Lecture 14: The C Standard Library Part 3

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/14-standard-library-3.

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

In this third and last lecture on the C standard library, we discuss the `strtol` function, used to convert strings into numbers, as well as stream-based file operations, a series of functions to perform file I/O.

Limitations of `atoi`

Until now we have been converting strings to integers using the function `atoi`. The advantage of atoi is that it is easy to use, you simply pass the string as parameter, and it returns the converted integer. However when the string is malformed or corresponds to a number that is too large or too small to be stored as an integer, atoi will not warn you that something has gone wrong. Things failing silently and the program continuing to execute in an inconsistent state can be very hard to debug.

Converting Strings to Integers with `strtol`

The solution to that problem is to use `strtol`:

```
long strtol(const char *nptr, char **endptr, int base);
```

Its usage is slightly more complicated than `atoi`, but it is also much more robust. `strtol` will convert the string pointed by `nptr` into a `long` that is returned. One can specify a base such as 10, 16 for hexadecimal numbers, etc.

`endptr` is used to check the validity of the string: after the call, what is pointed by `endptr` will point either to the first invalid character of the string, and to `'\0'` if the string is fully valid. In case of under or overflow, `errno` is set to `ERANGE` and `strtol` returns either `LONG_MIN` if it is an underflow, or `LONG_MAX` if it is an overflow.

Consider this example:

```
/* ... */
#include <errno.h>
#include <limits.h>

int main(int argc, char **argv) {
  if(argc != 2) { /* ... */ }

  char *endptr;
  long n = strtol(argv[1], &endptr, 10);

  if(*endptr != '\0') {
    printf("invalid string!\n");
    return -1;
```

```
  }

  if(errno == ERANGE) {
    if(n == LONG_MIN) printf("underflow!\n");
    if(n == LONG_MAX) printf("overflow!\n");
    return -1;
  }

  printf("n is: %ld\n", n);
  return 0;
}
```

We use `strtol` to detect malformed strings and under/overflows. When we call the function in question, we pass as parameter the string, a pointer of pointer of character for `endptr`, and `10` to indicate a decimal base. Note that because `strtol` wants to return a pointer through the `endptr` parameter, we need to pass a pointer of pointer, a `char **`. It is a pointer to a `char *` that we declared above.

We check if the string was valid by observing the value of what is pointed by `endptr`. If it is the `NULL` character the string is valid, otherwise we abort.

We also check that there was no overflow or underflow by looking at `errno`and at the return value of `strtol`.

Stream-based File I/O: Main Functions

`fopen` and `fclose`

We create a stream object, also called `FILE *`, with the `fopen` function:

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

It takes the file path as parameter, and returns the stream object or `NULL` if something went wrong. It also takes another parameter, `mode`, that precises how the file will be accessed, as well as what to do if the file does not exists or if it already exists when `fopen` is called:

- `"r"`: read-only.
- `"r+"`: read-write.
- `"w"`: write-only, truncate file if it exists, create it if it does not.
- `"w+"`: read-write, truncate file if it exists, create it if it does not.
- There are more options possible for `mode`, listed in the relevant manual page[1].

Once all operations on a stream are done, it needs to be released with `fclose()`:

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

`fread` and `fwrite`

To read and write in streams representing files, the functions `fread` and `fwrite` are used:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

`fread` reads `nmemb` contiguous items, each item being of file `size`, from the file represented by `stream`, into memory at address `ptr`. `fwrite` writes `nmemb` items, each of file `size`, into the file represented by `stream`, from memory at address `ptr`. These functions return the number of items read or written, which is only equal to the total number of bytes written when `size` is 1. Note that similarly to what happens with file descriptors, streams have an internal offset that gets incremented each time we access the file.

[1] https://linux.die.net/man/3/fopen

Stream-based File I/O: Example

Consider that example program:

```c
#include <stdio.h>

char *alphabet = "abcdefghijklmnopqrstuvwxyz";

int main(int argc, char **argv) {
  FILE *f1, *f2;
  char buffer[27];

  f1 = fopen("test-file.txt", "w");
  if(f1 == NULL) {
    perror("fopen");
    return -1;
  }

  if(fwrite(alphabet, 2, 13, f1) != 13) {
    perror("fwrite");
    fclose(f1);
    return -1;
  }

  fclose(f1);

  f2 = fopen("test-file.txt", "r");
  if(f2 == NULL) {
    perror("fopen");
    return -1;
  }

  if(fread(buffer, 1, 26, f2) != 26) {
    perror("fread");
    fclose(f2);
    return -1;
  }

  buffer[26] = '\0';
  printf("read: %s\n", buffer);

  fclose(f2);
  return 0;
}
```

We start by opening a first stream `f1` in write-only mode. This mode will create the file if it does not exist, and if it does, it will truncate its size to 0. Next we want to write the entirety of the string `alphabet` in the file. Using `fwrite`, we can do it for example as 13 chunks of size 2 bytes each. Once done we close the stream `f1` with `fclose`.

Next we open a new stream `f2`, in read-only mode this time. Note that because this is a new stream independent of the previous one, its offset will be set to 0 in the file. We aim to read what we previously wrote in the file. With `fread` we can do it for example as 26 chunks of 1 byte each. Similarly to `read`, the programmer needs to fix up the buffer storing the result with the termination character to make it a proper C string.