Fundamentals of Programming 1

Files

Arkadiusz Chrobot

Department of Information Systems

January 7, 2025

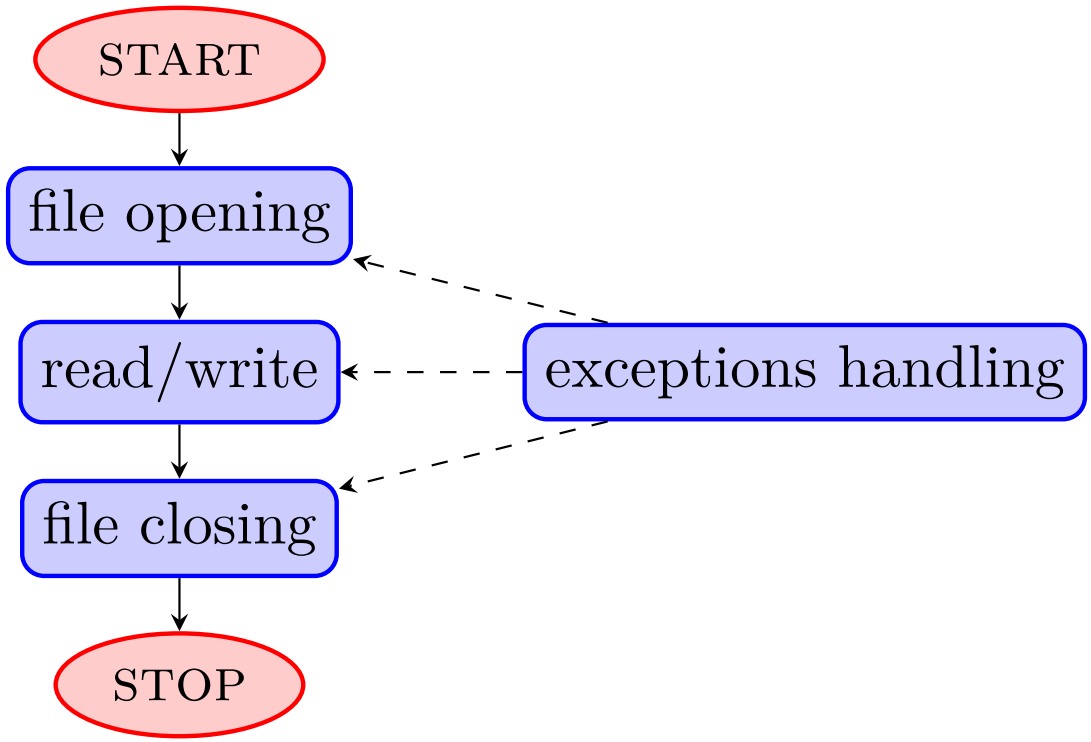
Outline

Introduction

Complex computer programs tend to generate and store huge amounts of data. The size of the data often exceeds the capacity of the RAM. Moreover, after the power supply is switched off all the information stored in the RAM vanishes. To store the valuable data of significant size efficiently and durably a mass storage devices have to be used. The problem with such a solution is that every type of a mass storage device has a different method for accessing the data that it stores. To unify the access to the information stored in different types of mass storages and in the RAM, a special data structure called a *file* was created.

Using a File

Using a file can be described by a following flowchart:



An exception is any situation that requires a special handling by the program because it may lead to errors in data processing. As an example please consider a situation in which program tries to read a file that does not exist.

File Handling in the C Language

The C language offers two ways of file handling: a low-level one with the use of so-called file descriptors and a high-level one that involves using so-called streams. In this lecture only the second way is discussed, because it offers many facilities, like automatic buffering of data that are read from or written to the file. A stream is a variable of the `FILE *` type[1]. All functions necessary for using the streams are declared in the `stdio.h` header file, that has to be included in the program.

[1] To be more specific, the variable is a pointer to the structure of the `FILE` type.

Standard Streams

Variables of the `FILE *` type are used in the C language not only for file handling. They have a few other applications. If the `stdio.h` header file is included in the program, then three variables of such a type become instantly available. Names of those variables are: `stdin`, `stdout` and `stderr`. The first one is also called the *standard input stream* and it is by default associated with the keyboard. The second one is referred to as *standard output stream* and by default is associated with the screen. The third one is also associated by default with the screen and it is described as the *standard error stream*. It is used for displaying all diagnostics messages, i.e. those which are generated as a result of exceptions handling.

File Opening

The file opening is an activity that starts file handling in a program. In the C language it associates a stream variable with the file. The opening is performed by the `fopen()` function which takes two arguments. The first one is a path to the file, and the second one consists of the following characters:

Mode	Description
`r`	Read only opening. The file pointer points to the beginning of the file.
`r+`	Read and write opening. The file pointer points to the beginning of the file.
`w`	Creating and opening the file for writing only. If the file exists then its content will be erased. The file pointer points to the beginning of the file.
`w+`	As above, but it is possible to read the file.
`a`	Opening for appending. The file pointer points to the end of the file. If the file doesn't exist, it will be created.

File Opening

Continuation

Mode	Description
`a+`	As previously, but reading the file is possible.
`b`	The opened file is interpreted as a binary file.

The mode characters from the table may be combined, provided they are not mutually exclusive (for example writing and appending mode cannot be used together). The `b` mode informs the function that it opens a binary, not a text file. Most of contemporary operating systems rarely need such information, Windows is an exception. The value returned by the `fopen()` function has to be stored in the variable of the `FILE *` type. If the variable has a `NULL` value after the function returns then it means that some exception has occurred and opening of the file failed.

Exceptions Handling

If a function, performing a file operation, signals an exception occurrence, then the code of the exception can be recognized with the use of the `ferror()` function. The function takes a variable of `FILE *` type as an argument and returns zero if no exception has occurred or a nonzero value, otherwise. The exception code can be zeroed out with the use of the `clearerr()` function which also takes the stream as an argument. However, this function should be used with caution because it also clears the end-of-file flag. Both functions are used for handling exceptions raised by functions other than `fopen()`. The textual description of an exception can be displayed on the screen with the use of the `perror()` function. The function takes as an argument a string which is the name of the function that caused the exception and prints a message that explains what happened. Both the functions (`ferror()` and `perror()`) should be called right after the invocation of a function that is likely to rise an exception.

Writing Files

Writing to a Text File

Writing to a text file[2] can be accomplished with the use of the `fputc()` function which takes two arguments — the character or its ASCII code (as a value of `int` type) and the stream associated with the file. The function returns the saved character or the `EOF` (END-OF-FILE) constant in case of a failure. A string of characters (except for the `'\0'` character) may be written to the file with the use of the `fputs()` function, which takes two arguments — the array of characters containing the string and the stream. The function returns the number of characters saved to the file or the `EOF` in case of a failure.

[2] A text file is a file that contains characters coded in the ASCII or similar code and can be edited with any text editor.

Writing Files

Writing to a Text Files — Continued

The data of different types can be converted to a string and saved to a file with the use of the `fprintf()` function, which is invoked similarly to the `printf()`, but as a first argument it takes a stream associated with file. The function returns the number of characters written to the file[3]. It is also applied when a message should be written to the standard error stream.

[3]The `printf()` returns the number of displayed characters, but usually this information is ignored in a program.

Writing Files

Writing to a Binary File

The `fwrite()` function is used for saving data to a binary file[4]. The function takes four arguments – pointer to the variable that contains the data to be written[5], the size of a single data item in the buffer, the number of data items to be written and the stream. It returns the number of saved data items. If it is smaller than the value of the third argument then a writing exception has occurred.

[4] The binary file contains data written in the same form as they are stored in the RAM, in other words as bit patterns, which usually are not ASCII characters.

[5] Such a variable is usually called a *buffer*.

Reading Files

The basic issue that has to be solved when reading a file is when to finish this activity. The C language standard defines the `feof()` function for streams, which signals reaching the end of file. It takes a stream as an argument and returns an `int` number. If the number is not a zero then it means that there is no more data in the file and the reading should be ended. The `feof()` function is usually invoked in condition expressions of the condition controlled loops, which is demonstrated in the example programs.

Reading Files

Reading a Text File

The `fgetc()` function reads a single character from a text file. It takes a stream as an argument and returns an `int` value which is an ASCII value of the character or the `EOF` constant if the end of file has been reached. The `fgets()` function reads a string from the text file. It takes three arguments — an array of characters where the string should be stored, the number of elements of the array and a stream associated with the file. The function reads at least one character less than the array length. If the data were saved in the file with the use of the `fprintf()` function then they could be read with the use of the `fscanf()` function and an appropriate formatting string as its argument. Aside from the formatting string the function has to take as many arguments as the formatting string indicates. Those additional arguments are addresses of variables in which the read data should be stored. The function returns the number of items from the file that it successfully matched to the formatting string and assigned to the buffers (aforementioned variables). In case of failure it returns the `EOF` constant.

Reading Files

Reading a Binary File

If the data were written to the file with the use of the `fwrite()` function then they can be read by the `fread()` function. The function takes the same arguments as the `fwrite()`, but they have a different meaning. The data from file are placed in the variable which address is passed as a first argument of the function. The `fread()` function returns the number of actually read data items from the file. If the number is smaller than the value of the third argument of the function then there are no more data to read in the file or an exception has occurred.

File Closing

The `fclose()` function closes an opened file. It takes a stream associated with the file and returns a nonzero value in case of a failure.

Other File Handling Functions

A *file pointer* is associated with every opened file that is handled with the use of streams. It has a similar role as an index in an array. The file pointer has been already mentioned in the `fopen()` modes table. Functions that read and write files increase the value of file pointer indirectly and automatically. If the value of the file pointer always grows then the file content is accessed *sequentially*. The *random* access allows the program to increase or decrease the value of the file pointer, which means that the file may be used in similar fashion as an array. The value of the file pointer can be modified with the use of the `fseek()` function. It takes three arguments — a stream associated with the file, the offset (a `long int` number), and one of the following constants: `SEEK_SET` — the offset will be stored in the file pointer, `SEEK_CUR` — the offset will be added to the current value of the file pointer and `SEEK_END` — the offset will be added to the currently last position in the file. The function returns `-1` in case of a failure.

Other File Handling Functions

The `rewind()` function changes the value of the file pointer so it points at the beginning of the file. It takes a stream as an argument and does not return any value. The `ftell()` function returns the current value of the file pointer (a `long int` value). The value of the pointer may also be modified with the use of `fgetpos()` and `fsetpos()` functions, but they are not discussed any further in this lecture.

Other File Handling Functions

While being written the data do not go directly to the file, but may be stored in a special buffers created automatically in the RAM. The reason for that is that the access time to a mass storage device is usually very long, so the program postpones the write operations as long as it can. For emptying the buffers the `fflush()` function is used. The function takes a stream as an argument and returns zero in case of a success or the `EOF` constant in case of a failure. Clearing the buffers does not mean that the data are immediately saved to the mass storage device, because also the operating system may buffer them. The `fclose()` function empties the same buffers as the `fflush()` function, so there is no need for calling the latter before the former. There are other functions in the standard C library that make it possible to manage the buffers but they are not discussed in this lecture.

File Management Functions

In the standard C library are also defined functions for the file management. A few of them are described here, in this slide. The `remove()` function deletes files and directories. As an argument it takes a string which is a name (path) of the file or directory to be removed. In case of failure it returns `-1`. The `rename()` function changes the name of the file or its location in the mass storage device. It takes two arguments, both are strings. The first one is the old name or location of the file, the second one is the new name or location of the file. The function returns `0` in case of a success or `-1` in case of failure.

First Example

Writing a Single Characters to a Text File

The first example is a program that saves 100 pseudorandom lowercase letters to a text file and then it reads them from the file and displays on the screen. The program also demonstrates how the exceptions can be detected and reported. However, the exception handling in this program is not perfect. For example an opened file is not closed when a writing exception occurs.

First Example

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define LENGTH 100
```

First Example

Comment

The code presented in the previous slide contains the preprocessor directives that include header files to the program. Aside from the `stdio.h` file also the `stdlib.h` and `time.h` files are included, because the program uses the PRNG. The `LENGTH` constant defines the number of elements in arrays of characters that store diagnostics messages about exceptions.

First Example

```
void display_exception_message(int code)
{
    char exception_description [][LENGTH] = {
        "A file opening for writing exception.",
        "A file writing exception.",
        "A file closing exception.",
        "A file opening for reading exception."
    };
    fprintf(stderr,"%s\n",exception_description[-code-1]);
}
```

First Example

Comment

The function in the previous slide displays messages that describe exceptions that may occur in other functions performed by the program. The messages are stored in the `exception_description` array. They are displayed by the `fprintf()` function, which takes as its first argument the standard error stream. To calculate the index of the exception message the `display_exception_message()` function uses an exception code passed by its parameter. If another function returns `0` then it means that its task has been successfully completed. The values of the exception code that signal a failure start with the `-1`. To get a correct array index from an exception code the function has to change the sign of the code and subtract one. The `-1` exception code means that opening of file for writing was unsuccessful, `-2` means that there was a problem with writing the file etc.

First Example

```c
int fill_file(char *file_name)
{
    FILE *file = NULL;
    srand(time(0));
    file = fopen(file_name,"w");
    if(file==NULL)
        return -1;
    int i;
    for(i=0; i<100; i++)
        if(fputc('a'+rand()%('z'-'a'+1),file)==EOF)
            return -2;
    if(fclose(file)!=0)
        return -3;
    return 0;
}
```

First Example

Comment

The `fill_file()` function writes 100 lowercase letters to a text file. The name of the file is passed to the function by the parameter. The function returns an exception code. If the code is zero, then it means that the function completed successfully its task. Any nonzero value of the code means that some exception has occurred. First, the function initializes the stream variable and the PRNG. Next, it opens the file. The operation creates the file if it doesn't exist or erases its content, if it does. Should the operation fail the function returns `-1` and terminates. Otherwise it chooses randomly 100 lower-cases and writes them one by one to the file. Each time it checks if the writing exception has occurred. If that happened the function would return `-2` and quit. After the function finishes the loop in which the letters are saved to the file, it closes the file. If the operation fails the function returns `-3`, otherwise it returns `0`.

First Example

```
int read_file(char *file_name)
{
    FILE *file = fopen(file_name,"r");
    if(file==NULL)
        return -4;
    while(!feof(file)) {
        int data_from_file = fgetc(file);
        if(data_from_file!=EOF)
            printf("%c ",data_from_file);
    }
    puts("");
    if(fclose(file)!=0)
        return -3;
    return 0;
}
```

First Example

Comment

The `read_file()` function reads characters from the file and displays them on the screen. The name of the file is passed to the function by the parameter. First, the function opens the file for reading. In case of failure, it returns `-4` and terminates. Otherwise, the function performs the `while` loop inside which it reads the file, character by character, with the use of the `fgetc()` function and displays the letters on the screen. The `!feof(file)` condition in the loop is equivalent to the `feof(file)==0` expression. If inside the loop the `fgetc()` function returns the `EOF` value, then the character will not be displayed on the screen, because it is not a letter, but a character that signals the end of the file. After the loop ends the `read_file()` function closes the file, checking if the operation is successful. If it isn't the function returns a corresponding exception code. Otherwise it returns zero.

First Example

```
int main(void)
{

    int result = fill_file("test.txt");
    if(result<0)
        display_exception_message(result);
    result = read_file("test.txt");
    if(result<0)
        display_exception_message(result);
    return 0;
}
```

First Example

Comment

In the `main()` function, first the `fill_file()` function is invoked and then the `read_file()` function is called. The number returned by each of the functions is stored in the `result` variable. If it is negative the `display_exception_message()` function is invoked with the number as its argument.

Second Example

The second program allows the user to save strings with spaces and other characters to the text file. The writing ends after the user enters the "stop" word. Similarly to the previous example, the exception handling is not perfect.

Second Example

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

#define LENGTH 100
#define SENTENCE_LENGTH 81
```

Second Example

Comment

Aside from the `stdio.h`, also the `string.h` header file is included because functions that perform operations on strings are used in the program. The `LENGTH` constant has the same meaning as in the previous program. The `SENTENCE_LENGTH` constants defines the number of elements in the character array that is used for storing strings that are saved to the file. Eighty is usually the maximal number of characters that can be displayed in one row on the screen.

Second Example

```
void display_exception_message(int code)
{
    char exception_description [][LENGTH] = {
        "A file opening for writing exception.",
        "A file writing exception.",
        "A file closing exception.",
        "A file opening for reading exception."
    };
    fprintf(stderr,"%s\n",exception_description[-code-1]);
}
```

Second Example

Comment

The `display_exception_message()` function is defined in exactly the same way as in the previous example.

Second Example

```
int write_sentences_to_file(char *file_name)
{
    char sentence[SENTENCE_LENGTH] = {'\0'};
    FILE *file=fopen(file_name,"w");
    if(file==NULL)
        return -1;
    while(strncmp(sentence,"stop",SENTENCE_LENGTH-1)!=0)
    {
        scanf("%80[^\n]s",sentence);
        while(getchar()!='\n');
        if(fprintf(file,"%s\n",sentence)!=strlen(sentence)+1)
            return -2;
    }
    if(fclose(file)!=0)
        return -3;
    return 0;
}
```

Second Example

Comment

The `write_sentences_to_file()` functions saves strings entered by user to a text file. The name of the file is passed by the parameter. The `sentence` variable, which is declared at the beginning of the function, is a local variable for storing a single string entered by user. It is initialized with an empty string. The `write_sentences_to_file()` function opens the file for writing. If the operation if successful then in the loop the function reads the strings entered by user and writes them to the file with the use of the `fprintf()` function. The function checks if the writing operation is successful by comparing the number of characters returned by `fprintf()` with the length of the entered string plus one. The internal `while` loop removes the new line character (associated with the `Enter` key) from the standard input stream, so the `scanf()` function can correctly read the next string. After the user enters the "stop" word, the external loop is terminated and the file is closed. If any of the file operations fails the function returns a negative number. Otherwise it returns zero.

Second Example

```
int read_sentences_from_file(char *file_name)
{
    char sentence[SENTENCE_LENGTH] = {'\0'};
    FILE *file = fopen(file_name,"r");
    if(file==NULL)
        return -4;
    while(feof(file)==0) {
        fscanf(file,"%[^\n]s",sentence);
        if(feof(file)==0) {
            puts(sentence);
            while(fgetc(file)!='\n');
        }
    }
    if(fclose(file)!=0)
        return -3;
    return 0;
}
```

Second Example

Comment

The `read_sentence_from_file()` function reads from the file strings saved by the previously presented function and displays them on the screen. The name of the text file is passed to the function by the parameter. In the function there is also declared a local character array which is initialized with an empty string. The function opens the file and then, in a loop, reads its content line by line and displays on the screen. After each call of the `fscanf()` function, the `fgetc()` function is called inside the internal loop, provided that the end of the file has not been yet reached. Its job is to read the new line characters from the file and move the file pointer to the beginning of the next string, so the subsequent `fscanf()` call may read it. As in previously described functions, any exception causes the `read_sentence_from_file()` to return a negative number and terminate. If all operations are completed successfully the function returns zero.

Second Example

```
int main(void)
{
    int result = write_sentences_to_file("test.txt");
    if(result!=0)
        display_exception_message(result);
    result = read_sentences_from_file("test.txt");
    if(result!=0)
        display_exception_message(result);
    return 0;
}
```

Second Example

Comment

The `main()` function is implemented in a similar fashion as in the previous example.

Third Example

The third example saves structures containing pseudorandom coordinates of points in the three dimensional space to a binary file. The program also demonstrates the operation of appending data to an existing file. As in the previous examples the exception handling in the program is simplified.

Third Example

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define LENGTH 100
```

Third Example

Comment

The beginning of the program is the same as in the case of the first program.

Third Example

```
void display_exception_message(int code)
{
    char exception_description[][LENGTH] = {
        "A file opening for writing exception.",
        "A file writing exception.",
        "A file closing exception.",
        "A file opening for appending exception.",
        "A file appending exception."
        "A file opening for reading exception."
    };
    fprintf(stderr,"%s\n",exception_description[-code-1]);
}
```

Third Example

Comment

In the `display_exception_message()` function two additional messages have been added to the `exception_description` array, which inform about exceptions associated with the operation of appending data to the existing binary file.

Third Example

```
struct coordinates {
    double x,y,z;
};
```

Third Example

Comment

The previous slide presents the definition of the type of structures that are written to the binary file.

Third Example

```
int write_to_file(char *file_name)
{
    FILE *file = fopen(file_name,"w");
    if(file==NULL)
        return -1;
    int i;
    for(i=0; i<10; i++) {
        const int RANGE = 20;
        struct coordinates point;
        point.x = rand()%RANGE;
        point.y = rand()%RANGE;
        point.z = rand()%RANGE;
        if(fwrite(&point,sizeof(point),1,file)!=1)
            return -2;
    }
    if(fclose(file)!=0)
        return -3;
    return 0;
}
```

Third Example

Comment

The `write_to_file()` function fills the fields of a structure, declared as a local variable, with the numbers ranging from `0` to `19` (chosen randomly) and it writes the structure to the file with the use of the `fwrite()` function. This activity is repeated 10 times. After each call of the `fwrite()` function the `write_to_file()` function checks what value has returned the former one. It should be equal to the value of the third argument of the `fwrite()`, which defines the number of data items that ought to be written to the file. In this program it is equal one, since only one data item is written to the file by each `fwrite()` call. The file is opened before writing happens and closed after it is finished. The `write_to_file()` function returns a negative number if any exception occurs or zero if all file operations complete successfully.

Third Example

```
int add_to_file(char *file_name)
{
    FILE *file = fopen(file_name,"a");
    if(file==NULL)
        return -4;
    struct coordinates point;
    const int RANGE = 40;
    point.x = rand()%RANGE;
    point.y = rand()%RANGE;
    point.z = rand()%RANGE;
    if(fwrite(&point,sizeof(point),1,file)!=1)
        return -5;
    if(fclose(file)!=0)
        return -3;
    return 0;
}
```

Third Example

Comment

The `add_to_file()` function has a similar task to the `write_to_file()` function, but this time the file is opened for appending. It means that if it exists then its content it not removed and the file pointer is set to its end. The values for the fields of the structure are chosen randomly, ranging from `0` to `39`. The structure is appended to the existing data in the file. Those operations are performed only once.

Third Example

```c
int read_from_file(char *file_name)
{
    FILE *file = fopen(file_name,"r");
    if(file==NULL)
        return -6;
    int i=0;
    struct coordinates point;
    while(fread(&point,sizeof(point),1,file)==1)
        printf("Point %d -- x: %f, y: %f z: %f\n",
                            ++i,point.x, point.y, point.z);
    if(fclose(file)!=0)
        return -3;
    return 0;
}
```

Third Example

Comment

The `read_from_file()` function opens the binary file for reading, reads all structures from the file and displays their content on the screen. The name of the file is passed to the function with the use of the parameter. Values of fields of a single structure are displayed in one row. After the function finishes reading it closes the file. Reaching the end of the file is tested by comparing the value returned by the `fread()` function with the value of its third argument. If they differ then it means that there are no more data in the file to read. The exception handling is implemented similarly to previously presented functions.

Third Example

```
int main(void)
{
    srand(time(0));
    int result = write_to_file("data.bin");
    if(result!=0)
        display_exception_message(result);
    result = add_to_file("data.bin");
    if(result!=0)
        display_exception_message(result);
    result = read_from_file("data.bin");
    if(result!=0)
        display_exception_message(result);
    return 0;
}
```

Third Example

Comment

The `main()` function is defined in a similar way as in the previously described examples, but there are also new elements. They are: the invocation of the function that appends a new structure to the end of the file and initialization of the PRNG.

Forth Example

The forth and last example program writes lowercase letters and pseudorandom natural numbers to a text file. Those values are divided into lines that contain one letter and three numbers. The values in a line are separated by spaces. Also, a slightly different approach to the exception handling is used in the program. It is more strict than in the previously described programs, but also has drawbacks. For example the function that reads the file is not informed if the function that writes the file successfully finished its job.

Forth Example

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
```

Forth Example

Comment

The same header files are included to the program as in the previous example.

Forth Function

```
void write_to_file(char file_name[])
{
    srand(time(0));
    FILE *file = fopen(file_name,"w");
    if(file!=NULL) {
        int i;
        for(i=0; i<5; i++) {
            fprintf(file,"%c %d %d %d\n",
                (char)('a'+rand()%('z'-'a'+1)),
                        rand()%5, rand()%7, rand()%3);
            if(ferror(file)) {
                perror("fprintf()");
                break;
            }
        }
        if(fclose(file))
            perror("fclose()");
    }
}
```

Forth Function

Comment

The `write_to_file()` function initializes the PRNG, opens a text file (which name is passed by the parameter) for writing and writes five lines to the file. Each line contains a pseudorandom lowercase letter and three natural number chosen randomly from different ranges. Please notice, that no writing operation is performed if the file is not opened correctly. The result of a writing operation is tested with the use of the `ferror()` function after each call to the `fprintf()` function. The file is being closed regardless if the writing operations finished successfully, or not. However, the closing operation is not performed if the file was not opened successfully. The messages about exceptions are displayed by the `perror()` function.

Forth Example

```
void read_from_file(char file_name[])
{
    char letter;
    int first_number, second_number, third_number;
    FILE *file = fopen(file_name,"r");
    if(file!=NULL) {
        while(!feof(file)) {
            fscanf(file,"%c %d %d %d\n",&letter,&first_number,
                                                &second_number,&third_number);
            if(ferror(file)) {
                perror("fscanf()");
                break;
            }
            printf("%c %d %d %d\n",letter,first_number,
                                                second_number,third_number);
        }
        if(fclose(file))
            perror("fclose()");
    }
}
```

Forth Example

Comment

The `read_from_file()` function reads a text file which name is passed by the parameter and displays the content of the file on the screen. The file is first opened by the function for reading and next it is read in a loop, line by line, with the use of the `fscanf()` function. Please observe, that the formatting string used in the `fscanf()` function call is the same as in the `fprintf()` function invocation from the previously described function. Also, the exception handling is implemented similarly.

Forth Example

```
int main(void)
{
    write_to_file("numbers.txt");
    read_from_file("numbers.txt");
    return 0;
}
```

Forth Example

Comment

The `main()` function in this program is very simple. It contains only the invocations of the two previously described functions, that don't return any values.

Thanks

Many thanks to Grzegorz Łukawski, PhD and Leszek Ciopiński, PhD for helping me to complete the Polish version of this slides.

Questions

?

The End

Thank You for Your attention!