

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 10**
 - 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

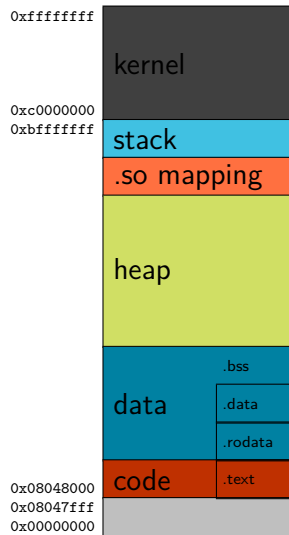
- 1 Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - 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 10**
 - 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

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?

```
1 /* A char* és char[] típusok különbsége inicializálás esetén.
2  *   A program "pointer" paraméterrel történő indítása futási hibát okoz.
3  *   2006. Augusztus 17. Gergely Tamás, gertom@inf.u-szeged.hu
4  */
5
6 #include <string.h>
7 #include <stdio.h>
8
9 int main(int argc, char *argv[]) {
10     char t[] = "Tömb_vagy_pointer";
11     char *p = "Tömb_vagy_pointer";
12     if (argc == 2) {
13         if (!strcmp(argv[1], "tomb")) {
14             strcpy(t, "Hello!");
15         } else if (!strcmp(argv[1], "pointer")) {
16             strcpy(p, "Hello!");
17         }
18         printf("p: %s\n", p);
19         printf("t: %s\n", t);
20     } else {
21         printf("Használat:\n\t%s_tomb\n\t%s_pointer\n", argv[0], argv[0]);
22     }
23     return 0;
24 }
```

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.

- `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?

```
1 /* Egy const értékének megváltoztatása.
2  * 2006. Augusztus 17. Gergely Tamás, gertom@inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6
7 int main() {
8     const int c = 100;
9     int *p;
10    p = (int*)&c;
11    *p *= 2;
12    printf("%d\n", c);
13    return 0;
14 }
```



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.



- 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.

- 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”?

- 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]

```
1 /* A volatile kulcsszó hatásának bemutatása
2  * 2016. December 13. Gergely Tamás, gertom@inf.u-szeged.hu
3  */
4
5 #include <pthread.h>
6 #include <stdio.h>
7 #include <unistd.h>
8
9 void *szal(void *arg) {
10     while (scanf("%d", (int*)arg) == 1);
11     *((int*)arg) = 0;
12     printf("Szál_befejezése.\n");
13     return NULL;
14 }
15
16 int main() {
17     pthread_t kiiro_szal;
18     int valtozo = 1;
19     printf("A_program_akkor_fejeződik_be,_ha_egy_0_értékű_input_adatután_egy\n"
20          "egészként_nem_értelmezhető_adatot_adunk_neki.\n");
21     pthread_create(&kiiro_szal, NULL, (void*)&valtozo);
22     while (valtozo);
23     printf("Várunk_a_szál_befejezésére.\n");
24     pthread_join(kiiro_szal, NULL);
25     return 0;
26 }
```

A volatile kulcsszó hatása [1/1]

volatile.c [1-26]

```
1 /* A volatile kulcsszó hatásának bemutatása
2  * 2016. December 13. Gergely Tamás, gertom@inf.u-szeged.hu
3  */
4
5 #include <pthread.h>
6 #include <stdio.h>
7 #include <unistd.h>
8
9 void *szal(void *arg) {
10     while (scanf("%d", (int*)arg) == 1);
11     *((int*)arg) = 0;
12     printf("Szál_befejezése.\n");
13     return NULL;
14 }
15
16 int main() {
17     pthread_t kiiro_szal;
18     volatile int változo = 1;
19     printf("A_program_akkor_fejeződik_be,_ha_egy_0_értékű_input_adatután_egy\n"
20          "egészként_nem_értelmezhető_adatot_adunk_neki.\n");
21     pthread_create(&kiiro_szal, NULL, (void*)&változo);
22     while (változo);
23     printf("Várunk_a_szál_befejezésére.\n");
24     pthread_join(kiiro_szal, NULL);
25     return 0;
26 }
```

- 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.

- 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]

```
1 /* Egy bináris adatállományba egész számok kiírása.
2  * Az adatállomány nevét a parancssorból kapjuk.
3  * 1998. Április 24. Dévényi Károly, devenyi@inf.u-szeged.hu
4  */
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <fcntl.h>
9 #include <unistd.h>
10 #include <errno.h>
11
12 #define DBSZAM 10
13
14 long int tomb[DBSZAM];
15 int i;                                /* innen fogunk írni */
                                        /* 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
27     for (i = 0; i < DBSZAM; ++i) {        /* a tomb feltöltése jól látható értékekkel */
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 }
```

- `home.inf.u-szeged.hu:`

```
$ 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|
```

- 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...|
```

- 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 bájtrend különbözik az egyes architektúrákon (x86-64 és SUN Sparc).
- Bájtrend alapján kétféle architektúrát különböztetünk meg:
 - Ha az alacsonyabb című memóiahelyekre az alacsonyabb helyiértékű bájtokat tároljuk (LSB – Least Significant Byte), akkor a bájtrend little-endian („kis indián”, pl. a PC típusú gépek).
 - Ha az alacsonyabb című memóiahelyekre a magasabb helyiértékű bájtokat tároljuk (MSB – Most Significant Byte), akkor a bájtrend big-endian („nagy indián”, pl. a `home` szerver).

Architektúra bájtrendjének meghatározása [1/1]

endian.c [1–16]

```
1 /* Eldönti, hogy milyen a bájtrend a számítógépen.
2  * 1998. Április 24. Dévényi Károly, devenyi@inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6
7 int main() {
8     int x = 1;
9
10    if (*(char *)&x == 1) {
11        printf("little-endian\n");
12    } else {
13        printf("big-endian\n");
14    }
15    return 0;
16 }
```



Architektúra bájtrendjének meghatározása

Az `endian.c` működése

little-endian

00 00 00 01

... 01 00 00 00 ...

... 01 00 00 00 ...

0x01

`int x = 1;`

`0x00000001`

`&x`

`(char*)&x`

`*(char*)&x`

big-endian

00 00 00 01

... 00 00 00 01 ...

... 00 00 00 01 ...

0x00

- 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 10**
 - 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

- 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 jelentőséget 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;
```

- 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 {  
    /* ... */  
};
```



- 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 értéként értelmezhetőek, az

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

is érvényes konstrukció, ami viszont mindig igaz és x-nek új értéket ad.

- Ezért a konstanssal való összehasonlítást sokan

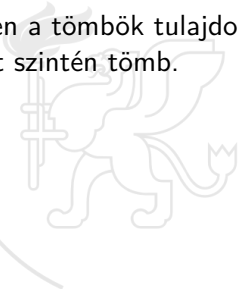
```
if (1 == x) { }
```

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")) ) { }
```

- A tömb indexelése 0-nál kezdődik, így az n elemű tömb index tartománya $[0 \dots n - 1]$.
- 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.



- 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.

- 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.

- A `printf()` értékeket vár, a `scanf()` címekeket.
- A `printf()` és `scanf()` esetén nagyon fontos a konverziós specifikáció és a paramétertípusok megfelelése.
 - 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 fájlnev 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.

- Ha egy típus elmarad, akkor a feltételezett típus `int` és nem `void`.
- Az `int` típus egyes variánsai nem teljesen kompatibilisek.
 - Alul- és túlcsoordulás lehet pl. szűkebb értékészletű 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.

- 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 **10**
 - Alapok
 - Adatállományok

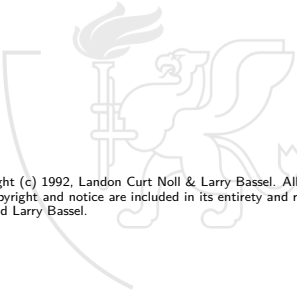
- 9 **C fordítás**
 - A fordítás folyamata
 - A preprocessor
 - A C fordító
 - Assembler
 - Linker és modulok

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

Hol vagyunk? [1/1]

where.c [1-12]

```
1      main(1
2      ,a,n,d) char**a;{
3      for(d=atoi(a[1])/10*80-
4      atoi(a[2])/5-596;n="@NKA\
5      CLCCGZAAQBEEADAFaISADJABBA^\
6      SNLGAQABDAXIMBAACTBATAHDBAN\
7      ZcEMMCCCCAAhEIJFAEAAAABafHJE\
8      TBdFLDAANEfDNBPHdBcBBBEA_AL\
9      HLEuLlLlO,uuuuWuOuRuLlD!u"
10     [1++-3];) for(;n-->64;)
11     putchar(!d+++33^
12     l&1);};
```



Copyright (c) 1992, Landon Curt Noll & Larry Bassel. All Rights Reserved. Permission for personal, educational or non-profit use is granted provided that this copyright and notice are included in its entirety and remains unaltered. All other uses must receive prior permission in writing from both Landon Curt Noll and Larry Bassel.

Hol vagyunk?

A program értelmezése – 0.

```
main(1
, a, n, d) char**a; {
for (d=atoi(a[1])/10*80-
atoi(a[2])/5-596; n="@NKA\
CLCCGZAAQBEEADAFAISADJABBA^\
SNLGAQABDAXIMBAACTBATAHDBAN\
ZcEMMCCCCAAhEIJFAEAAAABafHJE\
TBdFLDAANEfDNBPHdBcBBBEA_AL\
H_E_L_L_O, W_O_R_L_D!"
[1+-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(1,a,n,d)
{
  char**a;
  {
    for(d=atoi(a[1])/10*80-atoi(a[2])/5-596;
        n="@NKA\
CLCCGZAAQBEAADAFaISADJABBA^\
SNLGAQABDAXIMBAACTBATAHDBAN^\
ZcEMMCCCCAAhEIJFAEAAAABafHJE^\
TBdFLDAANEfDNBPHdBcBBBEA_AL^\
_H_H_E_L_L_L_0,_W_0_R_L_D!_"
        [1++-3]);
    for(;n-->64;)
      putchar(!d+++33^1&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(1,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^\
ZcEMMCCCCAAhEIJFAEAAAABafHJE^\
TBdFLDAANefDNBPHdBcBBBEA_AL^\
H_E_L_L_0,0000W_0_R_L_D!"
      [1++-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="@NKACLCCGZAAQBEEADAFaISADJABBA~SNLGAQABDAXIMBAACTBATAHDBANZcE\  
MMCCCCAAhEIJFAEAAAABafHJETBdFLDAANEfdNBPPhdBcBBBEA_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 preprozessor munkájából is, hogy a sztringünket kicsit szebbé húzzuk. Az értéke ezzel az átalakítással nem változik.

Hol vagyunk?

A program értelmezése – 4.

```
const char *str="@NKACLCCGZAAQBEEADAFaISADJABBA~SNLGAQABDAXIMBAACTBATAHDBANZcE\  
MMCCCCAAhEIJFAEAAAABafHJETBdFLDAANEfdNBPdHcBBBEA_ALH_E_L_L_O,uuuuWuO_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="@NKACLCCGZAAQBEEADAFAiSADJABBA~SNLGAQABDAXIMBAACTBATAHDBANZcE\  
MMCCCCAAaEIJFAEAAAABafHJETBdFLDAANEfdNBPHdBcBBBEA_ALH_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])/5v)+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 (N70°, W180°) koordinátáknak felel meg.)

Hol vagyunk?

A program értelmezése – 6.

```
const char *str="@NKACLCCGZAAQBEEADAFaISADJABBA~SNLGAQABDAXIMBAACTBATAHDBANZcE\  
MMCCCCAAaHEIJFAEAAAABafHJETBdFLDAANEfdNBPHdBcBBBEA_ALH_E_L_L_O,LLLLW_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="@NKACLCCGZAAQBEEADAFaISADJABBA~SNLGAQABDAXIMBAACTBATAHDBANZcE\  
MMCCCCAAhEIJFAEAAAABafHJETBdFLDAANEfdNBPfHdBcBBBEA_ALH_E_L_L_O,uuuuWuO_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="@NKACLCCGZAAQBEEADAFaISADJABBA~SNLGAQABDAXIMBAACTBATAHDBANZcE\  
MMCCCCAAhEIJFAEAAAABafHJETBdFLDAANEfdNBPHdBcBBBEA_AL_H_E_L_L_O,uuuuWuO_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="@NKACLCCGZAAQBEEADAFAiSADJABBA~SNLGAQABDAXIMBAACTBATAHDBANZcE\  
MMCCCCAAhEIJFAEAAAABafHJETBdFLDAANEfdNBPHdBcBBBEA_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.

Köszönöm a figyelmet!

