7 Vstupně-výstupní operace

- Knihovna stdio
- Koncepce proudů
- Vstupně-výstupní operace
- Formátování vstupů a výstupů
- Funkce pro práci s proudy a soubory
- Ošetřování chybových stavů
stdio (= Standard Input/Output)
Knihovna vstupně-výstupních operací

• Funkce a datové typy pro vstupně-výstupní operace jsou v knihovně \texttt{stdio} ⇒ připojit hlavičkový soubor příkazem preprocesoru: \texttt{#include <stdio.h>}

• \texttt{stdio} obsahuje mj. prototypy nejznámějších funkcí jazyka C – „základních” funkcí \texttt{printf(\texttt{) a scanf(\texttt{):}}

\begin{verbatim}
#include <stdio.h>

void main() {
    char name[128] = \{0\};

    printf("Type your name: ");
    scanf("%s", name);
    printf("Hello, %s!\n", name);
}
\end{verbatim}

\textcolor{red}{některé překladače, např. \texttt{gcc}, přeloží uvedený kód i bez připojení knihovny, jen s warningem...}
stdio (= Standard Input/Output)
Koncept tzv. streamů čili proudů

- Knihovna stdio používá k práci s fyzickými zařízeními (klávesnice, terminál, tiskárna, diskový soubor, apod.) koncept tzv. proudů bytů (Byte Streams) – původ. myšlenka pochází z vývoje UNIXu (Mike Lesk @ Bell Labs, 70. léta 20. stol.), proto ve Windows mnohdy nefunguje úplně přirozeně;

- stream je abstrakce, která dovoluje obsluhovat všechna zařízení jednotným způsobem;

- streamy mají společné vlastnosti nezávislé na konkrétních vlastnostech zařízení, se kterými jsou spojené – např. přístupová oprávnění (jen pro čtení/pro čtení i zápis), povaha dat (text/binární), vyrovnávací paměť (fully buffered, line buffered, unbuffered), orientace, atd.

- streamy mají určité vnitřní stavové proměnné, indikátory.
stdio (= **Standard Input/Output**)
Koncept tzv. streamů čili proudů

- Streamy mohou být **vstupní** a/nebo **výstupní**;

- streamy mají určité **vnitřní stavové proměnné**, indikátory: **indikátor chyby** (**Error Indicator**), **indikátor konce** (**End-Of-File Indicator**), **indikátor pozice** (**Position Indicator**);

- indikátory nejsou programátorovi přímo přístupné, ale pracují s nimi knihovní funkce, např. funkce **int feof(·)** vrací hodnotu indikátoru **End-Of-File**, tj. zjišťuje, zda bylo dosaženo konce streamu;

- stream je přístupný prostřednictvím tzv. **handle** – instance ukazatele na proměnnou typu **FILE**, např.: **FILE **f;**
Datový typ FILE
Stavová a referenční proměnná proudu

• Tzv. neprůhledný datový typ (*Opaque Type*) – norma jazyka C stanovuje, že tento typ musí být v knihovně definován, ale nepředepisuje jak! (tj. různé implementace knihovny mohou tento typ definovat různě, např. MSVC vs GCC)

⇒ programátor se nesmí pokoušet pracovat s tímto typem přímo (přístupovat k jeho složkám), ale vždy jej jen předává jako referenční proměnnou daného streamu příslušným funkcí...

```c
FILE *inpf = fopen("test.dat", "w");
fputs("Hello!
", inpf);
fclose(inpf);
```

Datový typ FILE
Různé překladače – různé implementace

- Složky struktury nejsou normou jazyka C předepsány;
- definice struktury FILE z knihovny překladače Open Watcom Compiler Suite for C, C++, and F77:

```c
typedef struct __iobuf {
    unsigned char * __ptr;   /* next char
    int __cnt;               /* number of
    struct __s_link * __link; /* location
    unsigned __flag;        /* mode of f
    int __handle;           /* file hand
    unsigned __bufsize;     /* size of b
    unsigned short __ungotten; /* used by u
} FILE;
```

⚠️ Jakýkoliv pokus o práci přímo s položkami struktury (byť může fungovat) vede k nepřenositelnému programu!
Datový typ FILE
Různé překladače – různé implementace

• Definice struktury FILE z knihovny překladače Borland C/C++ Compiler 5.5:

```c
typedef struct {
    unsigned char *curp;  /* Current active
    unsigned char *buffer; /* Data transfer
    int level; /* fill/empty lev*/
    int bsize; /* Buffer size */
    unsigned short istemp; /* Temporary file*/
    unsigned short flags; /* File status fl*/
    wchar_t hold; /* Ungetc char if*/
    char fd; /* File descriptor*/
    unsigned char token; /* Used for valid*/
} FILE;
```

⚠️ Jakýkoliv pokus o práci přímo s položkami struktury (byť může fungovat) vede k nepřenositelnému programu!
Vztah proudu a diskového souboru
Různé úrovně abstrakce

- Proud představuje **vyšší úroveň abstrakce**, diskový soubor je jedna konkrétní realizace proudu;

- proud definuje pouze **mechanismus čtení/zápisu dat**, nikoliv např. uspořádání dat na médiu, operace s daty na médiu (jako je např. kopírování, přesouvání, mazání, přejmenování, apod.) – to je věcí operačního systému, knihovní funkce většinou jen žádají OS o provedení příslušné akce;

- proudy lze definovat i pro zařízení bez souborového systému (např. I/O porty, paměť, sítová zařízení, konzole, atp.);

- proudy lze **přesměrovat** (*Redirect*) a **zřetězit** (*Pipeline*), tzn. data zapisovaná do jednoho proudu se mohou ukládat do jiného proudu a jeden proud může být zdrojem dat pro jiný.
Standardní proudy
otevřené od počátku vykonávání programu

• V okamžiku předání řízení funkci `int main()` jsou k dispozici otevřené 3 předdefinované, tzv. standardní proudy;

• představují základní komunikační kanály, které může běžící proces využívat:

  ```
  FILE *stdin;  // standardní vstup, tj. klávesnice konzole
  FILE *stdout; // standardní výstup, tj. display konzole
  FILE *stderr; // standardní chybový výstup, většinou také display konzole (na Macu samostatné okno)
  ```

  za normálních okolností směřují všechny 3 do konzole, ze které byl program spuštěn (ale lze je přesměrovat)...

Programování v jazyce C
Standardní proudy
Přesměrování a zřetězení (prostředky OS)

• Cíl/zdroj dat proudu lze přesměrovat prostředky operačního systému:

  proud `stdout` (výstup na display konzole)
  je přesměrován do souboru `outf.txt`

  E:\C-Prog\demos>test.exe > outf.txt
  E:\C-Prog\demos>test.exe < text1.txt

  proud `stdin` (vstup z klávesnice konzole)
  je „krmen“ daty ze souboru `text1.txt`

• Zřetězení (Pipeline) – výstup (`stdout`) příkazu `type` (ale obecně jakéhokoliv programu) „krmi“ vstup (`stdin`) programu `test.exe`:

  E:\C-Prog\demos>type outf.txt.txt | test.exe
Standardní proudy
Přesměrování prostředky knihovny jazyka C

- Přesměrování lze dosáhnout také programově:

```
#include <stdio.h>

void main() {
    fclose(stdout);
    stdout = fopen("test.txt", "w");

    printf("Hello, world!\n");
}
```

Standardní proudy jsou normální proměnné, které je možné přiřazovat (teorie).

⚠️ POZOR: V některých implementacích (GCC, MSVC) je `stdin`/`stdout`/`stderr` makro, tzn. nelze přiřadit (není to L-value)!
Standardní proudy
Přesměrování prostředky knihovny jazyka C

• Programové přesměrování v případě definice proudů std... jako maker:

```c
#include <stdio.h>

void main() {
    fclose(stdout);
    freopen("test.txt", "w", stdout);

    printf("Hello, world!\n");
}
```

Zde je R-value OK!

→ výstup řetězce funkci `printf(·)` proběhne do souboru místo na display konzole
Práce s proudy/soubory
Otevření a uzavření

• Každý otevřený proud/soubor musí mít svojí referenční proměnnou typu FILE *, jinak není možné s ním pracovat (nebylo by jak sdělit příslušným funkcím, s jakým souborem mají operovat):

```c
#include <stdio.h>

void main() {
    FILE *f;

    f = fopen("test.txt", "w");
    fputs("Hello!\n", inpf);
    fclose(f);
}
```

název souboru, který se má otevřít

režim dat
w = zápis (Write)

uzavření souboru velmi důležité – vyprázdnuje vyrovnávací paměť proudu, tj. data z v. p. jsou zapsána na disk – to lze vynutit kdykoliv voláním funkce int fflush();
Otevření souboru a otázka přenositelnosti
Zápis absolutních cest, apod.

- Je-li uvedena celá (absolutní) cesta, včetně např. disku (ve Windows), může nastat problém při přenosu na UNIX:

  ```
  f = fopen("C:\Windows\err.log", "r");
  ```

  **zpětné lomítko musí být v řetězci zdvojeno (jinak uvozuje escape sekvenci)**

  **disk (Win) a svazek (UNIX) jsou natolik odlišné koncepce, že knihovna neposkytuje žádné konverzní mechanismy**

  **UNIX: OK, přirozené; Win: ? – záleží na implementaci knihovny (většinou se to převede na \)**

  ```
  f = fopen("testdir/test.txt", "w");
  ```

- Řešit podmíněným překladem, konverzní mechanismy knihovny mohou působit nečekané problémy.
Otevření souboru
Specifikace režimu průdu

\[
f = \text{fopen("filename.ext", "rb+");}
\]

**Specifikátor režimu**
max. 3 znaky, řetězcová konstanta!
(nikoli znaková)

**Povinný specifikátor**
způsobu přístupu
- \( r \) = čtení (read)
- \( w \) = zápis (write)
- \( a \) = přidání (append)

**Nepovinný specifikátor typu dat**
- \( b \) = binární (binary)
  není uveden = text

**Nepovinný specifikátor možnosti kombinovat způsob přístupu**
- \( + \) = lze kombinovat způsob přístupu (r/w)
  není uveden = nelze
Otevření souboru
Význam některých specifikací režimu proudu

```c
f = fopen("filename.ext", "wb");
for (i = 1; i <= 10; i++)
    fprintf(f, "Line %d\n", i);
```

(i ve Win) jen LF

```
Line 1 0A
Line 2 0A
Line 3 0A
```

• Když není specifikátor binárního proudu (b) uveden, jsou data považována za textová:

```c
f = fopen("filename.ext", "w");
for (i = 1; i <= 10; i++)
    fprintf(f, "Line %d\n", i);
```

```
Line 1 0D 0A
Line 2 0D 0A
Line 3 0D 0A
```

• **Textový proud**: speciální znaky (\n) mohou být interpretovány v závislosti na platformě.
Otevření souboru
Význam některých specifikací režimu proudu

\[
f = \text{fopen}("filename.ext", "w") \quad \text{— otevření pro zápis}
\]

- Když soubor \textbf{neexistuje}, \textit{vytvoří} se nový;
- když už \textbf{existuje}, bude zápis probíhat od začátku, tj. uložená \textbf{data se budou přepisovat}!

\[
f = \text{fopen}("filename.ext", "r") \quad \text{— otevření pro čtení}
\]

- Když soubor \textbf{neexistuje} \implies \textbf{chyba}! (ale prog. nemusí havarovat!)

\[
f = \text{fopen}("filename.ext", "w+") \quad \text{— komb. otevření}
\]

- Specifikátor + (\textit{update}) znamená, že do souboru lze jak \textbf{psát} (w), tak z něj \textbf{i číst}, avšak po zápisu \textbf{nesmí} bezprostředně následovat čtení – nastal by \textbf{problém s vyrovnávací pamětí}...

Je nutné vynutit její vyprázdění, např. funkcí \textbf{fsetpos}(), \textbf{fseek}(), \textbf{rewind}(), \textbf{fflush}() (totéž platí pro r+).
Otevření souboru
Testování úspěšnosti operace

- Funkce `FILE *fopen(·)` vrací v případě neúspěchu konstantu `NULL`:

  ```c
  FILE *fin = fopen("filename.ext", "r");
  if (fin == NULL) perror("Error! ...");
  else {
    ::
    fclose(fin);
  }
  ```

- Po otevření souboru je **vždy nutné zkontrolovat**, zda funkce `fopen(·)` vrátila platný ukazatel na referenční proměnnou – pokud ne, okamžitě ukončit vykonávání programu;

- některým funkcím (třeba `fscanf(·)`) totiž nevadí, když se jim předá jako ref. proměnná proudu hodnota `NULL`, akorát že v takovém případě nic nedělají!
Další souborové operace
Zavírání, vyprázdnování bufferu, atd.

FILE *freopen(char *filename, const char mode, FILE *stream);

- (i) zavře soubor daný referenční proměnnou stream;
- (ii) pokud je filename != NULL, otevře soubor stejně jako fopen(·);

je-li jméno souboru filename == NULL, pokusí se znovu otevřít původní soubor (je možné změnit režim).

zapíše všechna data z vyrovnávací paměti proudu na přiřazené výstupní zařízení (jde-li o výstupní proud, pro vstupní proud needef., stejně jako v případě komb. proudu, kdy posl. operace bylo čtení)

int fflush(FILE *stream);
int fclose(FILE *stream);

uzavře proud, zajistí zápis dat z vyrovnávací paměti
Bufferování
Nastavení bufferů a deaktivace bufferování

- Programátor může z různých důvodů chtít, aby měl k vyrovnávací paměti I/O operací přístup → je třeba nastavit v programu deklarováno pole jako vlastní buffer.

```c
void main() {
    char buf[BUFSIZ];
    setbuf(stdin, buf);
    ...
}
```

- Někdy je nutné, aby se objekt do proudu zapsal okamžitě, nikoliv až při naplnění bufferu – bufferování lze zakázat:

```c
void main() {
    setbuf(stdout, NULL);
    putchar('a');
    sleep(1);
    putchar('b');
}
```

Konstanta BUFSIZ určuje optimální velikost bufferu

Nebufferované I/O operace jsou výrazně pomalejší!
Bufferování
Speciální nastavení bufferů

• Funkce `setvbuf(·)` zajišťuje specifické nastavení bufferu pro konkrétní proud:

```c
int setvbuf(FILE *stream, char *buffer,  
            int mode, size_t size);
```

úplné bufferování (Full Buffering)
řádkové bufferování (Line Buffering)
bez bufferování (No Buffering)

• Změna velikosti bufferu (je-li známa nějaká lepší):

```c
if (setvbuf(fout, NULL, _IOFBF, 65536) != 0) {
    perror("Buffer setting failed.");
    return 1;
}
```

není předán nový buffer, tzn. mění se parametry stávajícího
Operace s otevřenými proudy
Neformátovaný vstup/výstup znaků

```c
int getchar(); // načte znak z konzole (ze stdin)
int putchar(int c); // zapíše znak na konzoli (do stdout)

int fgetc(FILE *stream); // načte znak z proudu
int getc(FILE *stream); // (totéž impl. jako makro)
int fputc(int c, FILE *stream); // uloží znak do proudu (fceu)
int putc(int c, FILE *stream); // (makro)
int ungetc(int c, FILE *stream); // „vrátí“ znak do proudu, takže další volání
getc() předá tento znak v návrat. hodn.

Pro práci se znaky (zejm. načítanými z proudů) se musí použít
typ int, protože #define EOF (-1), tj. integer!

getchar() ≡ getc(stdin) ≡ fgetc(stdin)
```
Operace s otevřenými proudy
Neformátovaný vstup/výstup řetězců

čte ze `stdin` znaky do str

```c
char *gets(char *str);
```

NEPOUŽÍVAT – NEBEZPEČNĚ
netestuje, zda se řetězec ze `stdin` do pole `str` vejde!

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

čte z proudu `stream` maximálně `sz - 1` znaků do pole `str`

- `fgets(·)` načítá znaky dokud (i) nenačte znak konce řádky,
  (ii) nedosáhne konce souboru, (iii) nenačte `sz - 1` znaků;
- za načtené znaky přidá `'\000'` ⇒ řetězec je řádně ukončen.

zapíše řetězec do `stdout` a odřádkuje

```c
int puts(const char *str);
```

```c
int fputs(const char *str, FILE *stream);
```

zapíše řetězec bez ukončovacího znaku `'\000'` do `stream`

Funkce `gets(·)` je z bezpečnostních důvodů ve standardu C99 označena jako deprecated a v C11 již úplně odstraněna!

💡
Operace s otevřenými proudy
Co na to „nové“ standardy jazyka C?

Funkce `gets()` je z bezpečnostních důvodů ve standardu C99 označena jako `deprecated` a v C11 již úplně odstraněna!

- Novější standardy také již neomezují I/O operace jen na 8-bitové znaky, ale zavádějí typ `wchar_t` (Wide Char), který představuje znak z rozšířené znakové sady (UTF-8);
- ke každé základní funkci z ANSI C existuje verze pro práci s „širokými“ znaky: `int getc(FILE *stream)`
  `wint_t getwc(FILE *stream)`

```c
do {
    wc = getwc(inpf);
    if (wc == L'\$') n++;
} while (wc != WEOF);
wprintf(L"\'$\' appeared %d times.\n", n);
```

Funkce jsou deklarované v headeru `wchar.h`
Operace s otevřenými proudy
Přepínání módu proudu

• Od standardu C95 je zavedena možnost přepínat mód všech I/O operací (pak není nutné volat spec. funkce `w...()`)  
  ```c
  int fwide(FILE *stream, int mode);
  ```
  mode < 0  nastaví 8-bitový mód
  mode == 0  dotaz na mód proudu
  mode > 0  nastaví wide-char mód

• Ve Windows působí problémy, protože např. konzole používá CP 852, tj. výstup UTF-8 je stejně nečitelný.
  ```c
  fmode = fwide(finp, 0);
  printf("The stream mode is %s.\n", fmode > 0 ? "wide" : "8-bit");
  if (fmode <= 0) fwide(finp, 1);
  ```
Operace s otevřenými proudy
Formátovaný výstup

formatovaný výstup na konzoli (\texttt{stdout})

\begin{verbatim}
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *buffer, const char *format, ...);
\end{verbatim}

formatovaný výstup \texttt{f} \equiv do souboru
\texttt{s} \equiv do pole charů (řetězce)
platí stejné pravidla jako u \texttt{printf}()

\begin{verbatim}
\ldots
char str[1024] = {0};
const float pi = 3.141592;

sprintf(str, "%.2f %.4f\n", pi, pi);
puts(str);
\ldots
\end{verbatim}

3.14 3.1416
**Formátovací řetězec**
Použitelný pro výstupní i vstupní* funkce

```c
printf("Hello, %s!\n", name);
```

---

**specifikátor formátu**
*(Format Specifier)*

---

**Obecný formát specifikátoru:**

```
% 
vlajky šířka .přesnost délka specifikace
```

---

**Základní specifikace:**

<table>
<thead>
<tr>
<th>Specifikace</th>
<th>Odejde</th>
<th>Popis</th>
<th>Ukládané hodnoty</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d</code> nebo <code>i</code></td>
<td>celé číslo – znaménkové</td>
<td>–279</td>
<td></td>
</tr>
<tr>
<td><code>u</code></td>
<td>celé číslo – neznaménkové</td>
<td>5386</td>
<td></td>
</tr>
<tr>
<td><code>x</code></td>
<td>hexadecimální celé číslo</td>
<td>f7ba</td>
<td></td>
</tr>
<tr>
<td><code>X</code></td>
<td>hexadecimální celé číslo <em>(upcase)</em></td>
<td>F7BA</td>
<td></td>
</tr>
<tr>
<td><code>f</code></td>
<td>reálné číslo</td>
<td>392.65</td>
<td></td>
</tr>
<tr>
<td><code>e</code></td>
<td>vědecká notace reál. čísla</td>
<td>3.9265e+2</td>
<td></td>
</tr>
<tr>
<td><code>c</code></td>
<td>znak</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td><code>s</code></td>
<td>řetězec znaků</td>
<td>Hello!</td>
<td></td>
</tr>
</tbody>
</table>
Formátovací řetězec
Méně běžné specifikace

Další specifikace:

<table>
<thead>
<tr>
<th>Specifikace</th>
<th>Obrázek</th>
<th>Popis</th>
<th>Externí příklad</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td></td>
<td>oktalové celé číslo</td>
<td>6105</td>
</tr>
<tr>
<td>F C99</td>
<td></td>
<td>reálné číslo (upcase)</td>
<td>-37.25</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>vědecká notace (upcase)</td>
<td>-3.725E1</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td>nejkratší výpis %e či %f</td>
<td>-37.25</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>nejkratší výpis %e či %f (upcase)</td>
<td>-37.25</td>
</tr>
<tr>
<td>a C99</td>
<td></td>
<td>hexadecimální reál. číslo</td>
<td>-0xc.90fep-2</td>
</tr>
<tr>
<td>A C99</td>
<td></td>
<td>hexadecimální reál. číslo (upcase)</td>
<td>-0XC.90FEP-2</td>
</tr>
<tr>
<td>p</td>
<td></td>
<td>ukazatel</td>
<td>7FA02000</td>
</tr>
</tbody>
</table>

printf("%+010.3f\n", pi);  
printf("%10d\n", 2016);  

Chceme-li vypsát znak %, musí se ve formátovací řetězci zdvojit: printf("100%%");
Formátovací řetězec
Vlajky a šířka zobrazení

**Vlajky (Flags):**
- (mínus) zarovnávání vlevo (default vpravo)
+ tiskné znaménko vždy (i kladné)

```c
printf("%-7d
", 1);
printf("%+7d
", 1);
printf("%07d
", 1);
```

(mezera) u nezáporné hodnoty místo znaménka mezeru vyplňuje zleva nulami do určené šířky
0 | x | X: vypíše 0 | 0x | 0X před hodnotu
# a | A | e | E | f | F | g | G: vypisuje vždy desetinnou čárku (tečku), i když následuje 0

```c
printf("%7.f
", 2.0);
printf("%#7.f
", 2.0);
printf("%#7x
", 65535);
```
Formátovací řetězec
Přesnost a šířka zobrazení

Přesnost: číslo, určující kolik číslic za desetinnou čárkou se vytiskne (default = 6), a tedy řád zaokrouhlení
* ≡ přesnost není určena specifikátorem, ale argumentem (uvedeným před formátovanou hodnotou):

```c
printf("%07.3f\n", pi);
printf("%07.*f\n", 5, pi);
```

Šířka: číslo, určující kolik minimálně znaků se vytiskne
* ≡ šířka není určena specifikátorem, ale argumentem (uvedeným před formátovanou hodnotou):

```c
printf("%8d\n", 1);
printf("%*d\n", 9, 1);
```
Formátovací řetězec
Délka (modifikace dat. typu) zobrazení

Modifikátor délky způsobuje „změnu“ interpretace datového typu argumentu (tj. přetypování pro potřeby výstupu):

<table>
<thead>
<tr>
<th>Délka</th>
<th>Specifikátor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(nespec.)</td>
<td>int unsigned int double int char* void* int*</td>
</tr>
<tr>
<td>hh</td>
<td>signed char unsigned char signed char*</td>
</tr>
<tr>
<td>h</td>
<td>short int unsigned short int short int*</td>
</tr>
<tr>
<td>l</td>
<td>long int unsigned long int wint_t wchar_t* long int*</td>
</tr>
<tr>
<td>ll</td>
<td>long long int unsigned long long int long long int*</td>
</tr>
<tr>
<td>j</td>
<td>intmax_t uintmax_t intmax_t*</td>
</tr>
<tr>
<td>z</td>
<td>size_t size_t size_t*</td>
</tr>
<tr>
<td>t</td>
<td>ptrdiff_t ptrdiff_t ptrdiff_t*</td>
</tr>
<tr>
<td>L</td>
<td>long double</td>
</tr>
</tbody>
</table>

↑ http://www.cplusplus.com/reference/cstdio/printf/

žluté podbarvení ≡ C99

```
printf("%hhX\n", 0x12345678);
printf("%hd\n", 0x7FFFFFFF);
printf("%lld\n", 0x7FFFFFFF);
```

5678
-1
2147483647
Operace s otevřenými proudy
Formátovaný vstup

formátovaný vstup z konzole (*stdin*)

```c
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(char *buffer, const char *format, ...);
```

formátovaný vstup \(f \equiv \) ze souboru
\(s \equiv \) z pole charů (řetězce)

platí stejné pravidla jako u printf(·)

```c
char buf[80] = "3.14 2.72";
float x = 0, y = 0;

sscanf(buf, "%f %f", &x, &y);
printf("%f %f\n", x, y);
```

3.140000 2.720000
#include <stdio.h>

int main() {
    FILE *fr;
    int x;

    fr = fopen("numbers.txt", "r");
    if (!fr) exit(1);

    while (fscanf(fr, "%d", &x) == 1)
        printf("\%d\n", x);
    fclose(fr);

    return 0;
}

• Funkce scanf(·) vrací počet načtených hodnot → lze využít: testování/zastavovací podm.
Operace s otevřenými proudy
Detekce konce řádku

- Různé platformy – různé realizace konce řádku (Windows, OS/2, Symbian, Palm OS – CR LF; Unix a Unix-like systémy jen LF; MacOS do verze 9 jen CR) \(\Rightarrow\) nelze konstantou překladač ale "umí" escape pro novou řádku

```c
while ((c = getc(finp)) != '\n') ...
```

- Ve Windows následuje při čtení po znaku ale za CR ještě LF ('\n' je vlastně 2-znaková konstanta), tj. je třeba o něm vědět a případně ho odfiltrovat:

```c
while ((c = getc(finp)) != '\n') {
    if (c >= ' ') ...
}
```

- Filtr ASCII hodnotou, v ASCII mají řídicí znaky nižší hodnotu než mezera (0x20).
Operace s otevřenými proudy
Detekce konce souboru

- V headeru `stdio.h` je definovaný symbol EOF (End-Of-File), většinou prostřednictvím preprocesoru takto:
  ```c
  #define EOF (-1)
  ```
  tj. po rozvinutí makra je to dekadická konstanta v kódu → int

- `getc()` vrací EOF při pokusu číst za koncem souboru:
  ```c
  while ((c = getc(finp)) != EOF) ...
  ```
  čtení po znaku až do konce souboru

⚠️ POZOR! Porovnáváme-li načitaný znak s konstantou EOF, musí být typu `signed int`, jinak může dojít k implicitnímu přetypování, např.:  
  ```c
  ((unsigned char) -1) == 255
  ```

- Bezpečnější (ale pomalejší) je použití funkce `feof()`.
#include <stdio.h>

```c
int main() {
    FILE *fin, *fout;
    int c;

    fin = fopen("source.txt", "r");
    fout = fopen("target.txt", "w");

    while ((c = getc(fin)) != EOF)
        putc(c, fout);

    fclose(fin);
    fclose(fout);

    return 0;
}
```
Operace s otevřenými proudy
Přímý (nebufferovaný) vstup a výstup

načítá blok dat z proudu

```c
size_t fread(void *buffer, size_t size,
            size_t count, FILE *stream);
size_t fwrite(const void *buffer, size_t size,
             size_t count, FILE *stream);
```

zapisuje blok dat do proudu

- Čtený/zapisovaný blok dat je typu `void *`, tzn. funkci je předána pouze adresa – na této adrese může být cokoliv (např. struktura, pole, proměnná);
- param. `size` určuje, jaká je velikost zapisovaného objektu (v bytech), param. `count` říká, kolik takových objektů se má načíst/zapsat → tzn. `size` ~ `count`: lze s tím laborovat; ⚠️
- funkce vrací počet přečtených/zapsaných objektů.
Operace s otevřenými proudy
Přímý (nebufferovaný) vstup a výstup

#include <stdio.h>
#define SIZE 5

void main() {
    int nr = 0;
    double arr[SIZE] = {1, 2, 3, 4, 5};

    FILE *f = fopen("doubles.bin", "wb");
    nr = fwrite(arr, sizeof(arr[0]), SIZE, f);
    nr = fwrite(arr, 1, sizeof(arr[0]) * SIZE, f);
    nr = fwrite(arr, sizeof(arr[0]) * SIZE, 1, f);
    fclose(f);

    f = fopen("doubles.bin", "rb");
    nr = fread(arr, sizeof(arr[0]), SIZE, f);
    fclose(f);
}
Úspěšnost vstupně-výstupních operací
Indikace chybových stavů

• Obecně platí, že funkce z knihovny `stdio` nějak indikují (ne)úspěch operace, často návratovou hodnotou:

```c
#include <stdio.h>
...
if ((finp = fopen("data.bin", "r")) == NULL) {
    printf("Cannot open file...
");
    exit(1);
}
...
if (fclose(finp) == EOF) {
    printf("Cannot close file...
",
    exit(1);
}
...
```

• Naneštěstí v tom není moc systém (každá funkce indikuje chybový stav jinak).
Úspěšnost vstupně-výstupních operací
Získávání informací o chybě

• Dojde-li k chybě při I/O operaci, je její kód uložen do globální proměnné `errno` definované v knihovně `errno`:

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

::
errno = 0;
finp = fopen("data.bin", "r");
if (!finp) {
    printf("Error \"%s\" while opening file.\n",
            strerror(errno));
    exit(1);
}
::

• `strerror()` převede kód chyby na text. chybové hlášení.

před testovanou operací vynulovat chybový kód (může tam být něco z minula)
Přímý přístup k souboru
Nastavování pozice v souboru pro čtení/zápis

```c
int fseek(FILE *stream, long offs, int from);
```

- Nastavuje pozici v otevřeném souboru `stream` — hodnota `offs` udává, o kolik byte se pozice změní vzhledem k (1) začátku souboru, (2) aktuální pozici či (3) konci souboru;

- vrací 0 v případě úspěchu, jinak chybový kód;
- `offs` může být i záporné číslo (ne však se SEEK_SET);
- je-li `offs` kladný a `from` SEEK_END, soubor se zvětší o daný počet bytů – obsah takto vzniklé oblasti je nedef.;
- z bezpečnostních důvodů používat jen na binární proudy;
- u textových proudů je bezpečné jen nastavení na začátek: `fseek(tf, 0, SEEK_SET)` a na konec souboru: `fseek(tf, 0, SEEK_END)`. interpretace znaků 'n'
Přímý přístup k souboru
Zjišťování pozice v otevřeném souboru

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

- Zjišťuje pozici v otevřeném souboru `stream` – návratová hodnota představuje počet bytů od začátku souboru;
- v případě chyby vrací `EOF`.

⚠️ **POZOR!** Návratová hodnota je typu `long int`, tzn. na většině platform nemůže pracovat se soubory většími než 2 GB → viz `fsetpos()` a `fgetpos()`.

- `ftell()` selhává při pokusu o zjištění pozice v proudech, které nejsou spojeny se soubory uloženými na paměťovém médii, a tedy nemají definovanou délku (např. konzolové proudy `stdin`, `stdout`, apod.).
Přímý přístup k souboru
Problém velmi velkých souborů (> 2GB)

```c
fpos_t position;
fgetpos(finp, &position);
wb = fwrite(..., finp);

fsetpos(finp, &position);
```

- ukažatel na strukturu, do které se ukládá aktuální pozice ve velkém souboru
- uložení pozice
- souborová operace
- návrat na původní pozici před zápisovou operací

- Obě funkce vrací v případě úspěchu 0, jinak kód chyby ⇒
  je rozumné v programu testovat výsledek operace:

```c
fpos_t pos = {0};
int rc = fgetpos(stdin, &pos);
if (rc) perror("Unable to tell position!");
```
Přímý přístup k souboru
Nastavení na začátek a detekce konce souboru

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

- Nastavuje pozici v otevřeném souboru `stream` na začátek, tj. shodné s voláním `fseek(stream, 0, SEEK_SET)`, ale jsou vynulovány příznaky chyby a konce souboru;
- funkce neposkytuje informaci o úspěchu operace.

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

- Testuje, zda bylo dosaženo konce soubor. proudu `stream`, pokud ano, vrací nenulovou hodnotu;
- funkce **jen předává informaci ze stavové struktury** proudu (hodnotu příznaku konce souboru), sama zdroj dat nezkou- má, tj. indikuje dosažení konce až poté, co nějaká I/O funkce (např. `fgetc(·)` ) na konec narazí a upraví hodnotu příznaku ve stavové struktuře proudu `FILE`. 
Management chybových stavů proudů
Funkce pro detekci chybového stavu

```c
int ferror(FILE *stream);
```

- Zkoumá, zda při poslední I/O operaci s proudem `stream` nastala chyba – pokud ano, vrací její kód (jinak 0);
- jakmile se chyba objeví, funkce jí hlásí stále, dokud není příznak chyby vynulován.

```c
void clearerr(FILE *stream);
```

- Nuluje příznak chyby a konce souboru ve stavové struktuře `FILE` otevřeného souborového proudu `stream`;

```c
... 
fwrite(...);
if (ec = ferror(finp)) {
    printf("I/O error %#d!\n", ec);
    clearerr(finp);
}
```
Vysokoúrovňové operace se soubory
Přejmenování a odstranění

```c
int rename(const char *old, const char *new);
```

- Změní jméno souboru `old` na `new` – nepracuje s otevřeným proudem prostřednictvím stav. strukt. `FILE`, ale pouze se jmény ⇒ provedení operace je svěřeno OS;
- v případě chyby vrací její kód, jinak 0;
- pokud soubor se jménem `new` již existuje, závisí chování na implementaci knihovny a OS (norma je nedefinuje).

```c
int remove(const char *filename);
```

- Odstraní soubor se jménem `filename` (stejný mechanismus jako ↑) – OS určuje, co se vlastně přesně stane (koš);
- v případě chyby vrací její kód, jinak 0;
Vysokoúrovňové operace se soubory
Vytvoření dočasného souboru

```c
FILE *tmpfile();
```

- Vytvoří dočasný soubor s nekonfliktním jménem (tj. nedojde k přepsání žádného souboru) a otevře ho v režimu "wb+";
- takových dočasných souborů si může proces vytvořit přinejmenším `TMP_MAX` (může být dále omezeno `FOPEN_MAX`);
- soubor není třeba zavírat, OS ho zlikviduje při ukončení procesu, jinak se funkce chová stejně jako `fopen()`.

```c
FILE *ftmp = tmpfile();
if (!ftmp) {
    perror("Cannot create temp file!");
    exit(1);
}
fwrite(buf, sizeof(buf[0]), BLEN, ftmp);
...```
Vysokoúrovňové operace se soubory
Vygenerování nekonfliktního jména souboru

- Nevyhovuje-li chování `tmpfile()`, může si programátor založit dočasný soubor sám (např. nechce, aby zanikl):

  ```c
  char *tmpnam(char *filename);
  ```

- Vygeneruje unikátní jméno souboru (o maximální délce `L_tmpnam`) a uloží do řetězce `filename`;
- pokud `filename` ← NULL, pak vrací ukazatel na statickou oblast naplněnou vygenerovaným jménem;
- není-li možné unikátní jméno vygenerovat, vrací NULL.

```c
char tmpname1[L_tmpnam] = {0};
char *tmpname2 = tmpnam(NULL);

if (tmpnam(tmpname1)) {
    printf("Temp file name: %s\n", tmpname1);
    ftmp = fopen(tmpname1, "wb");
    ...
```