Week 3

- Text and binary files
- Functions

File operations

So far the operations are done on the terminal and not stored anywhere.

But in the software industry, most of the programs are written to store the information fetched from the program.

One such way is to store the fetched information in a file. Different operations that can be performed on a file are:

- Creation of a new file
- Opening an existing file
- Reading from file
- Writing to a file
- Closing a file

Text files vs binary files

- **Text files** store text. One reads and writes strings to text files.
- **Binary files** are very similar to array (like strings are arrays of characters), but the “structures” are in a disk file rather than in an array in memory.

- Binary files also usually have faster read and write times than text files, because a binary image of the data is stored directly from memory to disk (or vice versa).
- In a text file, everything has to be converted back and forth to text, and this takes time.

Opening a file

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program:

```
FILE *fptr;
```

Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file:

ptr = fopen("filename","mode");

- The basic ***modes*** are “r” (read), “w” (write), “a” (append)

Writing text files

- `fprintf()` is used to print content in file instead of stdout (screen)
- `fclose()` closes a file.

File: Week3 > `Example1.c`

Reading text files

`fgets(line, n, fptr)`

`fgets()` keeps on reading characters until:

- (*n* - 1) characters have been read.
- a newline character is encountered.
- end of file (EOF) is reached.

File: `Example2.c`

Some observations

- If a file to open for writing does not exist, it will be created
- If a file to open for writing exists, the old content ***will be lost***
- If there is no file to open for reading, an error occurs: NULL is returned
- If the file is opened for append, the new information will be added following the exiting one
- It is the user's responsibility not to break the file!

Some error handling

```
if ((fptr = fopen(fileName, "r")) == NULL){
     perror("Failed to open file, terminating");
     exit(0);
    }
```

- `fopen` open returns `NULL` if the file doesn't exist
- `perror` prints your text plus system's error declaration
- `return` returns from the current function
- `exit()` terminates the **whole program**. Defined in `stdlib.h`

File: `Example3.c`

Writing binary files `fwrite()`

```
for (i=0; i < 10; i++)
    fwrite(&i, sizeof(int), 1, fptr);
```

- `&i` points to the block of memory which contains the data items to be written.
- `sizeof(int)` specifies the number of bytes of each item to be written.
- `1` is the number of items to be written.
- `fptr` is a pointer to the file where data items will be written.

Reading binary files `fread()`

```
while (fread(&nr, sizeof(int), 1, fptr) != 0)
        printf("%d,", nr);
```

- `&nr`: Pointer to the variable where data will be stored.
- `sizeof(int)`: The size of each element to be read in bytes
- 1: Number of elements to be read
- `fptr`: Pointer to the FILE object from where data is to be read

File: `Example4.c`

Functions

- A **function** is a group of statements that together perform a task. Every C program has at least one function, which is `main()`.

- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

- A function ***declaration*** tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Functions

A function **declaration** tells the compiler about a function's name, return type, and parameters (as in the actual function definition):

```
int max(int num1, int num2);
```

A function **definition** provides the actual body of the function:

```
return_type function_name(parameter list) {
   // body of the function
}
```

File: `Example5.c`

Parameters

- To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

File: `Example6.c`

Formal parameter: the identifier used in a method to stand for the value that is passed into the method by a caller.

Actual parameter: the actual value that is passed into the method by a caller.

Parameters

Call by value: values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are ***not*** reflected in actual parameters of the caller.

Call by reference: the address of the variable is passed into the function call as the actual parameter. Both the actual and formal parameters refer to the same locations, so any changes made inside the function ***are*** reflected in actual parameters of the caller.

File: `Example7.c`

Defining macros

We can define simple “functions” as macros:

```
#define square(x) x*x
```

File: `Example8.c`

```
gcc -E Example8.c > out
```