System-Level Programming

22 Supplements: In-/Output

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

http://sys.cs.fau.de/lehre/ss24

Input/Output

- I/O functionality is not part of the programming language
- Realized by “normal” functions
  - part of the standard library
  - simple programming interface
  - efficient
  - portable
  - close to the operation system
- Functionality
  - open/close files
  - read/write single characters, lines, or arbitrary blocks of data
  - formatted input/output

Standard Input/Output

Every C program gets 3 I/O channels assigned automatically upon starting:

`stdin`: standard input

- usually connected to the keyboard
- "end of file" (**`EOF`**) gets signaled by input of **`CTRL-D`** at the begin of a line
- this can be redirected to a file upon calling the program

```
~> prog < inputfile
```

`stdout`: standard output

- usually connected to the display (or the window from which the program was started)
- this can be redirected to a file upon calling the program

```
~> prog > outputfile
```

`stderr`: output channel for error messages

- usually also connected to the display

Standard Input/Output (continued)

- Pipes
  - The standard output of a program can be connected with the standard input of another program:

    ```
    ~> prog1 | prog2
    ```

    The redirection of the standard I/O channels is not detectable for the called programs.
- Automatic buffering
  - Input from the keyboard is usually buffered line-by-line by the operating system and only passed to the program when a **`NEWLINE`** symbol (**`'\n'`**) occurs!
  - Output for the display is usually buffered line-by-line by the program and only written to the display when a **`NEWLINE`** symbol occurs!

Opening and Closing Files

- Besides the standard I/O channels, a program can open further I/O channels
  - access to files
- Opening an I/O channel
  - function `fopen` (file open)
- Closing an I/O channel
  - function `fclose` (file close)

Opening and Closing Files (continued)

- Interface **fopen**

```
#include <stdio.h>

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

name: path name of the file to be opened

mode: mode, how the file has to be opened

”r”: read

”w”: write

”a”: write at the end of the file (append)

”rw”: read and write

- opens file **name**
- result of **fopen**: pointer to a data type **FILE** that describes a file channel; on error **NULL**

Opening and Closing Files (continued)

- Interface **fclose**

```
#include <stdio.h>

int fclose(FILE *fp);
```

  - closes I/O channel `fp`
  - result is either **`0`** (no errors) or **`EOF`** if an error occured

Opening and Closing Files – Example

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

int main(void)
{
    FILE *fp; int ret;

    fp = fopen("test.dat", "w"); /* Open "test.dat" for writing. */
    if (fp == NULL) {
        /* Error */
        perror("test.dat"); /* Print error message. */
        exit(EXIT_FAILURE); /* Terminate program. */
    }

    ... /* Program can now write to file "test.dat". */

    ret = fclose(fp); /* Close file. */
    if (ret == EOF) {
        /* Error */
        perror("test.dat"); /* Print error message. */
        exit(EXIT_FAILURE); /* Terminate program. */
    }
    return EXIT_SUCCESS;
}
```

Reading and Writing single Characters

Reading a single character

- from standard input

```
#include <stdio.h>
int getchar(void);
```

- from a file

```
#include <stdio.h>
int fgetc(FILE *fp);
```

- read the next character
- return the character as `int` value
- return `EOF` at the end of file or when CRTL-D is pressed

Writing a single character

- to the standard output

```
#include <stdio.h>
int putchar(int c);
```

- into a file

```
#include <stdio.h>
int fputc(int c, FILE *fp);
```

- write the character `c`
- return `EOF` in case of an error

Reading and Writing single Characters – Example

Copy program:

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

int main(int argc, char *argv[]) {
    FILE *src, *dst;
    int c;

    if (argc != 3) { ... }

    if ((src = fopen(argv[1], "r")) == NULL) { ... }
    if ((dst = fopen(argv[2], "w")) == NULL) { ... }

    while ((c = fgetc(src)) != EOF) {
        if (fputc(c, dst) == EOF) { ... }
    }

    if (fclose(dst) == EOF) { ... }
    if (fclose(src) == EOF) { ... }

    return EXIT_SUCCESS;
}
```

Reading and Writing Line-by-Line

- Reading one line

```
#include <stdio.h>
char *fgets(char *buf, int bufsize, FILE *fp);
```

  - reads characters from the file channel **fp** into the **char** array **buf** until either **bufsize-1** characters have been read or **'\n'** or **EOF**
  - **s** (returned string) gets terminated by **'\0'** (**'\n'** does not get removed)
  - returns **NULL** on **EOF** or when an error occurs
  - for **fp**, **stdin** can be used to read from the standard input

- Writing one line

```
#include <stdio.h>
int fputs(char *buf, FILE *fp);
```

  - writes the characters from the array **s** to the file channel **fp**
  - returns **EOF** when an error occurs
  - for **fp** **stdout** or **stderr** can be used

Formatted Output

- Interface

```
#include <stdio.h>
int printf(char *format, ...);
int fprintf(FILE *fp, char *format, ...);
int sprintf(char *buf, char *format, ...);
int snprintf(char *buf, int bufsize, char *format, ...);
```

- The parameters given instead of `...` are outputted according to the specifications in the `format` string
  - when using `printf` to the standard output channel
  - when using `fprintf` to the file channel `fp` (`fp` can be substituted by `stdout` or `stderr`)
  - `sprintf` writes the output into the `char`-array `buf` (but does not consider the length of the array ⇒ buffer overflow possible!)
  - `snprintf` works analogously, but writing at most `bufsize` characters (`bufsize` therefore should not be greater than the size of the array!)

Formatted Output (continued)

- Characters in the **`format`** string have different meanings
  - normal (printable) characters:
    are copied to the output
  - escape characters:
    e. g., `\n` or `\t` are substituted by the corresponding characters in the output (here: new line or tabulator)
  - format instructions:
    start with `%` character and describe, how the corresponding parameter in the list after the **`format`** string has to be interpreted
- For more specific information refer to the manuals
  (`man 3 printf`, ...)

Formatted Output (continued)

- Format-instructions

%d, %i: output **`int`** parameter as a decimal number
%ld, %li: correspondingly for **`long int`**

%f: output **`float`** parameter as floating point number (e. g., **`13.153534`**)
%lf: correspondingly for **`double`**

%e: output **`float`** parameter as a floating point number with powers of 10 (e. g., **`2.71456e+02`**)
%le: correspondingly for **`double`**

%c: output **`char`** parameter as single character
%s: output **`char`** array until **`'\0'`** is reached

%%: output a **`%`** character

...: ...

Formatted Output – Example

```c
int day = 25;
int month = 6;
int year = 2009;
char *name = "Michael Jackson";
printf("On %d/%d/%d\n%s died.\n",
       month, day, year, name);

printf("\n");

double pi = asin(1.0) * 2.0;
double e = exp(1.0);
fprintf(stdout,
        "Important value are:\n");
fprintf(stdout,
        "pi=%lf and e=%lf\n", pi, e);
```

```
~> ./test
On 6/25/2009
Michael Jackson died.

Important value are:
pi=3.141593 and e=2.718282
~>
```

Formatted Input

- Interface

```c
#include <stdio.h>

int scanf(char *format, ...);
int fscanf(FILE *fp, char *format, ...);
int sscanf(char *buf, char *format, ...);
```

Format string analogously works to the formatted output.
For more specific information, read the manuals (`man 3 scanf`, ...).

But: since values have to be read, pointers to the variables have to be passed to the functions (mimic call-by-reference semantics with C's call-by-value approach)!

Formatted Input – Example

```c
double pi, e;
int ret;

ret = scanf("pi=%lf, e=%lf\n", &pi, &e);
if (ret != 2) {
    fprintf(stderr, "Bad input!\n");
    exit(EXIT_FAILURE);
}
printf("I got\n\tpi=%lf\n\te=%lf\n", pi, e);
```

```
~> ./test
3.14 2.718
Bad input!
~>
```

```
~> ./test
pi=3.14, e=2.718
I got
        pi=3.140000
        e=2.718000
~>
```