School of Computing and Information Systems

COMP30023: Computer Systems

File Reading (Linux & C)

1 Introduction

This material is intended to be a introduction/refresher to file reading in C, for the purpose of Project 1, where the path of an input file is specified by command-line arguments.

2 Linux Standard Streams

Before diving into file reading, let's first revise the standard streams[1] which are automatically opened when a program's `main` function is invoked. These are:

- `stdin` - The standard input stream, file descriptor 0
- `stdout` - The standard output stream, which is used for normal output from the program (e.g. `printf`), file descriptor 1
- `stderr` - The standard error stream, which is used for error messages and diagnostic messages (e.g. debug messages with `fprintf(stderr, ...)`), file descriptor 2

Recommended reading on file descriptors and streams: https://www.gnu.org/software/libc/manual/html_node/Streams-and-File-Descriptors.html

2.1 File Redirection and Pipes

With bash (and many other shells), we can redirect input and output to and from programs. e.g.:

- Redirecting file to `stdin`: `$ ./my_program < file.txt`
- Redirecting `stdout` to file: `$ ./my_program > stdout.txt`
- Redirecting `stderr` to file: `$ ./my_program 2> stderr.txt`

There are also pipes, which connects the `stdout` of one process to the `stdin` of another. e.g.:
`$ cat file.txt | grep COMP30023`[2]

3 Taking command line arguments

Command-line arguments are relatively simple to read in C, when the `main` function is defined to take two arguments `argc` and `argv`.

[1] `https://www.gnu.org/software/libc/manual/html_node/Standard-Streams.html`
[2] `cat` is redundantly used here

An example function signature may be:
`int main(int argc, char* argv[]);`

The return value of the `main` function indicates the argument provided to the implicit call of `exit()`, e.g. `EXIT_SUCCESS`, `EXIT_FAILURE`, or any other code ≤ 255. This will then become the exit status of the program.

`argv` is a vector of C strings; its elements are the individual command line argument strings. The file name of the program being run is also included in the vector as the first element; the value of argc counts this element[3].

We can write a demo program to see this in action:

```
#include <stdio.h>
int main(int argc, char* argv[]) {
        for (int i = 0; i < argc; i++)
                printf("%s\n", argv[i]);
        return 0;
}
```

```
$ ./cli 123 -f helloworld.txt
./cli
123                  (a positional argument)
-f
helloworld.txt       (a named argument)
```

Because arguments may be presented in any order (as a general rule of thumb), we will need a loop to parse the arguments or use something like `getopt`[4], which provides a useful abstraction to parse arguments.

4 Filenames

Modern filesystems support the nesting of directories. When a filename starts with `/`, it's an *absolute* path and is a full path starting from the root directory `/`, e.g. `/bin/ls`, `/home/comp30023/.ssh`. Otherwise, it is a *relative*, e.g. `~/.ssh`, `hello.txt`, `./hello.txt` etc.

If there is a need, the `realpath` command can be used to get the absolute path of a file from the command line, e.g.:

```
$ realpath ~/.ssh
/home/comp30023/.ssh
```

Here are some assumptions which you shouldn't make about files in Project 1:

- Your program will have write access to test files or the directory where it is invoked.
- That filenames will be shorter than a specific length.
  In any case, you **should not** need to copy the string from `argv` and store it somewhere else.
- That test files will always be under the directory where your project is stored.
  i.e. The path given can be relative or absolute. Please do not hard code it.
- That your compiled program will be stored and invoked from a specific location.

Note that we **will** provide copies of test files in the testing/marking environments and provide your program with valid paths to those files.

[3] `https://www.gnu.org/software/libc/manual/html_node/Program-Arguments.html`
[4] `https://www.gnu.org/software/libc/manual/html_node/Getopt.html`

5 Reading an example file

```
1 f 100
23 t 1000
4 t 2000
```

Above is the example file which we will attempt to parse.

Opening the file

We can use `fopen`, to open the file as a stream.
`"r"` is specified to open the file in readonly mode.

```
FILE *fp = fopen(filename, "r");
fclose(fp);
```

Reading the file

Based on the characteristics of this data (which you should also verify against the data and specification - *especially regarding data types*), I've came up with the following struct.

```
#include <stdbool.h>
typedef struct {
        int col1;
        bool col2;
        int col3;
} Example;
```

Since we guarantee that the input file is well-formed, a simple way to continuously read input is to use `fscanf`.

```
#include <stdio.h>
...
int col1, col3; char col2;
while (fscanf(fp, "%d %c %d", &col1, &col2, &col3) == 3) {
        Example example = {
                .col1 = col1,
                .col2 = col2 == 't',
                .col3 = col3
        };
        printf("%d %d %d\n", example.col1, example.col2, example.col3);
}
```

Storing the read values into memory

For best practices, it's best to avoid reading all of the input into memory before processing it[5]. However, this may be necessary at times, e.g. if the input needs to be sorted before it can be processed.

Since we must sort our input and there are no restrictions on how long the file can be, we must use dynamic memory allocation.

Here's the full program:

[5] There is overhead incurred in storing the data in memory and also to loop over it

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

typedef struct {
        int col1;
        bool col2;
        int col3;
} Example;

int main(int argc, char* argv[]) {
        // Open file
        FILE* fp = fopen(argv[1], "r");
        if (fp == NULL) {
                perror("fopen");
                exit(EXIT_FAILURE);
        }

        // Read into memory
        size_t examples_len = 0, examples_size = 2;
        Example* examples = malloc(sizeof(Example) * examples_size);
        if (examples == NULL) {
                fprintf(stderr, "Malloc failure\n");
                exit(EXIT_FAILURE);
        }

        int col1, col3;
        char col2;
        while (fscanf(fp, "%d %c %d", &col1, &col2, &col3) == 3) {
                if (examples_len == examples_size) {
                        examples_size *= 2;
                        examples = realloc(examples, sizeof(Example) * examples_size);
                        if (examples == NULL) {
                                fprintf(stderr, "Realloc failure\n");
                                exit(EXIT_FAILURE);
                        }
                }
                examples[examples_len].col1 = col1;
                examples[examples_len].col2 = col2 == 't';
                examples[examples_len++].col3 = col3;
        }

        // Print values
        for (int i = 0; i < examples_len; i++) {
                Example example = examples[i];
                printf("%d %d %d\n", example.col1, example.col2, example.col3);
        }

        free(examples);
        fclose(fp);
        return 0;
}
```