The enumerated data type – File handling

Basics of Programming 1

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

4 November, 2020

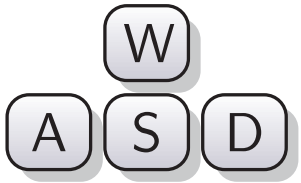
Content

Chapter 1

The enumerated type

The enumerated type – Motivation

- We are writing a game, in which the user can control direction of the player with 4 keys.



- As the input from user needs to be read (checked) frequently, we create a `read_direction()` function for this task.
- This function reads from the keyboard and returns the direction to the calling program segment.
- What type should the function return with?

The enumerated type – Motivation

- Idea Nr. 1: Let's return with the key pressed. ('a','s','w','d'):

```c
char read_direction(void)
{
  char ch;
  scanf("%c", &ch);
  return ch;
}
```

link

The enumerated type – Motivation

- Idea Nr. 1: Let's return with the key pressed. (`'a'`,`'s'`,`'w'`,`'d'`):

```c
char read_direction(void)
{
  char ch;
  scanf("%c", &ch);
  return ch;
}
```

link

- Problems:
  - We have to decode characters into directions many times at different parts of the source code.
  - If we change to use the arrow keys $\leftarrow \downarrow \uparrow \rightarrow$ for control, we have to modify the source code a thousand time and place.

The enumerated type – Motivation

- Idea Nr. 1: Let's return with the key pressed. (`'a'`,`'s'`,`'w'`,`'d'`):

```
1  char read_direction(void)
2  {
3    char ch;
4    scanf("%c", &ch);
5    return ch;
6  }
```

link

- Problems:
  - We have to decode characters into directions many times at different parts of the source code.
  - If we change to use the arrow keys $\leftarrow \downarrow \uparrow \rightarrow$ for control, we have to modify the source code a thousand time and place.
- Solution:
  - We have to decode in place (inside the function), and should return with direction.
  - But how can we do that?

The enumerated type – Motivation

- Idea Nr. 2: Let's return with `int` values 0,1,2,3:

'a'	0	←
'w'	1	↑
'd'	2	→
's'	3	↓

```c
int read_direction(void) {
  char ch;
  scanf("%c", &ch);
  switch (ch) {
  case 'a': return 0; /* left */
  case 'w': return 1; /* up */
  case 'd': return 2; /* right */
  case 's': return 3; /* down */
  }
  return 0; /* default is left :) */
}
```

The enumerated type – Motivation

- Idea Nr. 2: Let's return with `int` values 0,1,2,3:

'a'	0	←
'w'	1	↑
'd'	2	→
's'	3	↓

```c
int read_direction(void) {
  char ch;
  scanf("%c", &ch);
  switch (ch) {
  case 'a': return 0; /* left */
  case 'w': return 1; /* up */
  case 'd': return 2; /* right */
  case 's': return 3; /* down */
  }
  return 0; /* default is left :) */
}
```

- Problem:
  - In other parts of the program we have to use numbers 0-3 for the directions, so the programmer must remember the number-direction assignments.

The enumerated type – Motivation

- We need a type named `direction`, that can store `LEFT`, `RIGHT`, `UP`, `DOWN` values.
- We can do such thing in C!

  Declaration of the appropriate enumerated type (`enum`):

```
1 enum direction {LEFT, RIGHT, UP, DOWN};
```

- How to use the type:

```
1 enum direction d;
2 d = LEFT;
```

The enumerated type – Motivation

- The final solution with the new type

```
1  enum direction {LEFT, RIGHT, UP, DOWN};
2  typedef enum direction direction; /* simplification */
3
4  direction read_direction(void)
5  {
6    char ch;
7    scanf("%c", &ch);
8    switch (ch)
9    {
10   case 'a': return LEFT;
11   case 'w': return UP;
12   case 'd': return RIGHT;
13   case 's': return DOWN;
14   }
15   return LEFT;
16 }
```

link

The enumerated type – Motivation

- Usage of the function:

```
1  direction d = read_direction();
2  if (d == RIGHT)
3    printf("You were eaten by a tiger\n");
```

link

The enumerated type – Motivation

- Usage of the function:

```
1 direction d = read_direction();
2 if (d == RIGHT)
3   printf("You were eaten by a tiger\n");
```

link

- Without the enumerated type, it would look like this:

```
1 int d = read_direction();
2 if (d == 2) /* "magic" constant, what does it mean? */
3   printf("You were eaten by a tiger\n");
```

link

The enumerated type – Motivation

- Usage of the function:

```
1 direction d = read_direction();
2 if (d == RIGHT)
3   printf("You were eaten by a tiger\n");
```

link

- Without the enumerated type, it would look like this:

```
1 int d = read_direction();
2 if (d == 2) /* "magic" constant, what does it mean? */
3   printf("You were eaten by a tiger\n");
```

link

- The enumerated type...
  - replaces "magic constants" with informative code,
  - focuses on content instead of representation,
  - allows a higher level programming.

The enumerated type – Definition

The enumerated (enum) type

Joins into one type integer type constants referenced by symbolic names.

```
enum [<enumeration label>]opt
{ <enumeration list> }
[<variable identifiers>]opt;
```

The enumerated type – Definition

The enumerated (enum) type

Joins into one type integer type constants referenced by symbolic names.

```
enum [<enumeration label>]opt
{ <enumeration list> }
[<variable identifiers>]opt ;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

The enumerated type – Definition

The enumerated (enum) type

Joins into one type integer type constants referenced by symbolic names.

```
enum [<enumeration label>]opt
{ <enumeration list> }
[<variable identifiers>]opt;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

The enumerated type – Definition

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

The enumerated (enum) type

Joins into one type integer type constants referenced by symbolic names.

```
enum [<enumeration label>]opt
{ <enumeration list> }
[<variable identifiers>]opt;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

The enumerated type – Definition

The enumerated (enum) type

Joins into one type integer type constants referenced by symbolic names.

```
enum [<enumeration label>]opt
{ <enumeration list> }
[<variable identifiers>]opt;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

enum examples

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  enum month {
2    JAN,   /*  0 */
3    FEB,   /*  1 */
4    MAR,   /*  2 */
5    APR,   /*  3 */
6    MAY,   /*  4 */
7    JUNE,  /*  5 */
8    JULY,  /*  6 */
9    AUG,   /*  7 */
10   SEPT,  /*  8 */
11   OCT,   /*  9 */
12   NOV,   /* 10 */
13   DEC    /* 11 */
14 };
15
16 enum month m=OCT; /*9*/
```

enum examples

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
enum month {
  JAN,   /*  0 */
  FEB,   /*  1 */
  MAR,   /*  2 */
  APR,   /*  3 */
  MAY,   /*  4 */
  JUNE,  /*  5 */
  JULY,  /*  6 */
  AUG,   /*  7 */
  SEPT,  /*  8 */
  OCT,   /*  9 */
  NOV,   /* 10 */
  DEC    /* 11 */
};

enum month m=OCT; /*9*/
```

```c
enum {
  RED,        /*  0 */
  BLUE = 3,   /*  3 */
  GREEN,      /*  4 */
  YELLOW,     /*  5 */
  GRAY = 10   /* 10 */
} c;

c = GREEN;
printf("c: %d\n", c);
```

```
c:  4
```

Chapter 2

File handling

Files

File

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

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.

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

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:

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

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

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

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:

Files

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

Files

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

Text vs. Binary

Text file – contains text, divided into lines

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

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

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`

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.

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

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

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

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

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

Opening a file

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

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

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

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

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);
```

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

Closing a file


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

- It closes the file referenced by the `fp` identifier

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

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


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

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

Writing onto screen / into text file / into string


```
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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

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

Reading from text file

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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!

Reading from text file

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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  }
```

Reading from text file

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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);
```

Keyboard? Monitor?


```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)

Keyboard? Monitor?


```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

Keyboard? Monitor?


```
1  scanf("%c", &c);
2  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.

Keyboard? Monitor?


```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? Monitor?


```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

Keyboard? Monitor?


```
1 scanf("%c", &c);
2 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
```

Redirecting

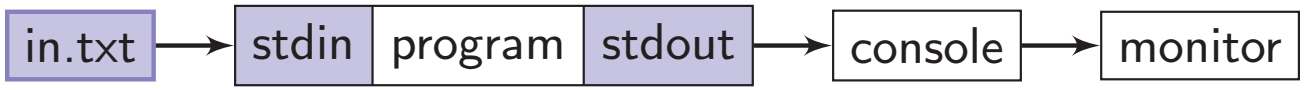
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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
```



Redirecting

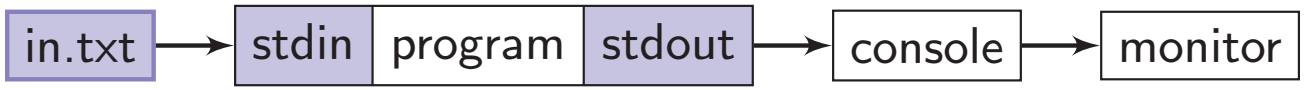
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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

stdin and stdout

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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);
```

stdin and stdout

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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

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);
```

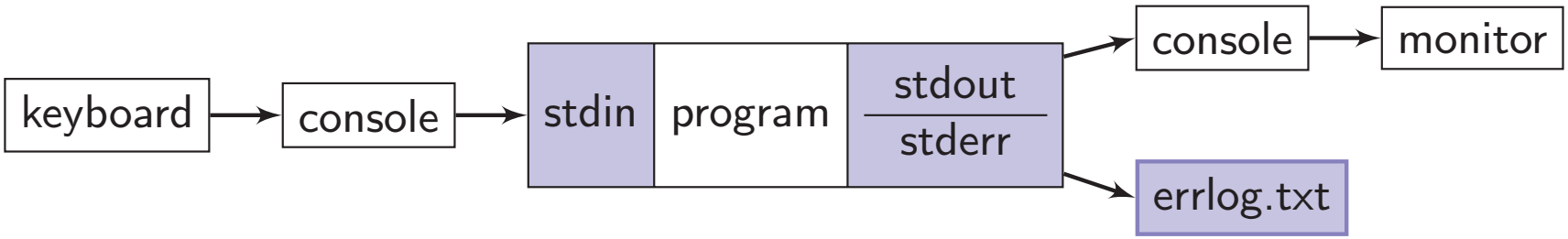
```c
char c;
fprintf(stdout, "Hello");
fscanf(stdin, "%c", &c);
fprintf(stdout, "%c", c);
```

- 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

[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.

Binary files

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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

[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.

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 

[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.

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);
```

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

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.

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);
```

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`

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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- This dog_array array contains 5 dogs

```
1  typedef enum { BLACK, WHITE, RED } color_t;
2
3  typedef struct {
4    char name[11];    /* name max 10 chars + terminating */
5    color_t color;    /* colour */
6    int nLegs;        /* number of legs */
7    double height;    /* height */
8  } dog;
9
10 dog dog_array[] = /* array for storing 5 dogs */
11 {
12   { "max",   RED,    4, 1.12 },
13   { "cesar",  BLACK, 3, 1.24 },
14   { "buddy",  WHITE, 4, 0.23 },
15   { "spider", WHITE, 8, 0.45 },
16   { "daisy",   BLACK, 4, 0.456 }
17 };
```

link

Binary files – examples

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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!!! */
```

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!

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.

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

Binary files – example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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

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

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?

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?

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?

Binary files – example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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?

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.

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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Use text files, it is beneficial for everyone!

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

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 }
```

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

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

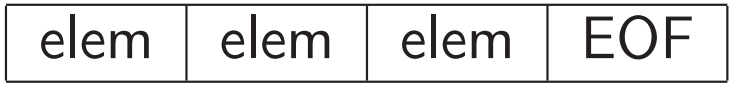
- Most of the time we don't need them: we can use the return value of read and write functions.

Statusflag functions

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Typical mistake

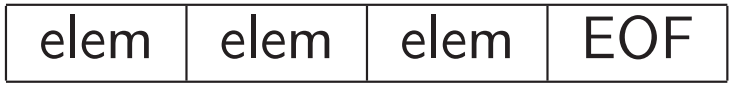
```
1  while (!feof(fp))
2  {
3      /* read data element */
4
5      /* process data element */
6  }
```



Statusflag functions

- Typical mistake

```
1  while (!feof(fp))
2  {
3      /* read data element */
4
5      /* process data element */
6  }
```



- `feof()` is true only if we already have read the end of file symbol.

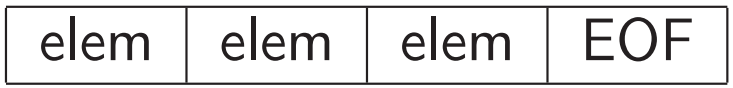
Statusflag functions

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Typical mistake

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

   /* process data element */
}
```



- `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 */
}
```

Thank you for your attention.