# Lecture 13: The C Standard Library Part 2

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/13-standard-library-2.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

---

Here we continue to discuss the standard library. We cover reading and writing to files, as well as error management.

## File I/O: Basic Access Functions

### `open`

Before reading or writing from a file, we first need to obtain a handle to that file with `open`:

```c
int open(const char *pathname, int flags, mode_t mode);
```

This handle is named a **file descriptor**. Open takes several parameters:

- `pathname` is the path of the file, for example `/home/pierre/test`.
- `flags` specifies how the file will be used. For that parameter several flags can be specified by piping particular keywords. The access mode indicates if we will only read the file (`O_RDONLY`), only write the file (`O_WRONLY`) or both (`O_RDWR`). We can indicate to create the file if it does not exist with `O_CREAT`. We can truncate the file size to 0 if it exists with `O_TRUNC`. There are more possible values for `flags`, described in `open`'s manual page[1].
- The final argument `mode` indicates file permissions if it is created, see in the man page the accepted values.

Open returns the file descriptor in the form of an `int`.

### `read`

Once we have the file descriptor we can access the file. Use the read function to read from the file:

```c
ssize_t read(int fd, void *buf, size_t count);
```

`read` takes 3 parameters:

- The first parameter is the file descriptor to read from.
- The second is the address of a buffer that will receive the data that is read.
- The third is the amount of bytes to read.

The `read` function returns the amount of bytes that were actually read. That value can be used to check for errors. Note that it is not an error if the number of bytes read is smaller than what is requested, for example the end of the file could have been reached. An actual error is indicated by a return value equal to `-1`.

[1] https://linux.die.net/man/2/open

### `write`

The `write` function is used to write in the file, it works similarly to `read`:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Its parameters are: - The file descriptor to write to. - The address of a buffer containing the data we wish to write. - The amount of bytes to write.

`write` returns the amount of bytes that were actually written. Same as with `read`, it is not an error if the number of bytes written is smaller than what is requested, for example the disk could be full. However, it's an error when it returns `-1`.

### `close`

Once the operations on a file are finished, the programmer must free the file descriptor using `close`:

```
int close(int fd);
```

## Example: Writing to a File

Consider the following example:

```
#include <stdio.h>
#include <sys/types.h>  // needed for open
#include <sys/stat.h>   // needed for open
#include <fcntl.h>      // needed for open
#include <unistd.h>     // needed for read and write
#include <string.h>

int main(int argc, char **argv) {
    int fd1;
    char *buffer = "hello, world!";

    fd1 = open("./test", O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR | S_IWUSR);
    if(fd1 == -1) {
        printf("error with open\n");
        return -1;
    }

    /* write 'hello, world!' in the file */
    if(write(fd1, buffer, strlen(buffer)) != strlen(buffer)) {
        printf("issue writing\n");
        close(fd1); return -1;
    }

    /* write it again */
    if(write(fd1, buffer, strlen(buffer)) != strlen(buffer)) {
        printf("issue writing\n");
        close(fd1); return -1;
    }

    close(fd1);
    return 0;
}
```

Here we first open a file. With the flags `O_WRONLY | O_TRUNC | O_CREAT` we specify that we will perform on write operations, that if the file exist we want the its size to reduced to 0 upon `open` (this effectively destroys all the content of the file if there was any), and also that we want to create the file if it does not exist. If the file needs to be created, its permission will be read and write permissions for the current user (`S_IRUSR | S_IWUSR`).

Next, we perform a write operation in the file. We have the file descriptor, and we write from `buffer` that contains `"hello world"`. We set the amount of bytes to write to be the size of that buffer. For simplicity, we exit if we cannot fully write the buffer. And then we perform again the same write operation. Next we close the file and exit.

This is the content of the file `test` after execution of that program:

```
hello, world!hello, world!
```

When the file is opened, associated with the file descriptor there is an internal **offset** value that is set at the beginning of the file on disk, that is the address 0 in the file. When we perform the first write the buffer is written in the file starting from the offset, then the offset is placed right after what was written. Then we write a second time the buffer in the file starting at the offset, and we shift the offset at the end of what was written.

## Example: Reading from a File

Things work very similarly for read operations. Consider the following example:

```c
// Assume the local file "test" was previously created with previous (write) example program
char buffer2[10];
int fd2 = open("./test", O_RDONLY, 0x0);
int bytes_read;

if(fd2 == -1) { printf("error open\n"); return -1; }

/* read 9 bytes */
if(read(fd2, buffer2, 9) != 9) {
    printf("error reading\n"); close(fd2); return -1;
}

/* fix the string and print it */
buffer2[9] = '\0';
printf("read: '%s'\n", buffer2);

/* read 9 bytes again */
bytes_read = read(fd2, buffer2, 9);
if(bytes_read != 9) {
    printf("error reading\n"); close(fd2); return -1;
}

/* fix the string and print it */
buffer2[9] = '\0';
printf("read: '%s'\n", buffer2);

close(fd2);
```

We open the file we previously created in read only mode. We read 9 bytes from it inside `buffer2` and we display the content of `buffer2`. We do this operation twice. Note how we manually write the string termination character in the last byte of `buffer2`, right after what was read. This program outputs the following:

```
read: 'hello, wo'
read: 'rldhello,'
```

When the file descriptor is created, the file offset is initialised to 0. The first read operation of 9 bytes reads the first 9 bytes of the file into the first 9 bytes of `buffer2` and shifts the offset right after that. A second read operation reads the next 9 bytes from the file and shifts the buffer again.

## Random Number Generation

The `rand` function returns a random integer ranging from 0 to a large constant named RAND_MAX:

```c
int rand(void);
```

If we want to constrain the number to fall within a particular interval we can use the modulo operator. In this example we'll get only numbers ranging between 0 and 99:

```
for(int i=0; i<10; i++)
    printf("%d ", rand()%100);
```

Running this program one may notice that the sequence is always the same among several runs. This is not a very random behaviour. It is due to the way the numbers are generated, it is done in sequence based on a value called the **seed**. A given seed will always yield the same sequence. So to get variable sequences among multiple execution, we can initialise the seed based on the current time:

```
srand(time(NULL));  // init random seed
for(int i=0; i<10; i++)
    printf("%d ", rand()%100);
```

Note that in some cases it is good to have a fixed seed to ensure reproducible results.

## Error Management

When a function from the C standard library fails, a variable maintained by the C library is set with an error code. This variable is `errno`. Consider this code:

```
#include <errno.h>  // needed for errno and perror

int main(int argc, char **argv) {
    int fd = open("a/file/that/does/not/exist", O_RDONLY, 0x0);

    /* Open always returns -1 on failure, but it can be due to many different reasons */
    if(fd == -1) {
        printf("open failed! errno is: %d\n", errno);

        /* errno is an integer code corresponding to a given reason. To format
         * it in a textual way use perror: */
        perror("open");
    }
    return 0;
}
```

Here we try to open a file that does not exist and open `fails`, it returns `-1`. We can print the value of `errno`, it's an integer, so it's not very helpful by itself. We can get a better description of the error with the function `perror`. It internally looks at `errno` and prints a textual description of the error on the terminal.