Computer Programming

File I/O

Marius Minea

marius@cs.upt.ro

28 November 2016

Files and streams

A *file* is a data resource on persistent storage (e.g. disk).
File contents are typically sequences of bytes.

A *stream* is a program's view (logical view) of a file, also as sequence of characters (bytes).

To work with files, a program must

1. associate a stream with a file, by *opening* the file.
   A stream is associated with the C datatype `FILE *`
2. work with the stream, just like with `stdin` and `stdout` (two standard streams), with the *same or similar functions*
3. finish by *closing* the file

All functions discussed are in `stdio.h` unless noted

That's it!

Simple: show contents of text file

File name is 1st commandline argument (check that argc is 2)

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  FILE *f;
  char buf[80];
  if (argc == 2 && (f = fopen(argv[1], "r"))) {
    while (fgets(buf, sizeof(buf), f)) fputs(buf, stdout);
    fclose(f);
  } // else report error
}
```

Text and binary streams

Text files are files with human-readable content:
`.txt` files, programs `.c`, `.c++`, web pages `.html`, `.xml` files, etc.

Text streams contain *characters* grouped in *lines* terminated by `\n`

Conversions may occur in reading/writing text streams.
  e.g. end of line is `\r\n` in Windows vs. `\n` in Unix
Standard guarantees one-to-one correspondence if:
  text contains only printable chars, tab and newline
  no newline is immediately preceded by spaces
  last character is a newline

Binary files are not human-readable as character sequences:
`.exe`, `.mp3`, though they may contain text: `.doc`, `.pdf`

Binary streams record internal data as-is .
The sequence of characters read is *exactly the same* as was written

$\Rightarrow$ Any (text) file may also be opened as binary stream

File opening modes

`r`: open for reading (file must exist)
`w`: open for writing (truncated to length 0 if existing, else created)
`a`: open for appending (writing at end of file; created if inexistent)
any writes go to *current* end-of-file, regardless of using `fseek`

First character (`r`, `w`, `a`) of opening mode may be followed by:
`+` (`r+`, `w+`, `a+`): open as stated, but can use for input *and* output
must position (`fseek`) for write after read, unless `EOF`
must position or `fflush` for read after write
`a+`: initial read position implementation-defined (glibc: at start)

`b`: opens binary file (otherwise: text; no explicit text mode)

`x`: (eXclusive) may be last char *only* in `w` mode
file must not exist; no shared access allowed (if system support)

Examples: `rb+` (read/write, binary), `wx`, `wb+x`, `a+x`, etc.

Opening and closing files

```
FILE *fopen (const char *pathname, const char *mode)
```

arg. 1: *file name* (absolute or relative to current directory)

arg. 2: *string* with *open mode*: `r`, `w`, or `a`; optionally `+`, `b`, `x`

```
FILE *f1 = fopen("/home/u/t.txt", "r"); // fixed name, avoid
FILE *f2 = fopen(argv[2], "w"); // second arg in command line

char name[128];                    // example with user-given name
if (scanf("%127s", name) == 1) {
  FILE *f = fopen(name, "ab+"); // open binary, append+read
  if (!f) { /* not opened, handle error */ }
}
```

`fopen` returns `NULL` on error (*MUST test!*)

Otherwise, returned value (a `FILE *`) *used for all other functions*

work with *stream* (logical), not with name (physical)

```
int fclose(FILE *stream)
```

Writes any buffered data to disk, closes file

Returns 0 on success, `EOF` on error. *SHOULD also test!*

(tell user if save of precious data failed!)

Standard streams. Redirection

`stdin`: standard input stream (default: from keyboard)
  `getchar`, `scanf`, etc. read from here
`stdout`: standard output stream (default: to screen)
  `putchar`, `printf`, `puts` write here
`stderr`: standard error stream (default: to screen)

These streams are automatically open when program runs

Write error messages to `stderr`, separate from output (results)!

From command line: can *redirect* standard streams to files,
  input: `program < in.txt` (will read from `in.txt`)
  output: `program > out.txt` (will write to `out.txt`)
  both: `program < in.txt > out.txt`

Can also redirect from within program (with `freopen`)

Remember: can run command from C with `system` (in `stdlib.h`)

File input/output

character-based

```
int fputc(int c, FILE *stream) // write char to file; also putc
int fgetc(FILE *stream) // read char from file; also getc
int ungetc(int c, FILE *stream) // puts ONE char back in stream
```

line-based (one text line)

```
int fputs(const char *s, FILE *stream) // writes string as is
int puts(const char *s) // writes string + \n to stdout
char *fgets(char *s, int size, FILE *stream)
// reads line into s, max. size-1 chars incl. \n, adds \0
```

formatted I/O (same as `printf/scanf`, from file in first arg)

```
int fscanf (FILE *stream, const char *format, ...)
int fprintf(FILE *stream, const char *format, ...)
```

Working with files

Typical sequence for working with files (name on command line)

```
#include <errno.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "correct usage: program filename\n");
    return 1;  // or some other error code
  }
  FILE *fp = fopen(argv[1], "r");
  if (!fp) { perror("error on open"); return errno; }

  // use file: getc, fscanf, fgets, fprintf, etc.

  if (fclose(fp)) { perror("error on close"); return errno; }
  return 0;
}
```

Error functions

`int feof(FILE *stream)` — nonzero if at `EOF`

`int ferror(FILE *stream)` — nonzero if file had errors

Do *NOT* loop while ~~`!feof(f)`~~ :

`EOF` is *NOT detected* when *at* end, only when trying to read *past* it

$\Rightarrow$ loop *while read OK*; if not, check `feof(f)` or `ferror(f)`

Error codes

global variable `int errno` declared in `errno.h`

contains code of last error in a library function

(illegal operation, file not found, not enough memory, etc.)

Function `void perror(const char *s)` from `stdio.h`

prints user message `s`, a colon `:` and then the error description

(same as given by `char *strerror(int errnum)` from `string.h`)

Direct I/O (binary format)

Read/write bytes as-is, without conversion, from/to binary streams

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *strm)
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *strm)
```

read/write to/from address `ptr` `nmemb` objects of `size` bytes each
just like repeated calls to `fgetc`/`fputc`

Return value: *number* of *complete* objects read/written
If smaller than requested, find reason from `feof` and `ferror`

Use `fread`/`fwrite` if *byte order* same in memory and in file
(as specified in docs for file format: `.bmp`, `.jpg`, `.zip` etc.)

big endian, most significant byte first: `0xcafebabe`=`0xca` `0xfe` `0xba` `0xbe`
little endian, least significant byte first: Intel x86 (`0xbe` `0xba` `0xfe` `0xca`)

Otherwise, read/write number byte by byte, (de)compose in needed order

File positioning

Reading and writing use the same *file position indicator*

`long ftell(FILE *stream)` returns position from start of file

`int fseek(FILE *stream, long offset, int whence)`
Sets file position indicator to `offset`; 3rd arg is reference point:
start (`SEEK_SET`), current point (`SEEK_CUR`), end(`SEEK_END`)

`void rewind(FILE *stream)` sets file position indicator to start
same as `fseek(stream, 0L, SEEK_SET); clearerr(stream);`

Use (re)positioning to skip parts of the file on reading,
or to write a selected part

MUST use `fseek`/`fflush` when switching between read and write!

Positioning may not be possible in any file (e.g. `stdin/stdout`)

`int fflush(FILE *stream)`
writes unwritten data buffers for the given file

Chars, ints and EOF revisited

Files (and standard input) contain *bytes (chars)*

`EOF` is NOT a char (the *point* is to distinguish it from any char!)

- chars read by `getchar` or `getc` are `unsigned`, `EOF` is `-1`
- variable read with `getchar`/`getc` must be `int` so it can fit either

`scanf`, `fgets`, `fread` read arrays of *bytes (chars)*

- need no `int`, since they report end-of-file differently
- `EOF` can never be in an array read (since it's *NOT a char*)

Don't mix `signed` *and* `unsigned`*!* `char` *may* be signed

If reading char as int, compare to int: `0xFF`, `0xDA`, etc.

- or if declaring `unsigned char buf[]`

If declared as `char`, compare with `char`: `'\xff'`, `'\xda'`, etc.