Computer Programming

Files

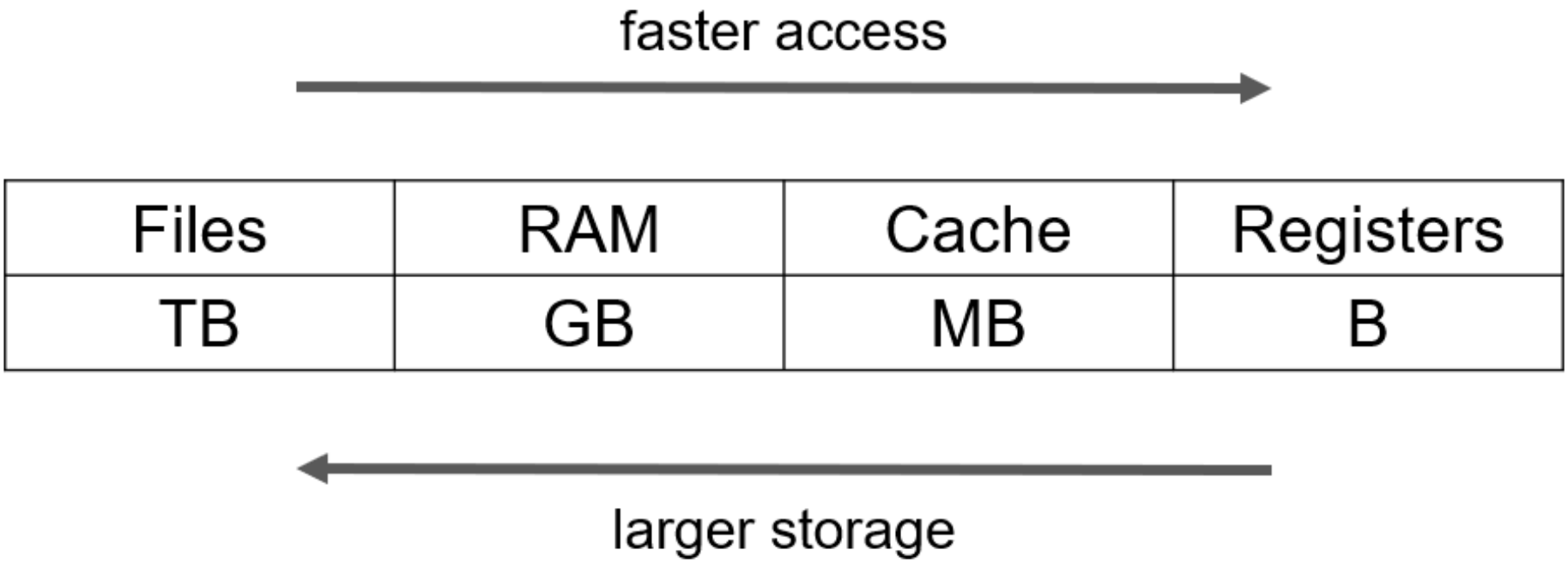
Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 11

Contents

Data storage



Files

- a file is a resource used to store data on a storage device (hard-drive, solid-state drive, DVD, USB stick)
- used for long-term storage
- usually means slower access compared to RAM
- for our purposes, it can be modeled as an array of bytes
- reading and writing can be done in a similar way to reading from the keyboard and writing on the screen
- requires permissions from the operating system
- all processing functions require a structure called FILE that is defined in stdio.h

FILE structure

- there are 3 files that are created automatically for every program:
  - stdin - standard input - linked to the keyboard
  - stdout - standard output - linked to the screen
  - stderr - error output - linked to the screen
- scanf/printf functions work directly with these files
- for operations on different files, the programmer must open them manually
- the state of a file is stored in a structure called FILE
- every operation requires such a structure

FILE structure

```
1  typedef struct _iobuf
2  {
3     char* _ptr;
4     int _cnt;
5     char* _base;
6     int _flag;
7     int _file;
8     int _charbuf;
9     int _bufsiz;
10    char* _tmpfname;
11 } FILE;
12
13 #define EOF (-1)
14 #define SEEK_CUR 1
15 #define SEEK_END 2
16 #define SEEK_SET 0
17 #define FILENAME_MAX 260
18 #define FOPEN_MAX 20
```

- structure defined in stdio.h
- ptr points at the current byte/character
- every IO operation requires a pointer to FILE
- maintains the state of the file

- constants declared in stdio.h
- end of file
- origin positions
- maximum length for a filename
- maximum number of files that can be opened simultaneously

File types

- text files
  - content is saved as text
  - human-readable
  - usually occupies more and takes longer to read/write
- binary files
  - content is saved as it is represented in memory
  - machine-readable
  - usually occupies less and is faster during read/write
  - can hide information
- text files can be considered as binary files

Processing pipeline

- File IO steps:
  - open the file with desired permissions
  - process the file using the FILE pointer
  - close the file
- Interaction with OS
  - we require the path to the file
  - need permission to read/write
  - processing functions return values which should be checked for errors

fopen function

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

- attempts to open a file with the specified permissions
- input parameters:
  - filename - the absolute/relative path to the file
  - mode - encodes the desired processing mode
- return value:
  - a valid file pointer in case of success
  - null pointer in case of failure
- possible errors: incorrect path, no permission

Processing modes

The second input parameter for `fopen` has at most 3 characters

- 1st character (required) - indicates the type of IO operation
  - r - is for read - read from an existing file
  - w - is for write - write a new file or overwrite an existing one
  - a - is for append - write at the end of an existing file
- 2nd character (optional) +
  - requests permission to read and to write in the file
- 3rd character (optional) b
  - signifies that the file is processed as a binary file

Program 11.1 - fopen example

```
1  #include <stdio.h>
2
3  int main(){
4
5    FILE* pf = fopen("file.txt", "r");
6    if (pf == NULL)
7      return -1;
8
9    FILE* pf2;
10   pf2 = fopen("..//date.out", "wb");
11   if (pf2 == NULL){
12     puts("cannot open");
13     return -2;
14   }
15
16   return 0;
17 }
```

- open the first file for reading
- if it cannot be opened, terminate the program
- in most cases we cannot recover from IO errors

- open the second file for writing in binary mode
- it is located in the parent directory
- if it cannot be opened, terminate the program with a different error code

File path

The file path can be in different forms

- filename + extension (specific case of relative path)
  - the file is expected to be in the current build folder
- relative path
  - specifies the location of the file starting from the current build folder
  - to specify the parent folder use ..
  - use / to separate folders
- absolute path
  - specifies the location of the file starting from the partition, the folders, the file name and extension along the path
  - Windows uses \ for separating directories
  - the character \ must be doubled in C strings like \\
  - alternatively, it can be replaced by forward slash / (Linux style)

File path - common mistakes

- incorrect file path - to avoid respect the following:
  - copy the file path from OS
  - change \ to / in string
  - check build folder, current working directory
- the extension of the file must be provided
  - extension can be hidden (txt files in Windows Explorer)
  - file extension indicates its format, but it can be anything
- no read/write privileges
  - a file cannot be opened to write if it is read-only or if it is opened by another application
- always check the pointer returned from `fopen` for errors

fscanf function

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

- read from the file represented by the pointer pf based on the format specifiers from format, in a similar manner to scanf
- input parameters:
  - pf - pointer to FILE structure
  - format - string containing format specifiers
  - ... - addresses where to save the data read
- return value:
  - number of arguments read successfully
  - EOF constant if the end of file is reached before a read
- possible errors:
  - invalid file pointer
  - invalid address

fprintf function

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

- prints in the file represented by the pointer pf based on the format specifiers from format, in a similar manner to printf
- input parameters:
  - pf - pointer to FILE structure
  - format - string containing format specifiers
  - ... - variables to print
- return value:
  - number of characters which were printed
  - negative number in case of an error
- possible errors:
  - invalid file pointer
  - file not accessible to write operations

fclose function

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

- close the file represented by the file pointer, signaling that no further operations will be performed on it
  - pf - pointer to FILE structure
- return value:
  - 0 in case of success
  - EOF constant in case of an error
- possible errors:
  - invalid file pointer
  - file already closed

Program 11.2 - Simple IO with files

```c
#include <stdio.h>

int main(){
    FILE* pf = fopen("file.txt", "w");
    if (pf == NULL)
      return -1;

    int x = 7;
    fprintf(pf, "%d\n%d", x, 5);
    fclose(pf);

    pf = fopen("file.txt", "r");
    int y;
    fscanf(pf, "%d%d", &x, &y);
    printf("x = %d y = %d\n", x, y);
    fclose(pf);
    return 0;
}
```

- open the file to write
- it will be created if it doesn't exist
- write x and y on different lines
- close the file
- open it for reading
- read x and y from the file
- confirm the values

File IO recommendations

- we should view file read/write operations as almost the same as normal read/write operations
  - processing functions reflect this
  - their names start with f and they require an additional FILE pointer structure
  - in fact, scanf and printf are just read and write operations on the stdin and stdout files respectively
- formatted reading/writing allows us to read tokens directly
  - in most cases, it is more lengthy to read the whole file character by character or line by line and then to tokenize the data
  - recall, that read functions jump over white-spaces and they can skip certain characters if the correct format specifier is employed
- if the number of elements to be read is unknown, the return value from fscanf should be monitored

Program 11.3 - Reading data of unknown size

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

int main(){
  FILE* pf = fopen("file.txt", "r");
  if (pf == NULL)
    return -1;

  int n = 10;
  int* a = calloc(n, sizeof(int));
  int ai;
  int pos = 0;
  while(fscanf(pf, "%d", &ai) == 1){
    if (pos == n){
      n *= 2;
      a = realloc(a, n * sizeof(int));
      if (a == NULL)
        return -2;
    }
    printf("a[%d] = %d\n", pos, ai);
    a[pos++] = ai;
  }
```

- open the file for reading
- allocate an array of size 10
- read one number at a time and check if the operation was successful
- if the number of elements exceeds the size of a, double its dimension by reallocating
- alternatively, allocate a at the beginning with maximum size and shrink it after reading

feof function

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

- check if the end of the file was reached after the last operation
  - pf - pointer to FILE structure
- return value:
  - non-zero value if the end of the file was reached
  - 0 if not
- possible errors:
  - invalid file pointer
- it is recommended to use the return value from the fscanf function instead

freopen function

```
FILE* freopen(const char* filename, const char* mode, FILE* pf);
```

- open a file under a different name
- redirect read/write operations to files
- input parameters:
  - filename - name of the file
  - mode - type of processing operations (like for fopen)
  - pf - pointer to FILE structure
- return value:
  - copy of the FILE pointer in case of success
  - NULL pointer in case of failure
- possible errors:
  - invalid file pointer
  - no permission

fseek function

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

- move the cursor to the position offset relative to the origin
- used to jump to a position or to rewind the file
  - pf - pointer to FILE structure
  - offset - positive or negative shift relative to the origin
  - origin - reference points (see below)
- return value:
  - 0 in case of success
  - non-zero value in case of failure
- possible values for origin (constants found in stdio.h):

  SEEK_SET - the beginning of the file
  SEEK_CUR - the current position in the file
  SEEK_END - the end of the file

ftell function

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

- returns the position of cursor in the file
  - pf - pointer to FILE structure
- return value:
  - 0 or positive value in case of success
  - negative value in case of failure
- can be used in conjunction with fseek to determine the size of a file programmatically
  - for larger files use `fgetpos`

Binary files - introduction

- most files are binary
- the content is not human-readable
- in order to extract meaningful data we need to know the structure of the file
- on Linux there is no difference between a binary and a text file
- the extension of a file does not matter, it is just a convention to indicate the format of the file
- binary file content:
  - data are saved as they are represented in the memory
  - usually occupies less space and is faster to read
  - can be interpreted only if we know the file structure

fread function

```
size_t fread(void* buffer, size_t size, size_t count, FILE* pf);
```

- reads in binary mode to the address indicated by buffer size * count bytes from the file represented by pf
  - buffer - pointer where the data should be stored
  - size - size of a single element to read
  - count - number of elements to read
  - pf - pointer to FILE structure
- return value:
  - number of elements read successfully
- possible errors
  - invalid file pointer
  - buffer pointer holds incorrect/inaccessible address

fwrite function

```
size_t fwrite(void* buffer, size_t size, size_t count, FILE* pf)
```

- writes in binary mode from the address indicated by buffer size * count bytes in the file represented by pf
  - buffer - pointer where the data to be written is stored
  - size - size of a single element to write
  - count - number of elements to write
  - pf - pointer to FILE structure
- return value:
  - number of elements read successfully
- possible errors
  - invalid file pointer
  - buffer pointer holds incorrect/inaccessible address

Program 11.4 - Binary read/write example

```
1  #include <stdio.h>
2
3  int main(){
4      FILE* pf = fopen("in.bin", "rb");
5      if (pf == NULL)
6          return -1;
7      int n;
8      fread(&n, sizeof(n), 1, pf);
9      int a[n];
10     fread(a, sizeof(int), n, pf);
11     fclose(pf);
12
13     for(int i=0; i<n; i++)
14         a[i] = -a[i];
15     pf = fopen("out.bin", "wb");
16     if (pf == NULL)
17         return -2;
18     fwrite(&n, sizeof(n), 1, pf);
19     fwrite(a, sizeof(int), n, pf);
20     fclose(pf);
21     return 0;
22 }
```

- open the file in binary mode
- read the number of elements from the array
- read the whole array by reading n ints
- change the sign of each number
- write n
- write the whole array with a single instruction

Study problem - Editing a bmp image

- you are given an image in bmp format
- Bitmap Microsoft - similar formats exist on other operating systems
- edit the image to create a red square on blue background
- retain the resolution of the original image

Bitmap format

Three main parts:

- bitmap file header - signature, file size
- bitmap DIB header - information about image resolution, color encoding
- array of pixels - color in each position stored in a linearized manner

More detailed structure here.

Structures from windows.h

```
typedef struct
    tagBITMAPINFOHEADER {
  DWORD biSize;
  LONG biWidth;
  LONG biHeight;
  WORD biPlanes;
  WORD biBitCount;
  DWORD biCompression;
  DWORD biSizeImage;
  LONG biXPelsPerMeter;
  LONG biYPelsPerMeter;
  DWORD biClrUsed;
  DWORD biClrImportant;
} BITMAPINFOHEADER;
```

```
typedef struct tagRGBTRIPLE {
  BYTE rgbtBlue;
  BYTE rgbtGreen;
  BYTE rgbtRed;
} RGBTRIPLE;

typedef struct
    tagBITMAPFILEHEADER {
  WORD bfType;
  DWORD bfSize;
  WORD bfReserved1;
  WORD bfReserved2;
  DWORD bfOffBits;
} BITMAPFILEHEADER;
```

BYTE - char

WORD - short

DWORD - int

LONG - long

Program 11.5 - Editing a bmp image - pt.1

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

int main(){
  FILE* fp = fopen("img.bmp", "rb+");
  BITMAPFILEHEADER head;
  fread(&head, sizeof(head), 1, fp);
  BITMAPINFOHEADER info;
  fread(&info, sizeof(info), 1, fp);
  int h = info.biHeight;
  int w = info.biWidth;

  RGBTRIPLE* pixels = calloc(sizeof(RGBTRIPLE), h*w);
  fread(pixels, sizeof(RGBTRIPLE), h*w, fp);

  for(int i = 0; i<h*w; i++) {
    pixels[i].rgbtBlue = 255;
    pixels[i].rgbtGreen = 0;
    pixels[i].rgbtRed = 0;
  }
```

- read the bitmap header
- read the info (DIB) header
- find the resolution
- read all the pixels as an array of RGB triplets

Program 11.5 - Editing a bmp image - pt.2

```
21    int h2 = h/2, h4 = h/4, w2 = w/2;
22    for(int i = h2 - h4; i < h2 + h4; i++){
23      for(int j = w2 - h4; j < w2 + h4; j++){
24        pixels[i * w + j].rgbtBlue = 0;
25        pixels[i * w + j].rgbtRed = 255;
26      }
27    }
28    fseek(fp, sizeof(head) + sizeof(info), SEEK_SET);
29    fwrite(pixels, sizeof(RGBTRIPLE), h*w, fp);
30    fclose(fp);
31    return 0;
32  }
```

- change the local pixel array
- move the file cursor back
- copy the pixel array to the file