

# Programozás 2

1.

# **DINAMIKUS TÖMBÖK**

# Dinamikus tömb típus

- Ha a programban deklarálunk egy tömböt, azzal az lehet a baj, hogy a méretét fordítási időben meg kell adni.
- Ez viszont nem mindig ismert, így előfordulhat, hogy a tömb számára kevés helyet foglaltunk, de az is, hogy feleslegesen sokat.
- A megoldás: tömb helyett pointert deklarálunk, és ha tudjuk a kívánt méretet, memóriát már a megfelelő számú elemnek foglaltunk.

# Dinamikus tömb típus

- ▶ Mivel a pointert tömbként kezelhetjük, a programban kódjában ez semmilyen más változást nem eredményez:

```
int tomb[MAX];  
...  
  
for(i=0; i<n; ++i) {  
    tomb[i]=i;  
}  
...  
  
...
```

```
int *tomb;  
...  
  
tomb=malloc(n*sizeof(int));  
for(i=0; i<n; ++i) {  
    tomb[i]=i;  
}  
...  
  
free(tomb);  
...
```

# Flexibilis tömb típus

- A C nyelvben lehetőség van arra, hogy egy pointer számára már lefoglalt memóriaterület méretét megváltoztassuk. A

```
void *realloc(void *ptr, size_t size)
```

függvény a **ptr** által mutatott területet méretezi (és ha kell mozgatja) át.

- Ez sem használható viszont, ha a memória túlságosan „széttöredezett”, azaz nincsennek benne nagy egybefüggő részek.

# Flexibilis tömb típus

- Egy megoldás a problémára a flexibilis tömb adattípus, ami a dinamikus tömb általánosítása.
- Ennek van olyan művelete amivel az indextípus felső határát módosíthatjuk, ezáltal változó elemszámú sorozatokat kezelhetünk, továbbá a megvalósítása kis méretű tömbökkel dolgozik.
- Adott **E** elemtípus esetén a flexibilis tömb (FTömb) adattípus értékhalmaza az összes

**A : 0..N ----> E**

függvény.

# Flexibilis tömb műveletei

- Kiolvas(  $\rightarrow$  A:FTömb;  $\rightarrow$  i:int;  $\leftarrow$  X:E)
  - Az A függvény értékének kiolvasása
- Módosít(  $\leftrightarrow$  A:FTömb;  $\rightarrow$  i:int;  $\rightarrow$  X:E)
  - Az A függvény értékének módosítása
- $X=Y$  értékadás, ha X és Y FTömb típusú változók.
- Létesít(  $\leftrightarrow$  A:FTömb;  $\rightarrow$  N:int)
  - N elemű flexibilis tömböt létesít.
- Megszüntet(  $\leftrightarrow$  A:FTömb)
  - Törli az A flexibilis tömbhöz foglalt memóriát.

# Flexibilis tömb műveletei

- Felső(  $\rightarrow$  A:FTömb): int
  - A felső határ lekérdezése
- Növel(  $\rightarrow$  A : FTömb; d:int)
  - Az aktuális indextípus felső határát a d értékkel növeli.
- Csökkent(  $\rightarrow$  A : FTömb; d:int)
  - Az aktuális indextípus felső határát a d értékkel csökkenti.

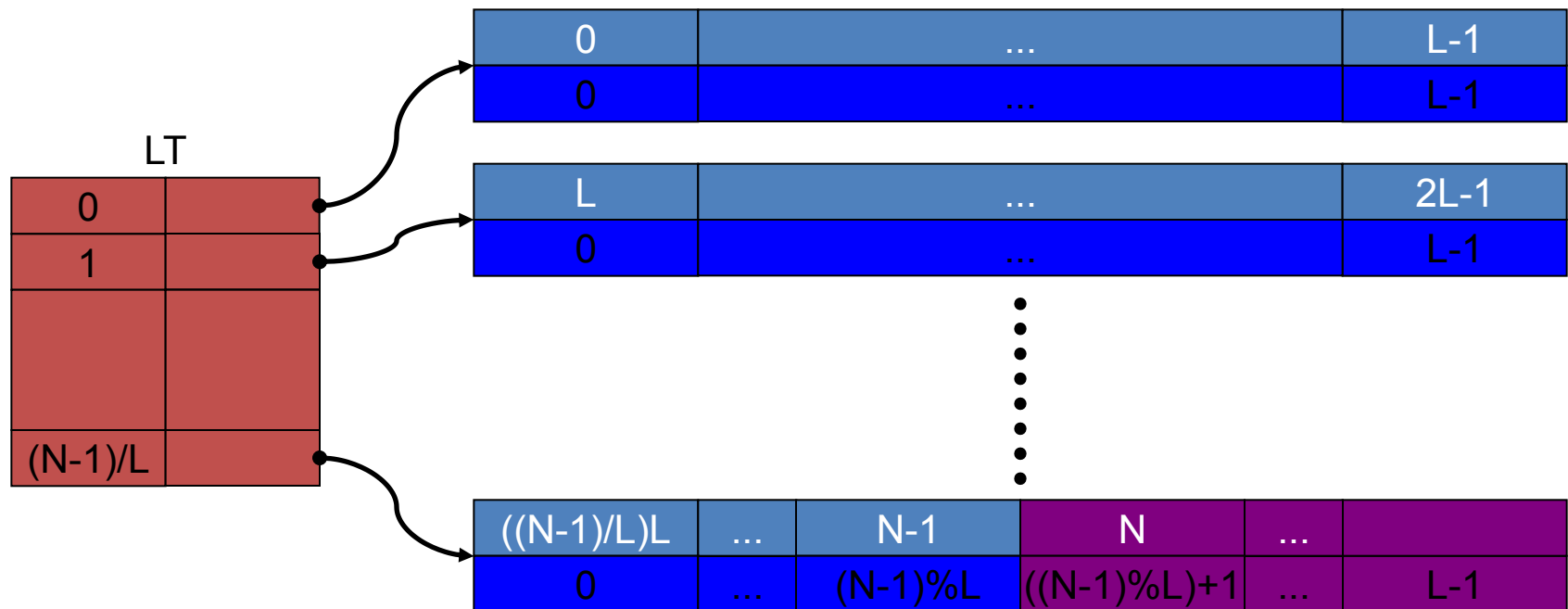


# Flexibilis tömb megvalósítása

- A megvalósításhoz válasszunk egy  $L$  konstanst.
- $L$  elemszámú dinamikussá tett tömbökből, amiket lapoknak nevezünk, állítsuk össze a nagy tömböt úgy, hogy felvesszünk egy  $LT$  laptérkép tömböt, amelynek elemei lapokra mutató pointerek.

# Flexibilis tömb megvalósítása

- Ezt szemlélteti az ábra.



# Flexibilis tömb megvalósítása

```
/* Globális elemek a flexibilis tömb megvalósításához */

#define L ???                                /* lapméret */

typedef enum {false, true} bool;             /* logikai típus */

typedef ???      elemtip;                   /* a tömb elemtípusa */
typedef elemtip *laptop;
typedef laptop   *lapterkeptip;

typedef struct ftomb {
    lapterkeptip lt;                        /* laptérkép */
    unsigned int hatar;                    /* aktuális indexhatár */
} ftomb;
```



# Flexibilis tömb megvalósítása

```
/* A műveletek megvalósítása: */  
  
void kiolvas(ftomb a, unsigned int i, elemtip *x)  
{  
    if(i < a.hatar) {  
        *x = a.lt[i / L][i % L];  
    }  
}  
  
void modosit(ftomb a, unsigned int i, elemtip x)  
{  
    if(i < a.hatar) {  
        a.lt[i / L][i % L] = x;  
    }  
}
```



# Flexibilis tömb megvalósítása

```
void letesit(ftomb *a, unsigned int n)
{
    int j;
    if(n) {
        a->hatar = n;
        a->lt=(elemtip**)malloc(
            (1+((n-1)/L))*sizeof(lapterkeptip));
        for(j=0; j<=((n-1) / L); ++j) {
            a->lt[j]=(elemtip*)malloc(L*sizeof(elemtip));
            /* lapok létesítése */
        }
    } else {
        a->hatar = 0;
        a->lt = NULL;
    }
}
```



# Flexibilis tömb megvalósítása

```
void megszuntet(ftomb *a)
{
    int j;
    if(a->hatar) {
        for(j=0; j<=((a->hatar-1) / L); ++j) {
            free(a->lt[j]);          /* lapok törlése */
        }
        free(a->lt);
        a->hatar=0;
    }
}

unsigned int felso(ftomb a)
{
    return a.hatar;
}
```



# Flexibilis tömb megvalósítása

```
void novel(ftomb *a, unsigned int d)
{
    int j;
    a->lt = (elemtip**)realloc(a->lt,
                               (1+((a->hatar+d-1)/L))*sizeof(lapterkeptip));
    for(j=(a->hatar ? ((a->hatar-1)/L)+1 : 0) ;
        j<=(a->hatar+d-1)/L; ++j) {
        a->lt[j]=(elemtip*)malloc(L*sizeof(elemtip));
    }
    a->hatar += d;
}
```



# Flexibilis tömb megvalósítása

```
void csokkent(ftomb *a, unsigned int d)
{
    int j;
    if(d <= a->hatar) {
        for(j=(a->hatar-d-1)/L +1; j<=(a->hatar-1)/L; ++j) {
            free(a->lt[j]);          /* lapok törlése */
        }
        a->hatar -= d;
        a->lt = (elemtip**)realloc(a->lt,
                                   (1+((a->hatar-1)/L))*sizeof(lapterkeptip));
    }
}
```



# Rendezés több szempont szerint

- Problémafelvetés
  - A beolvasott adatokat rendezzük több szempont szerint is egy egyszerű rendezési algoritmussal és minden rendezés után legyen kiíratás is.
- Specifikáció
  - Flexibilis tömbbel dolgozzunk
  - Input
    - A tömb elemei.
  - Output
    - A különböző szempontok szerint rendezett tömb.

# Rendezés több szempont szerint

- Algoritmustervezés:
  - A fő algoritmusban csak az elemeket kell beolvasni egy végjelig, majd rendre aktivizálni kell a különböző szempontok szerinti rendezést, végül az eredményt kiíratni.
  - A rendezés a beszűrőrendezés lesz.

# Beszűrő rendezés

- Problémafelvetés
  - Rendezzük egy tömb elemeit
- Specifikáció
  - Input
    - Egy tömb melynek elemtípusán értelmezett egy rendezési reláció.
  - Output
    - A reláció alapján rendezett tömb.

# Beszűrő rendezés

- Algoritmustervezés:
  - A tömböt logikailag egy már rendezett és egy még rendezetlen részre osztjuk, és a rendezetlen rész első elemét beszűrjük a rendezett elemek közé úgy, hogy azok rendezettek maradjanak.

3	1	5	4	2
---	---	---	---	---

# Rendezés több szempont szerint

```
/* Rendezzük névsorba illetve átlag szerint a hallgatókat!  
 * Flexibilis tömbbel történik a megvalósítás, tehát a  
 * névsor hosszát nem kell előre megmondani.  
 * Készítette: Dévényi Károly, devenyi@inf.u-szeged.hu  
 *           1998. Február 16.  
 * Módosította: Gergely Tamás, gertom@inf.u-szeged.hu  
 *           2006. Augusztus 15.  
 */
```

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

```
#define L 10
```

```
/* lapméret */
```



# Rendezés több szempont szerint

```
typedef enum {false, true} bool;           /* logikai típus */

typedef struct elemtip {                   /* a tömb elemtípusa */
    char nev[21];
    float adat;
} elemtip;

typedef elemtip *laptop;

typedef laptop *lapterkeptip;

typedef struct ftomb {
    lapterkeptip lt;                       /* laptérkép */
    unsigned int hatar;                   /* aktuális indexhatár */
} ftomb;

typedef bool (*RendRelTip)(elemtip, elemtip);
```



# Rendezés több szempont szerint

```
/* A műveletek megvalósítása: */  
  
void kiolvas(ftomb a, unsigned int i, elemtip *x)  
{  
    if(i < a.hatar) {  
        *x = a.lt[i / L][i % L];  
    }  
}  
  
void modosit(ftomb a, unsigned int i, elemtip x)  
{  
    if(i < a.hatar) {  
        a.lt[i / L][i % L] = x;  
    }  
}
```



# Rendezés több szempont szerint

```
void letesit(ftomb *a, unsigned int n)
{
    int j;
    if(n) {
        a->hatar = n;
        a->lt=(elemtip**)malloc(
            (1+((n-1)/L))*sizeof(lapterkeptip));
        for(j=0; j<=((n-1) / L); ++j) {
            a->lt[j]=(elemtip*)malloc(L*sizeof(elemtip));
            /* lapok létesítése */
        }
    } else {
        a->hatar = 0;
        a->lt = NULL;
    }
}
```





# Rendezés több szempont szerint

```
void megszuntet(ftomb *a)
{
    int j;
    if(a->hatar) {
        for(j=0; j<=((a->hatar-1) / L); ++j) {
            free(a->lt[j]);          /* lapok törlése */
        }
        free(a->lt);
        a->hatar=0;
    }
}

unsigned int felso(ftomb a)
{
    return a.hatar;
}
```



# Rendezés több szempont szerint

```
void novel(ftomb *a, unsigned int d)
{
    int j;
    a->lt = (elemtip**)realloc(a->lt,
                               (1+((a->hatar+d-1)/L))*sizeof(lapterkeptip));
    for(j=(a->hatar ? ((a->hatar-1)/L)+1 : 0) ;
        j<=(a->hatar+d-1)/L; ++j) {
        a->lt[j]=(elemtip*)malloc(L*sizeof(elemtip));
    }
    a->hatar += d;
}
```



# Rendezés több szempont szerint

```
void csokkent(ftomb *a, unsigned int d)
{
    int j;
    if(d <= a->hatar) {
        for(j=(a->hatar-d-1)/L +1; j<=(a->hatar-1)/L; ++j) {
            free(a->lt[j]);          /* lapok törlése */
        }
        a->hatar -= d;
        a->lt = (elemtip**)realloc(a->lt,
                                   (1+((a->hatar-1)/L))*sizeof(lapterkeptip));
    }
}
```



# Rendezés több szempont szerint

```
void beszuroRend(ftomb t, RendRelTip kisebb)
{ /* A Kisebb rendezési reláció szerinti helyben rendezés */
  int i,j;
  elemtip e,f;
  for(i = 1; i < felso(t); ++i) {
    kiolvas(t, i, &e);
    j = i-1;
    while(true) {
      if(j<0)
        break;
      kiolvas(t, j, &f);
      if(kisebb(f, e))
        break;
      modosit(t, ((j--)+1), f);
    }
    modosit(t, j+1, e);
  }
} /* BeszuroRend */
```



# Rendezés több szempont szerint

```
bool NevSzerint(elemtip X, elemtip Y)
{
    /* a névsor szerinti rendezési reláció */
    return strcmp(X.nev, Y.nev) <= 0;
}

bool AdatSzerint(elemtip X, elemtip Y)
{
    /* az adat szerinti rendezési reláció */
    return X.adat <= Y.adat;
}

bool CsokAdatSzerint(elemtip X, elemtip Y)
{
    /* az adat szerint csökkenő rendezési reláció */
    return X.adat >= Y.adat;
}
```



# Rendezés több szempont szerint

```
int main()
{
    ftomb sor;
    elemtip hallg;                /* beolvasáshoz */
    int i;
    letesit(&sor, 0);             /* a flexibilis tömb létesítése */
                                   /* beolvasás */
    printf("Kérem az adatsort, külön sorban név és adat!\n");
    printf("A végét a * jelzi.\n");
    scanf("%20[^\\n]*[^\\n]", hallg.nev); getchar();
    i = 0;                        /* az i. helyre fogunk beírni */
    while(strcmp(hallg.nev, "*")) {
        novel(&sor, 1);           /* a flexibilis tömb bővítése */
        scanf("%f*[^\\n]", &hallg.adat); getchar();
        modosit(sor, i++, hallg);
        scanf("%20[^\\n]*[^\\n]", hallg.nev); getchar();
    }
}
```



# Rendezés több szempont szerint

```
beszuroRend(sor, NevSzerint); /* Rend. névsor szerint */
printf("Névsor szerint rendezve:\n");
for(i = 0; i < felso(sor); ++i) {          /* Kiíratás */
    kiolvas(sor, i, &hallg);
    printf("%6.2f %s\n", hallg.adat, hallg.nev);
}
beszuroRend(sor, AdatSzerint); /* Rend. adat szerint */
printf("Adat szerint rendezve:\n");
for(i = 0; i < felso(sor); ++i) {          /* Kiíratás */
    kiolvas(sor, i, &hallg);
    printf("%6.2f %s\n", hallg.adat, hallg.nev);
}
beszuroRend(sor, CsokAdatSzerint); /* Rendezés újra */
printf("Adat szerint csökkenő sorba rendezve:\n");
for(i = 0; i < felso(sor); ++i) {          /* Kiíratás */
    kiolvas(sor, i, &hallg);
    printf("%6.2f %s\n", hallg.adat, hallg.nev);
}
megszuntet(&sor);          /* a flexibilis tömb törlése */
}
```

**LÁNCOK**

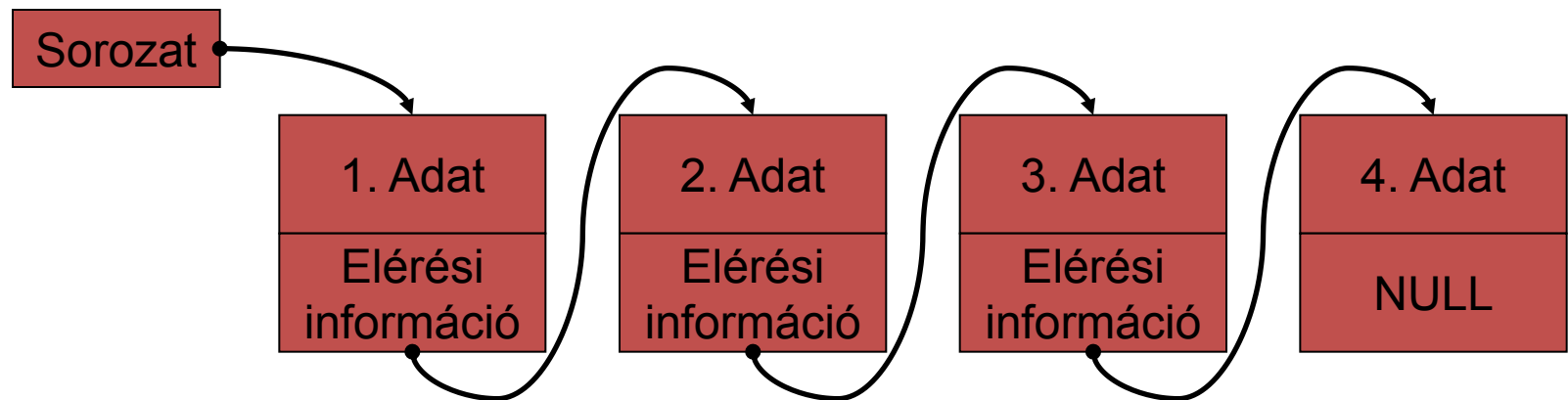


# Láncok

- Tekintsük azonos típusú adatelemek sorozatát.
- Ha a sorozat bármely pozíciójára vonatkozó bővítés és törlés műveletet is akarunk végezni, akkor a tömbös reprezentálás nem lehet hatékony, ehelyett a sorozatnak láncolással történő reprezentálása ajánlott.
- Láncoláson olyan adatszerkezetet értünk, amely a tárolandó sorozat minden elemét egy olyan rekordban (cellában) tárolja, amely a sorozat következő elemének elérését biztosító információt is tartalmazza.

# Láncok

- Az elérési információ lehet egy pointer érték, amely a sorozat következő elemét tartalmazó cellára (dinamikus változóra) mutató pointer.
- A sorozatot az első elemére mutató pointerrel adhatjuk meg.



# Láncok

- C-ben így hozhatunk létre láncolt lista típust:

```
typedef ??? elemtip;      /* a sorozat elemtípusa */

typedef struct cellatip {
    elemtip adat;          /* adatalem */
    struct cellatip *csat; /* a következő elem cellájára */
} cellatip;

typedef struct cellatip *pozicio; /* pointer */
```

# Láncok

- Deklaráljuk a **p** pointert! Lehet a következő módokon:

```
cellatip *p;
```

```
struct cellatip *p;
```

```
pozicio p;
```

- A **p** pointer által megmutatott cella egyes mezőire így hivatkozhatunk:

```
– p->adat
```

```
– p->csat
```

# Láncok

- A `->` és `.` struktúraoperátorok a precedencia-hierarchia csúcsán állnak, és ezért nagyon szorosan kötnek.
- A  
`++p->adat`  
nem `p`-t, hanem az `adat` mezőt  
inkrementálja, mivel az alapértelmezés szerinti  
zárójelezés:  
`++ (p->adat)`

# Láncok

- Zárójelek használatával a kötés megváltoztatható:  
    **(++p) ->adat**  
    az **adat**-hoz való hozzáférés előtt növeli **p**-t, míg  
    **(p++) ->adat**  
    azt követően inkrementál.
- Láncok esetén vigyázzunk ezekkel a műveletekkel, mert egyáltalán nem biztos, hogy két **cellatip** típusú láncelem közvetlenül egymás után helyezkedik el a memóriában!

# Láncok

- A lánc soron következő elemét a  
    `p->csat`  
pointeren keresztül a  
    `*p->csat`  
hozza be.

# Láncok

`*p->csat++`

azután inkrementálja **csat**-t, miután hozzáfért ahhoz, amire mutat (ekkor a lánc megszakadhat!)

`(*p->csat) ++`

azt növelné, amire csat mutat, (de ezt most nem lehet, mert ez egy **struct**)

`*p++->csat`

azután inkrementálja **p**-t, hogy hozzáfért ahhoz, amire csat mutat (de ekkor nem biztos, hogy **p** továbbra is a lánc valamelyik elemére mutat).



# Láncok

- Láncok bejárására írhatunk olyan függvényt amelynek paramétere az elvégzendő művelet:

```
typedef void (*muvelettip)(elemtip*);

void bejar(pozicio lanc, muvelettip mov)
{
    pozicio p;
    for(p = lanc; p != NULL; p = p->csat) {
        /* művelet a sorozat elemén */
        mov(&(p->adat));
    }
}
```

# **INPUT/OUTPUT, FÁJLOK**

# Az I/O alapjai

- 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 az írás/olvasás fizikailag nagyobb darabokban történik.
- A puffer hossza a `stdio.h`-ban van definiálva:
  - `#define BUFSIZ _IO_BUFSIZ`ahol
  - `#define _G_BUFSIZ 8192`
  - `#define _IO_BUFSIZ _G_BUFSIZ`

# Magas szintű fájlkezelés

- A UNIX-ban 3 előre definiált stream van:
  - **stdin**
  - **stdout**
  - **stderr**
- Mindhárom karakterfolyamnak tekinthető és a program indulásakor automatikusan úgy nyitódnak meg, hogy
  - az **stdin** a billentyűzethez
  - az **stdout** és a **stderr** a képernyőhöz rendelődik.

# Standard fájlok kezelése

- Az **stdin**, **stdout**, **stderr**
  - magas szinten,
  - szöveges módbankezelt fájlok
- A fájlkezelő függvények a **stdio.h**-ban vannak

# Magas szintű fájlkezelés

- Természetesen a UNIX-ban szokásos átírányításokkal az eredmény adatállományba is írható, illetve az input adatok adatállományból is vehetőek, de ez nem a C, hanem az operációs rendszer tulajdonsága.
- Sok függvény áll rendelkezésünkre, hogy egy karakterfolyamot kezeljünk.

# Alapvető függvények

- **int getchar(void)**
  - Makró, mely beolvas egy karaktert a **stdin**-ről.
  - Az **EOF** konstans értéket adja vissza, amikor az általa éppen olvasott, bármiféle bemenet végére ért (például lenyomtuk a **<ctrl> + d** billentyűkombinációt Linux-ban)
  - Implementációtól függően, de általában pufferelesen olvas:
    - A program csak akkor kapja meg a begépelt karakter(ek)e)t, ha a puffer betelt vagy sorvéget (fájlvégét) nyomtunk.
    - Közvetlen billentyűleütés érzékelésére nem alkalmas, de nem is ez a feladata. (A közvetlen billentyűleütést csak oprendszerfüggő módon tudnánk figyelni.)



# Alapvető függvények

- **char \*gets(char \*buf)**
  - Beolvas egy sort a **stdin**-ről a buf-ba.
  - A buf egy létező és elegendő hosszú karaktertömb kell, hogy legyen, mert:
    - Nem ellenőrzi, hogy szabad vagy foglalt memóriaterületet ír-e felül.
  - Válaszértéke
    - A karaktertömb címe sikeres olvasáskor .
    - **NULL** pointer sikertelen olvasáskor (hiba vagy fájlvége a sor elején).

# Alapvető függvények

- A sor közbeni fájlvége esetén
  - Lezárja a sztringet a buf-ban.
  - A buf pointerrel tér vissza.
- Az újsor karaktert nem olvassa be, hanem ' \0 ' karaktert tesz helyette a buf-ba (eltér az **fgets()** függvénytől!), ezért nem alkalmas az újsor karakter beolvasására (arra használhatjuk a **getchar()** makrót).
- Egy megjegyzés a gets manual oldalról:
  - Never use gets!
  - Soha ne használd a gets függvényt!
  - A buffer overflow súlyos biztonsági hiba lehet!

# Alapvető függvények

- **`int putchar(int ch)`**
  - Makró, mely kiír egy karaktert a **`stdout`**-ra.
  - A kiírt karakter kódját adja vissza sikeres művelet esetén, **`EOF`**-ot ha hiba történt.
  - Nem pufferelt.

# Alapvető függvények

- **`int puts(const char *buf)`**
  - A `buf` stringet kiírja a **`stdout`**-ra.
  - A sztringvége karakter helyett újsort ír.
  - Válaszértéke:
    - Egy nemnegatív érték sikeres végrehajtás esetén.
    - **`EOF`** hiba esetén.

# Példa

- A bemenet kisbetűssé alakítása:

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

main()
{
    int c;
    while ((c = getchar()) != EOF) {
        putchar(isupper(c) ? tolower(c) : c);
        /* putchar(tolower(c)); is jó megoldás */
    }
}
```

# Formatált I/O műveletek

- **`int printf(char *formátum, ...)`**
  - Kinyomtatja a **`stdout`**-ra az aktuális paraméterek értékeit a formátum szövegben előírt formában.
  - A visszatérési érték a kiírt karakterek száma.
  - A formázó karakterlánc kétféle típusú objektumot tartalmaz:
    - Közöséges karaktereket, amelyeket egyszerűen a kimeneti folyamra másol.
    - Konverzió-specifikációkat, amelyek mindegyike a soron következő paraméter konvertálását és kiíratását írja elő.

# Formatált I/O műveletek

- Formátumbeli speciális ESCAPE karakterek:
  - **\a** hangjelzés
  - **\n** újsor
  - **\r** CR
  - **\b** backspace
  - **\t** vízszintes tabulátor
  - **\v** függőleges tabulátor
  - **\f** lapdobás (form-feed)

# Formatált I/O műveletek

- Formátumbeli speciális ESCAPE karakterek:
  - `\"` `"`
  - `\\` `\`
  - `\NNN`
    - Az NNN oktális kódú karakter (legfeljebb 3 darab).
  - `\xnn`
    - Az nn hexadecimális kódú karakter.
  - `\unnnnn`
    - Az nnnn hexadecimális kódú 16 bites Unicode karakter.
  - `\Unnnnnnnnnn`
    - Az nnnnnnnnn hexadecimális 32 bites Unicode kódú karakter.



# Formatált I/O műveletek

- Egy konverzió-specifikáció (más néven formátumvezérlő szekvencia) általános alakja:

**% [flag] [n [.m]] [{h,l}] type**

(A [ ] közötti rész szabadon elhagyható.)

# Formatált I/O műveletek

- `%[flag][n[.m]][{h,l}]type`
- A **type** egy karakter, melyet konverziós karakternek nevezünk.
- A konverziós karakterek és jelentésük:
  - **d, i**
    - A paraméter decimális jelölésmódúvá alakul (**int, short, long, long long**).
  - **u**
    - A paraméter előjel nélküli decimális jelölésmódúvá alakul (**unsigned int, unsigned short, unsigned long, unsigned long long**).

# Formatált I/O műveletek

## – o

- A paraméter előjel nélküli oktális számmá konvertálódik (**unsigned int, unsigned short, unsigned long, unsigned long long**).

## – x, X

- A paraméter előjel nélküli hexadecimális számmá konvertálódik (**unsigned int, unsigned short, unsigned long, unsigned long long**).

## – c

- A paraméter egyetlen karakter (**char**).

# Formatált I/O műveletek

## – **f**

- A paramétert **float**-nak vagy **double**-nak tekinti, és a [-]mmm.nnnnnn decimális jelölésmódba konvertálja, ahol az n-ek karakterláncának hosszát a pontosság adja meg. Az alapértelmezés szerinti pontosság 6.

## – **e, E**

- A paramétert **float**-nak vagy **double**-nak tekinti, és a [-]m.nnnnnnE[-]xx decimális jelölésmódba konvertálja, ahol a számjegyek számát a pontosság adja meg. Az alapértelmezés szerinti pontosság 6.

## – **g, G**

- **%e** és **%f** közül a rövidebbet használja; az értéktelen nullákat elhagyja.

# Formatált I/O műveletek

- **s**
  - A paraméter karakterlánc (sztringre mutató pointer); a láncbeli karakterek a pontossággal megadott darabszámig vagy a nulla karakterig mindaddig nyomtatódnak.
- **p**
  - Cím (pointer) hexadecimális formában.
- **n**
  - Az eddig a pontig kiírt karakterek számát adja vissza a megfelelő paraméter által mutatott **int** változóban.
- **%**
  - Kiírja a % karaktert.
- Ha a %-ot követő karakter nem konverziós karakter, az illető karakter nyomtatásra kerül.

# Formatált I/O műveletek

- `% [flag] [n [.m]] [length] type`
- A **flag** hiányában a kiíratás jobbra igazítva, balról szóközökkel kitöltve történik
  - `-` balra igazítás.
  - `+` előjel kötelező kiírása.
  - `' '` (szóköz) kötelező előjel, `+` helyett szóközt ír.
  - `0` számok előtt a kitöltés a `'0'` karakterrel megy.
  - `#`
    - Lebegőpontosnál kötelező tizedespont.
    - `o, x, X` esetén `0`, `0x`, `0X` kiírása nullától különböző értékek előtt.

# Formatált I/O műveletek

- `% [flag] [n [.m] ] [length] type`
- Az `n` a mezőszélességet jelenti
  - Egy decimális szám, amely a minimális mezőszélességet határozza meg. (A számérték nem kezdődhet 0-val, mert azt flag-ként értelmezi.)
  - Ha a szám helyén egy `'*'` szerepel, akkor a szélességet a következő argumentum kifejezés értéke határozza meg:

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

# Formatált I/O műveletek

- A következő argumentum értéke legalább ilyen széles, vagy szükség esetén szélesebb mezőbe nyomtatódik ki.
- Ha a konvertált argumentum kevesebb karakterből áll, mint a mezőszélesség, akkor bal oldalon (vagy, ha a balra igazítás jelző szerepel, akkor jobb oldalon) a mező kitöltődik, hogy ezáltal az előírt mezőszélesség meglegyen.
- A kitöltő karakter közönséges esetben szóköz, ill. amennyiben a számot íratunk ki jobbra igazítva, és megadtuk a 0 flag-et, akkor a ' 0 ' karakter.



# Formatált I/O műveletek

- `% [flag] [n [.m] ] [length] type`
- Az **m** a pontosságot jelenti
  - Decimális szám:
    - **i, d, o, x, X, u** esetén legalább ennyi számjegyet ír ki, szükség esetén balról a '0' karakterrel kitöltve a helyet.
    - **f, e, E, g, G** esetén tizedesjegyek illetve értékes jegyek száma; az utolsó jegyet kerekíti. Ha 0, akkor a tizedespont sem kerül kiíratásra. Hiányában az alapértelmezés 6.
    - **s** esetén a sztringből kiírható karakterek maximális száma.
  - Ha a szám helyén egy ' \* ' szerepel, akkor a pontosságot a következő argumentum kifejezés értéke határozza meg.

# Formatált I/O műveletek

- `%[flag][n[.m]]length type`
- A **length** a hosszmodosító
  - **hh**
    - Az argumentumot **char**-ként kezeli.
  - **h**
    - Az argumentumot **short**-ként kezeli.
  - **l, L**
    - Az argumentumot **long**-ként illetve **long double**-ként kezeli.
  - **ll**
    - Az argumentumot **long long**-ként kezeli.

# Formatált I/O műveletek

- Figyeljük meg, hogy
  - A **printf()** kerekít
  - A konvertálás csonkít

```
#include <stdio.h>
int main() {
    int i;
    double d;
    for(d=-2.0; d<=2.0; d+=1.0/3.0) {
        i=d;
        printf("d=%23.201f; %5.21f; d=%2.01f; i=%2d;\n",
               d,          d,          d,          i);
    }
}
```

# Formatált I/O műveletek

```
d=-2.0000000000000000000000; -2.00; d=-2; i=-2;  
d=-1.666666666666666666666674068; -1.67; d=-2; i=-1;  
d=-1.333333333333333333333348136; -1.33; d=-1; i=-1;  
d=-1.000000000000000000000022204; -1.00; d=-1; i=-1;  
d=-0.666666666666666666666696273; -0.67; d=-1; i= 0;  
d=-0.333333333333333333333364790; -0.33; d=-0; i= 0;  
d=-0.000000000000000000000033307; -0.00; d=-0; i= 0;  
d= 0.3333333333333333333333298176; 0.33; d= 0; i= 0;  
d= 0.666666666666666666666629659; 0.67; d= 1; i= 0;  
d= 0.999999999999999999999955591; 1.00; d= 1; i= 0;  
d= 1.3333333333333333333333281523; 1.33; d= 1; i= 1;  
d= 1.666666666666666666666607455; 1.67; d= 2; i= 1;  
d= 1.999999999999999999999933387; 2.00; d= 2; i= 1;
```

# Formatált I/O műveletek

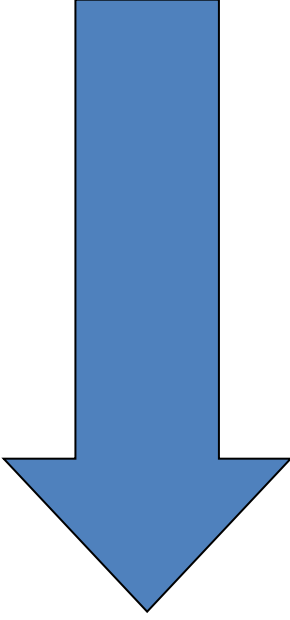
- Mi történik a **printf()** hívásakor ? Pl.:  
`printf("%d %d %d\n", 10, 20)`
- A verembe (stack) beletevődnek az argumentumok, de milyen sorrendben?
- Lényeges, hogy kiolvasásnál a formátum sztring megelőzze a többi argumentumot, hiszen abból tudjuk, hogy hány és milyen típusú értékeket kell kiolvasnunk.

# Formatált I/O műveletek

- Tehát a formátum sztringnek kell a verem „tetején” lennie, ezért utolsónak kell betenni.
- Tehát a paramétereket a C
  - jobbról balra haladva helyezi el a veremben,
  - egyre csökkenő címeken,
  - végül még beteszi a visszatérési címet

# Formatált I/O műveletek

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

	...
	20
	10
	"%d %d %d\n" címe
	Visszatérési cím

# Formatált I/O műveletek

- A függvényhívás után az argumentumokat a veremből törölni kell; de ki takarít?
  - Ha a hívott **printf()** visszatérés előtt takarítana, akkor a fenti esetben a formátum sztring alapján még egy további (nem ide tartozó) értéket kitörölné a veremből és az egész összezavarodna.
  - Tehát az argumentumokat a hívó függvény törli a visszatérés után, hisz ő tudja biztosan, hogy ténylegesen hány paramétert adott át.



# Formatált I/O műveletek

- A programozó felelőssége, hogy adott konverzió-specifikációhoz megfelelő típusú argumentumérték tartozzon:

```
#include <stdio.h>
int main()
{
    printf("%20lld %20lld\n", (long long)2005, 2005.0);
    printf("%20f %20f\n",      (long long)2005, 2005.0);
}
```

A program kimenete:

```
          2005  4656532898701115392
          0.000000          2005.000000
```

# Formatált I/O műveletek

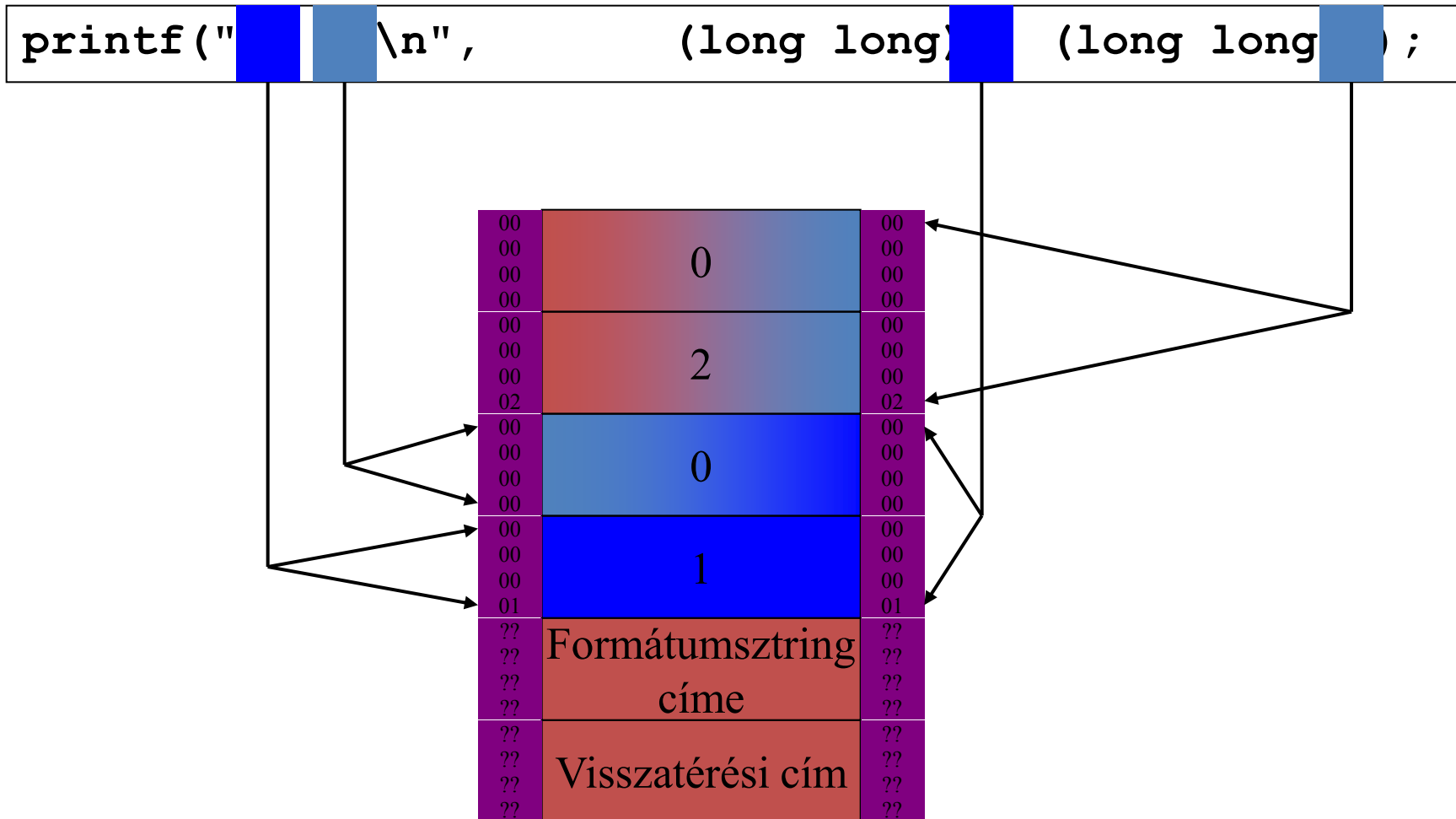
- Vagy:

```
#include <stdio.h>
int main()
{
    printf("%x %x %x %x\n", 1, 2, 3, 4);
    printf("%x %x %x %x\n",
        (long long)1, (long long)2, (long long)3, (long long)4);
    printf("%llx %llx %llx %llx\n",
        (long long)1, (long long)2, (long long)3, (long long)4);
    printf("%llx %llx %llx %llx\n", 1, 2, 3, 4);
}
```

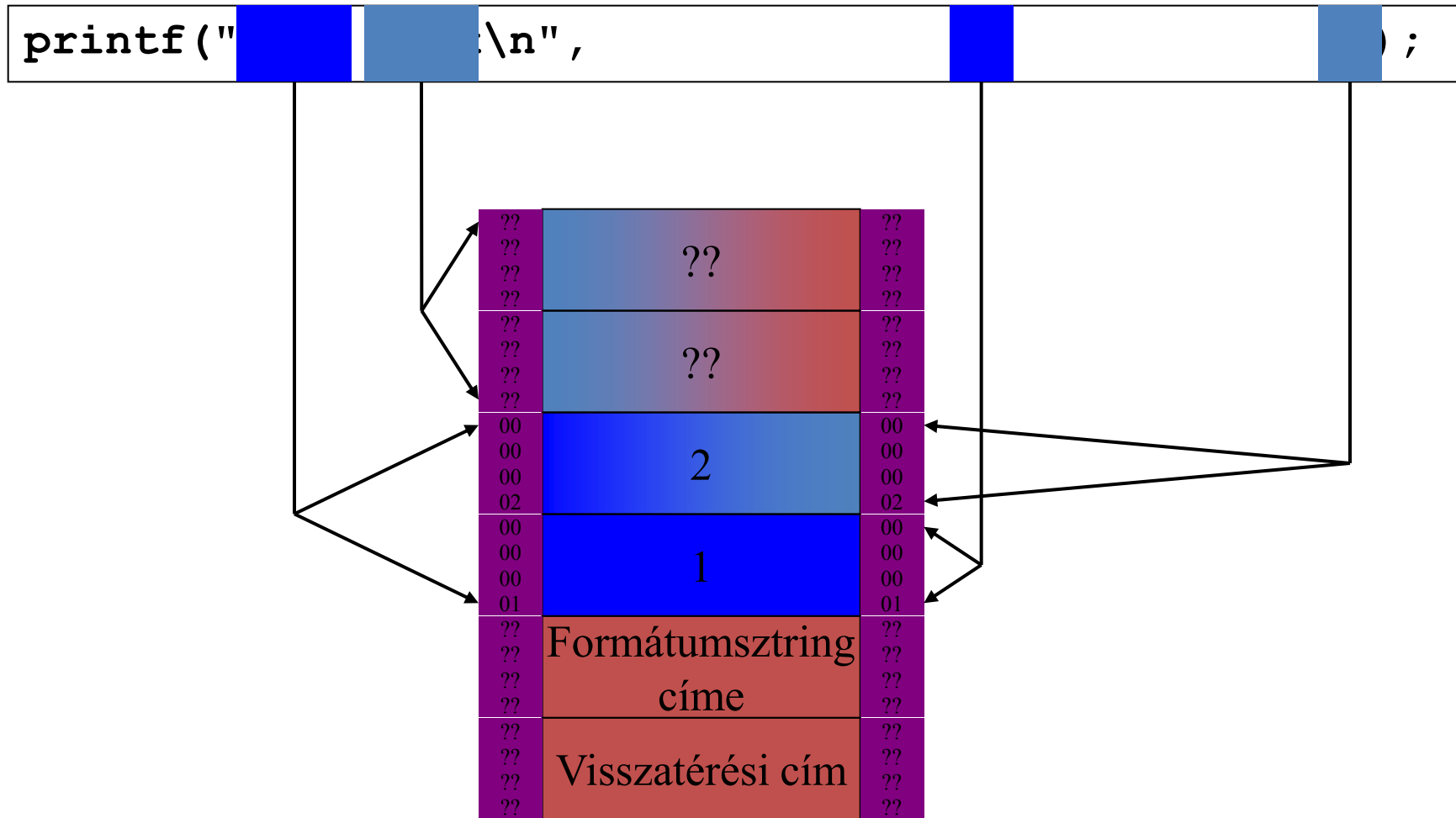
- A program kimenete:

```
1 2 3 4
1 0 2 0
1 2 3 4
200000001 400000003 bffd8d78bffd8df4 b7fabff40804844a
```

# Formátált I/O műveletek



# Formátált I/O műveletek



# Formatált I/O műveletek

- **int scanf(char \*formátum, ...)**
  - Beolvassa a **stdin**-ről a karaktereket.
  - Konvertálja a megfelelő formátumra.
  - Elhelyezi a paraméterlistában megadott memóriacímekre.
  - A visszatérési érték a beolvasott és sikeresen eltárolt értékek száma.
  - A formátumvezérlés technikája megegyezik a **printf()** formátumvezérlésével.

# Formatált I/O műveletek

- A formátum string fix karaktereit sorra összeveti a **stdin**-ről olvasottakkal.
- A láthatatlan karaktereket egyetlen szóköznek tekinti.
- A formátumparaméterek helyén megpróbálja elvégezni a konverziót.
- Megkeresi az első még fel nem használt (azaz következő) argumentumot és arra a címre elhelyezi az eredményt.

# Formatált I/O műveletek

- Például:

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

- Az stdin:

3 eredmény: 3 4 5<ENTER>

- Akkor

v1=3

v2=3

# Formatált I/O műveletek

- A formátum általános alakja:

`% [*] [width] [{h,l}] type`

- \*

- Az adatmezőt beolvassa, de nem helyezi el sehova (tehát ehhez nem kell változócímet megadni)



# Formatált I/O műveletek

- `%[*][width][length]type`
- `width` (szélesség)
  - A konverzió során figyelembe veendő karakterek száma; hiányában addig olvas, míg logikai alapon meg nem találja a megfelelő mező végét (pl. a típushoz nem illő karaktert).

# Formatált I/O műveletek

- `%[*][width][length]type`
- `length`
  - `hh`
    - A paraméter `char *`
  - `h`
    - A paraméter `short int*`
  - `l`
    - A paraméter `long int*` vagy `double*`
  - `L`
    - A paraméter `long long*` vagy `long double*`

# Formatált I/O műveletek

- `%[*][width][length] type`
- `type`
  - `c`
    - `width` darab (ha nincs megadva, akkor 1) karakter (`char*`)
  - `d,D`
    - decimális szám (`char*, ... int*`)
  - `o,O`
    - oktális szám (`char*, ... int*`)
  - `x,X`
    - hexadecimális szám (`char*, ... int*`)

# Formatált I/O műveletek

## – **i, I**

- (**char\***, ... **int\***)
- 0-val kezdődő szám oktális
- 0**x**-el kezdődő szám hexadecimális
- 0-tól különböző egészszel kezdődő decimális szám

## – **u, U**

- előjeltelen decimális érték  
(**unsigned char\***, **unsigned ... int\***)

## – **f**

- lebegőpontos érték (**float\***, **double\***, **long double\***)

# Formatált I/O műveletek

## – s

- Karakterlánc (sztring) (**char\***)
- Hosszúsága legfeljebb **width**, az első whitespace karakterig olvas, melyet sztringvégi null karakterrel helyettesít.

## – [**karakterek**]

- Sztring kizárólag a megadott karakterekből (**char\***)
- Hosszúsága: **width**, vagy az első olyan karakterig, amit nem soroltunk fel.

## – [**^karakterek**]

- Sztring kizárólag a meg NEM adott karakterekből (**char\***)
- Hosszúsága: **width**, vagy az első olyan karakterig, amit felsoroltunk.

# Formatált I/O műveletek

- Problémák:
  - Ha egy karaktert nem tud értelmezni, akkor abbahagyja az olvasást.
  - A maradékot benne hagyja az **stdin**-ben.
  - A program nem tudja, hogy mi volt a probléma.
  - Üres sztringet a **%s** segítségével nem tudunk beolvasni.
  - **%[^\\n]** esetén az újsor karaktert már nem dolgozza fel.
  - Ha az újsort betesszük a formátum sztringbe, akkor beolvassa a **scanf**, de a beolvasás végét egy újabb sorvéggel kell jeleznünk.

# Formatált I/O műveletek

- Megtehetjük, hogy minden **scanf** után **getchar()**-okkal ellépkedünk az újsor utánig, de lehet, hogy akkor értékes karaktereken is túlmegyünk.
- Másik megoldás:

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

# Alternatív I/O

- **int sscanf(char \*mp, char \*form,...)**
  - A memória adott területét megpróbálja a **scanf()** -hez hasonlóan értelmezni és átmásolni.
  - A **scanf()** kiváltására használhatunk **sscanf()** -et is, az legalább nem az **stdin** fájlmutatóját rontja el, hanem csak egy memóriapointert.
- **int sprintf(char \*mp, char \*form,...)**
  - A **printf()** adott memóriaterületre író változata, belső konverziók elvégzésére is használható.



# Hozzáférés az adatállományokhoz

- A már említett 3 fájl, az **stdin**, **stdout** és **stderr** úgy képzelhető el, mint egy folyam:
  - 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.
- Általánosságban viszont egy fájl tartalma megmarad, tehát akárhányszor újra ránézhetünk tetszőleges részére.

# Hozzáférés az adatállományokhoz

- Minden fájl felfogható elemek (jelen esetben 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

$$[a_0 \dots a_{i-1}][a_i \dots a_{n-1}]$$

alakban adható meg.

- Az író-olvasó fej által kijelölt elem a második sorozat első eleme.

# Hozzáférés az adatállományokhoz

- Bármilyen művelet a fájlban (egy elem írása vagy olvasása) 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.

# Hozzáférés az adatállományokhoz

- Minden fájlhoz egy **stdio.h**-beli **FILE** struktúrát rendelünk hozzá.
- 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.

# Hozzáférés az adatállományokhoz

- Ha tehát fájlokkal szeretnénk dolgozni, a következő műveletekre lesz szükségünk:
  - Először a **FILE** típusú változóhoz hozzárendelünk egy operációs rendszerbeli adatállományt vagy eszközt:

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

- Majd használjuk.
- Végül lezárjuk:

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

# Hozzáférés az adatállományokhoz

- **FILE \*fopen(const char \*path, const char \*mode)**
  - A **path** az operációs rendszerben létező fájlra való szabályos hivatkozás.
  - Ha a megnyitás nem sikerült, akkor a visszatérési érték **NULL**
  - Ha sikerült, akkor egy **FILE**-re mutató pointer.

# Hozzáférés az adatállományokhoz

- A **mode** a megnyitás módja, ami lehet:
  - **"r"** (read):
    - Egy már létező fájl megnyitása olvasásra. Ha nem létezik vagy nincs rá olvasási jogunk, az hiba.
  - **"w"** (write):
    - 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.
  - **"a"** (append):
    - 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.

# Hozzáférés az adatállományokhoz

- **"r+"** (read+write):
  - Egy már létező fájl megnyitása olvasásra és írásra. Ha nem létezik vagy nincs rá jogunk, az hiba.
- **"w+"** (write+read):
  - Egy fájl megnyitása írásra és olvasásra. Ha már létezett, a régi tartalma törlődik, egyébként létre lesz hozva. Ha nincs rá jogunk, vagy nem tudjuk létrehozni, az hiba.
- **"a+"** (append+read):
  - Egy fájl megnyitása hozzáfűzésre és olvasásra. 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 jogunk, vagy nem tudjuk létrehozni, az hiba.



# Hozzáférés az adatállományokhoz

- A **mode** bizonyos implementációkban tartalmazhatja még:
  - **'t'** (text):
    - Szöveges mód.
  - **'b'** (binary):
    - Bináris mód.
  - E két módnak linux alatt nincs jelentősége. Ha a megjelölés elmarad, a text mód az alapértelmezés.

# Hozzáférés az adatállományokhoz

- Az író-olvasó fej **r**, **r+**, **w**, **w+** esetén a fájl elejére, **a**, **a+** esetén a fájl végére áll.
- Ha a fájlt **r+**, **w+**, **a+** módok valamelyikében nyitottuk meg és váltani akarunk írás és olvasás között, akkor célszerű a puffert az **fseek()** vagy **fflush()** függvények valamelyikével a kiüríttetni.

# Hozzáférés az adatállományokhoz

- Az **fopen** használata:

```
FILE *fp;                                /* fájlpontier */  
if ((fp = fopen("/etc/motd", "r")) == NULL) {  
    printf("Nem sikerült megnyitni a /etc/motd adatállományt");  
    exit(1);  
}
```

- A zárójelezésre nagyon kell vigyázni!
- Biztonságosabb, ha az értékadás egy külön sorban történik meg:

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

# Hozzáférés az adatállományokhoz

- Alapvető műveletek:
  - **int fgetc(FILE \*fp)**
    - Egy karakter olvasása fp-ről.
  - **int getc(FILE \*fp)**
    - Egy karakter olvasása fp-ről (makró).
  - **int fputc(char ch, FILE \*fp)**
    - Egy karakter írása fp-re.
  - **int putc(char ch, FILE \*fp)**
    - Egy karakter írása fp-re (makró).

# Hozzáférés az adatállományokhoz

- `int fseek(FILE *fp, long mennyi, int honnan)`
  - Fájlok direkt elérése, vagyis a fej pozícionálása.
  - Lépteti a fájlpontert **mennyi** bájtnyit.
  - **honnan** (viszonyítva) lehet:
    - **SEEK\_SET** : a fájl elejétől.
    - **SEEK\_CUR** : jelenlegi fájlpozíciótól.
    - **SEEK\_END** : fájl végétől.

# Hozzáférés az adatállományokhoz

- A fájlpointer értéke negatív nem lehet, de mutathat a fájlvégén túlra is, ekkor a rendszer addig a pozícióig nyújtja (szeméttel kiegészíti) a fájlt, és akkor is ott fogja a műveleteket végezni.
- Ha a pozicionálás sikeres volt, akkor 0-t ad vissza.

# Hozzáférés az adatállományokhoz

- `char *fgets(char *buf, int n, FILE *fp)`
  - Egy sort (de legfeljebb n-1 karaktert) olvas be a fájlból, beleértve a sorvég karaktert is, ezt elhelyezi a buf karaktertömbbe, amelyet sztringvégi nulla karakterrel lezár.
  - Visszatérési értéke:
    - A tömb címe (valóban haszontalan, hisz ezt mi adtuk meg paraméterként) – ha sikerült a művelet.
    - **NULL** – hiba esetén (fájl vége, olvasási hiba).

# Hozzáférés az adatállományokhoz

- **int fputs(char \*buf, FILE \*fp)**
  - A buf sztring tartalmát kiírja a fájlba.
  - Nem tesz a sor végére automatikusan újsort.
  - Válaszértéke:
    - Az utolsó kiírt karakter kódja .
    - 0 – ha a string üres volt.
    - **EOF** – hiba esetén.



# Hozzáférés az adatállományokhoz

- `int fprintf(FILE *fp, char *form,...)`
  - A `printf()` függvény fájlba író változata.
- `int fscanf(FILE *fp, char *form,...)`
  - A `scanf()` függvény fájlból olvasó változata.

# Hozzáférés az adatállományokhoz

- **`int ungetc(int c, FILE *fp)`**
  - Visszateszi a `c` karaktert a fájlra úgy, hogy a legközelebbi olvasás ezt a karaktert fogja olvasni.
  - Felhasználás: pl. számjegyeket olvasunk be, amikor nem számjegy karakter jött, azt visszatesszük, a számjegyeket pedig egy számmá konvertáljuk.
  - Legfeljebb egy karaktert lehet visszatenni, az **`ungetc()`** többszöri egymás utáni meghívása nem megengedett.

# Hozzáférés az adatállományokhoz

- **long ftell(FILE \*fp)**
  - Lekérdezi a fej pozícióját.
- **int feof(FILE \*fp)**
  - Válaszérték:
    - 0: még nem értük el a fájl végét.
    - más: elértük a fájl végét.
- **int ferror(FILE \*fp)**
  - Válaszérték:
    - 0: a fájl megnyitása óta nem történt hiba.
    - más: hiba történt.

# Hozzáférés az adatállományokhoz

- `int fread(char *buf, int rsize, int rn, FILE *fp)`
  - Az **rsize** hosszúságú csomagokból **rn** darabot olvas be a **buf** címre.
  - Visszaadja a ténylegesen beolvasott csomagok számát.

# Hozzáférés az adatállományokhoz

- `int fwrite(char *buf, int rsize, int rn, FILE *fp)`
  - Az `rsize` hosszúságú csomagokból `rn` darabot ír ki a `buf` címről.
  - Visszaadja a ténylegesen kiírt csomagok számát.

# Hozzáférés az adatállományokhoz

- **`int fflush(FILE *fp)`**
  - Kiírja a puffer tartalmát.
  - Válaszértéke:
    - 0: sikeres
    - **EOF**: sikertelen
- **`int fclose(FILE *fp)`**
  - Lezárja a fájlt.
  - Válaszértéke:
    - 0: sikeres
    - **EOF**: sikertelen

# Hozzáférés az adatállományokhoz

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

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

# Hozzáférés az adatállományokhoz

```
#include <stdio.h>

main() {
    FILE *fp, *fpb;
    int a,b;

    fp=fopen("probat.txt","wt");
    fprintf(fp,"%d",34);
    fflush(fp); fclose(fp);

    fp = fopen("probat.txt","rt");
    fscanf(fp,"%d",&a);
    printf("%d\n",a);
    fclose(fp);
```



# Hozzáférés az adatállományokhoz

```
fpb =fopen("probab.dat","w+b");  
fwrite(&a,sizeof(int),1,fpb);  
fseek(fpb, 0, SEEK_SET);  
fread(&b,sizeof(int),1,fpb);  
printf("%d\n",b);  
fclose(fpb);  
}
```

- Az output:

34

34

# Felhasznált anyagok

- Dévényi Károly (SZTE): Programozás alapjai
- Simon Gyula (PE): A programozás alapjai
- Pohl László (BME): A programozás alapjai
- B. W. Kernighan - D. M. Ritchie: A C programozási nyelv