Yale-NUS College - 2018-2019

Bruno Bodin

YSC2227: INTRO TO C

week 03.1.io (auto-generated)

FILES

- A file is complex,
- this is the combination of *data* and *metadata*
- They can be stored on multiple *storage devices* , even across them.

DATA

- Character (text files)
- Binary (program)
- Numeric

META-DATA (OS SPECIFIC!)

- Name
- Location (where to find it)/path
- Type
- Creation date
- Last modification
- Size (real size and actual size)
- Protection/Access restriction

OPERATIONS

- The *POSIX standard* defines multiple function: `open`, `read`, `close`, ...
- The *C Standard library* provides higher-level functions: `fopen`, `fread`, `fclose`, ...
- Using the *f-functions* enables using `fscanf`, `fprintf` etc...
- The f-functions are also **more efficient**.

OPEN A FILE

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

- Opens a given file
- Returns a pointer to a `FILE` structure (`NULL` if error)
- Modes:
  - "r" : Opens a file in read only. The file must exist.
  - "w" : Creates new file in write only. Existing file is overwritten.
  - "a" : Creates a new file if it does not exist, otherwise starts writing at the end of the file.
  - "r+" : Opens a file in read/write mode. The file must exist
  - "w+" : Creates new file in read/write modes.
  - "a+" : Creates a new file if it does not exist, otherwise starts writing at the end of the file. Reading will happen from the start of file.
  - "b" : This specifies that the file is opened as a binary rather than a text file. In text mode, there might be some online conversions of the content.

CLOSE A FILE

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

- Returns `0` (success) or `EOF` (error)

Always close a file after opening it.

If not, you may have the following side effects:

- File not accessible from outside
- Empty file when program finishes

READ

- `size_t fread (void *ptr, size_t size, size_t count, FILE *stream );`
- Read as much as `count` blocks of size `size` from a file to the buffer `ptr`.
- Return the number of blocks actually read.

```
FILE* fd = fopen ( argv[1] , "rb" );
if (!fd) {printf("Could␣not␣open␣the␣file.\n");exit (1);}

unsigned char * buf = malloc (sizeof(unsigned char) * SIZE );
int read = fread(buf, sizeof(unsigned char) , SIZE  , fd);

if (read != SIZE) {printf("File␣is␣too␣small.\n");} else {
  printf("The␣header␣of␣this␣file␣is␣" ) ;
  for (int i = 0 ; i < SIZE ; printf("␣%2.2x",buf[i++]));
}
printf(".\n" ) ;
fclose(fd);
```

WRITE

```
size_t fwrite (const void *ptr, size_t size, size_t count, FILE *fd);
```

- Write as much as `count` blocks of size `size` in a file from the buffer `ptr`.
- Start writting from the stream position indicator (SPI) and advance the SPI by as many bytes written.
- Return the number of blocks actually written.

```
FILE* fdw = fopen ( argv[2] , "wb" );
if (!fdw) {printf("Could␣not␣open␣the␣file.\n");exit (1);}
int wt = fwrite(buf, sizeof(unsigned char) , SIZE  , fdw);
if (wt != SIZE) {printf("Erro␣while␣writting\n");}
fclose(fdw);
```

READ (FANCY MODE)

- `int fscanf(FILE *stream, const char *format, ...);`
  - Reads a sequence of values using similar *pattern matching* format of `printf` .
- `int fgetc(FILE *stream);`
  - Reads one char from *stream
- `char *fgets(char *str, int n, FILE *stream);`
  - Reads up to n characters from a file, and stores the result in str.
  - Returns pointer to str if success, NULL if error

WRITE (FANCY MODE)

- `int fprintf(FILE *stream, const char *format, ...);`
  - Can be used to write content in any file using similar pattern matching from `printf` .
- `int fputs(const char *str, FILE *stream);`
  - Write a string. return `EOF` if error.
- `int fputc(int char, FILE *stream);`
  - Write a character. return `EOF` if error.
- `int fflush (FILE *stream);`
  - The f-functions perform *double buffering* , this function flushes the buffer and thus actually writes the data.

double buffering

STANDARD I/O

- `stdin` for standard input
- `stdout` for standard output
- `stderr` for standard error

RESTRUCTURED TEXT

- Simple markup language used for formatting
  - Ex: "this *word* is in bold"
  - Output: "this **word** is in bold"
- We will only consider the following:
  - '*' is used for bold
  - '#' for underline
  - '$' for italic
  - line starting with '|' indicates a title of section

HTML

- Stands for HyperText Markup Language
- Uses tags:
  - Paragraph are decorated by ...
  - ...
- Every opened tag needs to be closed
- An HTML file starts with "<html>" and ends with "</html>"

TODAY'S EXERCISE

- Take a reStructuredText file as input
- Convert it in HTML
- Note:
  - HTML file should start with <html> and ends with </html>
  - '*' is used for bold (html: and)
  - '#' for underline (<u> and </u>)
  - '$' for italic (<i> and <i>)
  - A line starting with '|' indicates a title of section (<h1> and </h1>)
  - A newline in html is
. You can keep the newline character for readability

TOPICS COVERED

- Variables and Assignement Operators ✓ (T. Bailey, Chapter 1 and 2)
- Numeric Data Types and Conversion ✓ (T. Bailey, Chapter 2)
- Arrays ✓ (T. Bailey, Chapter 8)
- Arithhmetic and bitwise operators ✓ (T. Bailey, Chapter 2 and 12)
- Compilation, flags, and command-line arguments ✓ (D. Harris C.10)
- Pointers ✓ (T. Bailey, Chapter 7)
- C functions ✓ (T. Bailey, Chapter 4)
- Files and I/O ✓ (T. Bailey, Chapter 13)
- Control structures, logic operators, and loops ✓ (T. Bailey, Chapter 3)
- Scope ✓ (T. Bailey, Chapter 5)
- Structures and Unions ✓ (T. Bailey, Chapter 11 and 14)
- Memory management and segmentation ✓ (T. Bailey, Chapter 9)
- Basic libraries
- Makefile
- Debugging

KEY POINTS

- *data, metadata, storage devices*
- *posix standard, c standard library, f-functions*
- *pattern matching*
- *double buffering*

REFERENCES

- cook and magician:
  https://pixabay.com/en/users/graphicmama-team-2641041/