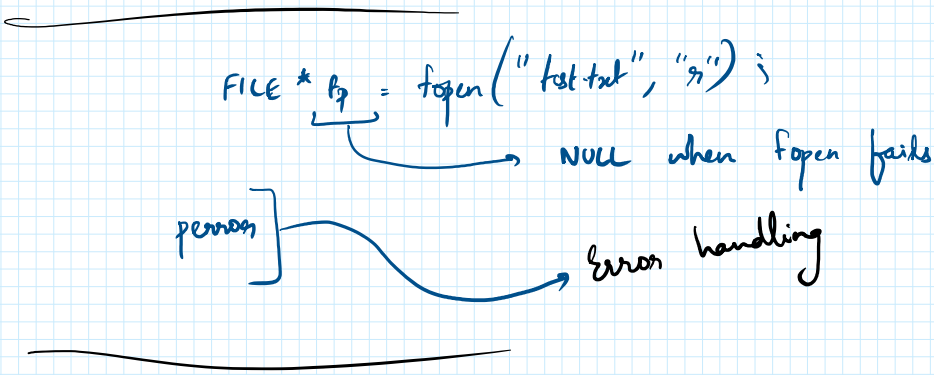
1. File Handling

07 May 2024 08:48



fclose(fp)

- Closes file stream, flushes buffer
- Return value → Success → 0
  → failure → EOF

fgetc() / fputc()

Same as getc and putc, difference being getc and putc may be implemented as macros

Syntax: int fgetc(FILE *fp);

Return value → Success → Next byte from input stream
→ EOF / failure → EOF

Syntax: int fputc(character, FILE *fp);

Return value → success → Character that is written
→ failure → EOF

Error handling

Here, EOF is returned either when the end of the file is reached or in the case of some error. To differentiate:

feof()

Syntax: int feof(FILE *fp);

Return value → success → Non-zero if EOF is reached
→ failure → Zero if not reached

fgets()

Reads a specified number of characters as a string

- Reading stops when:
  - (n-1) characters are read
  - Newline character is read

- EOF is reached

Syntax: char * fgets (char* char_array, int n, FILE *fp);

Return value → success → Pointer to char_array

→ EOF → NULL

→ failure → NULL

fputs()

Writes a string to the file

Syntax: int fputs (char * str, FILE * fp);

Return value → Non-negative number

→ EOF

Error handling

ferror()

Syntax: int ferror (FILE * fp);

Return value → success → Non-zero if error present

→ failure → Zero otherwise

fscanf()

Syntax: int fscanf (FILE * fp, format_string, parameters ...);

Return value → success → No. of characters read

→ failure → EOF

fprintf()

Syntax: int fprintf (FILE *fp, format_string, parameters ...);

Return value → success → No. of characters written

→ failure → EOF

fread()

Reads an entire block

Syntax: size_t fread (void *p, size_t size, size_t n, FILE *fp);

Return value → success → Reads n items from the file and returns n

→ EOF → Reads less than n items and returns a value less than n

→ failure → Reads less than n items and returns a value less than n

fwrite()

Writes data in the form of binary blocks

Syntax: size_t fwrite(void *p, size_t size, size_t n, FILE *fp);

Return value → success → Writes n items and returns n

→ failure → Writes less than n items and returns a value less than n

fseek()

Sets the file pointer position.

New position (in bytes) = whence + offset

Syntax: int fseek(FILE* fp, long int offset, int whence);

Return value → 0

→ Non-zero value

> NOTE: Special values for whence
> 0 ⇌ SEEK_SET: Start of file
> 1 ⇌ SEEK_CUR: Current location of pointer
> 2 ⇌ SEEK_END: EOF

rewind()

Sets file pointer to beginning of file

Syntax: void rewind(FILE* fp);

- Equivalent to (void) fseek(FILE *fp, 0L, SEEK_SET)

ftell()

Returns current position of file pointer

Syntax: long int ftell(FILE *fp);

Return value → Current position in bytes from the beginning of the file

→ -1

CSV FILES

Tokenisation : strtok()

Syntax: char * strtok(string/NULL, delimiter);

Return value → Pointer to token created

→ no token → NULL

Working

char str[] = "This is a string"

char delim[] = " "

- First, we call strtok like so:

```
char* token = strtok(str, delim);
```

```
char * token = strtok(str, delim);
```

- This replaces the first instance of the delimiter with a NULL character. It also maintains an internal pointer to the next character after the NULL character.

"This \0 is a string"
↑ internal pointer

- To get the next token, we need to continue from the location specified by the internal pointer. To do this, we pass NULL instead of the string's starting address.

```
char * token2 = strtok(NULL, delim);
```

"This \0 is \0 a string"
↑

atoi function

- We repeat this until the entire string is tokenized, i.e., strtok returns NULL.