Low-level programming

Lecture 7

Functions – continued

I/O operations

Marcin Radom, Ph.D.

Institute of Computing Science
Faculty of Computing and Telecommunication

Pointers to the function

- Functions in C are not variables, yet there can be pointers for them.
- Such pointers can be assigned values, stored in arrays, given to other functions as arguments and also can be returned as a function values.

- **Example of function pointer definition:**

```
int ( *pf )( );
```

Pointers to the function

- We read the example in the following way `int (*pf)();`
  - name of function is **`pf`**,
  - we go left, because inside where pf is there is nothing else in this parenthesis () - **`( *pf )`** – so: **pf** is in fact a **pointer**
  - we go right, next parenthesis () – **`( *pf )( )`** – **pf** is a pointer **to a function without arguments**
  - now going left – … a function **returning** **`int ( *pf )( )`** – value type **`int`**.
  - All in one: **pf** is a pointer to a function taking no arguments and returning value type **`int`**.
- If we skip the first parenthesis, we get: **`int *pf ( )`** – this would be however a declaration of a function which returns a pointer to type ***int*** and has no arguments.
- **Name of a function is also the address of it in the memory** (i.e., an address to the area of memory, where code for such a function resides).
- Because of that we can set it to point at address stored by pointer:

```
pointer = name_of_function; // IT IS NOT A CALLING OF A FUNCTION
// pointer = name_of_function() - error, this IS A CALLING and also we
// try to write the value of it into pointer
```

Pointers to the function

- **Example:**

```
int ( *pf )( int, int );  // type of result and arguments

int Ki( int x, int y ){
     return (x + y) * (x + 2 * y);
}

long Kl( int x, int y ){
     return (long)(x + 3 * y) * (x - y);
}

pf = Ki;       // correct (compiler will know that pf is
               // pointer and will set it up correctly)
pf = & Ki;     // correct (using operator & is not required, yet
               // we can still do it)
pf = Kl;       // error, wrong pointer (wrong type of result for
               // pointer (int) vs function (long) )
```

Pointers to the function

- **Calling functions - examples:**

```
void bubbleSort ( float T[ ], int n ) { ... }
void mergeSort ( float T[ ], int n ) { ... }
void heapSort ( float T[ ], int n ) { ... }
// pointer to a function can be assigned initial value:
void ( *wS )( float[ ], int ) = bubbleSort;
void ( *nS )( float[ ], int ) ;

nS = wS; // ns and ws point to function bubbleSort
float TAB[ 150 ];

/* equivalent to calling function bubbleSort: */
wS ( TAB, 150 ); // we can now use the pointer to call the function
//
// calling what pointer points to, that is: calling a function bubbleSort
( *nS )( TAB, 150 );    // parenthesis ARE REQUIRED for correct C syntax
```

Array of pointers to the functions

- We can declare an array of pointers to functions:

```
void (*(twf[10]) )();
```

  - Definitions above we can read as:
    - **`twf`** – name of array *twf*
    - **`[10]`** – which is 10-elements array…
    - **`*`** - of pointers…
    - **`()`** – to the functions having no arguments…
    - **`void`** - …which (functions that is) return nothing.
- In such an array we can store chosen functions of our program (e.g., elements of menu, which then can be easily changed).
- After defining such an array its pointer can be assigned as before (but using indexes):

```
twf[1] = search;     // set second pointer (index [1]) for function search
```

- **Calling example:**

```
twf[1]();            // call the second function from array twf
```

Array of pointers to the functions - Example

- **Examples:**

```
int fA(double x) { ... }
int fB(double y) { ... }
int fC(double z) { ... }
 // initialization – assigning functions to elements of a table
int ( *fX [3] )( double ) = { fA, fB };
fX[2] = fC;

fX[1]( 37.18 );          // calling for fB

// definition with default initial values
int ( *Test [10] )( char* ) = { /* ...... */ };
int Results [10];  // result of function from pointer table
char* Texts [10];  // functions arguments from pointer table Texts
for (int i = 0; i < 10; ++i )
     Results[i] = Texts[i] ( Texts[i] ); // calling functions
```

Array of pointers to the functions

- We can establish a „shortcut” for a complex type using **typedef**.
- **Example:**
  - We could provide a synonym for type being a *pointer to a function which have two **float** arguments and its return value type is **int***.
  - Then we can use it for declaration/definition of function pointers

```
typedef int ( *PF )( float, float ); // PF is a type name

int Fun( float a, float b ) { ... }

PF f1, f2 = Fun;
PF Tfun[15]; // pointers to function array
```

Pointer to the function as a function arguments

- Pointer to a function can also be the argument of some other function.
- **Example:**

```
// argument here is a specific sorting method
void Sort( float[ ], int, void (*) ( float[ ], int ) = mergeSort );

void Sort( float TAB[ ], int size, void (*fS) (float[ ], int) )
{  ..................
      fS( TAB, size );
      ..................
}

float T1[100], T2[200];
void ( *wF )( float[ ], int ) = heapSort;

Sort( T1, 100, bubbleSort );   // bubbleSort
Sort( T2, 200, wF );           // heapSort
Sort( T2, 200 );               // default: mergeSort
```

Pointer to the function as a function arguments

- Pointer to a function can be returned by another function as its value.
- **Example:**

```
typedef char* ( *NP )( char*, char* ); // NP is type name
struct FUNCTIONS {
     int feature; // we use it to identify some specific function
     NP function; // pointer to function type NP
} TABLE[15];

// returns a pointer to a function
NP Search( struct FUNCTIONS TAB[ ], int size, int pattern ) {
     for (int i = 1; i < size; ++i)
          if (TAB[i].feature == pattern)
               return TAB[i].function;
     return TAB[0].function;
}

// calling chosen function
printf( "%s\n" , Search ( TABLE, 15, 1527 )( "Alf","Ogi") );
```

Pointer to the function - Example

- **Example - calculator**

```
#include <math.h>
// Counts values for sin, cos, tan, cotan, sqrt, log, recip, sqr

double FSin ( double x, bool &err)
{ return sin(x); }

double FCos ( double x, bool &err)
{ return cos(x); }

double FTan ( double x, bool &err) {
     if (cos(x) != 0)
          return tan(x);
     else {
          err = true;
          return 0;
     }
}
```

Pointer to the function - Example

```
double FCotan ( double x, bool &err){
     if (sin(x) != 0)
          return 1 / tan(x);
     else {
          err = true;
          return 0;
     }
}

double FSqrt ( double x, bool &err){
     if (x >= 0)
          return sqrt(x);
     else {
          err = true;
          return 0;
     }
}
```

Pointer to the function - Example

```
double FLog ( double x, bool &err) {
     if (x > 0)
          return log(x);
     else {
          err = true;
          return 0;
     }
}
double FRecip ( double x, bool &err){
     if (x != 0)
          return 1 / x;
     else {
          err = true;
          return 0;
     }
}

double FSqr ( double x, bool &err)
{ return x * x; }
```

Pointer to the function - Example

```
int main() {
	double (*TabFun[8])(double, bool&)={FSin,FCos,FTan,FCotan,FSqrt,FLog,FRecip,FSqr};
	int option;
	double arg, result;
	bool go_on = true, invalid;
	while (go_on) {
		printf("\nChose function: \n0 - sin, \n1 - cos, \n2 - tan, \n3 - cotan,"
			"\n4 - sqrt,\n5 - log,\n6 - recip,\n7 - sqr, \ninna  - end: ");
		scanf("%d", &option);
		if(option < 0 || option > 7){
			printf("\nEnd.\n");
			go_on = false;
		} else {
			printf("Provide value x : ");
			scanf("%lf", &arg);
			invalid = false;
			result = TabFun[option](arg, invalid); // calling proper function
			if (invalid)
				printf("Invalid argument.\n");
			else
				printf("Result = %lf\n", result);
		}
	}
	return 0;
}
```

I/O operations

Files, access, write and read

Files

- A **file** is a certain block of disk memory with its own name.
- From C language point of view file is a sequence of bytes, of which every single one can be read separately.
- According to the ANSI standard two ways of looking at files are: *binary* and *text* perspective.
- In **binary look** each byte of file is accessible to the program.
  - Single element of such a file is byte.
  - Binary file format: [file bytes][EOF]
- From **text perspective** what program „sees" can vary – it not necessarily be the real bytes-written sequence (e.g. \n and \r special characters for PC/Mac architectures).
  - Single element of such a file is a single character.
  - Text file format:

    [characters in 1 line][/r][/n]
    ...
    [characters in last line][EOF]

Files – text and binary ones

- **Example:** how to write number 25
  - **Text file**
    - It will be coded as two ASCII digits: '2' and '5' and for example additional space at the end.

32H	35H	20H

    - Functions (e.g. **fprintf**) write numerical values to text file after they transform them into chains.
    - It can lead to a loss of accuracy (e.g. value 0.33 can we written using 4 characters and we could loose all additional fraction values).
  - **Binary file**
    - The most precise way to write a number is to write the exact structure of its bytes as coded in the given type, e.g., type **double** variable should be written in using same number of bytes as the size of type **double**.

Files – text and binary ones

- If data in the file are represented in the same type as in the program, we say they are in *binary form.*
- In such case there is no conversion between numerical value and a string.
- In a binary file number 25 (assuming it is *int*) is written using 4 bytes in *little-endian* conversion (in which a less significant byte (a lower one in other words) is written first).

19H	0H	0H	0H

- In standard input/output form data exchange in binary format is realized using **fread** and **fwrite**.

- In reality all data are written in binary form (even signs).
- If all data in file are interpreted as character codes, we say file contains text data.
- In some or all data are interpreted as numerical binary values, we say that file contains binary data.

Files

- To represent files in programs file streams are used.
- Such a stream is represented in C by file structural variable **FILE**.
- Structure **FILE** is declared in **<stdio.h>** and stores the following information:
  - buffer location,
  - current character position in buffer,
  - type of access to the file (for reading, writing, etc.),
  - error signals or end of file information.
- All operations on a file stream require pointer to the **FILE** structure.
- Declaration for such a file pointer can be:

```
FILE *fp; //fp is a pointer to FILE structure
```

Files

- In **<stdio.h>** three pointers to files are defined. They are assigned to 3 typical „files" opened by C programs:
  - **stdin** – standard input (usually keyboard)
  - **stdout** – standard output (usually screen)
  - **stderr** – standard output for errors (usually screen)
- Above pointers are type FILE, so they all can be used as arguments for standard input/output functions, e.g., pointer ***fp*** from last example.
- **Phases of working with a file:**
  - File opening,
  - Write or read operations (always from current position, index is moved by reading/writing the next values),
  - Closing file

Files

- For file operations in C we use some common functions from **<stdio.h> (#include <stdio.h>).**
- **There are 3 groups of functions:**
  - **fopen, fclose, fcloseall** (opening and closing files)
  - **ftell, fseek, rewind, feof** (establishing current position within file, checking if end of file is reached)
  - **fread, fwrite, fgetc, fputc, fgets, fputs, fscanf, fprintf** (reading and writing data)
- **Opening file:**
  - To open a file a function ***fopen()*** must be used. While using it we need to provide a path to file (and its name) and type of access to it.
  - Function **fopen** return pointer to the file (pointer to structure of type **FILE**).
  - Function **fopen** returns NULL is opening was not possible.

fopen – open access to file

- Function ***fopen***

```
FILE* fopen(const char* name, const char* mode);
```

Function: opening file "name"

Mode:

r	: reading existing file
w	: creating file for writing
a	: writing at the end of the existing file (append mode)
r+	: writing or reading existing file
w+	: creating new file for writing and reading
a+	: writing or reading starting from the end of existing file

Additionally in some systems binary na text files are distinguished:

t	: text file
b	: binary file

- Opening non existing file for writing/append will create it.
- Opening existing file for writing will clear its content!

fopen – open access to file

- **Example:**

```
#include <stdio.h>
int main(void)
{
     ...
     FILE *list;
     list = fopen("LIST.TXT","rt+");
     if (list== NULL)
        printf("\nFile cannot be opened.");
     ...
     return 0;
}
```

fclose – closing access to file

- Function ***fclose***

```
int fclose ( FILE *file);
```

Function: closes access to the file.

Result: 0 – if action was successful **or EOF** – if it was not.

Example:

```
#include <stdio.h>
int main(void)
{ ...
     FILE *personel = fopen ( "PER.TXT",  "rt" );
     ...
     fclose ( personel );
     ...
}
```

fclose – closing access to file

- Function ***fclose()*** 'breaks' the connection between pointer (established by function ***fopen***) and a real disk file, releasing such pointer to be used for some other file for example.
- It is very important because operating systems limits number of access to the single file from a single program.
- Access to file should be closed if the file is no longer necessary.
- Function ***fclose()*** is called automatically for all opened files if the program ends **normally**.
- Function ***fcloseall()***

```
int fcloseall ( void );
```

Function: closes all opened files in the program.

Returns: number of closed files – if the action was successful **or EOF** – if it was not.

fseek – establishing position in file

- Function ***fseek()*** allows to treat file as it was an array – it allows access to the specific byte within the file.
- Function ***fseek()*** has 3 arguments:
  - Pointer to the file.
  - Offset (position):
    - Specifies distance and direction from the starting location.
    - Must be type **long**.
    - It can be *positive* (moving forward), *negative* (moving back) or *zero* (remain in place).
  - Mode (goal), which define starting location.
    - In <**stdio.h**> there are string constant for such modes::
      - **SEEK_SET** – the very start of a file
      - **SEEK_CUR** – current location
      - **SEEK_END** – end of file location

fseek – establishing position in file

- Function ***fseek()***

```
int fseek (FILE *file, long position, int mode);
```

Function: sets the current location in file.

mode:

SEEK_SET - start of file,

SEEK_CUR - current position

SEEK_END - end of file

position : +/- , from 0, can go beyond file size

Result: 0 : success, !0 : error

fseek – establishing position in file

- **Example:**

```
#include <stdio.h>
int main(void)
{
     ...
     long pp;
     FILE *description = fopen("OP1.DOC","rt+");
     // move to the place located 0 bytes beyond end of file
     // so in simple words: to the end of file
     fseek(description , 0L, SEEK_END);
     ... // write at the end of file
     // move 0 bytes beyond start of file
     fseek(description , 0L, SEEK_SET);
     ... // write at the beginning of file
     pp = 15453l; // move from current location
     fseek(description , pp, SEEK_CUR);
     ... // read something
     ...
}
```

rewind – establishing position in file

- Function ***rewind*** sets file pointer to the initial position.
- Calling ***rewind****(fp)* is equal to : ***fseek****(fp, oL, SEEK_SET)*

```
void rewind ( FILE *file);
```

Function: set the current location to the beginning of file

Example:

```
#include <stdio.h>
int main(void)
{
     ...
     FILE *working = fopen("R1.DAT","rt+");
     fseek(working , 0L, SEEK_END);
     ...                  // write at the end of file
     rewind(working );    // go to the beginning of file
     ...
}
```

feof – checking end of file

- Function **feof** allows checking if the end of file is reached.

```
int feof ( FILE *file);
```

Function: reads the status of end-of-file indicator

Result: returns !o : the is **EOF** (indicator of the end of file), o if not

Example:

```
#include <stdio.h>
int main(void)
{ ...
     FILE *file1= fopen("P1.MAN","rt");
     ...          // read from file
     if (feof(file1)!= 0)
          printf("\nEnd of file reached.");
     ...
}
```

fgetc – reading

- Function ***fgetc*** reads from stream *file* a single sign and returns its value as *unsigned char* (and transformed to *int*). If there was any error or end of file is reached it returns **EOF**.

```
int  fgetc ( FILE *file);
```

Function: read next sign

Result: integer number ooo | sign_code

Example:

```
#include <stdio.h>
int main(void)
{
      ...
      char cc;
      FILE *info = fopen("INF.DOC","rt");
      cc = fgetc(info);
      ...
}
```

fputc – writing

- Function ***fputc*** writes *sign* (transformed to *unsigned char*) to the stream *file*. It returns a written sign (its ASCII code) or **EOF** in there was any error.

```
int  fputc ( int sign,  FILE *file);
```

Function: writes next sign (integer number: ooo | code)

Result: sign or **EOF**

Example:

```
#include <stdio.h>
int main (void)
{
     char cc = 'K';
     FILE *data= fopen("DATA.DOC","rt+");
     ...
     fputc(cc, data);
     ...
}
```

fgets – reading

- Function ***fgets*** reads at maximum *number-1* signs and puts them into array *text*. Function stops after reading new line sign – it is also put into array. Whole text end with \o.

```
char* fgets ( char *text,  int number,  FILE *file);
```

Function: reads sequence of signs, at maximum *number-1* signs.

Result: text or **NULL** in case of error or reaching end of file

Example:

```
#include <stdio.h>
int main(void)
{
      ...
      char surname[16];
      FILE *dir = fopen("DIR.DOC","rt");
      fgets(surname, 16, dir);
      ...
}
```

fputs – writing

- Function ***fputs*** writes text from table *text* into a stream *file*. It returns written character or **EOF** if there was any error.

```
int fputs ( char *text, FILE *file);
```

Function: writes sequence of characters.

Result: last written character or **EOF**

Example:

```
#include <stdio.h>
int main(void)
{
     ...
     char *list = "Index";
     FILE *dok = fopen("DOK1.DOC","rt+");
     fputs(list, dok);
     ...
}
```

fscanf – reading from file

- Function ***fscanf*** works similarly to ***scanf***, but it requires additional argument – file stream reference.
- It end its work when whole text has been formatted.
- It returns **EOF** if there is any error during text transformation or end of file has been reached. Otherwise it returns the number of formatted and written characters.
- Function **fscanf**

Def:
```
int fscanf(FILE *file, const char *format,
           pointer, pointer, ...);
```

Function: reads sequences of characters and convert them into binary format (just like scanf).

Result: number of written characters or EOF.

fscans – reading – example

- **Example:**

```
#include <stdio.h>
int main(void)
{
    ...
    int numbers;
    float price;
    FILE *good = fopen("INVENTORY.DOC","rt");
    ...
    fscanf(good, "%d%f", &number, &price);
    ...
}
```

fprintf – writing to file

- Function ***fprintf*** works similarly to *printf*, but it writes data to file (so additional argument defining file is required).
- It returns negative number if there was any error. Otherwise it returns the number of written characters.

```
int fprintf ( FILE *file, const char *format,
 statement, statement, ...);
```

Function: writes sequence of characters (like **printf**).

Result: number of written bytes or **EOF**

fprintf – example

- **Example:**

```
#include <stdio.h>
int main(void)
{
    ...
    int currency_code = 15;
    float price = 0.23547;
    FILE *prices_table = fopen("STOCK_EXCHANGE.TAB","wt");
    ...
    fprintf(prices_table, "\n%3d\t%8.3f",
    currency_code, price );
    ...
}
```

fread – reading from file

- Function ***fread*** reads from stream *file* into table given as *pointer* at maximum *number* of objects having specific *size.*
- Function ***fread*** should be used to read data which have been written using ***fwrite***.

```
int fread (pointer,  int size, int number,  FILE *file);
```

Function: reads *number* of data, each having specific *size*.

Result: number of read objects or o

Example:

```
double prices[50];
...
fread(prices, sizeof(double), 50, fp);
// it will copy 50 values of type double from file
// into array prices
```

fread – example

- **Example:**

```
#include <stdio.h>
int main(void)
{
     ...
     struct book
     {
          char author[25];
          char title[50];
     } books[100];
     ...
     FILE *store = fopen("MAGAZINE.DOC","rt");
     fread(books, sizeof(book), 100, store);
     ...
}
// it will copy 100 objects type book from file into
// array books
```

fread – example

```
void main(void)
{
     int number_of_reads;
     struct position{
          double coordinates[2];
          double height;
     } path[1000];
     ...
     FILE *tour = fopen("W1.DAT","rb");
     // read how many elements the table has
     fread(&number_of_reads, sizeof(int), 1, tour);
     // reads the elements and put them into tour
     fread(path, sizeof(position), number_of_reads, tour);
     ...
}
```

fwrite – writing to files

- Function **fwrite** writes *number* of objects having specific *size*, from array *pointer* into stream *file*.

```
int fwrite (pointer, int size, int number, FILE *file);
```

Function: writes *number* of data each having its *size.*

Result: number of written data or 0.

Example:

```
double prices[50];
...
fwrite(prices, sizeof(double), 50, fp);
// it will write the 50 elements type double from prices
// array into file
```

fwrite – example

- **Example:**

```
#include <stdio.h>
int main(void)
{
    ...
    long double measures[row][col];
    ...
    FILE *archive= fopen("ARCH.TAB","wb");
    fwrite(measures, sizeof(measures), 1, archive);
    ...
}
```

Examples

Different file operations

Files – Example 1

```
#include <stdio.h>
// analyze file Data.txt containing integer number: even one writes into
// Even.txt, odd number into Odd.txt

int main (int n, char* TS[ ]){ // Even / odd
     FILE *Data, *Even, *Odd;
     int Number;

     Data = fopen ("Data.txt", "rt");
     Even = fopen ("Even.txt", "wt");
     Odd = fopen ("Odd.txt", "wt");

     if (Data == NULL || Even == NULL || Odd == NULL) {
          printf ("\nFile could not be read.\n\n");
          return 0;
     }
     // how many number are in Data.txt? :
     fscanf(Dane, "%d", &Liczba);
```

Files – Example 1

```
    while ( feof(Data) == 0 ){
        if( Number & 1)
            fprintf(Even, "%d ", Number);
        else
            fprintf(Even, "%d ", Number);

        fscanf(Data, "%d", & Number);
    }

    fcloseall();
    printf ("\nEnd.\n\n");
    return 0;
}
```

Files – Example 2

- Input file contains a sequence of integer numbers separated by #. Program opens such file and writes the numbers to files *.fir and *.sec

```
#include <stdio.h>
#include <string.h>

int main (int n, char* TS[ ]) {// split using #
     char Ntxt[64];
     char Nfir[64];
     char Nsec[64];

     FILE *Ptxt, *Pfir, *Psec;

     bool End = false, Which = true;
     int Sign;
     int Dot;
     char *Ptr;
```

Files – Example 2

```
if ( n < 2 ){
     printf ("\nNo name of file.\n" "Run program with a parameter"
          "being file name.\n\n");
     return 0;
}
strncpy(Ntxt, TS[1], 58);
strncpy(Nfir, TS[1], 58);
strncpy(Nsec, TS[1], 58);
// search for first '.' and calculates length counting from the beginning.
if ((Ptr = strchr(Ntxt, '.')) != NULL)
     Dot = Ptr - Ntxt;
else {
     printf("\nWrong file number.\n\n");
     return 0;
}
// adds new suffix keeping the old file name
strcpy(Nfir + Dot + 1, "fir");
strcpy(Nsec + Dot + 1, "sec");

Ptxt = fopen (Ntxt, "r");
Pfir = fopen (Nfir, "w+");
Psec = fopen (Nsec, "w+");
```

Files – Example 2

```
if (Ptxt == NULL || Pfir == NULL || Psec == NULL) {
     printf ("\nFile could no be opened\n\n");
     return 0;
}
while ( !End ){
     // reads input file sign by sign
     Sign = fgetc(Ptxt);

     // writes signs: first to one file, second to another, and so on
     if ( !(End = (feof(Ptxt) != 0)))
          if (Sign == '#’)
               Which = !Which;
          else
               if (Which)
                    fputc(Sign, Pfir);
               else
                    fputc(Sign, Psec);
}
printf ("\nEnd.\n\n");
return 0;
}
```

Files – Example 3

- Program makes a list of employees (max 50). Each one is described using structure containing surname and salary. There are the following options:
  - R : reads number employees and array of structures describing them from a specific file,
  - N : new employee – reads data and put them into next table element,
  - W : shows info about all employees,
  - Z : writes number of employees and array of structures describing them info specific file,
  - K : quit.

Files – Example 3

```
#include <stdio.h>
struct Employee{
     char Surname[32];
     double Salary;
};

void From_file(Employee Tab[], int& how_many){
     FILE* file;
     char name[16];
     printf("Enter file name to read: ");
     fflush(stdin);
     scanf("%15s", name);
     file = fopen(name,"rt");
     if (file == NULL) {
          printf("No such file.");
          return;
     }
     fscanf(file,"%d",&how_many); // number of employees
     fread(Tab, sizeof Employee, how_many, file);
     fclose(file);
}
```

Files – Example 3

```
void To_file(Employee Tab[], int how_many){
     FILE* file;
     char name[16];
     printf("Enter file name to write : ");
     fflush(stdin);
     scanf("%15s", name);

     plik = fopen(name,"wt");

     // number of employees
     fprintf(plik,"%d",ile);

     // array of employees
     fwrite(Tab, sizeof Employee, how_many, file);
     fclose(file);
}
```

Files – Example 3

```
void New_em(Employee Tab[], int& how_many) {
     if (how_many == 50) {
          printf("Table is full.\n");
          return;
     }
     printf("Enter surname : ");
     fflush(stdin);
     scanf("%31s",Tab[how_many].Surname);
     printf("Enter salary : ");
     scanf("%lf",&Tab[how_many++].Salary);
}

void ShowAll(Employee Tab[], int how_many) {
     if (how_many == 0) {
          printf("Empty table.\n");
          return;
     }
     for(int i = 0; i < how_many; i++)
          printf("Employee %d : %s , %.2lf\n", i, Tab[i].Surname, Tab[i].Salary);
}
```

Files – Example 3

```
int main(int argc, char* argv[]){
     Employee TaPa[50];
     bool go_on = true;
     int how_many = 0;
     char option;
     while (go_on) {
          printf("Choose option [R,N,W,Z,K] : ");
          fflush(stdin);
          scanf("%c",&option);
          switch(opcja & 0x5F){
               case 'R' :From_file(TaPa, how_many); break;
               case 'N' :New_em(TaPa, how_many); break;
               case 'W' :ShowAll(TaPa, how_many); break;
               case 'Z' :To_file(TaPa, how_many); break;
               case 'K' :go_on = false; break;
               default : printf("Wrong option.\m");
          }
     }
     return 0;
}
```

Questions?