HIGH LEVEL FILE PROCESSING

1. Overview

The learning objectives of this lab session are:

- To understand the functions used for file processing at a higher level.
  - These functions use special structures of type FILE.
  - The main operations which can be performed on files at this level are: open (including to create a new file), read/write of a character or a character string, read/write binary, file positioning, file close, flushing the buffer area for a file.
- To acquire hands-on experience in using high level IO functions.

2. Brief theory reminder

At the higher level of the file management system, a pointer to a structure of type FILE is attached to each file:

FILE *p;

The type **FILE** and all function prototypes are found in **stdio.h.**

2.1. File open

To open an existing file, and also to create a new one you can use the function **fopen**, with the following prototype:

FILE *fopen(const char *path_name, const char *mode);

where:
path_name – pointer to a character string which defines the path name for a file;
mode – pointer to a character string which defines the file processing mode for the opened file as follows:

"r" – open for reading;
"w" – open for writing;
"a" – open for appending;
"r+" – open for updating (both read and write);
"rb" – open for binary reading;
"wb" – open for binary writing;
"r+b" – open for binary reading/writing ;
"w+b" – open for binary writing/reading;
"ab" – open for binary appending.

Notes.

- The content of an existing file opened for writing (mode **"w"**) will be overwritten.
- If a file is opened in mode append (mode **"a"**), new records can be added after the last existing record in the file.
- A non-existent file opened in modes **"w"** or **"a"** will be automatically created..
- The function **fopen** returns a pointer of type FILE if successful, or the NULL pointer if not.

The standard I/O files are automatically opened at run time. They are:

stdin – standard input;
stdout – standard output;
stderr – standard error;

On a PC, in MS-DOS the following are also opened:

stdprn – standard printing output;
stdaux – standard auxiliary device (serial communication).

2.2. Character-wise processing of a file

2.2.1. Read/write character by character

A file can be processed by reading opr writing character by character, using functions **getc** and **putc,** of prototypes:

```
int getc (FILE *pf);
int putc (int ch, FILE *pf);
```

where:
pf - is a pointer to the type FILE returned by the function **fopen**;
ch - is the ASCII code of the character to write.
Both return the ASCII code of a character (read or written) on success; on error -1 is returned.
For example, a sequence used to copy the standard input to the standard output is:

```
while ((c=getc(stdin))!=EOF)
    putc(c, stdout);
```

2.2.2. Read/write character strings

To read a character string form a file use **fgets.** This has the prototype:

```
char *fgets(char *s, int n, FILE *pf);
```

where:
s – a pointer to the memory area where the character string will be stored;
n – the size of the memory area where the string will be read. Reading stops when the EOL (End Of Line, '\n') is met or at most n-1 characters were read. In both cases, the last character in the string will be '\0'.
Pf – a pointer to a structure of type FILE.
fgets returns the value of **s** on success, and a NULL pointer on failure.

To write a character string, ('\0' is included) use the function **fputs,** of prototype:

```
int fputs (const char *s, FILE *pf);
```

where **s** is a pointer to the beginning of the memory area containing the string to be written.
fputs returns the ASCII code of the last written character if successful, or –1 on failure.

2.2.3. Formatted read/write

Formatted read/write can be performed using the functions **fscanf/fprintf,** similar to the functions **sscanf/sprintf,** presented in laboratory session no. 1. The difference consists in the fact that for **sscanf/sprintf** the first argument is a pointer to the area where the character string will be stored, and for **fscanf/fprintf** the first argument is a pointer to the file used for reading/writing as their prototypes show:

```
int fscanf(FILE *pf, const char *format {, address});
int fprintf(FILE *pf, const char *format {, expression});
```

fscanf returns the number of correctly read fields. When the end of a file is met, **fscanf** returns EOF. **fprintf** returns the number of characters written to the file if successful, and –1 if it fails.

2.3. Flushing a file buffer

To flush a file buffer, use **fflush**, which has the following prototype:

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

where **pf** is a pointer to a structure of type FILE, opened for output.
Notes.

- If the file is opened for writing, the contents of its buffer area is *written to* the corresponding file.
- If the file is opened for reading, the unread characters are lost.
- **fflush** returns zero upon success, and –1 if an error occurs.

2.4. Positioning in a file

To position the file pointer on a certain byte use **fseek.** Prototype:

```
int fseek (FILE *pf, long offset, int origin);
```

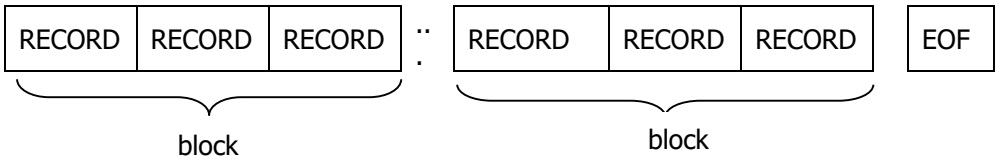
This function is similar to **lseek.** The difference consists in the fact that for **fseek** one passes a pointer to type FILE as the first argument, and for **lseek** a file descriptor was given.

The current position in the file is given as an offset in bytes from its beginning is returned by the function **ftell.** Its prototype is:

```
long ftell(FILE *pf);
```

2.5. Processing binary files

Binary files are considered to be sequences of records, each block containing a number records, as show below:



Records have fixed lengths. A record is data of predefined or user defined type.

To read/write records use **fread**/**fwrite**. Prototypes:

```
unsigned fread(void *buf, unsigned dim, unsigned nrart, FILE *pf);
unsigned fwrite(void *buf, unsigned dim, unsigned nrart, FILE *pf);
```

where:

buf – pointer to the buffer area containing the read/to write records;
dim – size of a record in bytes;
nrart – number of records in a block;
pf – pointer to structure of type FILE.

Both functions return the number of records read/written on success, and –1 on error..

2.6. File close

To close a file use **fclose**. Prototype:

int fclose(FILE *pf);

where **pf** is a pointer to a structure of type FILE returned by **fopen.** The function returns 0 if successful close, and –1 if not.

2.7. File delete

A closed file can be deleted using the function **unlink,** of prototype:

int unlink (const char *path_name);

where **path_name** is a pointer to a character string representing the path to the file.The function returns 0 if successful close, and –1 if not.

2.8. Examples

2.2.4. Example 1. Character and string oriented file processing.

Program L11Ex1.c creates a file character by character, the characters being read from the standard input. Then it illustrates the way to append character strings, and then it prints the file, line by line, lines being preceded by their line numbers.

```
/* Program L11Ex1.c */

#include <stdio.h>
/* Shows character and character string oriented file processing*/

int main(void)
{
    char ch, s[100], file_name[50]="file1.txt";
    int i;
    FILE *pf;

    /* create a file; write all characters read from stdin (including '\n' */
    pf=fopen(file_name, "w");
    printf("\nPlease input a text to store in file \"%s\".\nEnd with Ctrl+Z\n",
file_name);
    while ((ch=getc(stdin))!=EOF)
    {
        putc(ch, pf);
    }
    fclose(pf);
    /* append character strings*/
    pf=fopen(file_name,"r+");
    fseek(pf, 0l, 2);
    printf("\nPlease input the strings to append to the created file. End with an empty\
line.\n");
    while(fgets(s, 100, stdin)!=(char*)0)
    {
        fputs(s, pf);
    }
    fclose(pf);
    /* show file contents */
```

```
    printf("\nLines of the files (numbered):\n");
    i=0;
    pf=fopen(file_name,"r");
    while(fgets(s,100,pf)!=(char *)0)
    {
        printf("%d  %s", i, s);
        i++;
    }
    fclose(pf);
    getchar();
    unlink(file_name);
    return 0;
}
```

2.2.5. Example 2. Binary processing of files.

Program L11Ex2.c illustrates binary processing of files. It creates and then lists the contents fof the file.

```
/* Program L11Ex2.c */

#include <stdio.h>
/* Illustrates binary file processing */
typedef struct
{
    char name[40];
    long sum;
    /*other members */
} RecordT;

void show_contents(char *file_name)
{
    FILE *pf;
    RecordT buf;
    int i;

    pf=fopen(file_name, "rb");
    printf("\nNO.   SUM       FAMILY NAME-FIRST NAME\n");
    i=0;
    while(fread(&buf, sizeof(RecordT), 1, pf) > 0)
   {
        printf("\n%6d %10ld      %-40s", i, buf.sum, buf.name);
        i++;
   }
    fclose(pf);
 }

int main(void)
{
    FILE *pf;
    RecordT buf;
    int i, n;
    char s[40], file_name[40]="file.dat";

    /*Create file */
    printf("\nInput the number of persons, n=");
```

```
    scanf("%d", &n);
    pf=fopen(file_name, "wb");
    for(i=1; i<=n; i++)
    {
        printf("Family name and first name of the person:: ");
        fgets(buf.name, 40, stdin);
        printf("Sum = ");
        scanf(" %ld", &buf.sum);
        fwrite(&buf, sizeof(RecordT), 1, pf);
    }
    fclose(pf);
    printf("\nFile contents:\n");
    show_contents(file_name);
    getchar();
    return 0;
}
```

3. Lab Tasks

3.1. Analyze and execute the examples L11Ex1.c and L11Ex2.c.

3.2. Create a file containing a list of items a shop sells. An item is represented as a structure which contains:
- a code;
- a name;
- a unit of measure;
- the amount;
- price per unit

3.3. Starting with this file get another one where the items are sorted ascending on their code field.

3.4. Using the file created at 3.2, write functions to input and output items in the shop.

3.5. Write a program to process the results of the admission exam at our University. The program shall include the creation of the file with candidate data. Finally, you must obtain the files with the candidates admitted for each specialization in the order of their average scores and the ones rejected in order of their names. An average score is calculated using the formula: ((baccalaureate + 4 * math_test / 5.0).

3.6. Consider a file directory. Each entry contains a 8 character name and a 3 character extension, the number of blocks allocated to store it, the address of the first allocated block, and the date and time of last update (year, month, day, hour, minute, second). You shall:
- Create a file directory.
- List the files in the directory in the ascending order of file names.
- List the files in the directory in the ascending order of their last update date and time.

3.7. Write programs which use the high level functions to implement problems 3.2.-3.10 of laboratory session no. 10.