Maciej Sobieraj

Lecture 12

Outline

1. Connecting to the real world: files and streams
    1. **Reading from the stream**
    2. Writing to the stream
    3. Dealing with the stream's position
2. Quiz

fgetc() function

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

- the function name comes from the words file get character;
- the function expects one parameter of type FILE *; the parameter must be a pointer to a stream opened for reading or updating;
- the function attempts to read one character (byte) from the stream identified by the parameter; if possible, the function returns th code of the retrieved character as its result

fgetc() function

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

- if the attempt fails (e.g. because the current file position is already located after the last character in the file), fgetc returns the value of EOF (-1) and the file position is not changed;
- the function might be used for reading characters from a text file as well as reading bytes from a binary file.

fgetc() function

- There’s an additional function with the following prototype:
  - int getchar(void);
- and it causes the same effects as the following invocation:
  - fgetc(stdin);
- The first function is used for reading a single character from the ***stdin*** stream.

fgetc() function

- fgetc() function: reading one character from the stream

```c
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[]) {
    FILE *inp;
    int  chr;

    /* check if there is one argument */
    if(argc != 2) {
        printf("usage: show file_name\n");
        return 1;
    }

    /* check if we are able to open the input file */
    if((inp = fopen(argv[1],"rt")) == NULL) {
        printf("Cannot open the file %s\n", argv[1]);
        return 2;
    }
```

fgetc() function

- fgetc() function: reading one character from the stream

```
    /* we will try to read the file char by char and print the chars to screen */

    while((chr = fgetc(inp)) != EOF)
        printf("%c",chr);

    /* it's time to close the stream */
    fclose(inp);
    return 0;
}
```

fgets() function

- fgets() function: reading one string from the stream

```
char *fgets(char *str, int maxsize, FILE *stream);
```

- the function name is from the words file get string;

fgets() function

- fgets() function: reading one string from the stream

```
char *fgets(char *str, int maxsize, FILE *stream);
```

- the function expects the following three parameters:
  - *str*: **a pointer to a string** in which *fgets* **will store** one line taken from the stream;
  - *maxsize*: the **maximum number of characters** that can be safely stored inside the *str*;
  - *stream*: a **pointer to the stream opened** for reading or **updating**;

fgets() function

```
char *fgets(char *str, int maxsize, FILE *stream);
```

- **the function attempts to read one line** of text from the stream; if it succeeds, the function stores at most *maxsize* characters in the string pointed to by *str*; if the file contains lines of a greater length, they will be read part by part;
- if the reading is successful, the function **returns the value** of the *str* parameter, and the current position of the file is moved to the place after the last retrieved character;
- otherwise the function returns NULL as a result and current file position is not changed.

fgets() function

- There’s an additional function with the following prototype:
  - **char*** gets(**char** *str);
- which causes the same effects as the following invocation:
  - fgets(str, INT_MAX, stdin);
- where *INT_MAX* is a symbolic constant representing the maximum value of type *int*.

fgets() function

```c
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[]) {
        FILE *inp;
        char line[128];
        if(argc != 2) {
                printf("usage: show2 file_name\n");
                return 1;
        }
        if((inp = fopen(argv[1],"rt")) == NULL) {
                printf("Cannot open the file %s\n", argv[1]);
                return 2;
        }
        while((fgets(line,sizeof(line),inp)) != NULL)
                printf("%s",line);
        fclose(inp);
        return 0;
}
```

fread() function

- **fread() function: reading bytes from the stream**

```
int fread(void *mem, int size, int count, FILE *stream);
```

- the function name derives from the words **f**ile **read**;
- the function expects the following four parameters:
  - *mem*: a **pointer to a memory** in which fread will store a portion of bytes read from the stream;
  - *size*: the **size** (in bytes) of the portion to be read;
  - *count*: the **number of portions** to be read;
  - *stream*: a **pointer to the stream** opened for reading or updating;

fread() function

- **fread() function: reading bytes from the stream**

```
int fread(void *mem, int size, int count, FILE *stream);
```

- the function attempts to read *size * count* bytes from the stream; if it succeeds, the function stores the read bytes in the memory pointed to by *mem*;
- the function returns the number of successfully read portions; it may, but doesn't have to, be equal to the *count* value; a value of 0 says that the function was unable to read any portion; the current position of th is moved to the place after the last read byte.

fread() function

- *fread* uses two parameters to specify the size of the data to read: *size* and *count*. How do we deal with it?
- Suppose that we want to retrieve the value from the *input* stream and store the bytes in the *number* variable. The following declarations apply:
  - **int** number;
    FILE *input;
- The reading could be performed in two equivalent ways:
  - fread(&number, **sizeof**(**int**), 1, input);
    fread(&number, 1, **sizeof**(**int**), input);

fread() function

- In the first case, *fread* will read one portion of **sizeof**(**int**) size.
- In the second case, *fread* will read **sizeof**(**int**) portions of **1** byte each.
- In both cases, the *fread* attempts to read *sizeof(int)* bytes, but the results returned by the function will differ. Since *fread* returns the number of correctly read portions, it will be equal to *1* in the first case and to *sizeof (int)* in the second, as long as everything is correct.

fread() function

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

int main(int argc, char *argv[]) {
        FILE                    *inp;
        unsigned char           buffer[1024]; /* 1 kilobyte */
        int             i, read;

        if(argc != 2) {
                printf("usage: show3 file_name\n");
                return 1;
        }
        if((inp = fopen(argv[1],"rb")) == NULL) {
                printf("Cannot open the file %s\n", argv[1]);
                return 2;
        }
        do {
                read = fread(buffer,1,sizeof(buffer),inp);
                for(i = 0; i < read; i++)
                        printf("%02X",buffer[i]);
        } while (read > 0);
        fclose(inp);
        return 0;
}
```

fscanf() function

- We need a function that can read a string representing any value and convert it directly to internal representation.
- Such a function exists and we've used it already, but in a form that allowed us to read the data from the *stdin* stream only.
  - **int** scanf(**char** * format, ...);

fscanf() function

- **fscanf() function: formatted reading from the stream**

```
int fscanf(FILE *stream, const char *format, ...);
```

- This function expects the following parameters:
  - *stream*: a **pointer to the stream** opened for reading or updating;
  - *format*: a **pointer to a string** describing what data should be read from the stream;
  - ... : a **list of pointers to variables** to be assigned with values read from the data stream.

fscanf() function

```
int fscanf(FILE *stream, const char *format, ...);
```

- The function returns the **number of values correctly read** from the stream.
- The invocation like this:
  - scanf("%d", &number);
- is the same as this:
  - fscanf(stdin, "%d", &number);

fscanf() function

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main(int argc, char *argv[]) {
        int   numbers[1000];
        int   i,aux;
        int   numbersread = 0;
        int   swapped;
        FILE        *inp;
        if(argc != 2) {
                printf("usage: intsort input_file\n");
                return 1;
        }
        if((inp = fopen(argv[1],"rt")) == NULL) {
                printf("Cannot open %s - %s\n",argv[1],strerror(errno));
                return 2;
        }
        while(fscanf(inp,"%d",&numbers[numbersread]) == 1) {
                numbersread++;
                if(numbersread == 1000)
                        break;
}
```

fscanf() function

```
    fclose(inp);
    if(numbersread == 0) {
        printf("No numbers found in the files %s\n",argv[1]);
        return 3;
    }
    do {
        swapped = 0;
        for(i = 0; i < numbersread - 1; i++)
            if(numbers[i] > numbers[i + 1]) {
                swapped = 1;
                aux = numbers[i];
                numbers[i] = numbers[i + 1];
                numbers[i + 1] = aux;
            }
    } while(swapped);
    printf("The sorted values: ");
    for(i = 0; i < numbersread; i++)
        printf("%d ",numbers[i]);
    printf("\n");
    return 0;
}
```

Outline

1. Connecting to the real world: files and streams
    1. Reading from the stream
    2. **Writing to the stream**
    3. Dealing with the stream's position
2. Quiz

fputc() function

- **fputc() function: writing one character to the stream**

```
int fputc(int chr, FILE *stream);
```

- the function name comes from the words *file put character;*
- the function expects two parameters:
  - *chr:* a **code of the character** (or the character itself) to be output to the stream;
  - *stream*: a **pointer to the stream** opened for writing or updating;

fputc() function

```
int fputc(int chr, FILE *stream);
```

- if the function succeeds, it **returns the** *chr* **character code** as its result; it will always be a number between 0 and 255; the current file position moves one byte toward the end of the file;
- if the function fails (e.g. because of insufficient disk space), *fputc* returns the value of *EOF* (-1) and the file position is not changed;
- this function can be used for writing characters to a file or bytes to a binary file.

fputc() function

- There's an additional function with the following prototype:
  - **int** putchar (**int** chr);
- which works exactly in the same way as:
  - fputc(chr, stdout);
- and is used for writing one character/byte to the *stdout* stream.

fputc() function

```c
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[]) {
    FILE *inp, *out;
    int   chr;

    /* check if we've got exactly two arguments */
    if(argc != 3) {
        printf("usage: copyc source_file target_file\n");
        return 1;
    }

    /* check if we are able to open the source file */
    if((inp = fopen(argv[1],"rb") )== NULL) {
        printf("Cannot open %s\n", argv[1]);
        return 2;
    }
```

fputc() function

```
/* check if we are able to open the target file */
if((out = fopen(argv[2],"wb")) == NULL) {
        printf("Cannot create %s\n", argv[2]);
        fclose(inp);
        return 3;
}

/* we are going to read char by char until we reach EOF */
while((chr = fgetc(inp)) != EOF)
        if(fputc(chr, out) == EOF)
                break;
fclose(inp);
fclose(out);
return 0;
}
```

fputs() function

- **fputs() function: writing a string to the stream**

```
int fputs(char *string, FILE *stream);
```

- the function name derives from the words *file put string*
- the function expects two parameters:
- *string*: a **pointer to the string** to be written to the stream; note: the function will **not** implicitly add a \n character at the end of the string (in contrast to puts);
- *stream*: a **pointer to the stream opened** for writing/updating

fputs() function

```
int fputs(char *string, FILE *stream);
```

- the function attempts to **write the content of the** *string* to the *stream*
- if the function is successful, it returns a **non-negative number** and the current position of the file is moved towards the end of the file
- in the event of an error the function returns *EOF* as a result; the current file position is unchanged;
- the function is definitely not intended to write data to binary files as it is not possible to write a byte of val

fputs() function

- Here is a function (you already know it) with the following prototype:

```
int puts(char *string);
```

which is an equivalent of:

```
fputs(string, stdout);
```

fputs() function

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

int main(int argc, char *argv[]) {
        FILE        *inp, *out;
        char         line[128];

        if(argc != 3) {
                printf("usage: copys source_file target_file\n");
                return 1;
        }

        if((inp = fopen(argv[1],"rt")) == NULL) {
                printf("Cannot open %s\n", argv[1]);
                return 2;
        }
```

fputs() function

```
if((out = fopen(argv[2],"wt")) == NULL) {
        printf("Cannot create %s\n", argv[2]);
        fclose(inp);
        return 3;
}
while((fgets(line,sizeof(line),inp)) != NULL)
        if(fputs(line,out) == EOF)
                break;
fclose(inp);
fclose(out);
return 0;
}
```

fwrite() function

- **fwrite() function: writing bytes to the stream**

```
int fwrite(void *mem, int size, int count, FILE *stream);
```

- the function name comes from the words *file write*;
- the function expects four parameters:
  - mem: a pointer to the memory area to be written to the stream;
  - size: the size (in bytes) of one memory portion being written;
  - count: the number of portions intended to be written;
  - stream: a pointer to the stream opened for writing or updating;

fwrite() function

```
int fwrite(void *mem, int size, int count, FILE *stream);
```

- the function attempts to write (*size * count*) bytes from *mem* to the *stream*;
- the function returns the number of successfully (actually) written portions and the current position of the file is moved toward the end of the file; the result may differ from the count value, due to some errors preventing successful writing;
- the function is ideal for writing to binary files, but you can use it to create text files too if you provide the appropriate handling of the endline characters.

fwrite() function

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

int main(int argc, char *argv[]) {
        FILE        *inp, *out;
        char        buffer[16384]; /* 16 kilobytes */
        int   read, written;

        if(argc != 3) {
                printf("usage: copyw source_file target_file\n");
                return 1;
        }
        if((inp = fopen(argv[1],"rb")) == NULL) {
                printf("Cannot open %s\n", argv[1]);
                return 2;
        }
```

fwrite() function

```
if((out = fopen(argv[2],"wb")) == NULL) {
        printf("Cannot create %s\n", argv[2]);
        fclose(inp);
        return 3;
}
do {
        read = fread(buffer,1,sizeof(buffer),inp);
        if(read)
                written = fwrite(buffer,1,read,out);
} while (read && written);
fclose(inp);
fclose(out);
return 0;
}
```

fprintf() function

- **fprintf() function: formatted writing to the stream**

```
int fprintf(FILE *stream, char *format, ...);
```

- *stream*: a pointer to the stream opened for writing or updating;
- *format*: a pointer to a string describing the data to be written to the stream;

fprintf() function

```
int fprintf(FILE *stream, char *format, ...);
```

- ...: a list of expressions whose values will be converted into human-readable form and written to the stream;
- the function returns the number of characters (not values, as opposed to the *fscanf* function) correctly written to the stream.

This function enables us to send error messages directly to the *stderr* stream, which is, as you already know, both encouraged and welcome.

fprintf() function

- As you probably remember, the printf function prototype is as follows:
  - int printf (char *format, ...);
- This implies that the invocation:
  - printf("%d", number);
- is the functional equivalent of the following invocation:
  - fprintf(stdout, "%d", number);

fprintf() function

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main(int argc, char *argv[]) {
      int   numbers[1000];
      int   i,aux;
      int   numbersread = 0;
      int   swapped;
      FILE        *inp, *out;
      if(argc != 3) {
            fprintf(stderr,"usage: intsort2 source_file target_file\n");
            return 1;
      }
      if((inp = fopen(argv[1],"rt")) == NULL) {
            fprintf(stderr,"Cannot open %s: %s\n",argv[1],strerror(errno));
            return 2;
      }
```

fprintf() function

```
    if((out = fopen(argv[2],"wt")) == NULL) {
        fprintf(stderr,"Cannot create %s: %s\n",argv[2],strerror(errno));
        fclose(inp);
        return 3;
    }
    while(fscanf(inp,"%d",&numbers[numbersread]) == 1) {
        numbersread++;
        if(numbersread == 1000)
            break;
}
    fclose(inp);
    if(numbersread == 0) {
        printf("No numbers found in the file %s\n",argv[1]);
        return 3;
    }
```

fprintf() function

```
    do {
        swapped = 0;
        for(i = 0; i < numbersread - 1; i++)
            if(numbers[i] > numbers[i + 1]) {
                swapped = 1;
                aux = numbers[i];
                numbers[i] = numbers[i + 1];
                numbers[i + 1] = aux;
        }
    } while(swapped);
    for(i = 0; i < numbersread; i++)
        fprintf(out,"%d\n",numbers[i]);
    printf("\n");
    fclose(out);
    return 0;
}
```

Outline

1. Connecting to the real world: files and streams
    1. Reading from the stream
    2. Writing to the stream
    3. **Dealing with the stream's position**
2. Quiz

The ftell() function

- **The ftell() function: getting the stream's position**

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

- the function name comes from the words *file tell*;
- the function expects one parameter, which is a **pointer to the opened stream**;

The ftell() function

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

- the function returns the **distance** (in bytes) counted from the beginning of the file to the current file position; thus, the first byte of the file is located at position zero;
- in the event of an error, the function returns EOF (-1) as the result;
- the function affects neither the position of th nor its content.

The fseek() function

- **The *fseek*() function: setting the stream's position**

```
int fseek(FILE *stream, long offset, int whence);
```

- The fseek function allows us to **set the current position of the file**.

The fseek() function

```
int fseek(FILE *stream, long offset, int whence);
```

- the name of the function comes from the words *file seek*;
- the function expects the following three parameters:
  - stream: a **pointer** to an opened stream;
  - offset: a value describing the **target position** (may be negative);

The fseek() function

```
int fseek(FILE *stream, long offset, int whence);
```

- whence: a value indicating a reference point, i.e. saying how the new position is to be calculated; usually the role of this parameter is played by one of three symbolic constants:
  - SEEK_SET: the offset parameter specifies the position calculated from the **beginning** of the file;
  - SEEK_CUR: the offset parameter specifies the position calculated from the **current** file position;
  - SEEK_END: the offset parameter specifies the position calculated from the **end** of the file;

The fseek() function

```
int fseek(FILE *stream, long offset, int whence);
```

- in the event of an error, the function returns EOF (-1); otherwise, the return value is 0;
- the function obviously affects the current position of the file.

The fseek() function

- We assume that the following declaration is active:
  - FILE *F;
- Let’s perform some fseek invocations, describing their effects.
  - fseek(F, 0, SEEK_SET);
- sets the file in its starting position.
  - fseek(F, 100, SEEK_SET);
- sets the file at the 100th byte from the begin of the file.

The fseek() function

- fseek(F, 0, SEEK_END);
- sets the file at the end.
- fseek(F, 0, SEEK_CUR);
- does not change the position of the file (why?).
- fseek(F, -1, SEEK_CUR);
- offsets the current file position by 1 byte.

The fseek() function

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

int main(int argc, char *argv[]) {
        FILE            *file;
        long        size;

        if(argc != 2) {
                fprintf(stderr,"usage: getsize file_name\n");
                return 1;
        }
        if((file = fopen(argv[1],"rt")) == NULL) {
                fprintf(stderr,"Cannot open %s: %s\n",argv[1],strerror(errno));
                return 2;
        }
```

The fseek() function

```
    /* attempt to skip to the end of file */
    if(fseek(file,0,SEEK_END))
        if(errno == EBADF)
            fprintf(stderr,"The file has no size: %s\n",argv[1]);
        else
            fprintf(stderr,"Error in fseek: #%d\n",errno);
    else {
        size = ftell(file);
        printf("File: %s size: %d\n", argv[1], size);
    }
    fclose(file);
    return 0;
}
```

Rewinding the stream

- This function is a kind of an archaic artefact from the good old days when **magnetic tape storage devices** were in common use. These devices couldn’t perform random access, and you had to **rewind** the tape to the beginning for it to be re-read or re-written to.

Rewinding the stream

- The rewind function has the following prototype

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

- and when invoked it plays the same role as the following fseek invocation:
  - fseek(stream, 0, SEEK_SET);
- except for the fact that rewind doesn't return a value and it doesn't set the errno variable.

Checking the end of the file

- A few words of explanation are necessary here: the **end of file** (EOF) state occurs when there's nothing more to read in the file.

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

- This function returns a **non-zero value** if the stream is in the *EOF* state; and otherwise, the return value is 0.

Checking the end of the file

- Let's try to use this function in a short snippet taken from one of the previous programs. We'll carefully read the input file in the following way:

```
while(!feof (input)) {
    fgets (line, sizeof(line), input));
    fputs (line, output);
}
```

Outline

1. Connecting to the real world: files and streams
    1. Reading from the stream
    2. Writing to the stream
    3. Dealing with the stream's position
2. **Quiz**

Quiz

How big is the file created by the following program?

```
#include <stdio.h>
int main(void) {
   FILE *f;
      char s[] = "ABC";
      char *p = "DEF";
      int i=123;
      char c='\'';
      f = fopen("f","wb");
      fprintf(f,"%d %s %s %c",i,s,p,c);
      fclose(f);
      return 0;
}
```

- 9 bytes
- 11 bytes
- 13 bytes

Quiz

What happens when you try to compile and run the following program?

```
#include <stdio.h>
int main(void) {
   FILE *f;
      char s[] = "To be or not to be";
      long i;
      f = fopen("f","w+b");
      fputs(s,f);
      fseek(f,-2,SEEK_END);
      i = ftell(f);
      fclose(f);
      printf("%d",i);
      return 0;
}
```

- the program outputs -2
- the program outputs 18
- the program outputs 16

Quiz

What happens when you try to compile and run the following program?

```
#include <stdio.h>
int main(void) {
   FILE *f;
      int  i;
      f = fopen("f","wb");
      fputs("123",f);
      fclose(f);
      f = fopen("f","rt");
      fscanf(f,"%d",&i);
      fclose(f);
      printf("%d",i);
      return 0;
}
```

- the program outputs 12
- the program outputs 1
- the program outputs 123

Quiz

What happens when you try to compile and run the following program?

```
#include <stdio.h>
int main(void) {
   FILE *f;
      int  i;
      f = fopen("f","w+b");
      fputs("123",f);
      rewind(f);
      fputs("3",f);
      fclose(f);
      f = fopen("f","rt");
      fscanf(f,"%d",&i);
      fclose(f);
      printf("%d",i);
      return 0;
}
```

- ○ the program outputs 232
- ○ the program outputs 323
- ○ the program outputs 123

Quiz

What happens when you try to compile and run the following program?

```
#include <stdio.h>
int main(void) {
   FILE *f;
      int  i;
      f = fopen("f","wb");
      fclose(f);
      f = fopen("f","rb");
      fseek(f,0,SEEK_END);
      i = ftell(f);
      fclose(f);
      printf("%d",i);
      return 0;
}
```

- ○ the program outputs 2
- ○ the program outputs 1
- ○ the program outputs 0

Quiz

What happens when you try to compile and run the following program?

```
#include <stdio.h>
int main(void) {
   FILE *f;
   int  i;
   f = fopen("f","wb");
   fwrite(f,2,1,f);
   fclose(f);
   f = fopen("f","rb");
   fseek(f,0,SEEK_END);
   i = ftell(f);
   fclose(f);
   printf("%d",i);
   return 0;
}
```

- ◯ the program outputs 2
- ◯ the program outputs 1
- ◯ the program outputs 0