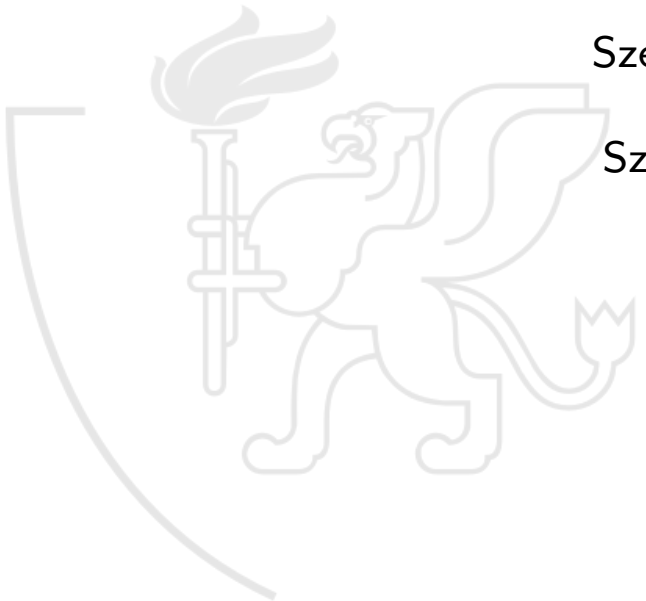
Programozás Alapjai

Dr. Gergely Tamás
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék



2023

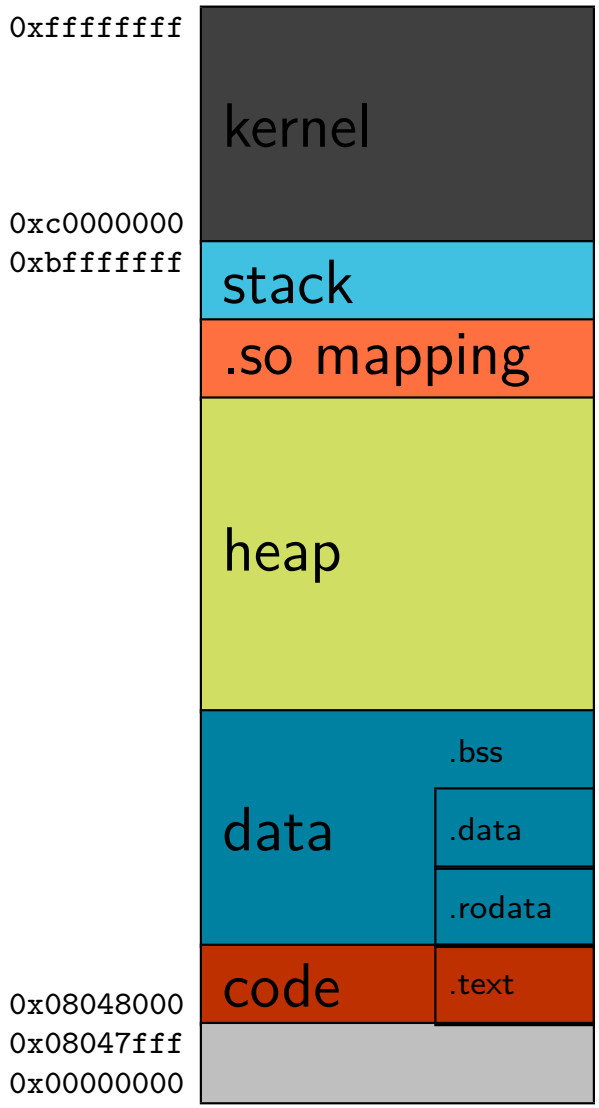
(v0911)

Tartalom

Tartalom

A memóriába töltött program részei

- Programkód
  - Saját kód, közös kód
  - Speciális (rendszer-, sőt akár hardverszintű) védelem
- Adat
  - Literálok, konstans értékek (és nem a konstans változók) tárolására csak olvasható terület
  - Globális változók területe
- Verem
  - Ideiglenes változók, technikai adatok
- Szabad memória
  - Szabadon foglalható memória



Konstansok és konstansok

Sztring tömb és pointer [1/3]

- Mi lesz az alábbi program kimenete `tomb` és `pointer` argumentummal?

```c
/* A char* és char[] típusok különbsége inicializálás esetén.
 *   A program "pointer" paraméterrel történő indítása futási hibát okoz.
 * 2006. Augusztus 17. Gergely Tamás, gertom@inf.u-szeged.hu
 */

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

int main(int argc, char *argv[]) {
    char t[] = "Tömb vagy pointer";
    char  *p = "Tömb vagy pointer";
    if (argc == 2) {
        if (!strcmp(argv[1], "tomb")) {
            strcpy(t, "Hello!");
        } else if (!strcmp(argv[1], "pointer")) {
            strcpy(p, "Hello!");
        }
        printf("p: \"%s\"\n", p);
        printf("t: \"%s\"\n", t);
    } else {
        printf("Használat:\n\t%s tomb\n\t%s pointer\n", argv[0], argv[0]);
    }
    return 0;
}
```

Konstans értékek

Sztring tömb és pointer [2/3]

- Mi a különbség a két deklaráció között?
  - `char  t[] = "Tomb vagy pointer";`
  - `char *p   = "Tomb vagy pointer";`
- A `"Tomb vagy pointer"` mindkét esetben egy konstans karaktertömb, ami a program konstans szekciójában, a csak olvasható adatok között kap helyet.
- A `t` egy tömb. Amikor létrejön, a veremben vagy a globális változóknak fenntartott memóriaterületen lefoglalódik a megfelelő méretű memóriaterület, és a sztring értéke a csak olvasható területről bemásolódik erre a területre.
- A `p` egy pointer. Amikor létrejön, a veremben vagy a globális változóknak fenntartott memóriaterületen lefoglalódik egy pointer számára megfelelő méretű memóriaterület, és ezen a területen eltárolódik a csak olvasható területen lévő sztring címe.

Konstans értékek

Sztring tömb és pointer [3/3]

- `strcpy(t, "Hello!");`
  - A `t` tömb területére bemásolódik a `"Hello!"` szöveg, azaz `t` felveszi ezt az értéket.
- `strcpy(p, "Hello!");`
  - A `p` által mutatott területre bemásolódna a `"Hello!"` szöveg. De ez a terület csak olvasható, így *futási hibát* kapunk. Azt, hogy egy memóriaterület csak olvasható az operációs rendszer tartja számon (hardveres segítséggel).

Konstansok és konstansok

const [1/2]

- Mi lesz az alábbi program kimenete?

```c
/* Egy const értékének megváltoztatása.
 * 2006. Augusztus 17. Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdio.h>

int main() {
    const int c = 100;
    int *p;
    p = (int*)&c;
    *p *= 2;
    printf("%d\n", c);
    return 0;
}
```

Konstansok és konstansok

`const` [2/2]

- Bármilyen meglepőnek tűnik is, a program nem okoz futási hibát, és a kimenet: 200
- A c tehát nem egy, a fordító számára szóló érték lesz, és nem is a csak olvasható memóriaterületen eltárolt változó, hanem egy valódi változó, egy kisebb megszorítással: nem szabad megváltoztatni az értékét. Vagyis a fordítás során bármely direkt c=... alakú kifejezés hibát eredményez, de a memóriaterület átírásával a c értéke mégis megváltoztatható.
- Valódi konstansokat a C előfeldolgozóval (`#define`) készíthetünk.

Konstansok és optimalizálás

- Az előző programot többféle fordítóval és optimalizálással fordítva már nem egyértelmű a végeredmény:

```
$ gcc-5 -O0 -o const const.c && ./const
200
$ gcc-5 -O2 -o const const.c && ./const
200
$ gcc-4.4 -O0 -o const const.c && ./const
200
$ gcc-4.4 -O2 -o const const.c && ./const
100
$ clang -O0 -o const const.c && ./const
100
$ clang -O2 -o const const.c && ./const
100
```

- A fordító ugyanis adott esetben azt látja, hogy a c egy konstans változó, tehát a `printf` hívásánál nem vesződik a változó kiolvasásával, hanem az inicializáló értéket fogja fixen átadni, mert az szerinte úgysem változik.

Változók és optimalizálás

- Az előző eset helyessége még véleményes lehet, ugyanis a programozó „mondta”, hogy az érték konstans, a fordító csak elhitte neki. (Öreg hiba! Bemondásra nem hiszünk el semmit!)
- Van viszont, amikor a fordító úgy ítéli meg, hogy valamilyen változó értéke nem változik, de téved, mert a környezete más szabályok szerint játszik.
- Helyes-e például, az alábbi fordítás?

```
1     b = a + 5;
2     c = a + 8;
```

```
1     mov $a, r1
2     mov $5, r2
3     add r1, r2, r3
4     mov r3, $b
5     mov $8, r2
6     add r1, r2, r3
7     mov r3, $c
```

- Nos, ha senki nem piszkál bele kívülről a memóriába (amit azért feltételezni szoktunk), akkor igen. De ha mégis, és ez nem hiba, hanem „feature”?

Változók és optimalizálás

- Az előző, és hasonló optimalizálások jelentősen tudják csökkenteni a kód méretét és a futásidőt is, de van, amikor kifejezetten károsak.
- Bizonyos rendszerekben például a párhuzamosan futó programok néha közös memórián keresztül kommunikálnak.
- Ilyen esetekben az adott változókat a `volatile` kulcsszóval kell deklarálni. Ez figyelmezteti a fordítót, hogy a változó értéke a program futásától függetlenül is megváltozhat, tehát ne hagyatkozzon a regiszterekben korábban eltárolt értékekre.

A `volatile` kulcsszó hiánya [1/1]

non-volatile.c [1–26]

```c
/* A volatile kulcsszó hatásának bemutatása
 * 2016. December 13.  Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *szal(void *arg) {
    while (scanf("%d", (int*)arg) == 1);
    *((int*)arg) = 0;
    printf("Szál befejezése.\n");
    return NULL;
}

int main() {
    pthread_t kiiro_szal;
    int valtozo = 1;
    printf("A program akkor fejeződik be, ha egy 0 értékű input adat után egy\n"
           "egészként nem értelmezhető adatot adunk neki.\n");
    pthread_create(&kiiro_szal, NULL, szal, (void*)&valtozo);
    while (valtozo);
    printf("Várunk a szál befejezésére.\n");
    pthread_join(kiiro_szal, NULL);
    return 0;
}
```

A volatile kulcsszó hatása [1/1]

volatile.c [1–26]

```c
/* A volatile kulcsszó hatásának bemutatása
 * 2016. December 13.  Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *szal(void *arg) {
    while (scanf("%d", (int*)arg) == 1);
    *((int*)arg) = 0;
    printf("Szál befejezése.\n");
    return NULL;
}

int main() {
    pthread_t kiiro_szal;
    volatile int valtozo = 1;
    printf("A program akkor fejeződik be, ha egy 0 értékű input adat után egy\n"
           "egészként nem értelmezhető adatot adunk neki.\n");
    pthread_create(&kiiro_szal, NULL, szal, (void*)&valtozo);
    while (valtozo);
    printf("Várunk a szál befejezésére.\n");
    pthread_join(kiiro_szal, NULL);
    return 0;
}
```

Változók és optimalizálás

- Az előző programokat futtatva látszik a különbség:

```
$ ./non-volatile
A program akkor fejeződik be, ha egy 0 értékű input adat után egy
egészként nem értelmezhető adatot adunk neki.
1
0
-1
x
Szál befejezése.
-2
^C
$ ./volatile
A program akkor fejeződik be, ha egy 0 értékű input adat után egy
egészként nem értelmezhető adatot adunk neki.
1
0
Várunk a szál befejezésére.
-1
x
Szál befejezése.
$
```

A `non-volatile.c` verzióban a `main` ciklusának változója csak egyszer kerül kiolvasásra (mert a ciklusban nem írjuk át), de akkor még 1 az értéke, és készen is van a végtelen ciklus.

Bináris adatállományok

- Világos, hogy ha egy karakterfolyamot ASCII kódrendszerben elkészítünk és ezt egy másik, szintén ASCII kódrendszerben dolgozó gépen beolvassuk, akkor ugyanúgy tudjuk értelmezni az adatállományt. (pl. UNIX, Linux, Mac, PC)
- Mi a helyzet bináris adatállományok esetén?
- Ennek megvizsgálásához készítsünk egy kis programot, amelyik 10 egész számot ír ki egy adatállományba és nézzük meg az eredményt több különböző rendszeren.

Memóriaterület kiírása binárisan [1/2]

keszit.c [1–15]

```c
/* Egy bináris adatállományba egész számok kiírása.
 * Az adatállomány nevét a parancssorból kapjuk.
 * 1998. Április 24.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#define DBSZAM 10

long int tomb[DBSZAM];                          /* innen fogunk irni */
int i;                                           /* ciklusváltozó */
```

Memóriaterület kiírása binárisan [2/2]

keszit.c [17–40]

```
17 int main(int argc, char **argv) {
18     int fd;                                  /* fájlleíró = file descriptor */
19     int bytedb;                                      /* kiírt bájtok száma */
20
21     if ((fd = open(argv[1], O_WRONLY)) == -1) {
22         errno = ENOENT;
23         perror("Megnyitási hiba");
24         exit(EXIT_FAILURE);
25     }
26                                /* a tomb feltöltése jól látható értékekkel */
27     for (i = 0; i < DBSZAM; ++i) {
28         tomb[i] = i + 'a';
29     }
30
31     if ((bytedb = write(fd, tomb, DBSZAM * sizeof(long int))) == -1) {
32         perror("Írási hiba");
33         exit(EXIT_FAILURE);
34     }
35
36     close(fd);
37
38     printf("Sikerült kiírni %d darab bájtot\n", bytedb);
39     return 0;
40 }
```

Bináris adatállományok

- 64 bites architektúrájú PC linux:

```
$ touch proba.bin && ./keszit proba.bin
Sikerült kiírni 80 darab bájtot
$ hd proba.bin
00000000  61 00 00 00 00 00 00 00  62 00 00 00 00 00 00 00  |a.......b.......|
00000010  63 00 00 00 00 00 00 00  64 00 00 00 00 00 00 00  |c.......d.......|
00000020  65 00 00 00 00 00 00 00  66 00 00 00 00 00 00 00  |e.......f.......|
00000030  67 00 00 00 00 00 00 00  68 00 00 00 00 00 00 00  |g.......h.......|
00000040  69 00 00 00 00 00 00 00  6a 00 00 00 00 00 00 00  |i.......j.......|
```

- 32 bites architektúrájú PC linux:

```
$ touch proba.bin && ./keszit proba.bin
Sikerült kiírni 40 darab bájtot
$ hd proba.bin
00000000  61 00 00 00 62 00 00 00  63 00 00 00 64 00 00 00  |a...b...c...d...|
00000010  65 00 00 00 66 00 00 00  67 00 00 00 68 00 00 00  |e...f...g...h...|
00000020  69 00 00 00 6a 00 00 00                           |i...j...|
```

- Régi `home.inf.u-szeged.hu` szerver (már nem él):

```
$ touch proba.bin && ./keszit proba.bin
Sikerült kiírni 40 darab bájtot
$ hexdump proba.bin
00000000  00 00 00 61 00 00 00 62  00 00 00 63 00 00 00 64  |...a...b...c...d|
00000010  00 00 00 65 00 00 00 66  00 00 00 67 00 00 00 68  |...e...f...g...h|
00000020  00 00 00 69 00 00 00 6a                           |...i...j|
```

Bináris adatállományok

- Láthatjuk, hogy a három adatállomány különböző.
  - A 32 és 64 bites architektúra közötti különbség abból adódik, hogy a `long int` adattípus nem azonos méretű a két rendszerben. A `long int` 32 bites architektúrán 4 bájt, 64 bites architektúrán 8 bájt. Ezen az eltérésen bizonyos esetekben könnyen lehet segíteni a pontosabb típusdefinícióval.
  - A `home` és `linux` közötti különbség pedig abból adódik, hogy a bájtsorrend különbözik az egyes architektúrákon (`x86-64` és `SUN Sparc`).
- Bájtsorrend alapján kétféle architektúrát különböztetünk meg:
  - Ha az alacsonyabb című memóriahelyekre az alacsonyabb helyiértékű bájtokat tároljuk (LSB – Least Significant Byte), akkor a bájtsorrend little-endian („kis indián", pl. a PC típusú gépek).
  - Ha az alacsonyabb című memóriahelyekre a magasabb helyiértékű bájtokat tároljuk (MSB – Most Significant Byte), akkor a bájtsorrend big-endian („nagy indián", pl. a `home` szerver).

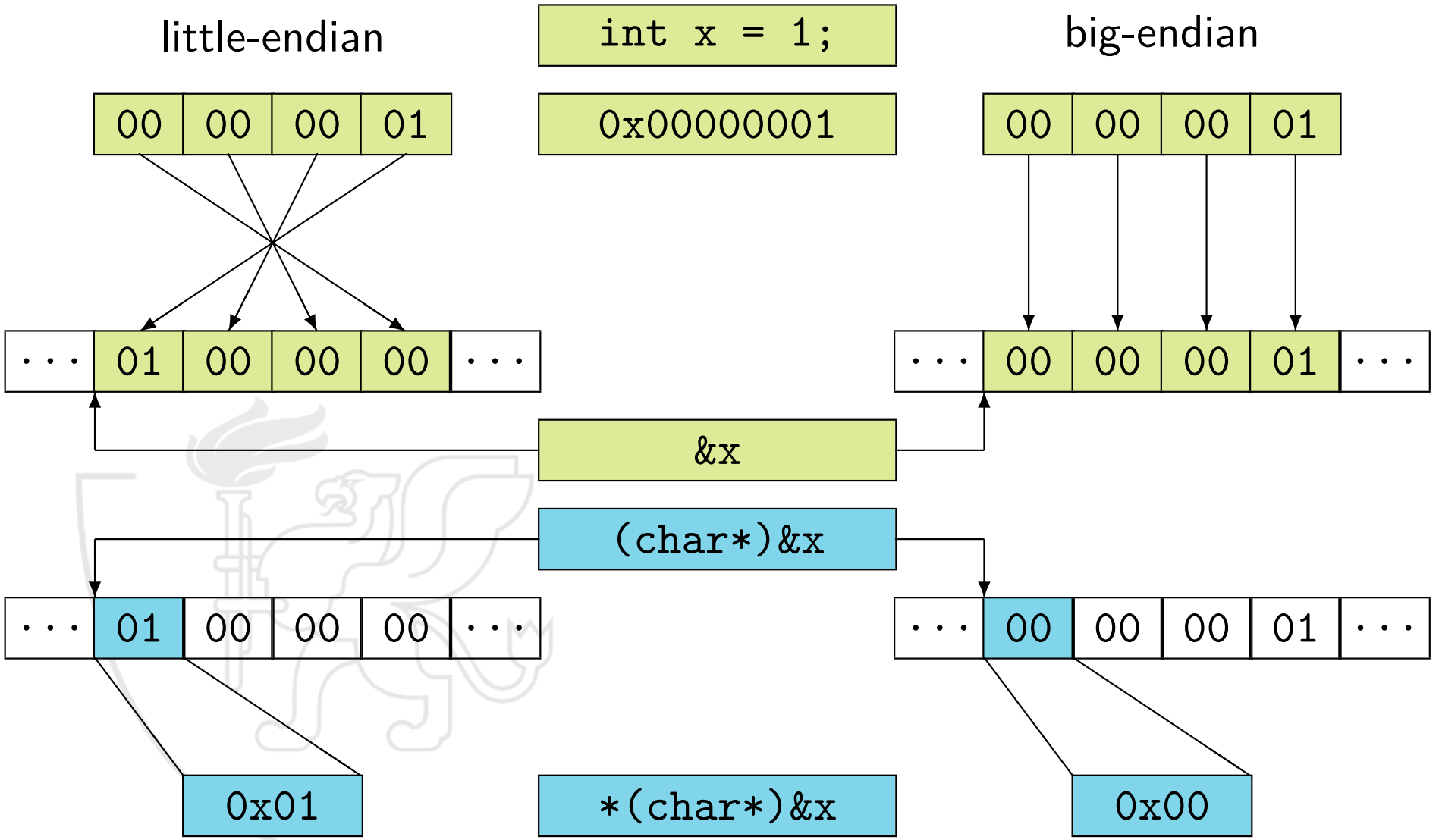
Architektúra bájtsorrendjének meghatározása [1/1]

endian.c [1–16]

```c
/* Eldönti, hogy milyen a bájtsorrend a számítógépen.
 * 1998. Április 24.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

int main() {
    int x = 1;

    if (*(char *)&x == 1) {
        printf("little-endian\n");
    } else {
        printf("big-endian\n");
    }
    return 0;
}
```

Architektúra bájtsorrendjének meghatározása

Az `endian.c` működése



Tartalom

Gyakori C hibák

- Nagyon sok „triviális” hibát követünk el programozás során, melynek számos oka lehet:
  - Egyszerű félregépelés
  - Programozási rutin, tapasztalat hiánya
  - Nyelv nem megfelelő ismerete
  - Módosítandó kód nem megfelelő ismerete
  - Kommunikációból eredő félreértések
  - Szélsőséges tesztesetekre nem gondolunk

Általános hibaforrások

A kód tabulálása

- Sok tapasztalatlan programozó nem tulajdonít kellő jelentősséget a tabulálásnak, mert a C nyelv nem érzékeny rá. **De az ember igen!**
- Itt például spórolhatnánk egy ciklust (`csuszoatlag.c [19-23]`):

```
19      for (i = 0; i < N-2; ++i)
20          atlagtomb[i] = atlag3(ertektomb[i], ertektomb[i + 1], ertektomb[i + 2]);
21      for (i = 0; i < N-2; ++i)
22          printf("%lf;", atlagtomb[i]);
23      putchar('\n');
```

De nem így:

```
19      for (i = 0; i < N-2; ++i)
20          atlagtomb[i] = atlag3(ertektomb[i], ertektomb[i + 1], ertektomb[i + 2]);
21          printf("%lf;", atlagtomb[i]);
22      putchar('\n');
```

mert bár vannak programozási nyelvek, ahol ezt így KELL csinálni, C-ben **nem**, és ez megfelelő tabulálással azonnal látszik:

```
19      for (i = 0; i < N-2; ++i)
20          atlagtomb[i] = atlag3(ertektomb[i], ertektomb[i + 1], ertektomb[i + 2]);
21      printf("%lf;", atlagtomb[i]);
22      putchar('\n');
```

Általános hibaforrások

Tabulátor vagy szóköz?

- Alapvetően mindegy (bár a szóköz használata talán egységesebb képet mutat mert nem függ a TAB beállításától), de **ne keverjük**!

Egységes szóköz(4) vagy TAB(4)

```
1     for (i = 0; path[i] != 0; ++i)
2         if (path[i] == '/')
3             lastsep = i;
4     ++lastsep;
5     i = 0;
6     while (path[lastsep] != 0)
7         base[i++] = path[lastsep++];
8     base[i] = 0;
```

Vegyes szóköz(4) és TAB(2)

```
1     for (i = 0; path[i] != 0; ++i)
2     if (path[i] == '/')
3             lastsep = i;
4   ++lastsep;
5     i = 0;
6   while (path[lastsep] != 0)
7         base[i++] = path[lastsep++];
8   base[i] = 0;
```

Vegyes szóköz(4) és TAB(8)

```
1     for (i = 0; path[i] != 0; ++i)
2                 if (path[i] == '/')
3             lastsep = i;
4         ++lastsep;
5     i = 0;
6         while (path[lastsep] != 0)
7         base[i++] = path[lastsep++];
8         base[i] = 0;
```

Vegyes szóköz(2) és TAB(8)

```
1   for (i = 0; path[i] != 0; ++i)
2                 if (path[i] == '/')
3       lastsep = i;
4         ++lastsep;
5   i = 0;
6         while (path[lastsep] != 0)
7     base[i++] = path[lastsep++];
8         base[i] = 0;
```

Gyakori C hibák

Szintaktika

- A C érzékeny a kis- és nagybetűkre.
- A ; lezár és nem elválaszt.
  - Minden utasítást le kell zárni, egy blokk utolsó utasítását is.
  - Az utasításblokkot ugyanakkor nem kell lezárni (a blokkzáró } után nincs ;).
  - A típus deklarációt/definíciót ; követi akkor is, ha az }-re végződik.

```
struct valami {
    /* ... */
};
```

Gyakori C hibák

Üres utasítások

- A ; egy üres kifejezésből is (üres) utasítást csinál.

```
if (F); M;
while (F); M;
for (I; F; N); M;
```

```
if (F); { M; }
while (F); { M; }
for (I; F; N); { M; }
```

- A fentiek mind üres ciklusmagok, a vezérlési szerkezethez ugyanis a ;-vel lezárt üres utasítás tartozik.

Gyakori C hibák

Értékadás, reláció

- Az = az értékadás, a == az összehasonlítás műveletének a jele.
- Mivel az értékadás is egy művelet, és az egész értékek logikai értékként értelmezhetőek, az

```
if (x = 1) { }
```

  is érvényes konstrukció, ami viszont mindig igaz és x-nek új értéket ad.
- Ezért a konstanssal való összehasonlítást sokan

```
if (1 == x) { }
```

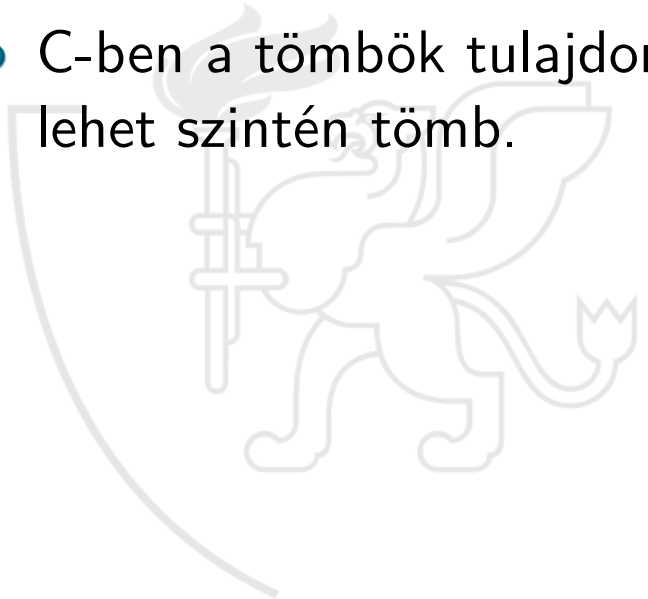
  alakban írják, mert elgépelésnél (== helyett =) így fordítási hibát kapunk.
- Ugyanakkor néha kifejezetten az értékadást szeretnénk használni az `if`-ben, ilyenkor érdemes az értékadást külön zárójelbe tenni:

```
if ( (f = fopen("data.txt", "rt")) ) { }
```

Gyakori C hibák

Tömbök

- A tömb indexelése 0-nál kezdődik, így az n elemű tömb index tartománya $[0 \ldots n-1]$.
- Nincs indexhatár-ellenőrzés, a tömb méretétől függetlenül akárhanyadik elemére hivatkozhatok.
- Ha tömböt adok át paraméterként, annak elemeit a függvény megváltoztathatja!
- C-ben a tömbök tulajdonképpen egydimenziósak, de elemtípusuk lehet szintén tömb.



Gyakori C hibák

Pointerek

- A pointer deklarálásánál a *-ot érdemes szorosan a változóhoz írni. Ha nem ezt tesszük, akkor pl.

```
int* ptr1, ptr2;
```

  esetén nem vesszük észre, hogy `ptr2` nem pointer, hanem csak `int`.
- Minden „kézzel” lefoglalt területet fel kell szabadítani! Bár a program bezárásával az operációs rendszer a programhoz tartozó minden memóriát felszabadít, ne szokjuk meg, hogy erre hagyatkozunk.
- Ha a program „véletlenszerű” hibákat produkál, akkor nagyon valószínű, hogy pointert rontottunk el.

Gyakori C hibák

Függvények

- A paraméterátadás érték szerinti, vagyis bemenő módú. Ha nem ezt akarjuk, akkor az aktuális paraméterek címét kell átadni/átvenni.
- Ha tömböt adok át paraméterként az már (a tömb értékét tekintve) cím szerinti paraméterátadás!
- Függvénynek deklarációjánál akkor is ki kell tenni a `()`-t, ha a függvénynek nincs paramétere.
  - Az ANSI szabvány szerint az `f` deklarációja ilyenkor `f(void)`.
- A függvényhívás paraméter nélküli függvény esetén is a `()` segítségével történik. A zárójelpár nélküli függvénynév a függvényre mutató pointert jelöli.

Gyakori C hibák
IO

- A `printf()` értékeket vár, a `scanf()` címeket.
- A `printf()` és `scanf()` esetén nagyon fontos a konverziós specifikáció és a paramétertípusok megfelelősége.
  - A `scanf()`-nél különösen fontos a méretmódosítók megadása.
- A `getchar()` visszaadott értéke `int`, amit *nem szabad* átkonvertálni `char`-ra mielőtt az `EOF`-fal összehasonlítanánk.
- A C programban megadott fájl útvonal neveket a shell már nem dolgozza fel, így a `"~/data"` vagy `"$HOME/data"` mint fájlnév megadás hibás.
  - Környezeti változók értékeit a `char *getenv(const char *name);` (`stdlib.h`) függvény segítségével érhetjük el (pl. `getenv("HOME")`).
- Bináris fájlműveleteknél fontos az „endianizmus”, vagyis hogy honnan származik a bináris adatállomány.
  - Az endianizmus nagyon alacsony szintű memóriabeli adatkezelésnél szintén fontos.

Gyakori C hibák

Vegyes

- Ha egy típus elmarad, akkor a feltételezett típus `int` és nem `void`.
- Az `int` típus egyes variánsai nem teljesen kompatibilisek.
  - Alul- és túlcsordulás lehet pl. szűkebb értékkészletű vagy előjelkezelés tekintetében eltérő változóba íráskor.
- Nem szabad összetéveszteni a logikai és bináris, vagyis a `&&` és `&`, illetve a `||` és `|` műveletet.
- A lebegőpontos értékekkel vigyázzunk!

```
float  f = 0.67;
double d = 0.67;
printf((f == d) ? "yes\n" : "no\n");
```

- Bár a warning-ok nem akadályozzák meg a program fordítását, sok esetben komoly problémát okozhat a warning-ot kiváltó kódrészlet.

Tartalom

Hol vagyunk? [1/1]

where.c [1–12]

```
1            main(l
2        ,a,n,d)char**a;{
3      for(d=atoi(a[1])/10*80-
4     atoi(a[2])/5-596;n="@NKA\
5    CLCCGZAAQBEAADAFaISADJABBA^\
6    SNLGAQABDAXIMBAACTBATAHDBAN\
7    ZcEMMCCCCAAhEIJFAEAAABAfHJE\
8    TBdFLDAANEfDNBPHdBcBBBEA_AL\
9    ␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
10      [l++-3];)for(;n--->64;)
11         putchar(!d+++33^
12              l&1);}
```

Hol vagyunk?

A program értelmezése – 0.

```
            main(l
      ,a,n,d)char**a;{
  for(d=atoi(a[1])/10*80-
 atoi(a[2])/5-596;n="@NKA\
CLCCGZAAQBEAADAFaISADJABBA^\
SNLGAQABDAXIMBAACTBATAHDBAN\
ZcEMMCCCCAAhEIJFAEAAABAfHJE\
TBdFLDAANEfDNBPHdBcBBBEA_AL\
␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
   [l++-3];)for(;n-->64;)
      putchar(!d+++33^
           l&1);}
```

- Az eredeti programot úgy alakítjuk át, hogy az algoritmus, amit leír, ne változzon.

Hol vagyunk?

A program értelmezése – 1.

```
main(l,a,n,d)
  char**a;
{
  for(d=atoi(a[1])/10*80-atoi(a[2])/5-596;
      n="@NKA\
CLCCGZAAQBEAADAFaISADJABBA^\
SNLGAQABDAXIMBAACTBATAHDBAN\
ZcEMMCCCCAAhEIJFAEAAABAfHJE\
TBdFLDAANEfDNBPHdBcBBBEA_AL\
␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
      [l++-3];)
    for(;n-->64;)
      putchar(!d+++33^l&1);
}
```

- Kezdjük egyszerű tördeléssel, mindössze szóközöket és sorvégeket helyezünk át a programban. Így már látszik, hogy régi típusú C függvényleírást használtunk.

Hol vagyunk?

A program értelmezése – 2.

```
int main(l,a,n,d)
  int l; char**a; int n; int d;
{
  for(d=atoi(a[1])/10*80-atoi(a[2])/5-596;
      n="@NKA\
CLCCGZAAQBEAADAFaISADJABBA^\
SNLGAQABDAXIMBAACTBATAHDBAN\
ZcEMMCCCCAAhEIJFAEAAABAfHJE\
TBdFLDAANEfDNBPHdBcBBBEA_AL\
␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
      [l++-3];)
    for(;n--->64;)
      putchar(!d+++33^l&1);
}
```

- A jelzett kiegészítésekkel a program működése nem változik, hiszen a C nyelv alapértelmezett típusa az `int` típus, azaz ha valahonnan elhagyjuk a típust, oda a fordító `int`-et képzel.

Hol vagyunk?

A program értelmezése – 3.

```
const char *str="@NKACLCCGZAAQBEAADAFaISADJABBA^SNLGAQABDAXIMBAACTBATAHDBANZcE\
MMCCCCAAhEIJFAEAAABAfHJETBdFLDAANEfDNBPHdBcBBBEA_AL␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
int main(l,a,n,d)
  int l; char**a; int n; int d;
{
  for(d=atoi(a[1])/10*80-atoi(a[2])/5-596;
      n=str[l++-3];)
    for(;n-->64;)
      putchar(!d+++33^l&1);
}
```

- Most a programban megadott hosszú sztringet kiemeljük egy külön globális változóba, hogy ne zavarjon a program megértésében.
- Egy keveset elvégzünk a preprocesszor munkájából is, hogy a sztringünket kicsit összébbhúzzuk. Az értéke ezzel az átalakítással nem változik.

Hol vagyunk?

A program értelmezése – 4.

```
const char *str="@NKACLCCGZAAQBEAADAFaISADJABBA^SNLGAQABDAXIMBAACTBATAHDBANZcE\
MMCCCCAAhEIJFAEAAABAfHJETBdFLDAANEfDNBPHdBcBBBEA_AL␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
int main(l,a,n,d)
  int l; char**a; int n; int d;
{
  for(d=(atoi(a[1])/10*80)-(atoi(a[2])/5)-(7*80+36);
      n=str[l++-3];) {
    for(;n-->64;) {
      putchar(!d+++33^l&1);
    }
  }
}
```

- Az átláthatóság kedvéért kiteszünk pár zárójelet, és láthatóvá tesszük a 80 karakter széles terminál oszlopszámát a kódban.

Hol vagyunk?

A program értelmezése – 5.

```
const char *str="@NKACLCCGZAAQBEAADAFaISADJABBA^SNLGAQABDAXIMBAACTBATAHDBANZcE\
MMCCCCAAhEIJFAEAAABAfHJETBdFLDAANEfDNBPHdBcBBBEA_AL␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
int main(l,a,n,d)
  int l; char**a; int n; int d;
{
  int sor=80, px, py;
  py=(atoi(a[1])/10)-7;
  px=(atoi(a[2])/5)+36;
  for(d=py*sor-px;
      n=str[l++-3];) {
    for(;n-->64;) {
      putchar(!d+++33^l&1);
    }
  }
}
```

- A külső ciklus inicializálását egy kicsit átalakítjuk, hogy érthetőbb legyen.
- Látszik, hogy d-be a paraméterben kapott kétdimenziós koordináta 80 oszlopot feltételező sorfolytonossá alakított értéke kerül, méghozzá úgy, hogy az origót a 7. sor 36. oszlopába toltuk el. (A bal felső sarok az $(\mathrm{N}70°, \mathrm{W}180°)$ koordinátáknak felel meg.)

Hol vagyunk?

A program értelmezése – 6.

```
const char *str="@NKACLCCGZAAQBEAADAFaISADJABBA^SNLGAQABDAXIMBAACTBATAHDBANZcE\
MMCCCCAAhEIJFAEAAABAfHJETBdFLDAANEfDNBPHdBcBBBEA_AL␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
int main(l,a,n,d)
  int l; char**a; int n; int d;
{
  int sor=80, px, py;
  py=(atoi(a[1])/10)-7;
  px=(atoi(a[2])/5)+36;
  for(d=py*sor-px;
      n=str[(l++)-3];) {
    for(;n-->64;) {
      putchar(!d+++33^l&1);
    }
  }
}
```

- Most nézzük a sztring indexelését. Hozzávéve azt, hogy a programot 2 paraméterrel történő futtatásra tervezték, azaz az `argc` szerepét betöltő `1` értéke 3-ról indul, a ciklus egyszerűen végiglépked a sztring karakterein, egészen a 0 kódú karakterig.

Hol vagyunk?

A program értelmezése – 7.

```
const char *str="@NKACLCCGZAAQBEAADAFaISADJABBA^SNLGAQABDAXIMBAACTBATAHDBANZcE\
MMCCCCAAhEIJFAEAAABAfHJETBdFLDAANEfDNBPHdBcBBBEA_AL␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
int main(l,a,n,d)
  int l; char**a; int n; int d;
{
  int sor=80, px, py;
  py=(atoi(a[1])/10)-7;
  px=(atoi(a[2])/5)+36;
  for(d=py*sor-px;
      n=str[(l)++-3];) {
    while((n--)>'@') {
      putchar(!d+++33^l&1);
    }
  }
}
```

- A belső ciklusból csak a feltételt használjuk, így az ekvivalensen átalakítható egy `while` ciklussá, ami annyiszor fut le, amennyivel a sztring kiolvasott karakterének kódja nagyobb a `@` karakter kódjánál.

Hol vagyunk?

A program értelmezése – 8.

```
const char *str="@NKACLCCGZAAQBEAADAFaISADJABBA^SNLGAQABDAXIMBAACTBATAHDBANZcE\
MMCCCCAAhEIJFAEAAABAfHJETBdFLDAANEfDNBPHdBcBBBEA_AL␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
int main(l,a,n,d)
  int l; char**a; int n; int d;
{
  int sor=80, px, py;
  py=(atoi(a[1])/10)-7;
  px=(atoi(a[2])/5)+36;
  for(d=py*sor-px;
      n=str[(l)++-3];) {
    while((n--)>'@') {
      putchar(((!(d++))+33)^(l&1));
    }
  }
}
```

- A prioritásoknak megfelelő zárójelezéssel most is láthatóvá tehetjük a kifejezés felépítését. (Ehhez persze felhasználjuk, hogy a +++ sorozatot a fordító mohó módon ++ + műveletekként értelmezi.)

Hol vagyunk?

A program értelmezése – 9.

```
const char *str="@NKACLCCGZAAQBEAADAFaISADJABBA^SNLGAQABDAXIMBAACTBATAHDBANZcE\
MMCCCCAAhEIJFAEAAABAfHJETBdFLDAANEfDNBPHdBcBBBEA_AL␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
int main(l,a,n,d)
  int l; char**a; int n; int d;
{
  int sor=80, px, py;
  py=(atoi(a[1])/10)-7;
  px=(atoi(a[2])/5)+36;
  for(d=py*sor-px;
      n=str[(l)++-3];) {
    while((n--)>'@') {
      putchar(((!(d++))+'!')^(l&1));
    }
  }
}
```

- A sorfolytonos pozíciót jelző `d` változót lépésenként növeljük, majd egy negációval 0 (keresett pozíció) vagy 1 (egyéb pozíció) értékre konvertáljuk és hozzáadjuk egy karakterkódhoz. Az ügyes karakterválasztás miatt ez meghatározza az utolsó előtti bitet.
- Ezután `l` párosságától függően az utolsó bitet is megváltoztatjuk. Így összesen 4-féle karaktert kaphatunk.

Hol vagyunk?

A program kimenete

```
$ ./where 46 20
               !!!!!!!!!!! !!!             !!!   !!!!!!
!  !!!!!!!!!!!!!!!!  !!!!! !    !      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 !!!!!!!!!!!!!!!!!! !!!!          !  !! !!!!!!!!!!!!!!!!!!!!!!!!!!!!!
          !!!!!!!!!!!!!           !!!!"!! !!!!!!!!!!!!!!!! !!    !
             !!!!!!!!            !! !   !!!!!!!!!!!!!!!!!!  !
     !        !!!!  !             !!!!!!!!!!!!!!!!!!!!!!!!
               !!!!!            !!!!!!!!!!!!   !!!   !!! !
                    !!!!!        !!!!!!!!!!      !     ! !  !
                    !!!!!!!!         !!!!!                  !!
                     !!!!!!           !!!! !              !!!!!
                      !!!!             !!                !!!!!!!!
                      !!                                  !!  !!     !
                      !
$
```

- A kiírt karakter ’ ’ vagy ’!’, ha `d` nem nulla és `l` páratlan vagy páros, illetve ’"’ vagy ’#’, ha `d` nulla és `l` páros vagy páratlan.
- Vagyis a sztring karakterei felváltva a szárazföld-víz szekvenciák hosszát határozták meg, míg a `d` 0 értéke a speciális karakter kiírásának helyét.

Köszönöm a figyelmet!