

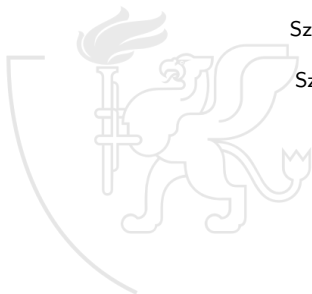
# Programozás Alapjai

Dr. Gergely Tamás  
Dr. Jász Judit

Szegedi Tudományegyetem  
Informatikai Intézet  
Szoftverfejlesztés Tanszék

2021

(v0901)



- 1 **Bemutakozás**
  - Kurzus információk
  - A SZTE és az informatikai képzés
- 2 **Linux**
  - Alapfogalmak
  - Linux parancsok
  - Linux shell
  - Felhasználók
  - Hálózat
- 3 **Gyors C áttekintés**
  - Bevezető
  - Pénzváltás (1. verzió)
  - Pénzváltás (2. verzió)
  - Röppálya számítás
  - Röppálya szimuláció
  - Az év napja
  - Csúszoátlag adott elemszámmra
  - Csúszoátlag parancssorból
  - Basename standard inputról
  - Basename parancssorból
  - Tér legtávolabbi pontjai
  - A nappalis gyakorlat értékelése

- 4 **Alapok**
  - Alapfogalmak
  - A programozás fázisai
  - Algoritmus vezérlése
  - A C nyelvű program
  - Szintaxis
  - A C nyelv elemi adattípusai
  - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
  - Bevezetés
  - Szekvenciális vezérlés
  - Függvények
  - Szelekciós vezérlések
  - Ismétléses vezérlések 1.
  - Eljárásvezérlés
  - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
  - Az adatkezelés szintjei
  - Elemi adattípusok
  - Pointer adattípus
  - Tömb adattípus

- Sztringek
  - Pointerek és tömbök C-ben
  - Rekord adattípus
  - Függvény pointer
  - Halmaz adattípus
  - Flexibilis tömbök
  - Láncolt listák
  - Típusokról C-ben
- 8 **IO**
    - Alapok
    - Adatállományok
  - 9 **C fordítás**
    - A fordítás folyamata
    - A preprocessor
    - A C fordító
    - Assembler
    - Linker és modulok
  - 10 **Gyakorlati kérdések**
    - Memóriahasználát
    - Gyakori C hibák
    - where.c felboncolva

- 1 Bemutakozás**
  - Kurzus információk
  - A SZTE és az informatikai képzés
- 2 Linux**
  - Alapfogalmak
  - Linux parancsok
  - Linux shell
  - Felhasználók
  - Hálózat
- 3 Gyors C áttekintés**
  - Bevezető
  - Pénzváltás (1. verzió)
  - Pénzváltás (2. verzió)
  - Röppálya számítás
  - Röppálya szimuláció
  - Az év napja
  - Csúszoátlag adott elemszámmra
  - Csúszoátlag parancssorból
  - Basename standard inputról
  - Basename parancssorból
  - Tér legtávolabbi pontjai
  - A nappalis gyakorlat értékelése

- 4 Alapok**
  - Alapfogalmak
  - A programozás fázisai
  - Algoritmus vezérlése
  - A C nyelvű program
  - Szintaxis
  - A C nyelv elemi adattípusai
  - A C nyelv utasításai

- 5 Vezérlési szerkezetek**
  - Bevezetés
  - Szekvenciális vezérlés
  - Függvények
  - Szelekciós vezérlések
  - Ismétléses vezérlések 1.
  - Eljárásvezérlés
  - Ismétléses vezérlések 2.

- 6 Folyamatábra és struktúradiagram**
- 7 Adatszerkezetek**
  - Az adatkezelés szintjei
  - Elemi adattípusok
  - Pointer adattípus
  - Tömb adattípus

- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

- 8 IO**
  - **Alapok**
  - Adatállományok

- 9 C fordítás**
  - A fordítás folyamata
  - A preprocessor
  - A C fordító
  - Assembler
  - Linker és modulok

- 10 Gyakorlati kérdések**
  - Memóriahasználát
  - Gyakori C hibák
  - where.c felboncolva

- Mint azt már említettük, a be- és kiviteli (I/O) szolgáltatások nem részei a C nyelvnek.
- A C szabvány szerint ugyanakkor rendelkezésre áll egy standard I/O függvénykönyvtár, melynek használatához a programunkban szerepelnie kell az

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

sornak.

- C-ben megkülönböztetünk
  - Alacsony szintű
  - Magas szintűfájlkezelést.
- Magas szintű fájlkezelés esetén külön beszélünk az úgynevezett standard fájlok kezeléséről.

# Magas szintű fájlkezelés

- A hordozhatóság érdekében ajánlott ezt használni.
- Az adatokat egy adatfolyamnak (stream) tekintjük.
- A stream I/O pufferezett, vagyis a fizikai írás/olvasás nagyobb darabokban történik, a műveleteket pedig ezután amíg lehet a sokkal gyorsabb memóriából szolgáljuk ki. A puffer hossza a `stdio.h`-ban van definiálva:

```
#define BUFSIZ _IO_BUFSIZ
#define _G_BUFSIZ 8192
#define _IO_BUFSIZ _G_BUFSIZ
```



# Magas szintű fájlkezelés

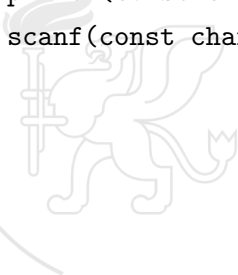
## Standard fájlok

- Linux alatt említettük a standard bemeneti (input), kimeneti (output) és hiba (error) csatornákat.
- A C programokban ezek
  - `stdin`,
  - `stdout` és
  - `stderr`néven elérhetőek.
- Mindhárom magas szinten, szöveges módban kezelt fájl, és az `stdio.h`-ban vannak deklarálva.
- A program indulásakor automatikusan úgy nyílnak meg, hogy
  - az `stdin` a billentyűzethez,
  - az `stdout` és az `stderr` a képernyőhözrendelődik (hacsak az operációs rendszer ezt felül nem írja, de a programunk abból semmit sem vesz észre).
- A csatornákat az `stdio.h`-beli fájlkezelő függvények segítségével kezelhetjük.

# Fájlműveletek standard fájlokon

## Lista

- `int getchar(void)`
- ~~`char *gets(char *buf)`~~
- `char *gets_s(char *buf, size_t n)`
- `int putchar(int ch)`
- `int puts(const char *buf)`
- `int printf(const char *format, ...)`
- `int scanf(const char *format, ...)`



- `int getchar(void)`

Egy karakter beolvasása a standard inputról (`stdin`).

*returns* Ha a művelet sikeres volt, akkor a beolvasott karakter kódja, és veszteség nélkül konvertálható `char` típusúvá. Ha hiba történt vagy a bemenet végére ért (például lenyomtuk a `<ctrl> + d` billentyűkombinációt Linux-ban), akkor EOF.

- Implementációtól függően, de általában pufferelten olvas: A program csak akkor kapja meg a begépelte karakter(ek)e)t, ha a puffer betelt vagy sorvéget (fájlvéget) nyomtunk.
- Közvetlen billentyűleütés érzékelésére nem alkalmas, de nem is ez a feladata. (A közvetlen billentyűleütést csak operációs rendszer függő módon tudnánk figyelni.)



# Fájlműveletek standard fájlokon

## gets

- ~~char \*gets(char \*buf)~~

Egy sor beolvasása a standard inputról (`stdin`) a `buf` által mutatott területre. **C11 óta nem használható!**

**buf** A puffer, amiben sztringként eltárolja a beolvasott sort.

**returns** Az eltárolt sor címe (`buf`), ha sikerült a művelet, különben `NULL`.

- A `buf` egy létező és elegendő hosszú karaktertömb kell, hogy legyen, mert a függvény nem ellenőrzi, hogy szabad vagy foglalt memóriaterületet ír-e felül.
- A sor közbeni fájlvége esetén lezárja a sztringet.
- Az újsor karaktert beolvassa, de nem tárolja el, hanem helyette lezárja a sztringet egy `'\0'` karakterrel (ezért nem alkalmas az újsor karakter beolvasására).

- `char *gets_s(char *buf, size_t n)`

Egy sor (de legfeljebb  $n-1$  karakter) beolvasása a standard inputról (stdin) a `buf` által mutatott területre. C11 óta használható.

**buf** A puffer, amiben sztringként eltárolja a beolvasott sort.

**n** Pufferméret. Legfeljebb annyi karaktert olvas be, hogy sztringként még el tudja tárolni  $n$  karakteren.

**returns** Az eltárolt sor címe (`buf`), ha sikerült a művelet, különben NULL.

- A `buf` egy létező és legalább  $n$  hosszú karaktertömb kell, hogy legyen.
- A sor közbeni fájlvége esetén lezárja a sztringet.
- Az újsor karaktert beolvassa, de nem tárolja el, hanem helyette lezárja a sztringet egy `'\0'` karakterrel (ezért nem alkalmas az újsor karakter beolvasására).

- `int putchar(int ch)`  
Egy karakter kiírása a standard outputra (`stdout`).
  - `ch` A kiírandó karakter.
  - `returns` Ha a művelet sikeres volt, akkor a kiírt karakter kódja, hiba esetén EOF.
- Általában nem pufferezt.



# Fájlműveletek standard fájlokon

puts

- `int puts(const char *buf)`

Egy sztring kiírása a standard outputra (`stdout`).

`buf` A sztring, amit ki kell írni.

`returns` Egy nemnegatív érték ha sikerült a művelet, különben EOF.

- A sztringvégi `'\0'` karakter helyett újsort (`'\n'`) ír a kimenetre.



# Fájlműveletek standard fájlokon

Példa (tolower.c [1-14])

- A bemenet kisbetűssé alakítása:

```
1 /* Input szöveg kisbetűssé alakítása.
2  * Dévényi Károly, devenyi@inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6 #include <ctype.h>
7
8 int main() {
9     int c;
10    while ((c = getchar()) != EOF) {
11        putchar(tolower(c));
12    }
13    return 0;
14 }
```



- `int printf(const char *form, ...)`

A standard kimenetre (`stdout`) kiírja az aktuális paramétereinek az értékét a formátumsztring (`form`) alapján.

*form* Formátumsztring.

*...* További paraméterek

*returns* A kiírt karakterek száma, hiba esetén negatív érték.

- A formátumsztring kétféle típusú objektumot tartalmaz:
  - Közönséges karaktereket, amelyeket egyszerűen a kimeneti folyamra másol.
  - Konverzió-specifikációkat, amelyek mindegyike a soron következő paraméter konvertálását és kiíratását írja elő.

- Kiíráskor speciálisan kezelt (vezérlő) karakterek:
  - `\a` Hangjelzés (ha a terminálon engedélyezett).
  - `\n` Új sor kezdése.
  - `\r` Sor elejére ugrás.
  - `\b` Egy karakterrel visszább ugrás a sorban.
  - `\t` Ugrás a következő tabulátor pozícióra.
  - `\v` Soremelés a kurzor vízszintes mozgatása nélkül.
  - `\f` Lapdobás.



- Egy konverzió-specifikáció (más néven formátumvezérlő szekvencia) általános alakja:
  - %[flag] [width] [.precision] [lmod] type
    - flag** Több karakter, ami többféle kiíratási jellemzőt kapcsolhat be. Elhagyható.
    - width** Egy szám, a kiíratás szélességét határozza meg. Elhagyható.
    - .precision** Egy szám, a kiíratás pontosságát adja meg. Elhagyható.
    - lmod** Több karakter, a megadott típus méretét módosítja. Elhagyható.
    - type** Egy karakter, a paraméter típusát és a kiíratás alapvető formáját határozza meg. Kötelező.



# printf

`%[flag][width][.precision][lmod]type - [1/3]`

- Egy karakter az alábbiak közül, ami a paraméter típusát és a kiíratás alapvető formátumát határozza meg:
  - `d, i` A paraméter valamilyen előjeles egész (`int`) típusú, és decimális alakban kell kiíratni.
  - `u` A paraméter valamilyen előjeltelen egész (`unsigned int`) típusú, és decimális alakban kell kiíratni.
  - `o` A paraméter valamilyen előjeltelen egész (`unsigned int`) típusú, és oktális alakban kell kiíratni.
  - `X, x` A paraméter valamilyen előjeltelen egész (`unsigned int`) típusú, és hexadecimális alakban kell kiíratni.
  - `c` A paraméter karakter (`char`) típusú, és karakterként kell kiíratni.

# printf

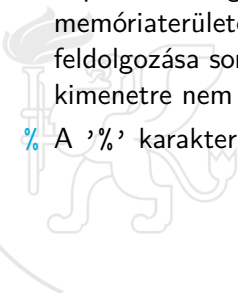
`%[flag] [width] [.precision] [lmod] type - [2/3]`

- f** A paraméter valamilyen valós (`double`, `float`) típusú, és tizedestört alakban kell kiíratni. A tizedesjegyek számát a pontosság fogja megadni (ez alapértelmezés szerint 6) és nincs exponens kiíratás.
- E, e** A paraméter valamilyen valós (`double`, `float`) típusú, és exponenciális alakban ( $x \cdot 10^{\text{exp}}$ ,  $1 \leq x < 10$ ) kell kiíratni. A tizedesjegyek számát a pontosság fogja megadni (ez alapértelmezés szerint 6) és mindig lesz exponens kiíratás.
- G, g** A paraméter valamilyen valós (`double`, `float`) típusú, és a tizedestört illetve az exponenciális alakok közül a rövidebbet fogja alkalmazni.

# printf

`%[flag] [width] [.precision] [lmod] type - [3/3]`

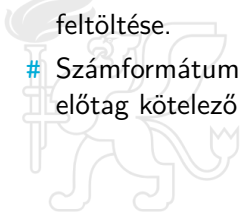
- s** A paraméter egy sztring (`char*`), ezt kell kiírni. A kiírt karakterek számát a pontosság korlátozza, ami alapértelmezett esetben végtelen.
- p** A paraméter egy pointer érték, ezt kell hexadecimális alakban kiírni.
- n** A paraméter egy `int*` pointer, és az ez által mutatott memóriaterületen kell eltárolni a formátumsztring eddigi feldolgozása során kiírt karakterek számát. A standard kimenetre nem írunk semmit.
- %** A `'%'` karakter kerül kiírásra.



# printf

`%[flag][width][.precision][lmod]type`

- Nulla vagy több karakter az alábbiak közül, ami a kiíratás formátumát módosíthatja:
  - A megadott szélességen belül a tartalom balra igazítása.
  - + Számok esetén az előjel kötelező kiírása.
  - szóköz Számok esetén az előjel helyének fenntartása.
  - 0 Számok esetén a megadott szélesség balról nullákkal való feltöltése.
  - # Számformátumra jellemző alak: tizedespont, kezdő 0, 0x előtag kötelező használata.



# printf

`%[flag] [width] [.precision] [lmod] type`

- Egy egész érték, ami a minimális mezőszélességet határozza meg. Ha a következő paraméter ennél kevesebb kiírandó karakterből áll, akkor a fennmaradt helyet szóközzel (vagy számok, jobbra igazítás és a 0 flag egyidejű használata esetén '0' karakterekkel) tölti ki. A - flag nélkül a rövidebb tartalmat a mező jobb széléhez igazítja. Ha a paraméter képe a szélességnél több karakterből áll, akkor nincs hatása.

**szám** Egy decimális számleírás.

- \* A mezőszélesség értékét a következő, `int` típusú paraméter határozza meg.
- Az alábbi két kiíratás egyenértékű:

```
printf("%5d", 125);  
printf("%*d", 5, 125);
```

# printf

%[flag] [width] [.precision] [lmod] type

- Egy egész érték, ami a kiíratás pontosságát határozza meg. Egész szám esetén ez legalább ennyi számjegy kiíratását jelenti, szükség esetén balról '0' karakterekkel kiegészítve. Valós szám esetén a kiírandó tizedesjegyek számát jelenti, sztringek esetén pedig a paraméterből kiírandó karakterek maximális számát határozza meg.

**.szám** Egy decimális száMLEÍRÁS (a pont előtte mindenképpen szükséges).

- \* A pontosság értékét a következő, int típusú paraméter határozza meg.

- Az alábbi két kiíratás egyenértékű:

```
printf("%.5lf", 12.5);  
printf("%.*lf", 5, 12.5);
```

# printf

`%[flag][width][.precision][lmod]type`

- Karakterek, amik a következő paraméter típusát pontosítják:
  - hh** Az egész int típusúként megadott paraméter valójában `char` méretű.
  - h** Az egész int típusúként megadott paraméter valójában `short int` méretű.
  - l** Az egész int típusúként megadott paraméter valójában `long int` méretű.
  - L** A valós `double` típusúként megadott paraméter valójában `long double` méretű.
  - ll** Az egész int típusúként megadott paraméter valójában `long long int` méretű.

# Valós értékek kezelése

## Kerekítés és csonkítás (io-printf-double.c [1-13])

```
1 /* Konverzió valós és egész számok között.
2  * 2014. November 19. Gergely Tamás, gertom.inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6
7 int main () {
8     double d;
9     for (d = -2.0; d <= 2.0; d += 1.0/3.0) {
10         printf("d=%23.20lf;□%5.2lf;□%2.0lf;□i=%2d;\n", d, d, d, (int) d);
11     }
12     return 0;
13 }
```

```
d=-2.00000000000000000000; -2.00; -2; i=-2;
d=-1.666666666666666666674068; -1.67; -2; i=-1;
d=-1.3333333333333333333348136; -1.33; -1; i=-1;
d=-1.0000000000000000000022204; -1.00; -1; i=-1;
d=-0.6666666666666666666696273; -0.67; -1; i= 0;
d=-0.3333333333333333333364790; -0.33; -0; i= 0;
d=-0.0000000000000000000033307; -0.00; -0; i= 0;
d= 0.33333333333333333333298176; 0.33; 0; i= 0;
d= 0.6666666666666666666629659; 0.67; 1; i= 0;
d= 0.9999999999999999999955591; 1.00; 1; i= 0;
d= 1.33333333333333333333281523; 1.33; 1; i= 1;
d= 1.6666666666666666666607455; 1.67; 2; i= 1;
d= 1.9999999999999999999933387; 2.00; 2; i= 1;
```



# Függvényparaméterek

## Argumentumok átadása

- Mi történik egy függvény meghívásakor? Ezt a fordító által használt ABI (Application Binary Interface), azon belül is a hívási konvenció határozza meg.
- Linux, i386 architektúra:
  - Minden argumentum a veremben kerül átadásra, jobbról balra sorrendben (vagyis az első argumentum utolsóként kerül a verembe). Technikai információk (pl. visszatérési cím) az argumentumok után szintén a verembe kerülnek. A hívó „takarít”.
- Linux, x86-64 architektúra:
  - Az első 6 egész (vagy egészként értelmezhető) és az első 8 valós argumentum megadott regiszterekben kerül átadásra. A további argumentumok (az i386 architektúrához hasonló módon) a veremben kerülnek átadásra, jobbról balra sorrendben (vagyis a korábbi argumentum később kerül a verembe). Technikai információk (pl. visszatérési cím) az argumentumok után szintén a verembe kerülnek. A hívó „takarít”.

- Mi történik a `printf()` hívásakor?

```
printf("%d□%d□%d\n", 10, 20);
```

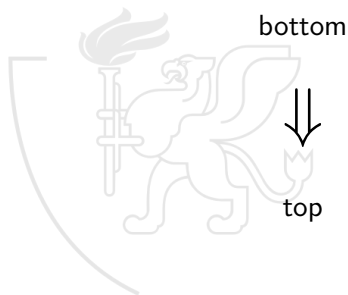
- A verembe (stack) belekerülnek az argumentumok (legalábbis i386 architektúrán), de hogyan?
- Lényeges, hogy a veremből kiolvasásnál a formátumsztringet a többi argumentum ismerete nélkül megtaláljuk.
- Mivel a verem teteje az egyetlen fix viszonyítási pont, a formátumsztringnek ide (vagy ebből fixen számolható címre) kell kerülnie, vagyis ezt a további argumentumok után kell a verembe rakni.

# Függvényparaméterek

## Veremkezelés [2/3]

- Egy C függvényhívásnál tehát (i386 architektúrán) az argumentumokat jobbról balra haladva helyezzük el a veremben egyre csökkenő címeken, majd ezek után jönnek a technikai információk (visszatérési cím, verem információk, stb.), melyek fix méretűek.

```
printf("%d_%d_%d\n", 10, 20);
```



...
20
10
&"%d %d %d\n"
technikai információk

- A függvényhívás után az argumentumokat a veremből törölni kell. De ki takarít?

```
printf("%d□%d□%d\n", 10, 20);
```

- Ha a hívott `printf()` a visszatérés előtt takarítana, akkor a fenti esetben a formátumsztring alapján még egy további (nem ide tartozó) értéket kitörölné a veremből és az egész verem összezavarodna.
- Tehát az argumentumokat a hívó függvény törli a visszatérés után, hisz ő tudja biztosan, hogy ténylegesen mit adott át.
- C-ben az a fajta veremkezelés biztosítja a változó számú paraméterrel rendelkező függvények megvalósíthatóságát.

# printf

## Formátumsztring és argumentumok [1/4]

- A programozó felelőssége, hogy adott konverzió-specifikációhoz megfelelő típusú argumentumérték tartozzon (io-formatumsztring.c [12-13]):

```
12 printf("%20lld□%20lld\n", 2005LL, 2005.0);  
13 printf("%20lf□%20lf\n", 2005LL, 2005.0);
```

- Az utasítások eredménye i386 architektúrán:

```
2005 4656532898701115392  
0.000000 2005.000000
```

- Az utasítások eredménye x86-64 architektúrán:

```
2005 ??????????????????  
2005.000000 0.000000
```

- Másik példa (io-formatunsztring.c [15-18]):

```
15 printf("%x_%x_%x%x\n", 1, 2, 3, 4);
16 printf("%llx_%llx_%llx%llx\n", 1, 2, 3, 4);
17 printf("%x_%x_%x%x\n", 1LL, 2LL, 3LL, 4LL);
18 printf("%llx_%llx_%llx%llx\n", 1LL, 2LL, 3LL, 4LL);
```

- Az utasítások eredménye i386 architektúrán:

```
1 2 3 4
200000001 400000003 ffba196cffba1964 f7771280f75c4225
1 0 2 0
1 2 3 4
```

- Az utasítások eredménye x86-64 architektúrán:

```
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

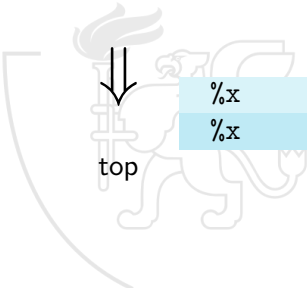
# printf

## Formátumsztring és argumentumok [3/4]

- i386 architektúrán (veremben átadott paraméterekkel)

```
printf("%x□%x\n", 1LL, 2LL);
```

	printf formátum	fizikai	printf hívás
bottom	...	...	...
	?	02 00 00 00	2LL
	?	00 00 00 00	
	%x	01 00 00 00	1LL
	%x	00 00 00 00	
top	...	...	...

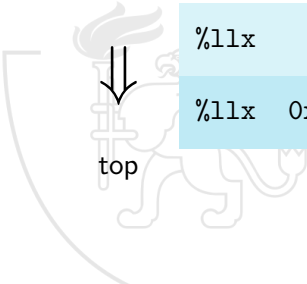


# printf

## Formátumsztring és argumentumok [4/4]

- i386 architektúrán (veremben átadott paraméterekkel)

```
printf("%11x□%11x\n", 1, 2);
```

	printf formátum	fizikai	printf hívás
bottom	...	...	...
	%11x	?? ?? ?? ?? ?? ?? ?? ??	? ?
	%11x	02 00 00 00 01 00 00 00	2 1
	...	...	...



- `int scanf(const char *form, ...)`

A standard bemenetről (`stdin`) a formátumsztring (`form`) alapján beolvassa az aktuális paraméterek értékeit és eltárolja azokat a paraméterlistában megadott memóriacímeken.

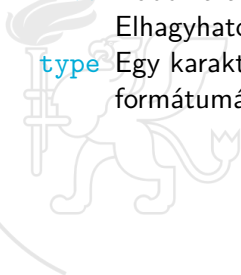
`form` Formátumsztring.

`...` További paraméterek

`returns` A beolvasott és sikeresen eltárolt értékek száma, hiba esetén EOF.

- A formátumsztring kétféle típusú objektumot tartalmaz:
  - Közöséges karaktereket, amelyeken az inputtal való egyezés esetén egyszerűen továbblép.
  - Konverzió-specifikációkat, amelyek mindegyike a soron következő paraméter konvertálását és tárolását írja elő.

- Egy konverzió-specifikáció (más néven formátumvezérlő szekvencia) általános alakja:
  - %[\*] [width] [lmod] type
    - \* Csillag karakter, beolvasás tárolás nélkül. Elhagyható.
    - width** Egy szám, a feldolgozandó karakterek maximális számát határozza meg. Elhagyható.
    - lmod** Több karakter, a megadott típus méretét módosítja. Elhagyható.
    - type** Egy karakter, a beolvasandó érték típusát és formátumát határozza meg. Kötelező.



# scanf

`%[*] [width] [lmod] type - [1/3]`

- Egy karakter az alábbiak közül, ami a beolvasandó érték típusát és formátumát, valamint a kapcsolódó paraméter típusát határozza meg:
  - d Előjeles decimális szám beolvasása és tárolása `int*` vagy `char*` típusú pointer által mutatott címen.
  - u Előjeltelen decimális szám beolvasása és tárolása `unsigned int*` vagy `unsigned char*` típusú pointer által mutatott címen.
  - o Egy oktális szám beolvasása és tárolása `int*` vagy `char*` típusú pointer által mutatott címen.
  - x Egy hexadecimális szám beolvasása és tárolása `int*` vagy `char*` típusú pointer által mutatott címen.

# scanf

`%[*] [width] [lmod] type - [2/3]`

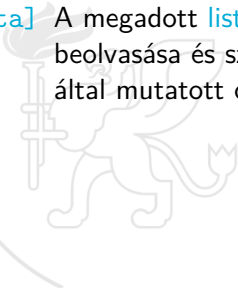
- i Egy egész szám beolvasása és tárolása `int*` vagy `char*` típusú pointer által mutatott címen. A beolvasandó szám leírása megfelel a C nyelvben használt egész számleíró formátumoknak (0 és `0x` előtagok oktális illetve hexadecimális számok jelölésére).
- f Valós szám beolvasása és tárolása `float*`, `double*` vagy `long double*` típusú pointer által mutatott címen.
- c Karakter(ek) beolvasása (`width` darab, vagy 1) és tárolása egy `char*` típusú pointer által mutatott címen.



**s** Egy whitespace karaktert nem tartalmazó karakterlác beolvasása és sztringként tárolása egy `char*` típusú pointer által mutatott címen.

**[lista]** A megadott **lista** karaktereket tartalmazó karakterlác beolvasása és sztringként tárolása egy `char*` típusú pointer által mutatott címen.

**[^lista]** A megadott **lista** karaktereket **nem** tartalmazó karakterlác beolvasása és sztringként tárolása egy `char*` típusú pointer által mutatott címen.



# scanf

`%[*][width][lmod]type`

- Beolvasás tárolás nélkül. A szöveg ezen helyén az előírt konverziónak megfelelő érték leírását kell beolvasni a standard inputról, viszont a beolvasott értéket nem kell eltárolni, így ehhez a konverzióhoz nem tartozik pointer sem.
  - \* A tárolás nélküli beolvasást a csillag karakter jelöli.



# scanf

`%[*] [width] [lmod] type`

- A maximális mezőszélességet határozza meg. A következő paraméter értékének beolvasásánál legfeljebb (illetve %c esetén pontosan) ennyi karakter lesz figyelembe véve. Ha az inputon ennél korábban szerepel az értékleírásban nem értelmezhető karakter, akkor természetesen már az sem lesz beolvasva. Hiányában addig olvas, míg a további karakterek logikailag az értékleírás részeként értelmezhetőek.

**szám** Egy decimális pozitív egész szám.

- Az alábbi beolvasás eredményeképpen például '123456' input esetén v értéke 123 lesz:

```
int v;  
scanf ("%3d", &v);
```

# scanf

`%[*][width][lmod]type`

- Karakterek, amik a következő paraméter típusát pontosítják:
  - hh** A megadott paraméter `int*` helyett `char*` típusú.
  - h** A megadott paraméter `int*` helyett `short int*` típusú.
  - l** A megadott paraméter `int*` vagy `float*` helyett `long int*` vagy `double*` típusú.
  - L** A megadott paraméter `int*` vagy `float*` helyett `long long int*` vagy `long double` típusú.





- Adott az alábbi programrészlet:

```
int v1, v2;  
scanf("%d_result:%d", &v1, &v2);
```

- A programrészlet eredménye különféle inputok esetén:

stdin beolvasás előtt	scanf vissz.	v1	v2	stdin beolvasás után
»3_result:147«	2	3	1	»47«
»3_results:147«	1	3	-	»s:147«
»result:147«	0	-	-	»result:147«



# scanf

## Tulajdonságok [1/2]

- A `scanf` elsőre talán sajátosnak tűnő logikával, de logikusan és konzekvensen működik.
- Ha egy karaktert nem tud értelmezni, akkor abbahagyja az olvasást. A program nem tudja, hogy mi volt a probléma (legfeljebb azt, hogy volt probléma). A maradékot ilyenkor benne hagyja az `stdin`-ben, és a következő beolvasás ennek a „maradéknak” a feldolgozásával kezdődik. De ha egy ciklusban ez ugyanaz a beolvasás mint az előbb

...

- Interaktív beolvasás esetén ezért érdemes lehet minden beolvasáskor kiüríteni az `stdin`-t:

```
scanf ("%d%* [^\n] ", &X);  
getchar ();
```

# scanf

## Tulajdonságok [2/2]

- Üres sztringet a %s segítségével nem tudunk beolvasni. Majdnem minden konverziós specifikációnál ugyanis az `stdin` elején lévő whitespace karaktereket a `scanf` átugorja.
- Ha az újsort (`'\n'`) betesszük a formátum sztringbe, akkor ezt feldolgozza a `scanf`, de ezzel kiürül az egysoros puffer, ezért egy újabb sort is beolvas az `stdin`-ről, mielőtt továbblépne. Ilyenkor működik úgy a program, mintha mindig „egy lépést késne” a beolvasással.



- Nem megfelelően használt I/O műveleteken keresztül megfelelően preparált adatokkal a program egyébként védett adatai is hozzáférhetőek lesznek, esetleg saját kód is futtatható.
  - `printf(str)` vagy `printf("%s", str)`  
`str="%x%x%x%x"`
  - `gets()` vagy `gets_s()`?  
Preparált inputtal a buffer overflow-t kihasználva a vezérlés tetszőleges függvénynek átadható.
- **C11** óta használható I/O függvények:
  - `int printf_s(const char *form, ...)`
    - Észreveszi a NULL sztring-pointereket, és nem engedi használni a `%n` konverziót (érték visszaírása a memóriába).
  - `int scanf_s(const char *form, ...)`
    - Észreveszi a NULL pointereket, és a `%c`, `%s`, `%[]` konverzióknál (sztringek beolvasása) plusz paraméterként várja a sztring méretét, amit figyelembe is vesz (érzékeli a puffer túlcsoordulását).

# Alternatív I/O

## sprintf, sprintf\_s

- `int sprintf(char *mp, const char *form, ...)`
- `int sprintf_s(char *mp, rsize_t n, const char *form, ...)`

A `printf` illetve `printf_s` függvények memóriába (sztringbe) író változatai.

`mp` A „kiírandó” adat helyére mutató pointer.

`n` A cél sztring mérete.

`form` Formátumsztring.

`...` További paraméterek

`returns` A kiírt karakterek száma, hiba esetén negatív érték.

- Az `sprintf_s` változat **C11** óta létezik. Ez `n` értéke alapján észreveszi, ha túlcsordulna a puffer.

# Alternatív I/O

snprintf, snprintf\_s

- `int snprintf(char *mp, int n, const char *form, ...)`
- `int snprintf_s(char *mp, rsize_t n, const char *form, ...)`

A `printf` illetve `printf_s` függvények memóriába (sztringbe) író változatai. Csak az első `n` karaktert írják ki ténylegesen.

`mp` A „kiírandó” adat helyére mutató pointer.

`n` A „kiírandó” sztring maximális hossza.

`form` Formátumsztring.

`...` További paraméterek

`returns` Az `n` figyelmen kívül hagyásával potenciálisan kiírandó karakterek száma, hiba esetén negatív érték.

- Az `snprintf_s` változat `C11` óta létezik.

# Alternatív I/O

sscanf, sscanf\_s

- `int sscanf(const char *mp, const char *form, ...)`
- `int sscanf_s(const char *mp, const char *form, ...)`

A `scanf` függvény memóriából (sztringből) olvasó változata.

`mp` A feldolgozandó adat memóriaterületére mutató pointer.

`form` Formátumsztring.

`...` További paraméterek

`returns` A sikeresen beolvasott paraméterek száma, hiba esetén EOF.

- Az `sscanf_s` változat **C11** óta létezik.



- 1 Bemutakozás**
  - Kurzus információk
  - A SZTE és az informatikai képzés
- 2 Linux**
  - Alapfogalmak
  - Linux parancsok
  - Linux shell
  - Felhasználók
  - Hálózat
- 3 Gyors C áttekintés**
  - Bevezető
  - Pénzváltás (1. verzió)
  - Pénzváltás (2. verzió)
  - Röppálya számítás
  - Röppálya szimuláció
  - Az év napja
  - Csúszoátlag adott elemszámra
  - Csúszoátlag parancssorból
  - Basename standard inputról
  - Basename parancssorból
  - Tér legtávolabbi pontjai
  - A nappalis gyakorlat értékelése

- 4 Alapok**
  - Alapfogalmak
  - A programozás fázisai
  - Algoritmus vezérlése
  - A C nyelvű program
  - Szintaxis
  - A C nyelv elemi adattípusai
  - A C nyelv utasításai

- 5 Vezérlési szerkezetek**
  - Bevezetés
  - Szekvenciális vezérlés
  - Függvények
  - Szelekciós vezérlések
  - Ismétléses vezérlések 1.
  - Eljárásvezérlés
  - Ismétléses vezérlések 2.

- 6 Folyamatábra és struktúradiagram**
- 7 Adatszerkezetek**
  - Az adatkezelés szintjei
  - Elemi adattípusok
  - Pointer adattípus
  - Tömb adattípus

- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

- 8 IO**
  - Alapok
  - **Adatállományok**

- 9 C fordítás**
  - A fordítás folyamata
  - A preprocessor
  - A C fordító
  - Assembler
  - Linker és modulok

- 10 Gyakorlati kérdések**
  - Memóriahasználat
  - Gyakori C hibák
  - `where.c` felboncolva



# Hozzáférés az adatállományokhoz

## Standard csatornák

- A már említett 3 fájl, az `stdin`, `stdout` és `stderr` úgy képzelhető el, mint egy folyamat:
  - Mi csak nyitni és zárni tudjuk a „zsilipet”, látjuk mi folyik át, de visszahozni azt már nem tudjuk.
  - Vagy másképpen: mi csak egy irányban (előre) lépegethetünk a fájlban lévő elemek sorozatán, mindig csak a következő elemre.
- Általános esetben viszont egy fájl tartalma megmarad, tehát akárhányszor újra ránézhetünk tetszőleges részére.



# Hozzáférés az adatállományokhoz

## Fájlok

- Minden fájl felfogható elemek (például karakterek) sorozataként.
- Ehhez a sorozathoz tartozik egy író-olvasó fej, ami a fájl értékét jelentő sorozatot két részsorozatra bontja, tehát egy fájl minden lehetséges értéke megadható  $[a_0, \dots, a_{i-1}][a_i, \dots, a_{n-1}]$  alakban.
  - Az író-olvasó fej által kijelölt elem a második sorozat első eleme.
- Bármilyen művelet a fájlban csak a fejen keresztül történhet.
- Lehetséges elemi műveletek:

**Olvasás:** Kiolvassuk a fej által kijelölt elem értékét és a fej egyet továbblép.

**Írás:** Megváltoztatjuk a fej által kijelölt elem értékét, vagy ha az a fájl vége volt, akkor hozzáfűzzük az új elemet; a fej egyet továbblép.

**Pozícionálás:** A fejet tudjuk mozgatni.

- Ha egy fájlt szeretnénk használni, akkor a C programban hozzárendelünk egy `stdio.h`-beli FILE struktúrát.
  - A FILE struktúra sokféle mezőt tartalmaz (fej pozíció, puffer, stb.), ezekkel azonban nem kell (és nem is érdemes) direkt módon dolgoznunk.
  - E helyett függvényeket használunk, amikben a fájlhoz rendelt struktúra címe fogja a fájlt azonosítani.
  - Ezért a fájlokat praktikusán FILE\* típusú változókkal tartjuk számon.
- A fájl használata során a következő műveleteket fogjuk elvégezni:
  - Először a FILE típusú változóhoz hozzárendelünk egy operációs rendszerbeli adatállományt vagy eszközt a FILE `*fopen(const char *path, const char *mode)` függvény segítségével, ami a megfelelő FILE struktúra címét adja vissza.
  - Ezután a kapott pointer segítségével használjuk a fájlt.
  - Végül az `int fclose(FILE *fp)` függvény segítségével megszüntetjük a hozzárendelést.

- `FILE *fopen(const char *path, const char *mode)`
- `int fgetc(FILE *fp)`
- `int getc(FILE *fp)`
- `int fputc(int ch, FILE *fp)`
- `int putc(int ch, FILE *fp)`
- `char *fgets(char *buf, int n, FILE *fp)`
- `int fputs(const char *buf, FILE *fp)`
- `int fprintf(FILE *fp, const char *form, ...)`
- `int fprintf_s(FILE *fp, const char *form, ...)`
- `int fscanf(FILE *fp, const char *form, ...)`
- `int fscanf_s(FILE *fp, const char *form, ...)`

- `int ungetc(int c, FILE *fp)`
- `int fseek(FILE *fp, long int offset, int where)`
- `long int ftell(FILE *fp)`
- `int feof(FILE *fp)`
- `int ferror(FILE *fp)`
- `size_t fread(char *bf, size_t rs, size_t rn, FILE *fp)`
- `size_t fwrite(char *bf, size_t rs, size_t rn, FILE *fp)`
- `int fflush(FILE *fp)`
- `int fclose(FILE *fp)`

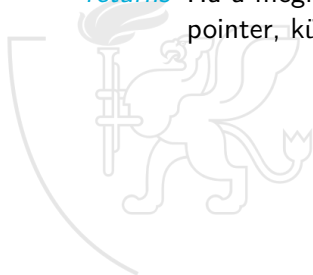
- FILE \*fopen(const char \*path, const char \*mode)

Fájl megnyitása.

**path** A fájlrendszeren létező fájlra való (az operációs rendszerben) szabályos hivatkozás.

**mode** A fájl megnyitási módját leíró sztring. Lehetséges karakterek: r, w, a, +, x (`C11`-től), b, t

**returns** Ha a megnyitás sikerült, akkor egy FILE-ra mutató pointer, különben a NULL pointer.



- A mode sztring a következő lehet:

"r" Egy már létező fájl megnyitása olvasásra. Ha nem létezik vagy nincs rá olvasási jogunk, az hiba.

"w" Egy fájl megnyitása írásra. Ha már létezett, a régi tartalma törlődik, egyébként létre lesz hozva. Ha nincs rá írásjogunk, vagy nem tudjuk létrehozni, az hiba.

"wx" C11 Mint "w", de már létező fájl esetén hibát kapunk.

"a" Egy fájl megnyitása hozzáfűzésre. Ha már létezett, a régi tartalma megmarad, az új a végéhez fűződik. Ha nem létezett a fájl, akkor egyenértékű a "w"-vel. Ha nincs rá írásjogunk, vagy nem tudjuk létrehozni, az hiba.

"r+" Egy már létező fájl megnyitása olvasásra és írásra.

"w+" Egy fájl megnyitása írásra és olvasásra.

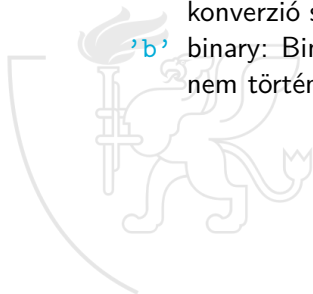
"w+x" C11 Mint "w+", de már létező fájl esetén hibát kapunk.

"a+" Egy fájl megnyitása hozzáfűzésre és olvasásra.

- Különböző operációs rendszereken a sorvége jelek eltérhetnek a C értelmezéstől (`'\n'`), ezért a `mode` sztringben jelezni kell azt is, hogy a fájl szövegfájl-e. Szövegfájlok esetén ugyanis beolvasáskor/kiíráskor megtörténik a konverzió, bináris fájlok esetén viszont nem.

`'t'` text: Szöveges fájl megnyitása. Linux alatt nincs jelentősége, de pl. Windows alatt `'\r\n'` ↔ `'\n'` konverzió szükséges. (Alapértelmezés.)

`'b'` binary: Bináris fájl megnyitása. Beolvasás és kiírás során nem történik konverzió.





# Fájlműveletek

## fopen – használat

- Az író-olvasó fej r, r+, w, wx, w+, w+x módok esetén a fájl elejére, a, a+ módok esetén a fájl végére áll.
- Ha a fájlt valamilyen író-olvasó módban (+) nyitottuk meg és váltani akarunk írás és olvasás között, akkor célszerű a puffert az fseek() vagy fflush() függvények valamelyikével a kiürítettetni.
- Az fopen visszatérési értékét illik ellenőrizni:

```
FILE *fp;                                /* fájlpointer */
if ((fp = fopen("/etc/motd", "r")) == NULL) {
    printf("Nem_sikerült_megnyitni_a_fájlt\n");
    exit(1);
}
```

vagy „biztonságosabban” (kevesebb zárójellel):

```
fp = fopen("/etc/motd", "r");
if (fp == NULL) ...
```

# Fájlműveletek

fgetc, getc, fputc, putc

- `int fgetc(FILE *fp)`

- `int getc(FILE *fp)`

Egy karakter beolvasása fájlból.

`fp` A fájlleíró adatra mutató pointer.

*returns* Ha a művelet sikeres volt, akkor a beolvasott karakter kódja, és veszteség nélkül konvertálható `char` típusúvá.  
Ha hiba történt, akkor EOF.

- `int fputc(int ch, FILE *fp)`

- `int putc(int ch, FILE *fp)`

Egy karakter kiírása fájlba.

`ch` A kiírandó karakter.

`fp` A fájlleíró adatra mutató pointer.

*returns* Ha a művelet sikeres volt, akkor a kiírt karakter kódja,  
hiba esetén EOF.

- `char *fgets(char *buf, int n, FILE *fp)`  
Egy sor (de legfeljebb  $n-1$  karakter) beolvasása fájlból sorvége jellel együtt.
  - `buf` A puffer, amiben sztringként eltárolja a beolvasott sort.
  - `n` Pufferméret. Legfeljebb annyi karaktert olvas be, hogy sztringként még el tudja tárolni  $n$  karakteren.
  - `fp` A fájlleíró adataira mutató pointer.
  - returns* Az eltárolt sor címe (`buf`), ha sikerült a művelet, különben `NULL`.
- `int fputs(const char *buf, FILE *fp)`  
Egy sztring kiírása fájlba (nem tesz a végére plusz sorvége jelet).
  - `buf` A sztring, amit ki kell írni.
  - `fp` A fájlleíró adataira mutató pointer.
  - returns* Az utolsó kiírt karakter kódja (üres sztring esetén 0) ha sikerült a művelet, különben `EOF`.

# Fájlműveletek

fprintf, fprintf\_s

- `int fprintf(FILE *fp, const char *form, ...)`
  - `int fprintf_s(FILE *fp, const char *form, ...)`
- A `printf` illetve `printf_s` függvények fájlba író változatai.

`fp` A fájlleíró adatra mutató pointer.

`form` Formátumsztring.

`...` További paraméterek

`returns` A kiírt karakterek száma, hiba esetén negatív érték.

- Az `fprintf_s` változat C11 óta létezik.



- `int fscanf(FILE *fp, const char *form, ...)`
- `int fscanf_s(FILE *fp, const char *form, ...)`

A `scanf` illetve `scanf_s` függvények fájlól olvasó változatai.

`fp` A fájlleíró adatra mutató pointer.

`form` Formátumsztring.

`...` További paraméterek

`returns` A sikeresen beolvasott paraméterek száma, hiba esetén EOF.

- Az `fscanf_s` változat `C11` óta létezik.



- `int ungetc(int c, FILE *fp)`  
Visszatesszi a `c` karaktert a fájlba úgy, hogy a legközelebbi olvasás ezt a karaktert fogja olvasni.
  - `c` A karakter.
  - `fp` A fájlleíró adatra mutató pointer.
  - *returns* A visszarakott `c` karakter, vagy hiba esetén EOF.
- Hasznos lehet ha például számokat olvasunk be: amikor nem számjegy karakter jön azt visszatesszük, az előtte beolvasott számjegyeket pedig egy számmá konvertáljuk.
- Az `ungetc` segítségével legfeljebb egy karaktert lehet visszatenni, a többszöri egymás utáni meghívása nem megengedett.

- `int fseek(FILE *fp, long int offset, int where)`

Az író-olvasó fej pozicionálása a fájlban.

`fp` A fájlleíró adatra mutató pointer.

`offset` Ennyivel fogja léptetni az író-olvasó fejet (előjelesen).

`where` A fej léptetésének indítópontja:

`SEEK_SET` Léptetés a fájl elejéről, tulajdonképpen az új pozíció direkt megadását jelenti.

`SEEK_CUR` Léptetés a jelenlegi pozíciótól, a fej mozgatása a megadott értékkel.

`SEEK_END` Léptetés a fájl végétől kezdve.

*returns* Ha a művelet sikeres volt, akkor 0, különben -1.

- A fej pozíciója negatív nem lehet, de mutathat a fájlvégén túlra is, ekkor a rendszer addig a pozícióig megnyújtja (szeméttel kiegészíti) a fájlt.

- `long int ftell(FILE *fp)`  
Lekérdezi az író-olvasó fej pozícióját.  
`fp` A fájlleíró adatra mutató pointer.  
`returns` A fej aktuális pozíciója, hiba esetén -1.
- `int feof(FILE *fp)`  
Elértük-e a fájl végét?  
`fp` A fájlleíró adatra mutató pointer.  
`returns` 0 ha nem értük el, egyébként nem 0.
- `int ferror(FILE *fp)`  
Történt-e hiba a fájl megnyitása óta?  
`fp` A fájlleíró adatra mutató pointer.  
`returns` 0 ha nem történt, nem 0 hibakód, ha volt hiba.



# Fájlműveletek

fread, fwrite

- `size_t fread(void *bf, size_t rs, size_t rn, FILE *fp)`  
Bináris adatbeolvasás adott méretű egységekben.
  - `bf` A beolvasott adat helye a memóriában.
  - `rs` Az adat egy egységének mérete (rekordméret).
  - `rn` A beolvasandó egységek száma.
  - `fp` A fájlleíró adatra mutató pointer.

*returns* A sikeresen beolvasott csomagok száma.
- `size_t fwrite(void *bf, size_t rs, size_t rn, FILE *fp)`  
Bináris adatkirás adott méretű egységekben.
  - `bf` A kiírandó adat helye a memóriában.
  - `rs` Az adat egy egységének mérete (rekordméret).
  - `rn` A kiírandó egységek száma.
  - `fp` A fájlleíró adatra mutató pointer.

*returns* A sikeresen kiírt csomagok száma.

# Fájlműveletek

`fflush`, `fclose`

- `int fflush(FILE *fp)`

Kiírja a fájl puffer tartalmát.

`fp` A fájlleíró adatra mutató pointer.

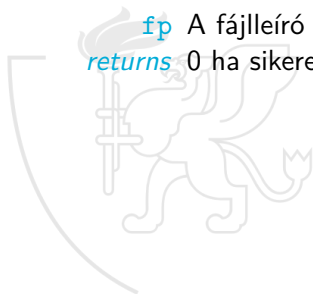
*returns* 0 ha sikeres, EOF ha sikertelen a művelet.

- `int fclose(FILE *fp)`

Lezárja a fájlt, megszakítja a hozzárendelést.

`fp` A fájlleíró adatra mutató pointer.

*returns* 0 ha sikeres, EOF ha sikertelen a művelet.



# Fájlműveletek és standard csatornák

- Természetesen az előre definiált 3 streamet is kezelhetjük ezekkel a függvényekkel, pl:

```
fprintf(stderr, "Nem működik a gépem\n");  
fputs(stdout, "Sztring\n");  
fscanf(stdin, "%s", nev);  
fgets(stdin, 50, sor);
```



# Fájlkezelés text és binary módban [1/1]

data.c [1–27]

```
1 /* Fájlkezelés többféle módon.
2  * Dévényi Károly, devenyi@inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6
7 int main() {
8     FILE *fp, *fpb;
9     int a, b;
10
11     fp = fopen("probat.txt", "wt");
12     fprintf(fp, "%d", 34);
13     fflush(fp); fclose(fp);
14
15     fp = fopen("probat.txt", "rt");
16     fscanf(fp, "%d", &a);
17     printf("%d\n", a);
18     fclose(fp);
19
20     fpb = fopen("probab.dat", "w+b");
21     fwrite(&a, sizeof(int), 1, fpb);
22     fseek(fpb, 0, SEEK_SET);
23     fread(&b, sizeof(int), 1, fpb);
24     printf("%d\n", b);
25     fclose(fpb);
26     return 0;
27 }
```