Files – Dynamic memory management

Basics of Programming 1

G. Horváth, A.B. Nagy, Z. Zsóka, P. Fiala, A. Vitéz

22 October, 2025

Content

Chapter 1

File handling

Files

File

Data stored on a physical media (hard disk, CD, USB drive)

- Data stored in a file is not lost after the program is finished, it can be reloaded.
- Independently of the media, files are handled in a uniform way
- File handling:
  1. Opening the file
  2. Data writing / reading
  3. Closing the file
- Two types of files:
  - Text file
  - Binary file

Text vs. Binary

Text file – contains text, divided into lines

- `txt`, `c`, `html`, `xml`, `rtf`, `svg`

Binary file – contains binary coded data of arbitrary structure

- `exe`, `wav`, `mp3`, `jpg`, `avi`, `zip`

- As long as it makes sense, use a text file – it is more friendly.
- It is a big advantage, if not only programs, but humans too are able to read and edit our data.

Writing into a text file

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
#include <stdio.h> /* fopen, fprintf, fclose */
int main(void)
{
  FILE *fp;
  int status;

  fp = fopen("hello.txt", "w");  /* file open */
  if (fp == NULL)                 /* no success */
    return 1;

  fprintf(fp, "Hello, World!\n");  /* writing */

  status = fclose(fp);             /* closing */
  if (status != 0)
    return 1;

  return 0;
}
```

link

FILE

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
struct _iobuf {
    char *_ptr;         // Current position in buffer
    int _cnt;           // Remaining chars in buffer
    char *_base;        // Start of buffer
    int _flag;          // File state flags
    int _file;          // File descriptor
    int _charbuf;       // Character buffer control
    int _bufsiz;        // Size of buffer
    char *_tmpfname;    // Temporary filename (if any)
};
typedef struct _iobuf FILE;
```

link

Opening a file


```
FILE *fopen(char *fname, char *mode);
```

- Opens the file whose name is specified in `fname` string, according to the mode given in `mode` string
- Main methods for text files:

mode		description
`"r"`	read	reading, the file must exist
`"w"`	write	writing, overwrites, if needed a new is created
`"a"`	append	writing, continues at the end, if needed a new is created

- return value is a pointer to a `FILE` structure, this is the identifier of the file
- If opening was not successfull, it returns with NULL

Closing a file


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

- It closes the file referenced by the `fp` identifier
- If the closing is successful[1], it returns with 0, otherwise it returns with `EOF`.

[1] closing a file may not be successful: for example somebody has removed the pendrive while we were writing onto it.

Writing onto screen / into text file / into string

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
int  printf(           char *control, ...);
int fprintf(FILE *fp,  char *control, ...);
int sprintf(char *str, char *control, ...);
```

- The text given in the `control` string will be written
  - onto the screen
  - into a text file (previously opened for writing) with `fp` identifier
  - into a string with `str` identifier (string must be long enough)
- Using of control character (eg. `%d`) is the same as with `printf`
- Return value is the number of successfully written characters[2], it is negative in case of error

[2] If we write into a string, it automatically adds the terminating 0, but it is not counted in the return value

Reading from keyboard / text file / string


```
int  scanf(           char *control, ...);
int fscanf(FILE *fp,  char *control, ...);
int sscanf(char *str, char *control, ...);
```

- Reads in the format specified in the `control` string from the
  - keyboard
  - a text file (previously opened for reading) with `fp` identifier
  - from a string with `str` identifier
- Return value is the number of read elements, it is negative in case of error

Reading from text file

Let's write a program, that prints (onto the screen) the content of a text file

```c
#include <stdio.h>
int main()
{
  char c;
  FILE *fp = fopen("file.txt", "r"); /* open file */
  if (fp == NULL)
    return -1; /* was not successfull */

  /* reading until successful (we read 1 character) */
  while (fscanf(fp, "%c", &c) == 1)
    printf("%c", c);

  fclose(fp); /* close file */
  return 0;
}
```

link

- Memorize the way we read until the end of the file!

Reading from text file

A text file contains the coordinates of 2D points. Each of its line has the following format
`x:1.2334, y:-23.3`
Let's write a program that reads and processes the coordinates!

```
1  FILE *fp;
2  double x, y;
3  ...
4  /* reading as long as it is successful */
5  /* (we read 2 numbers)                 */
6  while (fscanf(fp, "x:%lf, y:%lf", &x, &y) == 2)
7  {
8    /* processing */
9  }
```

- Once again, take a look at how we read until the end of the file!

Keyboard? Monitor?

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
scanf("%c", &c);
printf("%c", c);
```

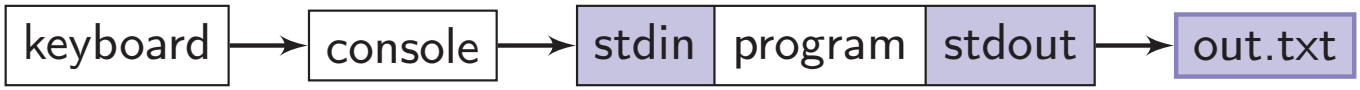
- The code segment above does not read directly from the keyboard and does not write directly onto the screen, but it reads from standard input (stdin), and writes to the standard output (stdout)
- stdin and stdout are text files
- The type of perphery or other file that is assigned to it depends on the operating system.
- Its default interpretation is as in the figure.
  - keyboard (through a console application) → stdin
  - stdout → (through a console application) monitor

Redirecting

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

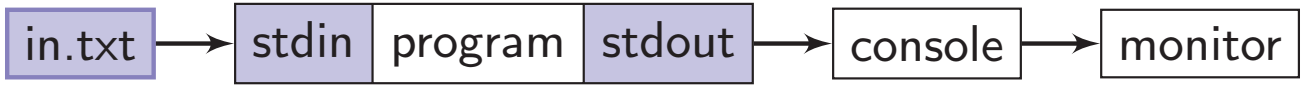
- If we start our program in the follwoing way, we can redirect the standard output: it will not print on the monitor, but into the `out.txt` text file

```
c:\>prog.exe > out.txt
```



- The standard input can also be redirected to a text file.

```
c:\>prog.exe < in.txt
```



- Of course, the 2 can be combined

```
c:\>prog.exe < in.txt > out.txt
```

stdin and stdout

- stdin and stdout are text files that are automatically opened when starting the program
- the code segments below are equivalent

```c
char c;
printf("Hello");
scanf("%c", &c);
printf("%c", c);
```

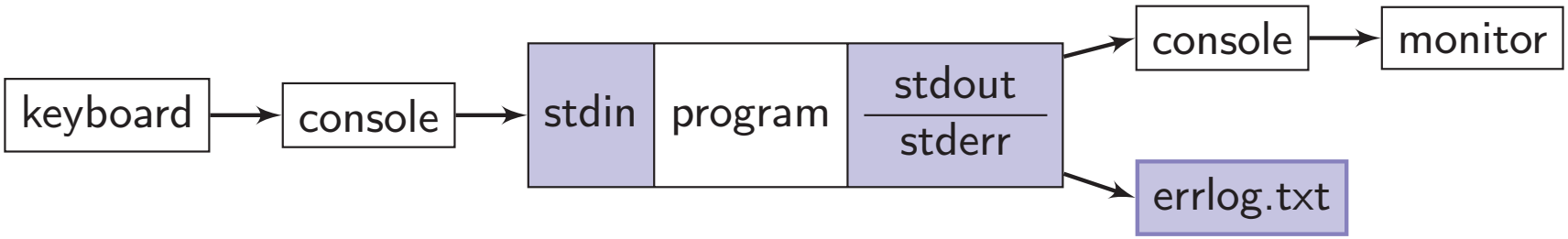
```c
char c;
fprintf(stdout, "Hello");
fscanf(stdin, "%c", &c);
fprintf(stdout, "%c", c);
```

- When writing data from a text file into a text file, instead of opening a file, use the standard input and output, and the redirection options of the operating system
- We can read from the console also until the end of the file: we can emulate the end of file by entering Ctrl+Z (windows) or Ctrl+D (linux).

stdout and stderr

- The output and the error messages of the program can be separated by using the standard error output `stderr`

```
c:\>prog.exe 2> errlog.txt
```




```c
if (error)
{
  /* useful information for the user */
  printf("Please, switch it off\n");
  /* detailed information to the error output */
  fprintf(stderr, "Error code 61\n");
}
```

Binary files

- Binary file: The bit-by-bit copy of the content of the memory onto a physical data media
- The actual data depends on the inner representation
- Use it only if storing as text would be very weird – and use it in tasks if asked 
- Opening and closing the file is similar to the case of text files, but now the `b` character must be used in the `mode` string[3]

`mode`		description
`"rb"`	read	reading, the file must exist
`"wb"`	write	writing, overwrites, if needed a new is created
`"ab"`	append	writing, continues at the end, if needed a new is created

[3] For the sake of analogy, in case of text file it is typical to use `t` (`text`), but actually `fopen` will not care about it.

Reading and writing a binary file


```
size_t fwrite (void *ptr, size_t size,
               size_t count, FILE *fp);
```

- Starting from address `ptr`, it writes `count` elements (that are placed one after the other in the memory), each having `size` size into a file with `fp` identifier
- Return value is the number of written elements.

```
size_t fread (void *ptr, size_t size,
              size_t count, FILE *fp);
```

- It reads `count` elements, each having `size` size from the file with `fp` identifier to the address `ptr`
- Return value is the number of read elements

Binary files – example

- This dog_array array contains 5 dogs

```c
typedef enum { BLACK, WHITE, RED } color_t;

typedef struct {
  char name[11];    /* name max 10 chars + terminating */
  color_t color;    /* colour */
  int nLegs;        /* number of legs */
  double height;    /* height */
} dog;

dog dog_array[] = /* array for storing 5 dogs */
{
  { "max",   RED,    4, 1.12 },
  { "cesar",  BLACK, 3, 1.24 },
  { "buddy",  WHITE, 4, 0.23 },
  { "spider", WHITE, 8, 0.45 },
  { "daisy",   BLACK, 4, 0.456 }
};
```

link

Binary files – examples

- Writing the dog_array array into a binary file is this easy!

```
1  fp = fopen("dogs.dat", "wb"); /* error handling!!! */
2  if (fwrite(dog_array, sizeof(dog), 5, fp) != 5)
3  {
4    /* error message */
5  }
6  fclose(fp); /* here also!!! */
```

- Re-reading the dog_array array is not less easier too.

```
1  dog dogs[5]; /* allocating memory */
2  fp = fopen("dogs.dat", "rb");
3  if (fread(dogs, sizeof(dog), 5, fp) != 5)
4  {
5    /* error message */
6  }
7  fclose(fp);
```

Binary files – example

- Do resist the temptation!
- If the representation of any members of the dog structure is different on mother's computer, the saved data cannot be re-read.
- Writing (saving) data into binary files without thinking makes our data non-portable
- Of course if we think, saving the data will become more difficult
  1. We have to agree on the representation
     - which bit is the LSB?
     - is it two-complement?
     - how long is mantissa?
     - are the members of the structure aligned to words? And how long is one word?
     - etc.
  2. The data must be converted first, and then written (saved)

Binary vs text

- Use text files, it is beneficial for everyone!
- Writing the dog_array array into text file

```
1 for (i = 0; i < 5; ++i) {
2   dog d = dog_array[i];
3   fprintf(fp, "%s,%u,%d,%f\n",
4     d.name, d.color, d.nLegs, d.height);
5 }
```

- Reading the dog_array array from text file[4]

```
1 dog dogs[5]; /* allocating memory */
2 for (i = 0; i < 5; ++i) {
3   dog d;
4   fscanf(fp, "%s,%u,%d,%lf",
5     d.name, &d.color, &d.nLegs, &d.height);
6   dogs[i] = d;
7 }
```

[4] we assume that the name of the dog has no whitespace characters in it

Statusflag functions


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

- true if we have reached the end of file, false otherwise

```
int ferror(FILE *fp);
```

- true if there was an error during read or write, false otherwise

- Most of the time we don't need them: we can use the return value of read and write functions.

Statusflag functions

- Typical mistake

```c
while (!feof(fp))
{
   /* read data element */

   /* process data element */
}
```

elem	elem	elem	EOF

- `feof()` is true only if we already have read the end of file symbol.
- What have we learned about data series with termination?

```c
/* read data element */
while (!feof(fp))
{
  /* process data element */
  /* read data element */
}
```

Chapter 2

Dynamic memory management

Dynamic memory management

- Let's read integer numbers and print them in a reversed order!
- The user will enter the number of the numbers to be read (count).
- Let's not use more memory than needed!

1. We read the count (`n`)
2. We ask memory from the operating system for storing `n` integer numbers
3. We read and store the numbers, and print them in reversed order
4. We give back (hand over) the reserved memory place to the operating system

Example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
int n, i;
int *p;

printf("How many numbers? ");
scanf("%d", &n);
p = (int*)malloc(n*sizeof(int));
if (p == NULL) return;

printf("Enter %d numbers:\n", n);
for (i = 0; i < n; ++i)
  scanf("%d", &p[i]);

printf("Reversed:\n");
for (i = 0; i < n; ++i)
  printf("%d ", p[n-i-1]);

free(p);
p = NULL;
```

link

p:0x0000

```
How many numbers?  5
Enter 5 numbers!
1 4 2 5 8
Reversed:
8 5 2 4 1
```

The malloc and free functions – <stdlib.h>

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
void *malloc(size_t size);
```

- Allocates memory block of size bytes, and the address of the block is returned as void* type value
- The returned void* "is only an address", we cannot de-refer it. We can use it only if converted (eg. to int*).

```
1 int *p; /* starting address of int array */
2 /* Memory allocation for 5 int */
3 p = (int *)malloc(5*sizeof(int));
```

- If there is not enough memory avaible, the return value is NULL. This must be checked always.

```
1 if (p != NULL)
2 {
3   /* using memory, and releasing it */
4 }
```

The malloc and free functions – <stdlib.h>

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
void free(void *p);
```

- Releases the memory block starting at address p
- The size of the block is not needed, the op.system knows it (it stored it just before the memory block, this is the reason for calling it with the starting address)
- free(NULL) is allowed (does not perform anything), so we can do this:

```
1 int *p = (int *)malloc(5*sizeof(int));
2 if (p != NULL)
3 {
4   /* using it */
5 }
6 free(p);  /* works even if NULL */
7 p = NULL; /* a useful step to remember */
```

- As a nullpointer points to nowhere, a good practice is to set a pointer to NULL after usage, so we can see it is not in use.

malloc – free

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- malloc and free go hand-in-hand,
- for each malloc there is a free

```
1 char *WiFi = (char *)malloc(20*sizeof(char));
2 int *Tibet = (int *)malloc(23*sizeof(int));
3 ...
4 free(WiFi);
5 free(Tibet);
```

- If we don't relelase the memory block, memory leak occurs
- Good practice rules:
  - Release in the same function where allocated
  - Don't modify the pointer that was returned by malloc, if possible, use the same pointer for releasing
- If we cannot keep these rules, make a note in the code about this (comment)

The calloc function – <stdlib.h>

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
void *calloc(size_t num, size_t size);
```

- Allocates memory block for storing `num` pieces of elements, each with `size` size, the allocated memory block is cleared (set to zero), and the address of the block is returned as `void*` type value
- Usage is almost the same as of `malloc`, except this performs the calculation `num*size`, and removes the garbage.
- The allocated block must be released in the same way: with `free`.

```
1 int *p = (int *)calloc(5, sizeof(int));
2 if (p != NULL)
3 {
4   /* using it */
5 }
6 free(p);
```

The realloc function – <stdlib.h>

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
void *realloc(void *memblock, size_t size);
```

- resizes to size bytes a memory block that was earlier allocated
- the new size can be smaller r larger than the earlier size
- if needed, the earlier content is copied to the new place, the elements are not initialized
- its return value is the starting address of the new place

```
1 int *p = (int *)malloc(3*sizeof(int));
2 p[0] = p[1] = p[2] = 8;
3 p = realloc(p, 5*sizeof(int));
4 p[3] = p[4] = 8;
5 ...
6 free(p);
```

Example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Let's create a function that concatenates the strings received as parameters. The function should allocate memory for the resulting string, and should return with its address.

1. The function determines the length of the two strings,
2. allocates memory for the result,
3. copies the first string into the result string,
4. copies the second string after it.

- Of course, this function cannot release the allocated memory, this must be done in the calling program segment

Example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
/* concatenate -- concatenating two strings
  Dynamic allocation, returning with address.
*/
char *concatenate(char *s1, char *s2){
    size_t l1 = strlen(s1);
    size_t l2 = strlen(s2);
    char *s = (char *)malloc((l1+l2+1)*sizeof(char));
    if (s != NULL)    {
        strcpy(s, s1);
        strcpy(s+l1, s2); /* or strcat(s, s2) */
    }
    return s;
}
```

link

Example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Usage of the function

```c
char word1[] = "partner", word2[] = "ship";

char *res1 = concatenate(word1, word2);
char *res2 = concatenate(word2, word1);
res2[0] = 'w';

printf("%s\n%s", res1, res2);

/* The function did allocate memory, release it! */
free(res1);
free(res2);
```

link

```
partnership
whippartner
```

Why to use fclose()?

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  FILE *__fdopen(int fd, int flags)
2  {
3      FILE *f;
4      struct stat st;
5
6      if (fcntl(fd, F_GETFL) < 0) return 0; /* verify fd vali
7      if (!(f = calloc(1, sizeof *f))) return 0;
8
9      f->fd = fd;
10     f->flags = flags;
11     f->buf = NULL;
12     f->buf_size = 0;
13     f->rpos = f->rend = f->wpos = f->wend = NULL;
14 ...
```

link

Thank you for your attention.