The `fclose()` function closes the file identified by its unique argument, flushing buffers as needed.
- `fclose()` returns 0 iff the file has been closed successfully.

The exit() function

- The `exit()` function, defined in `stdlib.h`, causes the program to terminate, closing any open file.
- The argument to `exit()` is an `int` that is passed on to the operating system where it can be used by other programs;
  by convention that value is `0` iff the program terminates normally, with different exit values enabling to distinguish between different causes of failure.
- In the initial call to `main()`, using `return 0;` is equivalent to using `exit(0);`
- `return` terminates the program iff it is used in the initial call to `main()`. Otherwise `return` passes control to the calling function (be it `main()` or another function), whereas `exit()` always terminates execution of the program.

These functions are illustrated in count.c

IO functions

- IO functions that work with standard input and output have counterparts that work with arbitrary files; they have an extra argument, namely, a pointer `fp` to the file to be processed.
- The program reducto.c illustrates some of the IO-functions that include
  - `getc(fp)` is the counterpart to `getchar()`;
    hence `getc(stdin)` is equivalent to `getchar()`.
  - `putc(c, fp)` is the counterpart to `putchar(c)`;
    hence `putc(c, stdout)` is equivalent to `putchar(c)`.
  - The counterparts to `printf()`, `scanf()`, `gets()` and `puts()` are `fprintf()`, `fscanf()`, `fgets()` and `fputs()`, respectively.
- More than one file can be opened simultaneously; the maximum number of files that can be opened at one time is system dependent.
- The `rewind()` function takes a pointer to a file as unique argument, and returns to the beginning of the file pointed to for reading and writing, as illustrated in addwords.c.

The fseek() and ftell() functions

- The `fseek()` function enables to treat a file `fn` opened with `fopen()` as an array, and to move to a particular byte in `fn`.
- `fseek()` takes three arguments:
  - a pointer to `fn`;
  - the offset, of type `long`, that tells how many bytes to move from the starting point—hence a positive value will force to move forward from the starting point, a negative value will force to move backward from the starting point, and a value of `0` will force to stay at the starting point;
  - the starting point, which can be:
    - `SEEK_SET`: the beginning of `fn`
    - `SEEK_CUR`: the current position
    - `SEEK_END`: the end of `fn`.
- The `ftell()` function works in *binary* mode; it takes a pointer to `fn` as an argument, and returns a value of type `long` equal to the number of bytes in `fn`, as illustrated in reverse.c.

The fread() and fwrite() functions (1)

- The IO functions we have used so far are good to process characters and strings. What about numeric data?
- If we use `fprintf()` with the `%f` format then we save a number as a string, which results in a loss in precision. *E.g.*:
  - The statements

    ```
    double num = 1./3;
    fprintf(fp, "%f", num);
    ```

    saves `num` as a string of 8 characters `0.333333`.
  - The statements

    ```
    double num = 1./3;
    fprintf(fp, "%.12f", num);
    ```

    saves `num` as a string of 14 characters `0.333333333333`.
- Hence changing the specifier changes the amount of space used to store the value, and changes the value itself.

The fread() and fwrite() functions (2)

- A number should be stored using the same pattern of bits that the program does: a `double` value should be stored in a unit of size `double`, using the same representation.
- For this we can use the `fread()` and `fwrite()` functions in binary view.
- `fread()` and `fwrite()` take 4 arguments:
  - the address of the chunk of data to be read or written;
  - the size, in bytes, of the chunks to be read or written;
  - the number of chunks to be read or written;
  - a pointer to the file being read from or written to.

This is illustrated in randbin.c.