Yrkes Akademin
Vi hjälper dig att lyckas!

C Programming

File Handling

File Handling

- In C, files have been abstracted in a data type as **FILE**
- **FILE** type contains all required information for different operations on files; including
  - A pointer to the buffer used when we write to a file.
  - A file position indicator which indicate to a position in a file
  - Some flags which are used for error handling
  - And etc.
- We shall always use **FILE** pointers (**FILE** *) and never dereference such pointers
- There are a set of functions in **stdio.h** for different operations on files and file systems
- Generally there are two types of files
  - **Text** file in which data is stored in a standard character format
  - **Binary** file in which data is stored in raw binary format.

File Handling

- File pointers are used to operate on files in the file system.
  - A file pointer can be declared like: FILE *fptr; // fptr is the name of the pointer
  - In most of the operations on a file, a file pointer as a handle to the file is required
    - Generally when we don't need to open files; file pointers are not required. For example:
      - When we use **remove** function to delete a file.
      - When we use **rename** function to change the name of a file
  - A file pointer is returned when we open a file using ***fopen*** function
    - Like; *FILE *file = fopen("filename", "mode");*
- Opening and creating files using **fopen**

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

- The **fopen** function opens a file specified by ***filename*** in a mode specified by ***mode***

File Handling

- The argument **filename** points to the name of a file as a string
- The argument **mode** points to a string as the mode which in the file is opened
  - If mode is one of the following, the file is open in the indicated mode.
    - **Otherwise, the behavior is undefined.**

Mode	**Description (r: reading, w: writing, a: appending, b: binary, x: exclusive, +: updating)**
"r"	Open text file for reading
"w"	Truncate to zero length or create text file for writing
"wx"	Exclusively create text file for writing (If the file already exists NULL is returned)
"a"	Append; open or create text file for writing at end-of-file
"rb"	Open binary file for reading
"wb"	Truncate to zero length or create binary file for writing

File Handling

Mode	**Description (r: reading, w: writing, a: appending, b: binary, x: exclusive, +: updating)**
"wbx"	Exclusively create binary file for writing (If the file already exists NULL is returned)
"ab"	Append; open or create binary file for writing at end-of-file
"r+"	Open text file for update (reading and writing)
"w+"	Truncate to zero length or create text file for update
"w+x"	Exclusively create text file for update (If the file already exists NULL is returned)
"a+"	Append; open or create text file for update, writing at end-of-file
"r+b" or **"rb+"**	Open binary file for update (reading and writing)
"w+b" or **"wb+"**	Truncate to zero length or create binary file for update
"w+bx" or **"wb+x"**	Exclusively create binary file for update (If the file already exists NULL is returned)
"a+b" or **"ab+"**	Append; open or create binary file for update, writing at end-of-file

File Handling

- If fopen fails, it returns NULL. Always we need to check for failures.
  - E.g. `FILE *file = fopen("text.txt", "r"); if(file == NULL) { /* error handling */ }`
- Max. number of simultaneously opened files in a system is defined as FOPEN_MAX
- Max. length of a filename is defined as FILENAME_MAX macro
- The same file shall not be open for read and write access at the same time on different streams.
  - E.g. `FILE *fw = fopen("tmp", "r+"); FILE *fr = fopen("tmp", "r"); // Not OK`
- There shall be no attempt to write to a stream which has been opened as read-only
- A pointer to a FILE object shall not be dereferenced

File Handling

- ❖ Closing files using **fclose**; int fclose(FILE * **fp**);
  - ➢ It flushes any unwritten buffered data to the file pointed by **fp** and then closes the file
  - ➢ It releases any memory used for the stream's input and output buffers.
  - ➢ It returns **zero** on success, or **EOF** if an error occurs
- ❖ When we don't need a file to be opened we shall close it.
- ❖ A pointer to a FILE shall not be used after the associated stream has been closed

- ❖ Reading from and writing to files
  - ➢ Reading from files and writing to files are done as streams of bytes
  - ➢ Streams in C can be either text (for text) or binary (for binary data) streams
  - ➢ Inefficient to read or write individual characters/bytes. Therefore I/O buffers are used

File Handling

- Reading from and writing to files
  - When we close a file, the output buffer is automatically flushed.
  - You can explicitly flush the output buffer of a file stream by calling **fflush** function

```
int fflush( FILE *fp );
```

  - The fflush() function empties the output buffer of the open file specified by **fp**
    - If the file was opened for writing, or for reading/writing and the last operation on it was a write operation, any unwritten data in its output buffer is written to the file.
    - In all other cases, the behavior is undefined and depends on the implementation
    - The function returns **0** if successful, or **EOF** if an error occurs in writing to the file.
  - Like elements in a char array, each character/byte in a file has a certain position in the file
  - A file ***position indicator*** determines the position of the next character to be read or written

File Handling

❖ Reading from and writing to files

- ➢ A file opened for reading/writing, the **position indicator** points to the **beginning** of the file (**0**)
- ➢ A file opened for appending, the **position indicator** points to the **end** of the file
- ➢ We can get the current value of the **position indicator** using the **ftell** function

```c
long int ftell(FILE * fp);
```

  - ■ On success, the current value of the position indicator is returned. On failure, -1L is returned.
- ➢ Random access to data in a file is possible using functions
  - ■ `int fseek(FILE * fp, long offset, int origin);`
    - ● Sets the position indicator to a position specified by an **offset** from a reference point (**origin**)
    - ● **origin** can be the beginning(**SEEK_SET**), current position(**SEEK_CUR**) and end(**SEEK_END**) of the file
    - ● Returns 0 on success and a non-zero value on failure.
  - ■ `void rewind(FILE * fp); // sets the file position indicator to the beginning of the file`

File Handling

- Reading from files
  - int feof(FILE *fp); // If you reach **EOF** a non-zero value is returned. Otherwise, zero is returned.
  - int fgetc(FILE *fp); // Used to read a character from the file
  - char *fgets(char *buf, int n, FILE *fp);
    - Used to read up to n − 1 characters from the file into the buffer, buf, and terminate the string
  - int fscanf(FILE *restrict fp, const char *restrict format, ...);
    - Used to read a formatted string from a file
  - size_t fread(void *buffer, size_t size, size_t count, FILE *fp); // It should be used only with binary files
    - It reads up to **count** elements whose size is **size** from a file and store them in the **buffer** array
    - The total number of elements successfully read is returned.
    - **ferror** and **feof** can be used to check errors

File Handling

❖ Writing to files

- int fputc(int c, FILE *fp); // Used to write a character to a file
- int ungetc(int c, FILE *fp); // Used to push the last character read, c, back onto the file
- int fputs(const char *s, FILE *fp); // Used to to write a null-terminated string to a file
- size_t fwrite(const void *buffer, size_t size, size_t count, FILE *fp);
  - Used to write count elements whose size is size from the buffer array to a file
  - The total number of elements successfully written is returned.
  - A return value less than **count** indicates that an error occurred.
  - It should be used only with binary files
- int fprintf(FILE *restrict fp, const char *restrict format, ...);
  - Used to write a formatted string to a file