

# Programozási nyelvek 1

Szathmáry László  
Debreceni Egyetem  
Informatikai Kar

## 12. előadás

- vegyes (malloc, calloc; stdbool.h; lebegőpontos számok összehasonlítása; optimalizált fordítás; hibák dinamikus mem.-foglalás esetén; hatáskör, élettartam; statikus változók; enum; változók deklarációja; szám -> sztring konverzió; Makefile)

(utolsó módosítás: 2025. febr. 20.)

2024-2025, 2. félév



# malloc, calloc

```
int *tomb = malloc(n * sizeof(int));
```

- A malloc() aktuális paraméterlistája egy (1) elemű!
- A malloc() által visszaadott void \* típusú mutatót nem muszáj cast-olni.
- A malloc() által lefoglalt terület tartalma: memóriaszemét.

```
int *tomb = calloc(n, sizeof(int));
```

- A calloc() aktuális paraméterlistája két (2) elemű!
- A calloc() a lefoglalt tárterületet kinullázza.

# stdbool.h

C-ben nincs külön logikai típus. Ezt eddig egész értékekkel oldottuk meg, ahol a 0 hamisnak számított, az 1-es pedig igaznak.

Ha mégis szeretnénk logikai típust használni, akkor a következő lehetőségünk van:

```
1  #include <stdbool.h>
2
3  bool paros_e(int n)
4  {
5      if (n % 2 == 0)
6      {
7          return true;
8      }
9      else
10     {
11         return false;
12     }
13 }
```

Logikai típusú változó létrehozása:

```
bool found = false;
```

A true és false valójában nevesített konstansok, melyek rendre az 1 és 0 értékkel rendelkeznek. Ezek az `stdbool.h`-ban vannak megadva.

# lebegőpontos összehasonlítás

```
2  #include <math.h>
3  #include <stdbool.h>
4
5  #define EPSILON 0.0000001
6
7  bool fequal(double x, double y)
8  {
9      if (fabs(x - y) < EPSILON)
10     {
11         return true;
12     }
13     else
14     {
15         return false;
16     }
17 }
18
```

```
31
32     if (fequal(x, y))
33     {
34         puts("egyenlők");
35     }
36     else
37     {
38         puts("NEM egyenlők");
39     }
40
```

# optimalizált fordítás

## gcc -O option flag

Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

forrás: <https://www.rapidtables.com/code/linux/gcc/gcc-o.html>

# optimalizált fordítás

<code>gcc prg.c</code>	<code>// alpertelmezés, fordítás debug módban</code>
<code>gcc -O2 prg.c</code>	<code>// gyors kód generálása, release mód</code>
<code>strip -s a.out</code>	<code>// exe méretének csökkentése</code>
<code>upx --best a.out</code>	<code>// exe röptömörítése</code>

# dinamikus memóriafooglalás

Néhány szabály:

- Ha egy memóriaterületet **malloc()** -kal lefoglaltunk, akkor azt valamikor a **free()** -vel szabadítsuk is fel!
- Ellenőrizzük le a **malloc()** által visszaadott értéket!
- Csak a **malloc()** -kal lefoglalt területet szabadítsuk fel a **free()** -vel!
- Egy memóriaterületet a **free()** -vel csak egyszer szabadítsunk fel!
- Ha a **free()** -vel felszabadítunk egy memóriaterületet, akkor ahhoz már semmi közünk! Ne akarjunk onnan olvasni, ne akarjunk oda írni!
- Mindig legyen egy mutatónk, ami a dinamikusan lefoglalt terület elejére mutat. Ez a mutató lehet egy másolat is.

# hatáskör

A hatáskör (*scope*) a nevekhez kapcsolódó fogalom. Egy név hatásköre alatt a program szövegének azon részét értjük, ahol az adott név ugyanazt a programozási eszközt hivatkozza, tehát jelentése, felhasználási módja, jellemzői azonosak.

hatáskör = láthatóság = *scope*

Egy progamegységben deklarált nevet a progamegység lokális nevének nevezzük.

Hatáskörkezelés: amikor egy név hatáskörét megállapítjuk. Beszélhetünk statikus és dinamikus hatáskörkezelésről. A statikus hatáskörkezelés fordítási időben történik, ezt a fordítóprogram végzi. Az eljárásorientált nyelvek a statikus hatáskörkezelést vallják.

Egy progamegység a lokális neveit bezárja a külvilág elől.



# élettartam

Egy elem élettartama a memóriába kerülésétől a számára lefoglalt memóriaterület felszabadításáig tart, ami megtörténhet futási időben, vagy akkor, amikor a program futása véget ér.

# statikus változók

```
0
7 int next_five()
8 {
9     static int number = 0;
10
11     number += 5;
12     return number;
13 }
14
15 int main()
16 {
17     printf("%d\n", next_five()); // 5
18     printf("%d\n", next_five()); // 10
19     printf("%d\n", next_five()); // 15
20
21     return 0;
22 }
23
```

# enum

```
8
9  typedef enum {
10      UP = 1,
11      DOWN,
12      LEFT,
13      RIGHT
14  } Direction;
15
16  int main()
17  {
18      Direction dir = LEFT;
19
20      printf("%d\n", dir);    // 3
21
22      return 0;
23  }
24
```

# változók deklarációja

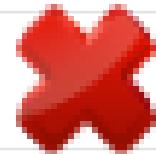
If declaring multiple variables of the same type at once, it's fine to declare them together, as in:

```
int quarters, dimes, nickels, pennies;
```



Just don't initialize some but not others, as in:

```
int quarters, dimes = 0, nickels = 0 , pennies;
```



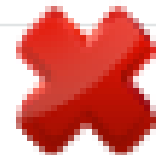
Also take care to declare pointers separately from non-pointers, as in:

```
int *p;  
int n;
```



Don't declare pointers on the same line as non-pointers, as in:

```
int *p, n;
```



# szám → sztring konverzió

```
10
11 int main()
12 {
13     char text[100];
14
15     int year = 2525;
16
17     sprintf(text, "%d", year);
18
19     printf("%s\n", text);    // 2525
20
21     return 0;
22 }
23
```

# Makefile

```
1
2 program: main.o mymath.o
3     gcc main.o mymath.o -o program
4
5 main.o: main.c
6     gcc -c main.c
7
8 mymath.o: mymath.c mymath.h
9     gcc -c mymath.c
10
11 clean:
12     rm *.o program
13
14 # target: dependencies
15 # action
16
```

# Házi feladat

- A K & R-féle „C Bibliában” nézzék át azokat a részeket, amikről szó volt az előadáson.
- Juhász István jegyzetéből nézzék át azokat a fogalmakat, amikről szó volt az előadáson ([link](#)).