Files

- You learned the difference between interactive and batch programs in CENG 103. If the program takes data from a data file rather than from the keyboard, it is named as *batch processing.*

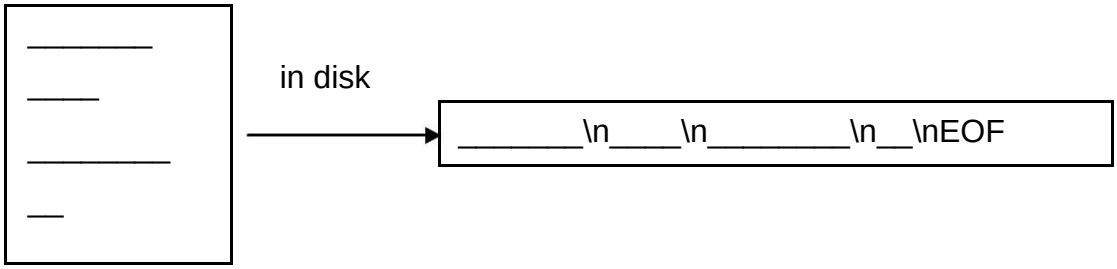
- Output to a disk file creates a permanent copy of it. You can get a hard copy of a disk file by sending it to the printer. In addition, output file of a program can be input to another program.

- There are two types of data files: *text files* and *binary files*.

- Upto now, you have been using text files in your programs. As you know, a text file can be created using an editor, or as the output of a program. It is a collection of characters ending with a special end-of-file character, which is denoted as **EOF**.

- While creating or editing a text file, each time the enter key is pressed, the newline character **'\n'** is placed in the file.

file



- Newline can be processed like any other character. It can be input using `getchar` or using `scanf` with the `%c` placeholder. It can be compared to **'\n'** for equality. It can be output using `printf` and `putchar`.

- Upto now, we checked for EOF only in batch programs. However, interactive programs also recognize EOF by:

  UNIX systems : <ctrl> d

  IBM PC / compatibles : <ctrl> z

- Thus, when the program is reading with `scanf`, if the user presses **<ctrl>z** and enter keys, `scanf` will return a negative integer associated with EOF as its value.

Example:

```
status = scanf("%d", &num);
while (status != EOF) {
     sum += num;
     status = scanf("%d", &num);
}
```

goes on reading integers until the user enters **<ctrl>z**. The C language allows you to write the above program segment also as follows:

```
for (status=scanf("%d", &num); status != EOF; status=scanf("%d",
     &num)) sum += num;
```

- All textual input and output data are actually a continuous *stream* of character codes. Streams provide communication channels between files and programs. When a file is opened, a stream is associated with the file.

- Three files and their associated streams are automatically opened when program execution begins: `stdin`, `stdout`, `stderr`.

  `stdin` : keyboard's input stream

  `stdout` : "normal" output stream for screen.

  `stderr` : "error" output stream for screen.

- We access an input or output stream with pointers. A *File Pointer* is the address of a structure that contains the information necessary to access an I/O file. As you know, we declare a file pointer as:

  ```
  FILE  *infilep;
  ```

- `stdin`, `stdout` and `stderr` are identifiers of type `FILE *`, which are initialized by the system prior to the start of a C program.

- Remember that we can give a value to a file pointer using `fopen` as:

  ```
  infilep = fopen("a:data.txt", "r");
  ```

- `fopen` prepares the file for input and output before permitting access. It returns NULL if it fails.

- A pointer whose value equals NULL is called a *null pointer*. Don't confuse the null pointer with the null character.

- Remember that using `fopen` with mode **"w"** to open a file that already exists for output usually causes loss of contents of the existing file (depends on OS).

- You already learned many functions of `stdio` library. Some of them are used to access `stdin` or `stdout` (e.g.: `scanf`, `printf`, `getchar`, `putchar`, `gets`, `puts`), some others can access any text file (e.g.: `fscanf`, `fprintf`, `getc`, `putc`, `fclose`).

- As you know, `getc` reads one character from a file whose file pointer is

  received. `ch = getc(infilep);`

  `getchar` reads one character from `stdin`.

  ```
  ch = getchar();      // ch = getc(stdin);
  ```

Example: Write a function that takes a file pointer as input argument and returns the number of characters appearing on the current line of the file.

```
int count_line(FILE *finp)
{
     int count = 0;
     char ch;
     for (ch = getc(finp); ch != '\n'; ch = getc(finp))
         count++;
     return (count);
}
```

Home Exercise: Rewrite the above function recursively.

- As you know, `putc` writes one character to a file whose file pointer is received. `putc(ch, outfilep);`

  `putchar` writes one character to `stdout`.

  ```
  putchar(ch);   // putc(ch, stdout);
  ```

Example: Write a function that takes two file pointers as arguments and copies everything from the first file to the second.

```
void copy_file(FILE *file1, FILE *file2)
{
     char ch;
     for (ch = getc(file1); ch != EOF; ch = getc(file1))
         putc(ch, file2);
}
```

- As you know, `fscanf` reads data from a file whose file pointer is received. It returns as its result the number of input values it has successfully stored through its output arguments. It returns the negative EOF value when it encounters the end of the file accessed by its file pointer argument.

  ```
  fscanf(infilep, "%[^\n]", line);
  ```

- The `stdio` library provides another function, `fgets`, to read a string from a file.

Example:

```
fgets(line, 80, infilep);
```

copies characters from the input file into the string `line` until it has read 79 characters or a newline character, whichever comes first. It marks the end of the string `line` with the null character. It returns a pointer to the string it reads. If it can not read a string, it returns NULL.

- Let's rewwrite the `copy_file` function using `fgets`:

```
void copy_file(FILE *file1, FILE *file2)
{
     char line[80];
     while (fgets(line, 80, file1) != NULL)
         fprintf(file2, "%s", line);
}
```

- Do not forget that, when a line is read using `fgets`, it also reads the newline character. That's why we did not use `\n` in the format string of the `fprintf` in the above function.

- We can use `fscanf` when the file is formatted, thus, if we can read data from it using some placeholders. Otherwise, it may be necessary to read the whole line with `fgets` and parse it to reach to the tokens of input data.

- While reading from a file, `feof` function can be used to check whether it is the end of a file or not. It returns a nonzero value (true) once the end-of-file indicator has been set; otherwise, zero is returned.

- Let's rewwrite the `copy_file` function using `feof`:

```
void copy_file(FILE *file1, FILE *file2)
{
     char ch;
     ch = getc(file1);
     while (!feof(file1))
     {
         putc(ch, file2);
         ch = getc(file1);
     }
}
```

Example: Write a main function that makes a back-up copy of a text file whose name is given as input, using `copy_file` function.

```
int main(void)
{
     char in_name[30];      // name of the input file
     FILE *inp, *outp;

     printf("Enter the name of the file you want to back up>
     "); scanf("%s", in_name);
     inp = fopen(in_name, "r");
     if (inp == NULL)
        printf("Can not open the file %s!\n", in_name);
     else
     {
        // The back-up file will have the prefix "bu_"
        outp = fopen(concat(in_name, "bu_", in_name, 30), "w");
        copy_file(inp, outp);
        fclose(inp);
        fclose(outp);
     }
     return (0);
}
```

- As in the above example, upto now, in all our batch programs, we checked if the input file could not be opened, if not gave an error message and finished the program.

- However, since the above program is getting the file name as input, we can make data validation, and want the user to enter another file name if that file does not exist:

```
...
inp = fopen(in_name, "r");
while (inp == NULL)
{
   printf("Can not open the file %s!\n", in_name);
   printf("Re-enter the file name> "); scanf("%s",
   in_name);
   inp = fopen(in_name, "r");
}
// The back-up file will have the prefix "bu_"
outp = fopen(concat(in_name, "bu_", in_name, 30),
"w"); ...
```