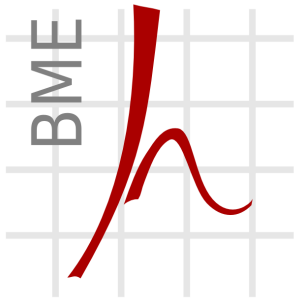
Dinamikus memóriakezelés – Fájlkezelés

A programozás alapjai I.



Hálózati Rendszerek és Szolgáltatások Tanszék
Farkas Balázs, Fiala Péter, Vitéz András, Zsóka Zoltán

2018. október 15.

Tartalom

1. fejezet

Dinamikus memóriakezelés

Dinamikus memóriakezelés

- Olvassunk be egész számokat, és írjuk ki őket fordított sorrendben!
- A beolvasandó egész számok számát is a felhasználótól kérjük be!
- Csak annyi memóriát használjunk, amennyi feltétlenül szükséges!

1. Beolvassuk a darabszámot (`n`)
2. `n` egész szám tárolására elegendő memóriát kérünk az operációs rendszertől
3. Beolvassuk és eltároljuk a számokat, kiírjuk őket fordítva
4. Visszaadjuk a lefoglalt memóriát az operációs rendszernek

Példa

```
1  int n, i;
2  int *p;
3
4  printf("Hányat olvassak be? ");
5  scanf("%d", &n);
6  p = (int*)malloc(n*sizeof(int));
7  if (p == NULL) return;
8
9  printf("Kérek %d számot:\n", n);
10 for (i = 0; i < n; ++i)
11   scanf("%d", &p[i]);
12
13 printf("Fordítva:\n");
14 for (i = 0; i < n; ++i)
15   printf("%d ", p[n-i-1]);
16
17 free(p);
18 p = NULL;
```

link

```
p:0x0000
```

```
Hányat olvassak be?  5
Kérek 5 számot:
1 4 2 5 8
Fordítva:
8 5 2 4 1
```

A malloc és free függvények – <stdlib.h>

```
void *malloc(size_t size);
```

- size bájt egybefüggő memóriát foglal, és a lefoglalt terület címét visszaadja void* típusú értékként
- A visszaadott void* „csak egy cím”, ami nem dereferálható. Akkor lesz használható, ha átkonvertáljuk (pl. int*-gá).

```
1 int *p; /* int adat címe */
2 /* Memóriafoglalás 5 int-nek */
3 p = (int *)malloc(5*sizeof(int));
```

- Ha nem áll rendelkezésre elég egybefüggő memória, a visszatérési érték NULL. Ezt mindig ellenőrizni kell.

```
1 if (p != NULL)
2 {
3   /* használat, majd felszabadítás */
4 }
```

A malloc és free függvények – <stdlib.h>

```
void free(void *p);
```

- A p címen kezdődő egybefüggő memóriaterületet felszabadítja
- Méretet nem adjuk meg, mert azt az op.rendszer tudja (felírta a memóriaterület elé, ezért a kezdőcímmel kell hívni)
- free(NULL) megengedett (nem csinál semmit), ezért lehet így is:

```
1 int *p = (int *)malloc(5*sizeof(int));
2 if (p != NULL)
3 {
4   /* használat */
5 }
6 free(p);  /* nem baj, ha NULL */
7 p = NULL; /* ez jó szokás */
```

- Mivel a nullpointer nem mutat sehova, jó szokás felszabadítás után kinullázni a mutatót, így látni fogjuk, hogy nincs használatban.

malloc – free

- a `malloc` és a `free` kéz a kézben járnak
- ahány `malloc`, annyi `free`

```
1  char *WiFi = (char *)malloc(20*sizeof(char));
2  int *Lunch = (int *)malloc(23*sizeof(int));
3  ...
4  free(WiFi);
5  free(Lunch);
```

- Ha a felszabadítás elmarad, memóriaszivárgás (memory leak)
- Jó szokások:
  - Amelyik függvényben foglalunk, abban szabadítsunk
  - A `malloc` által visszaadott mutatót ne módosítsuk, ha lehet, ugyanazon keresztül szabadítsunk
- Van, hogy nem lehet tartani a jó szokásokat, ezt külön (kommentben) jelezzük

A calloc függvény – <stdlib.h>

```
void *calloc(size_t num, size_t size);
```

- egybefüggő memóriát foglal num darab, egyenként size méretű elemnek, a lefoglalt területet kinullázza, és címét visszaadja void* típusú értékként
- Használata szinte azonos a malloc-kal, csak ez elvégzi a num*size szorzást, és kinulláz.
- A lefoglalt területet ugyanúgy free-vel kell felszabadítani

```
1 int *p = (int *)calloc(5, sizeof(int));
2 if (p != NULL)
3 {
4   /* használat */
5 }
6 free(p);
```

A realloc függvény – <stdlib.h>

```
void *realloc(void *memblock, size_t size);
```

- korábban lefoglalt meóriaterületet átméretez size bájt méretűre
- új méret lehet kisebb is, nagyobb is, mint a régi
- ha kell, új helyre másolja a korábbi tartalmat, az új elemeket nem inicializálja
- visszatérési értéke az új terület címe

```
1 int *p = (int *)malloc(3*sizeof(int));
2 p[0] = p[1] = p[2] = 8;
3 p = realloc(p, 5*sizeof(int));
4 p[3] = p[4] = 8;
5 ...
6 free(p);
```

Példa

- Írjunk függvényt, mely a paraméterként kapott két sztringet összefűzi. A függvény foglaljon helyet az eredménysztringnek, és adja vissza annak címét.

```c
/* concatenate -- két sztring összefűzése
   dinamikusan foglal, az eredmény címét adja vissza
*/
char *concatenate(char *s1, char *s2) {
    size_t l1 = strlen(s1);
    size_t l2 = strlen(s2);
    char *s = (char *)malloc((l1+l2+1)*sizeof(char));
    if (s != NULL) {
        strcpy(s, s1);
        strcpy(s+l1, s2); /* vagy strcat(s, s2) */
    }
    return s;
}
```

link

Példa

A függvény használata

```
1  char word1[] = "ló", word2[] = "darázs";
2
3  char *res1 = concatenate(word1, word2);
4  char *res2 = concatenate(word2, word1);
5  res2[0] = 'v';
6
7  printf("%s\n%s", res1, res2);
8
9  /* A függvény memóriát foglalt, felszabadítani! */
10 free(res1);
11 free(res2);
```

link

```
lódarázs
varázsló
```

2. fejezet

Többdimenziós tömbök

Többdimenziós tömbök

1D tömb	Azonos típusú elemek a memóriában egymás mellett tárolva
2D tömb	Azonos méretű és típusú 1D tömbök a memóriában egymás mellett tárolva
3D tömb	Azonos méretű és típusú 2D tömbök a memóriában egymás mellett tárolva
...	...

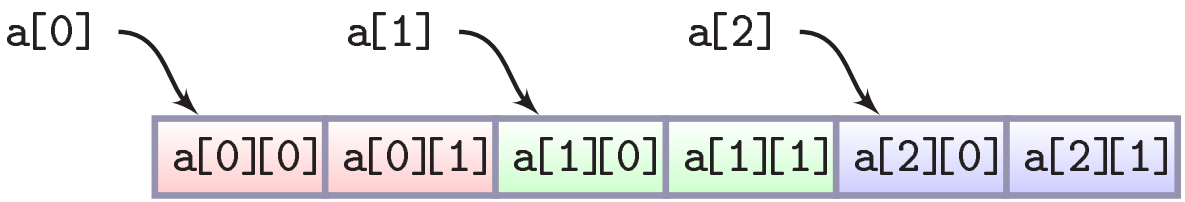
Kétdimenziós tömbök

- 2D tömb deklarációja:

```
1 char a[3][2]; /* 3 soros két oszlopos karaktertömb */
2               /* 2 elemű 1D tömbök 3 elemű tömbje */
```

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

- C-ben sorfolytonos tárolás, vagyis a hátsó index fut gyorsabban



- a[0], a[1] és a[2] 2 elemű 1D tömbök

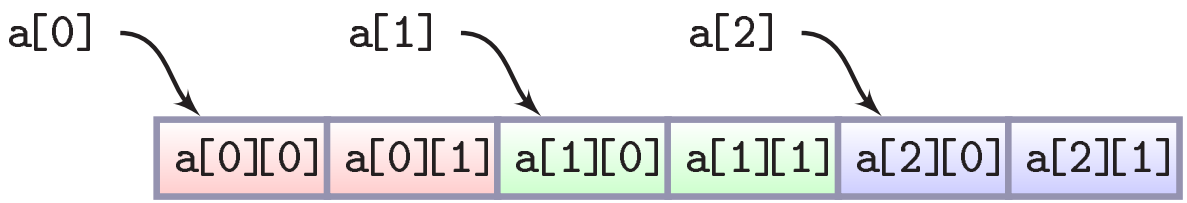
Kétdimenziós tömb átvétele soronként

- 1D tömb (sor) feltöltése adott elemmel

```c
void fill_row(char row[], size_t size, char c)
{
  size_t i;
  for (i = 0; i < size; ++i)
    row[i] = c;
}
```

- 2D tömb feltöltése soronként

```c
char a[3][2];
fill_row(a[0], 2, 'a'); /* 0. sor csupa 'a' */
fill_row(a[1], 2, 'b'); /* 1. sor csupa 'b' */
fill_row(a[2], 2, 'c'); /* 2. sor csupa 'c' */
```



Kétdimenziós tömb átvétele egyben

- átvétel 2D tömbként – csak ha az oszlopok száma ismert

```
1  void print_array(char array[][2], size_t nrows)
2  {
3    size_t row, col;
4    for (row = 0; row < nrows; ++row)
5    {
6      for (col = 0; col < 2; ++col)
7        printf("%c", array[row][col]);
8      printf("\n");
9    }
10 }
```

- A függvény használata

```
1  char a[3][2];
2  ...
3  print_array(a, 3);        /* 3 soros tömb kiírása */
```

Kétdimenziós tömb átvétele egyben

- 2D tömb átvétele mutatóként

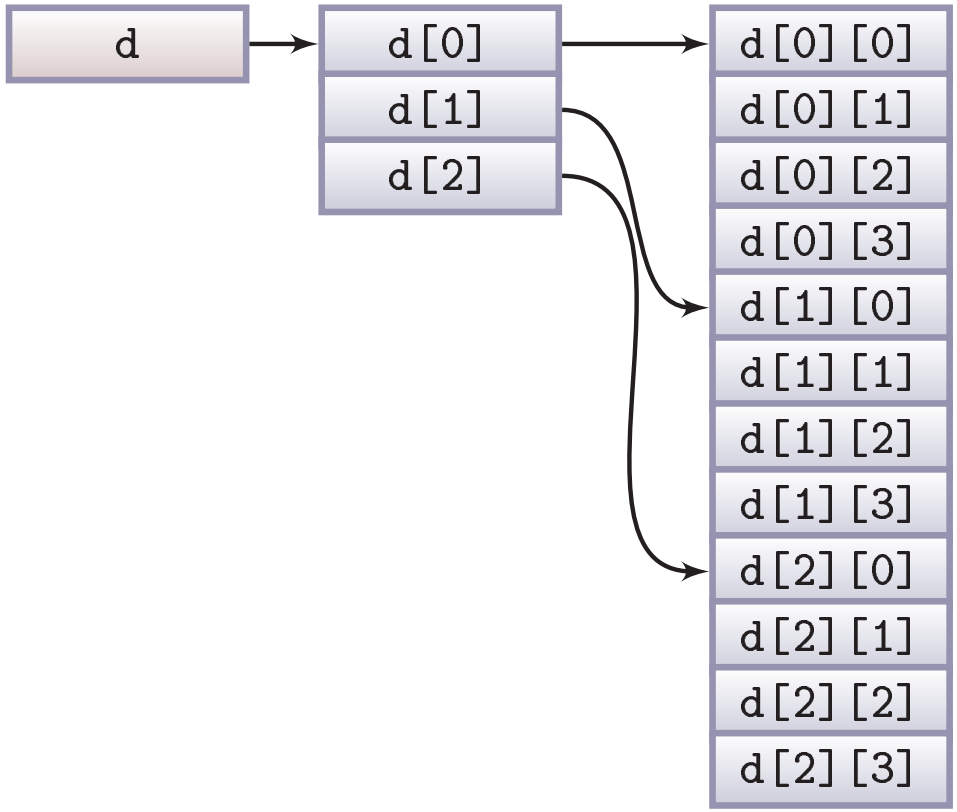
```
1  void print_array(char *array, int nrows, int ncols)
2  {
3    int row, col;
4    for (row = 0; row < nrows; ++row)
5    {
6      for (col = 0; col < ncols; ++col)
7        printf("%c", array[row*ncols+col]);
8      printf("\n");
9    }
10 }
```

- A függvény használata

```
1  char a[3][2];
2  ...
3  print_array((char *)a, 3, 2); /* 3 sor 2 oszlop */
```

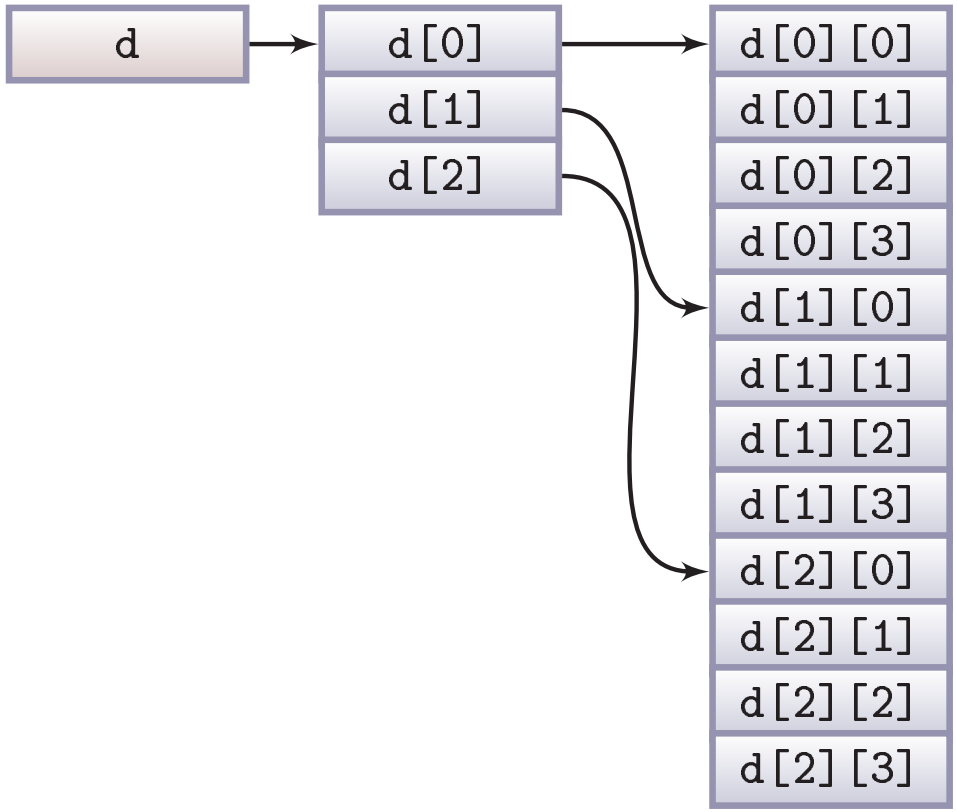
2D dinamikus tömb

Foglaljunk dinamikusan kétdimenziós tömböt, melyet a szokásos módon, `d[i][j]` indexeléssel használhatunk




```
1  double **d =(double**)malloc(3*sizeof(double*));
2  d[0] = (double*)malloc(3*4*sizeof(double));
3  for (i = 1; i < 3; ++i)
4    d[i] = d[i-1] + 4;
```

2D dinamikus tömb



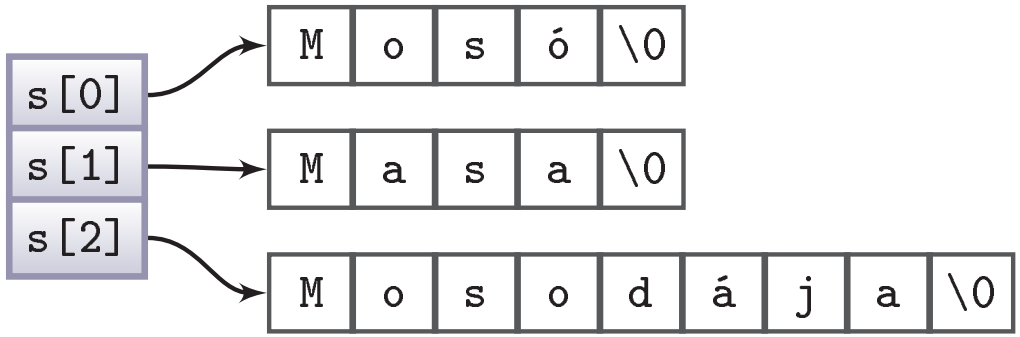
A tömb felszabadítása

```
1  free(d[0]);
2  free(d);
```

Mutatótömb

- Mutatótömb definiálása és átadása függvénynek

```
1  char *s[3] = {"Mosó", "Masa", "Mosodája"};
2  print_strings(s, 3);
```



- Mutatótömb átvétele függvénnyel

```
1  void print_strings(char *strings[], size_t size)
2  /*                  char **strings is lehet          */
3  {
4    size_t i;
5    for (i = 0; i < size; ++i)
6      printf("%s\n", strings[i]);
7  }
```

3. fejezet

Fájlkezelés

Fájlok

Fájl

Fizikai hordozón (merevlemez, CD, USB drive) tárolt adat

- A fájlba kimentett adat nem vész el a program futása után, következő futáskor visszaolvasható
- A különböző hordozókon tárolt fájlokat egységes felületen kezeljük
- Fájlkezelés:
  1. Fájl megnyitása
  2. Adatok írása / olvasása
  3. Fájl bezárása
- Kétféle fájltípus:
  - Szöveges fájl
  - Bináris fájl

Szöveges vs. Bináris

Szöveges fájl – szöveget tartalmaz, sorokra tagolódik

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

Bináris fájl – tetszőleges struktúrájú binárisan kódolt adatot tartalmaz

- `exe`, `wav`, `mp3`, `jpg`, `avi`, `zip`

- Amíg nem túl ésszerűtlen, ragaszkodjunk az emberbarát szöveges tároláshoz.
- Nagy előny, ha adatainkat nemcsak programok, hanem emberek is értik, szerkeszthetik.

Szöveges fájlba írás

```c
#include <stdio.h> /* fopen, fprintf, fclose */
int main(void)
{
  FILE *fp;
  int status;

  fp = fopen("hello.txt", "w");  /* fájlnyitás */
  if (fp == NULL)                 /* nem sikerült */
    return 1;

  fprintf(fp, "Szia, világ!\n");  /* beírás */

  status = fclose(fp);            /* lezárás */
  if (status != 0)
    return 1;

  return 0;
}
```

link

Fájl megnyitása

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

- Megnyitja az `fname` sztringben megadott nevű fájlt a `mode` sztringben megadott módon
- Szöveges fájlokhoz használt fontosabb módok:

mode		leírás
`"r"`	read	olvasásra, a fájlnak léteznie kell
`"w"`	write	írásra, felülír, ha kell, újat hoz létre,
`"a"`	append	írásra, végére ír, ha kell, újat hoz létre

- visszatérési érték mutató egy `FILE` struktúrára, ez a fájlpointer
- Ha a fájlnyitás nem sikeres, nullpointerrel tér vissza

Fájl bezárása

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

- Lezárja az `fp` fájlpointerrel hivatkozott fájlt
- Ha a lezárás sikeres[1], 0 értékkel, egyébként `EOF`-fal tér vissza

[1] fájlzárás lehet sikertelen. Pl. valaki kihúzta a pendrive-ot, miközben írtunk.

stdoutra / szöveges fájlba / sztringbe írás

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

- A `control` sztringben meghatározott szöveget írja a
  - képernyőre
  - `fp` azonosítójú (már írásra megnyitott) szöveges fájlba
  - `str` című (elegendően hosszú) sztringbe
- Visszatérési érték a beírt karakterek száma[2], hiba esetén negatív

[2] Ha sztringbe írunk, automatikusan beírja a lezáró 0-t is, de nem számolja bele a kimenetbe

stdinről / szöveges fájlból / sztringből olvasás

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

- A `control` sztringben meghatározott formátum szerint olvas a
  - billentyűzetről
  - `fp` azonosítójú (már olvasásra megnyitott) szöveges fájlból
  - `str` kezdőcímű sztringből
- Visszatérési érték a kiolvasott elemek száma, hiba esetén negatív

Szöveges fájlból olvasás

Írjunk programot, amely szöveges fájl tartalmát kiírja a képernyőre

```c
#include <stdio.h>
int main()
{
  char c;
  FILE *fp = fopen("fajl.txt", "r"); /* fájlnyitás */
  if (fp == NULL)
    return -1; /* sikertelen volt */

  /* olvasás, amíg sikeres (1 karakter jött) */
  while (fscanf(fp, "%c", &c) == 1)
    printf("%c", c);

  fclose(fp); /* lezárás */
  return 0;
}
```

link

- Jól figyeljük meg, hogyan olvasunk fájl végéig!

Szöveges fájlból olvasás

Egy szöveges fájl kétdimenziós pontok koordinátáit tartalmazza, minden sora az alábbi formátumú

`x:1.2334, y:-23.3`

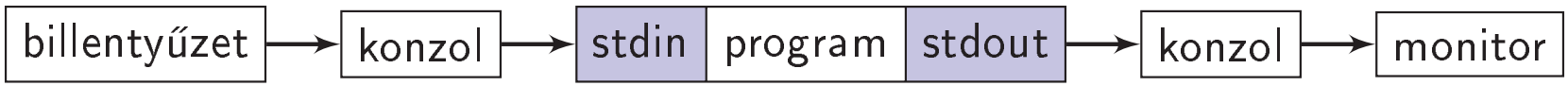
Írjunk programot, mely beolvassa és feldolgozza a koordinátákat!

```
1  FILE *fp;
2  double x, y;
3  ...
4  /* olvasás, amíg sikeres (2 számot olvastunk) */
5  while (fscanf(fp, "x:%lf, y:%lf", &x, &y) == 2)
6  {
7    /* feldolgozás */
8  }
```

- Ismét jól figyeljük meg, hogyan olvasunk fájl végéig!

Billentyűzet? Monitor?

```c
scanf("%c", &c);
printf("%c", c);
```

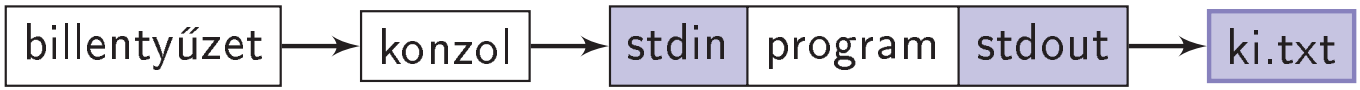


- A fenti kódrészlet nem közvetlenül a billentyűzetről olvas és monitorra ír, hanem a standard inputról (stdin) olvas, és a standard outputra (stdout) ír
- stdin és stdout szöveges fájlok
- Az operációs rendszeren múlik, hogy milyen periféria vagy egyéb fájl van hozzájuk rendelve
- Alapértelmezés az ábra szerint
  - billentyűzet (konzol programon keresztül) → stdin
  - stdout → (konzol programon keresztül) monitor

Átirányítás

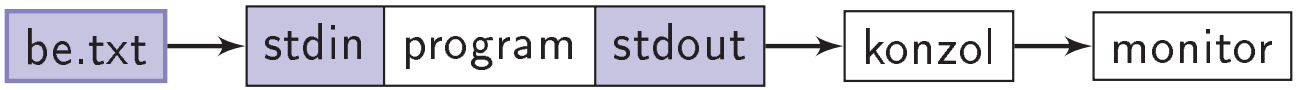
- Ha a programot az alábbi módon indítjuk, a standard output nem a monitorra megy, hanem a `ki.txt` szöveges fájlba

```
c:\>prog.exe > ki.txt
```



- A standard input is átirányítható szöveges fájlra

```
c:\>prog.exe < be.txt
```



- Természetesen együtt is lehet

```
c:\>prog.exe < be.txt > ki.txt
```

stdin és stdout

- Az stdin és stdout szöveges fájlok automatikusan nyitva vannak program indításakor
- az alábbi kódrészletek ekvivalensek

```c
char c;
printf("Hello");
scanf("%c", &c);
printf("%c", c);
```

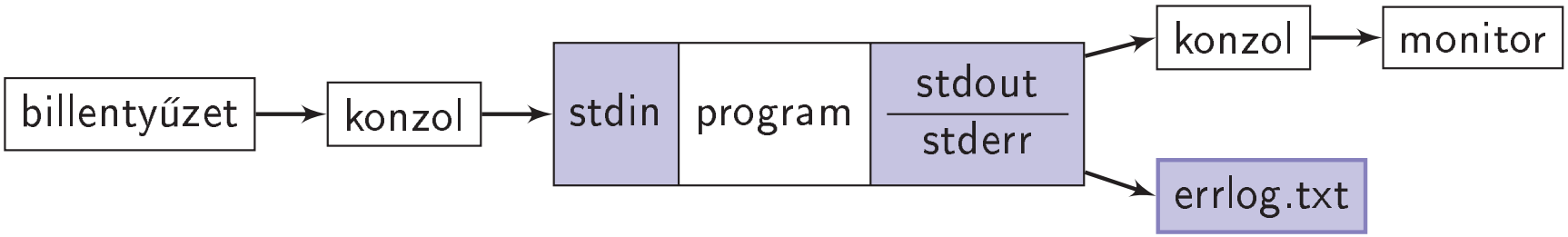
```c
char c;
fprintf(stdout, "Hello");
fscanf(stdin, "%c", &c);
fprintf(stdout, "%c", c);
```

- Ha szöveges fájlból szöveges fájlba dolgozó programot írunk, fájlnyitás helyett használjuk a standard be- és kimenetet és az operációs rendszer átirányítási lehetőségeit
- Konzolról is olvashatunk fájl végéig, amit Ctrl+Z (windows) vagy Ctrl+D (linux) leütésével szimulálhatunk.

stdout és stderr

- A program kimenete és hibaüzenetei is különválaszthatóak a stderr szabványos hibakimenet használatával

```
c:\>prog.exe 2> errlog.txt
```




```c
if (error)
{
  /* felhasználónak, ami rá tartozik */
  printf("Kérem, kapcsolja ki\n");
  /* hibakimenetre részletes információ */
  fprintf(stderr, "61. kódú hiba\n");
}
```

Bináris fájlok

- Bináris fájl: A memória tartalmának bithű másolata egy fizikai hordozón
- A tárolt adat természetesen belsőábrázolás-függő
- Csak akkor használjuk, ha a szöveges tárolás nagyon ésszerűtlen lenne – már a nagy háziban sem kötelező elem 

- Fájlnyitás és fájlzárás a szöveges fájlokhoz hasonlóan, csak a `mode` sztringben szerepelnie kell a `b` karakternek[3]

`mode`		leírás
`"rb"`	read	olvasásra, a fájlnak léteznie kell
`"wb"`	write	írásra, felülír, ha kell, újat hoz létre,
`"ab"`	append	írásra, végére ír, ha kell, újat hoz létre

[3] Az analógia kedvéért szöveges fájloknál bevett szokás a `t` (`text`) szerepeltetése, de ezt az `fopen` figyelmen kívül hagyja

Bináris fájl írása olvasása

```
size_t fwrite (void *ptr, size_t size,
               size_t count, FILE *fp);
```

- A `ptr` címtől `count` számú, egyenként `size` méretű, folytonosan elhelyezkedő elemet ír az `fp` azonosítójú fájlba
- Visszatérési érték a beírt elemek száma

```
size_t fread (void *ptr, size_t size,
              size_t count, FILE *fp);
```

- A `ptr` címre `count` számú, egyenként `size` méretű elemet olvas az `fp` azonosítójú fájlból
- Visszatérési érték a kiolvasott elemek száma

Bináris fájlok – példa

- Az alábbi dog_array tömb 5 kutyát tárol

```
1  typedef enum { BLACK, WHITE, RED } color_t;
2
3  typedef struct {
4    char name[11];    /* név max 10 karakter + lezárás */
5    color_t color;    /* szín */
6    int nLegs;        /* lábak száma */
7    double height;    /* magasság */
8  } dog;
9
10 dog dog_array[] = /* 5 kutya tömbje */
11 {
12   { "blöki",  RED,   4, 1.12 },
13   { "cézár",  BLACK, 3, 1.24 },
14   { "buksi",  WHITE, 4, 0.23 },
15   { "spider", WHITE, 8, 0.45 },
16   { "mici",   BLACK, 4, 0.456 }
17 };
```

link

Bináris fájlok – példa

- A dog_array tömb kiírása bináris fájlba enyire egyszerű!

```
1  fp = fopen("dogs.dat", "wb"); /* hibakezelés!!! */
2  if (fwrite(dog_array, sizeof(dog), 5, fp) != 5)
3  {
4    /* hibajelzés */
5  }
6  fclose(fp); /* ide is!!! */
```

- A dog_array tömb visszaolvasása sem bonyolultabb

```
1  dog dogs[5]; /* tárhely foglalás */
2  fp = fopen("dogs.dat", "rb");
3  if (fread(dogs, sizeof(dog), 5, fp) != 5)
4  {
5    /* hibajelzés */
6  }
7  fclose(fp);
```

Bináris fájlok – példa

- Álljunk ellen a csábításnak!
- Ha egy másik gépen a dog struktúra bármely tagjának ábrázolása eltérő, a kimentett adatokat ott nem tudjuk visszaolvasni
- Az átgondolatlanul kimentett bináris fájlok a programot hordozhatatlanná teszik
- Az átgondolt kimentés természetesen jóval bonyolultabb
  1. Megállapodunk az ábrázolásban
     - melyik bit az LSB?
     - kettes komplemens?
     - hány bites a mantissza?
     - struktúra elemei szóhatárra illesztettek? És az mekkora?
     - stb
  2. Az adatokat konvertáljuk, majd kiírjuk

Bináris vs szöveges

- Csináljuk inkább szövegesen, mindenki jobban jár!
- A dog_array tömb kiírása szöveges fájlba

```
1  for (i = 0; i < 5; ++i) {
2    dog d = dog_array[i];
3    fprintf(fp, "%s,%u,%d,%f\n",
4      d.name, d.color, d.nLegs, d.height);
5  }
```

- A dog_array tömb beolvasása szöveges fájlból[4]

```
1  dog dogs[5]; /* tárhely foglalás */
2  for (i = 0; i < 5; ++i) {
3    dog d;
4    fscanf(fp, "%s,%u,%d,%lf",
5      d.name, &d.color, &d.nLegs, &d.height);
6    dogs[i] = d;
7  }
```

[4] feltételezzük, hogy a kutya neve nem tartalmaz whitespace karaktert

Státuszjelző függvények

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

- igaz, ha elértük a fájl végét, hamis egyébként

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

- igaz, ha hiba történt íráskor-olvasáskor, hamis egyébként

- A legtöbbször nincs szükség rájuk, használjuk az író-olvasó függvények visszatérési értékét!

Státuszjelző függvények

- A tipikus hiba

```
1  while (!feof(fp))
2  {
3    /* elem beolvasása */
4
5    /* elem feldolgozása */
6  }
```

elem	elem	elem	EOF

- `feof()` akkor lesz igaz, ha már beolvastuk a fájl vége jelet.
- Mit is tanultunk a végjeles vektorról?

```
1  /* elem beolvasása */
2  while (!feof(fp))
3  {
4    /* elem feldolgozása */
5    /* elem beolvasása */
6  }
```

Köszönöm a figyelmet.