Week 08 Tutorial Answers

1. What does *fopen* do? What are its parameters?

Answer:

If you want to read from or write to a file, you need to open the file first. *fopen* is a (convenient) way to do this:

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

You tell *fopen* the name of the file you wish to open, and whether you want to open the file for reading (mode `"r"`), for writing (mode `"w"`), or for appending (mode `"a"`).

If the open request was successful, *fopen* will return a pointer to the file stream that has been opened, otherwise it returns `NULL`.

Files that are opened for writing that already exist are *truncated* to size 0: in other words, all contents are erased. If you want to write information to a file, but do not want to erase the current contents of the file, you should use the append mode, `"a"`.

2. What are some circumstances when *fopen* returns NULL?

Answer:

- If the file you tried to open for reading does not exist.
- If you try to open a file you do not have permission to access.
- If the "mode" string was invalid.
- If the system is out of memory.
- If you try to create a file and your quota of disk blocks or inodes has been exhausted.
- If the pathname was too long.

... and many more reasons.

3. How do you print the specific reason that caused *fopen* to return `NULL`?

Answer:

The global variable `errno` is given a value that indicates what went wrong (see tutorial 1, question 8).

The library function *strerror* turns known values of `errno` into strings, which you could just print out.

```
#include <errno.h>      // for `errno' values
#include <string.h>     // for `strerror(3)`

errno = ENOENT;
char *error = strerror(errno);
assert(strcmp(error, "No such file or directory") == 0);
```

Because this is such a common operation, there aren't many library functions that do this. *perror* is the only standard way to do this:

```
#include <errno.h>      // for `errno' values
#include <stdio.h>      // for `perror(3)'

errno = ENOENT;
perror("");             // prints "No such file or directory"
```

There are several “nonstandard” ways to do this; the most common is *warn*:

```
#include <err.h>        // for `err(3)', `warn(3)', etc.
#include <errno.h>      // for `errno' values

errno = ENOENT;
warn(NULL);             // prints "<progname>: No such file or directory"
```

Often, though, you want to immediately end the program if one of these errors occurs: there is no single standard function to do this, but *err* (related to *warn*) is one very common option:

```c
#include <err.h>        // for `err(3)', `warn(3)', etc.
#include <errno.h>      // for `errno' values

errno = ENOENT;
err(1, NULL);           // prints "<progname>: No such file or directory"
                        // and terminates the program with exit code 1.
```

Similarly, the GNU C library provides *error*:

```c
#include <error.h>      // for `error(3)', etc.
#include <errno.h>      // for `errno' values

error(1, ENOENT, NULL); // prints "<progname>: No such file or directory"
                        // and terminates the program with exit code 1.
```

4. Write a C program, `first_line.c`, which is given one command-line argument, the name of a file, and which prints the first line of that file to `stdout`. If given an incorrect number of arguments, or if there was an error opening the file, it should print a suitable error message.

Answer:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr,  "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // fopen returns a file pointer used to access file
    FILE *stream = fopen(argv[1], "r");
    if (stream == NULL) {
        // couldn't open the file, print an error message
        // to standard error
        fprintf(stderr, "%s: ", argv[0]);
        perror(argv[1]);
        return 1;
    }


    int c;
    while ((c = fgetc(stream)) != EOF) {
        fputc(c, stdout);
        if (c == '\n') {
            break;
        }
    }

    // close the file, as the program is about to exit
    // this isn't necessary but is good practice
    fclose(stream);
    return 0;
}
```

5. Write a C program, `write_line.c`, which is given one command-line argument, the name of a file, and which reads a line from `stdin`, and writes it to the specified file; if the file exists, it should be overwritten.

Answer:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr,  "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // fopen returns a file pointer used to access file
    FILE *stream = fopen(argv[1], "w");
    if (stream == NULL) {
        // couldn't open the file, print an error message
        // to standard error
        fprintf(stderr, "%s: ", argv[0]);
        perror(argv[1]);
        return 1;
    }


    int c;
    while ((c = fgetc(stdin)) != EOF) {
        fputc(c, stream);
        if (c == '\n') {
            break;
        }
    }

    // close the file, as the program is about to exit
    // this isn't necessary but is good practice
    fclose(stream);
    return 0;
}
```

6. Write a C program, `append_line.c`, which is given one command-line argument, the name of a file, and which reads a line from `stdin` and appends it to the specified file.

Answer:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr,  "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // fopen returns a file pointer used to access file
    FILE *stream = fopen(argv[1], "a");
    if (stream == NULL) {
        // couldn't open the file, print an error message
        // to standard error
        fprintf(stderr, "%s: ", argv[0]);
        perror(argv[1]);
        return 1;
    }


    int c;
    while ((c = fgetc(stdin)) != EOF) {
        fputc(c, stream);
        if (c == '\n') {
            break;
        }
    }

    // close the file, as the program is about to exit
    // this isn't necessary but is good practice
    fclose(stream);
    return 0;
}
```

7. Why should you not use fgets or fputs with binary data?

7. Why should you not use *fgets* or *fputs* with binary data?

Answer:

fgets or *fputs* work with C strings: arrays of bytes, terminated with a zero byte, '\0'.

Binary data may naturally contain zero bytes, and thus cannot be treated as a normal string, therefore functions that work on strings (including *fgets* or *fputs*) will not behave correctly.

8. What does the following *printf* statement display?

```
printf ("%c%c%c%c%c%c", 72, 101, 0x6c, 108, 111, 0x0a);
```

Try to work it out without simply compiling and running the code. The *ascii* manual page will help with this; read it by running "`man 7 ascii`". Then, check your answer by compiling and running.

Answer:

It prints the string `"Hello\n"`.

Easy to work out by looking at the ASCII table; e.g., 72 is the code for `'H'`.

9. How many different values can *fgetc* return?

Answer:

257.

It returns one of 256 values (0...255) if it manages to read a byte, and a special value `EOF` if it can't.

Often, `<stdio.h>` will contain something like:

```
#define EOF (-1)
```

10. Why are the names of *fgetc*, *fputc*, *getc*, *putc*, *putchar*, and *getchar* misleading?

Answer:

Programmers often think they are only for character I/O, but all of them just read or write a byte. `fgetb`, `fputb`, `getb`, `putb`, `putbyte`, and `getbyte` would have been better names.

11. Write a C program, `print_borts_file.c`, which prints the contents of a file containing ***borts***.

A ***bort*** is an unsigned two-byte big-endian integer (*bort* is a contraction of big-endian short).

The possible *bort* values are 0..65535.

For example:

```
$ ./print_borts_file test.borts.txt
bort    0: 34
bort    1: 35
bort    2: 36
bort    3: 37
bort    4: 38
bort    5: 39
bort    6: 40
bort    7: 41
bort    8: 42
```

Side note: the linux utilities `od` and `xxd` are good ways to inspect binary files such as *bort* files.

Answer:

```c
// Print contents of a file containing unsigned two-byte big-endian integers

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void print_borts_file(char *pathname);
int get_bort(FILE *f);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        print_borts_file(argv[arg]);
    }
    return 0;
}

void print_borts_file(char *pathname) {
    FILE *stream = fopen(pathname, "r");
    if (stream == NULL) {
        perror(pathname);
        exit(1);
    }

    size_t n_borts_read = 0;
    int bort;
    while ((bort = get_bort(stream)) != EOF) {
        printf("bort %4ld: %d\n", n_borts_read, bort);
        n_borts_read++;
    }

    fclose(stream);
}

// read an unsigned two-byte big-endian integer from stream f
// return the integer or EOF
int get_bort(FILE *f) {
    int byte0 = fgetc(f);
    if (byte0 == EOF) {
        return EOF;
    }
    int byte1 = fgetc(f);
    if (byte1 == EOF) {
        return EOF;
    }
    return (byte0 << 8) | byte1;
}
```

Revision questions

The following questions are primarily intended for revision, either this week or later in session. Your tutor may still choose to cover some of these questions, time permitting.

16. For each of the following calls to the `fopen()` library function, give an `open()` system call that has equivalent semantics relative to the state of the file.

    a. `fopen(`*`FilePath`*`, "r")`
    b. `fopen(`*`FilePath`*`, "a")`
    c. `fopen(`*`FilePath`*`, "w")`
    d. `fopen(`*`FilePath`*`, "r+")`
    e. `fopen(`*`FilePath`*`, "w+")`

    Obviously, `fopen()` returns a `FILE*`, and `open()` returns an integer file descriptor. Ignore this for the purposes of the question; focus on the state of the open file.

Answer:

You can work this out by looking at `man 3 fopen`, and then using the appropriate flags to `open()`.

a. `open(`*`FilePath`*`, O_RDONLY)`
   Opens for reading; fails if the file doesn't exist or isn't readable.

b. `open(`*`FilePath`*`, O_WRONLY|O_CREAT|O_APPEND)`
   Opens for writing and appending, creating the file if it doesn't exist.

c. `open(`*`FilePath`*`, O_WRONLY|O_CREAT|O_TRUNC)`

Opens for writing; creates the file if it doesn't exist, otherwise truncates it.

d. `open(FilePath, O_RDWR)`
Opens for reading and writing; fails if the file doesn't exist.

e. `open(FilePath, O_RDWR|O_CREAT|O_TRUNC)`
Opens for reading and writing; creates the file if it doesn't exist, otherwise truncates it.

17. Consider the `lseek(fd, offset, whence)` function.

    a. What is its purpose?

    b. When would it be useful?

    c. What does its return value represent?

Answer:

a. The `lseek()` function allows you to set the current file position for an open file descriptor.

b. It is vital when attempting to read from or write to a known position in a file. For example, in a file of fixed-size records, you can use `lseek()` to position the file at the start of a specific record, where the position can be computed from the record index and the size of each record.

c. The return value tells you the file position after the `lseek()` completed; it returns -1 if the `lseek()` failed — for example, if `fd` is not an open, seekable file descriptor.

18. Consider a file of size 10000 bytes, open for reading on file descriptor `fd`, initially positioned at the start of the file (offset 0). What will be the file position after each of these calls to `lseek()`? Assume that they are executed in sequence, and one will change the file state that the next one deals with.

    a. `lseek(fd, 0, SEEK_END);`
    b. `lseek(fd, -1000, SEEK_CUR);`
    c. `lseek(fd, 0, SEEK_SET);`
    d. `lseek(fd, -100, SEEK_SET);`
    e. `lseek(fd, 1000, SEEK_SET);`
    f. `lseek(fd, 1000, SEEK_CUR);`

Answer:

a. `lseek(fd, 0, SEEK_END);` moves the file position to the end of the file (i.e., to offset 10000, where there is no data to read).

b. `lseek(fd, -1000, SEEK_CUR);` moves the file position back 1000 from its current position, so it would end up at offset 9000.

c. `lseek(fd, 0, SEEK_SET);` moves the file position to the start of the file again (i.e., to offset 0).

d. `lseek(fd, -100, SEEK_SET);` attempts to move the file position 100 bytes before the start of the file; the file position is unchanged.

e. `lseek(fd, 1000, SEEK_SET);` moves the file position to offset 1000.

f. `lseek(fd, 1000, SEEK_CUR);` move the file position 1000 bytes forward from the current position; since it is currently at position 1000 (from the previous `lseek()`) its new position will be offset 2000.

19. If a file `xyz` contains 2500 bytes, and it is scanned using the following code:

```
int fd;         // open file descriptor
int nb;         // # bytes read
int ns = 0;     // # spaces
char buf[BUFSIZ]; // input buffer

fd = open ("xyz", O_RDONLY);
assert (fd >= 0);
while ((nb = read (fd, buf, 1000)) > 0) {
    for (int i = 0; i < nb; i++)
        if (isspace (buf[i]))
            ns++;
}
close (fd);
```

Assume that all of the relevant `#include`'s are done.

How many calls with be made to the `read()` function, and what is the value of `nb` after each call?

Answer:

Four calls to `read()`.

- The first call sets `nb = 1000`, and reads the first 1000 bytes from the file.
- The second call sets `nb = 1000`, and reads the next 1000 bytes from the file
- The third call sets `nb = 500`, and reads the remaining 500 bytes from the file
- The fourth call sets `nb = 0`, because end-of-file has been reached