

Programozási nyelvek 1

Szathmáry László
Debreceni Egyetem
Informatikai Kar

4. előadás

- memóriaszemét, képernyőre íratás
- függvények
- tömbök

(utolsó módosítás: 2024. jan. 31.)

2023-2024, 2. félév



Memóriaszemét

A memória 1 byte méretű rekeszekből épül fel. Minden rekesz rendelkezik egy egyedi azonosítóval (memóriacím).

Az egyszerűség kedvéért itt egyszerűsített memóriacímeket használunk (\$A0, ...).

\$A0

\$A1

\$A2

\$A3

\$A4

•

•

•

\$B0

\$B1

•

•

•

•

•

•

Memóriaszemét

Mi a szabad rekeszek tartalma?

A korábban befejeződött programok által visszahagyott „szemét”.

\$A0

56562

\$A1

876

\$A2

5

\$A3

34324

\$A4

78768

•

•

•

•

•

•

\$B0

345

\$B1

42

•

•

•

Memóriaszemét

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int x;
6
7      // Hiba! Nem adtunk
8      // neki kezdőértéket!
9      printf("%d\n", x);
10
11     return 0;
12 }

```

Az egyszerűség kedvéért az x változó értékét egy (1 db) rekeszben helyeztük el. Mivel egy int 4 byte-os, ezért a valóságban ez az x változó 4 rekeszt foglalna el.

	.	
	.	
	.	
\$A0	56562	
\$A1	876	
\$A2	5	X
\$A3	34324	
\$A4	78768	
.	.	
.	.	
.	.	
\$B0	345	
\$B1	42	

.

.

.

Memóriaszemét

- Fordítás
 - `gcc mem_szemet1.c`
 - `gcc mem_szemet1.c -Wall`
 - `gcc mem_szemet1.c -Wall -Werror`
 - `make mem_szemet1`

"C makes it easy to shoot yourself in the foot..."

Memóriaszemét

Egy másik tipikus hiba:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int counter;
6
7      for (int i = 0; i < 10; ++i)
8      {
9          // Hiba! Úgy növeljük az értékét, hogy nem adtunk
10         // neki kezdőértéket!
11         counter += 1;
12     }
13
14     printf("%d\n", counter);
15
16     return 0;
17 }
```



Képernyőre írás

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello\n");
6
7      puts("world");
8
9      putchar('A');
10     putchar('B');
11     putchar('B');
12     putchar('A');
13     putchar('\n');
14
15     puts("");
16
17     printf("__END__");
18
19     return 0;
20 }
```

ismerős

Csak sztring kiíratására
alkalmas, formázás nélkül.
A végén: automatikus újsor.

egy darab karakter
kiíratása

üres sor

```
hello
world
ABBA
```

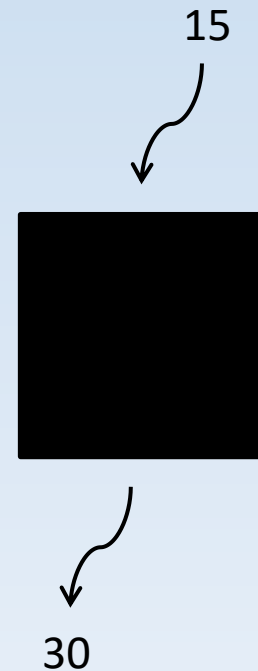
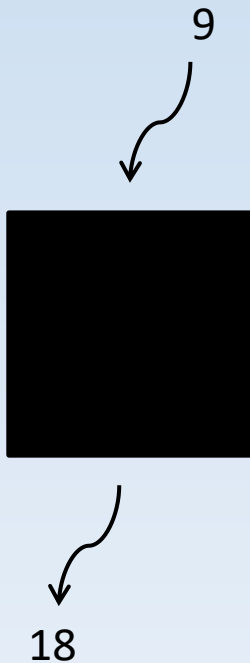
```
__END__
```

Függvények

- Eddig mindent a `main()` függvényen belülre írtunk.
- Viszont ahogy nő a programunk, egyre inkább felmerül az igény, hogy jó lenne a `main()` függvény tartalmát szétszedni kisebb logikai egységekre.
- Erre szolgálnak a **függvények**. (Egyéb elnevezések: **eljárás, szubrutin, metódus, alprogram**).

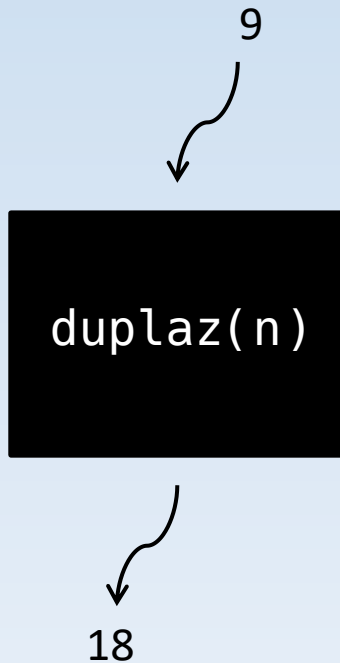
Függvények

- Mi a függvény?
 - Egy fekete doboz 0 vagy több bemenettel, s pontosan 1 kimenettel.



Függvények

- Mi a függvény?
 - Egy fekete doboz 0 vagy több bemenettel, s pontosan 1 kimenettel.



Függvények

- Mi a függvény?
 - Egy fekete doboz 0 vagy több bemenettel, s pontosan 1 kimenettel.

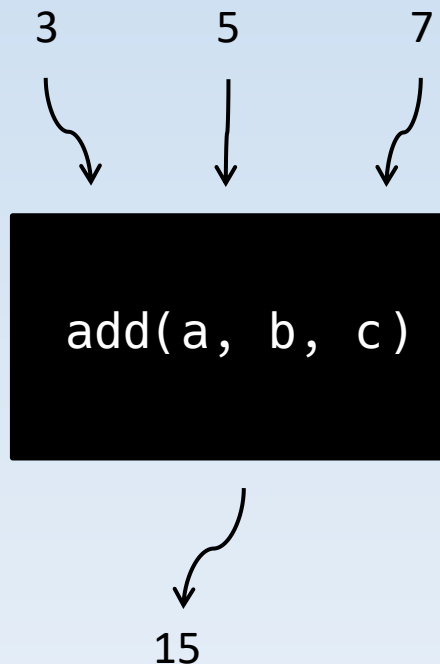
`mai_datum()`



`"2023-03-07"`

Függvények

- Mi a függvény?
 - Egy fekete doboz 0 vagy több bemenettel, s pontosan 1 kimenettel.



Függvények

- Miért hívjuk fekete doboznak?
 - Ha nem mi írtuk meg a függvényt (pl. a standard könyvtár része), akkor nem kell tudnunk, hogy hogyan lett implementálva.
 - Ha mi írtuk meg, s helyesen működik (leteszteltük), akkor „el lehet felejteni”, hogy hogyan implementáltuk.
 - Az a lényeg számunkra, hogy **mit csinál** az adott függvény. Az, hogy **hogyan csinálja**, az már nem érdekel, ui. az implementációs részletek nélkül is **tudom használni**.

```
mult(a, b):  
    output a * b
```

Függvények

- Miért hívjuk fekete doboznak?
 - Ha nem mi írtuk meg a függvényt (pl. a standard könyvtár része), akkor nem kell tudnunk, hogy hogyan lett implementálva.

```
mult(a, b):  
    set counter to 0  
    repeat b times:  
        add a to counter  
    output counter
```

Függvények

- Miért hívjuk fekete doboznak?
 - Ha nem mi írtuk meg a függvényt (pl. a standard könyvtár része), akkor nem kell tudnunk, hogy hogyan lett implementálva.
- Mit kell tudni egy függvény használatához?
 1. Mi a függvény neve?
 2. Mi a bemenet?
 3. Mi a kimenet?
 4. Mit csinál a függvény?
- A függvényeink számára válasszunk beszédes neveket (1-es pont), s a függvényeinket lássuk el dokumentációval (4-es pont, ill. 2-es és 3-as pontok). Egy jó függvénynévből már ki is lehet találni, hogy mit csinál az adott függvény!

Függvények

- Miért használunk függvényeket?
 - Szervezés
 - A függvények segítségével egy nagyobb problémát kisebb részproblémákra tudunk bontani.
 - Egyszerűsítés
 - Egy kisebb komponenst egyszerűbb megtervezni, implementálni és debuggolni.
 - Újrahasznosítás
 - Egy függvényt elég egyszer megírni, s többször, több helyen is fel tudjuk használni.

Függvények

- függvénydeklaráció

`return-type name(parameters);`



- `return-type`: a visszatérési érték típusa (output érték típusa)
- `name`: a függvény neve
- `parameters`: *formális paraméterlista* (input paraméterek)

Függvények

- függvénydeklaráció

```
int add_two_ints(int a, int b);
```

- Két egész összege szintén egész lesz.
- A függvény nevéből egyértelműen kiderül, hogy mit csinál a függvény.
- Két input paramétere van a függvénynek. Mindkettőnek nevet kell adni, ill. meg kell adni a típusukat is. Jelen esetben a függvény egyszerűsége miatt ezek a rövid nevek is elfogadhatóak.

Függvények

- függvénydeklaráció
 - Deklaráljunk egy függvényt, ami összeszoroz két valós számot!

Függvények

- függvénydeklaráció
 - Deklaráljunk egy függvényt, ami összeszoroz két valós számot!

```
float mult_two_reals(float x, float y);
```

- Két float szorzata szintén float lesz.
- Beszédes nevű a függvény.
- Rövid, egyszerű függvényről van szó, így az input változóknak lehet egyszerű, rövid nevet adni.

Függvények

- függvénydeklaráció
 - Deklaráljunk egy függvényt, ami összeszoroz két valós számot!

```
double mult_two_reals(double x, double y);
```

- Két double szorzata szintén double lesz.
- Beszédes nevű a függvény.
- Rövid, egyszerű függvényről van szó, így az input változóknak lehet egyszerű, rövid nevet adni.

Függvények

- függvény**definíció**
 - Egy függvény definíciója esetén megadjuk a függvény implementációját is.
 - Definiáljuk az előbb látott `mult_two_reals()` függvényt!

Függvények

- Egy függvény definíciója nagyon hasonlít a függvény deklarációjára.

- Deklaráció:

```
float mult_two_reals(float x, float y);
```



- Definíció:

```
float mult_two_reals(float x, float y)
{
    float result = x * y;
    return result;
}
```

Függvények

- Egy függvény definíciója nagyon hasonlít a függvény deklarációjára.

- Deklaráció:

```
float mult_two_reals(float x, float y);
```



- Definíció:

```
float mult_two_reals(float x, float y)
{
    return x * y;
}
```


Függvények

- Definiáljuk a korábbi `add_two_ints()` függvényt!
- Deklaráció:

```
int add_two_ints(int a, int b);
```

- Definíció:

```
int add_two_ints(int a, int b)
{

}
}
```

Függvények

- Definiáljuk a korábbi `add_two_ints()` függvényt!
- Deklaráció:

```
int add_two_ints(int a, int b);
```

- Definíció:

```
int add_two_ints(int a, int b)
{
    int sum;           // deklaráció
    sum = a + b;       // eredmény kiszámítása
    return sum;        // eredmény visszaadása
}
```

Függvények

- Definiáljuk a korábbi `add_two_ints()` függvényt!
- Deklaráció:

```
int add_two_ints(int a, int b);
```

- Definíció:

```
int add_two_ints(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

Függvények

- Definiáljuk a korábbi `add_two_ints()` függvényt!

```
int add_two_ints(int a, int b)
{
    int sum = 0;
    while (a > 0) {
        ++sum;
        --a;
    }
    while (a < 0) {
        --sum;
        ++a;
    }
    while (b > 0) {
        ++sum;
        --b;
    }
    while (b < 0) {
        --sum;
        ++b;
    }
    return sum;
}
```

Ez csak egy példa arra, hogy így is lehetne implementálni a feladatot!

A valóságban nehogy ilyen kódot írjanak!

Függvények

- Definiáljuk a korábbi `add_two_ints()` függvényt!
- Deklaráció:

```
int add_two_ints(int a, int b);
```

- Definíció:

```
int add_two_ints(int a, int b)
{
    return a + b;
}
```

Függvények

- Függvények (meg)hívása
 - Miután megírtunk egy függvényt, itt az ideje használni is!
 - Egy függvényt a neve alapján tudunk meghívni. A hívás helyén adjuk meg a megfelelő paramétereket (*aktuális paraméterlista*), majd a visszatérési értékét használjuk fel (pl. rendeljük hozzá egy megfelelő típusú változóhoz).
 - Példa: `osszead*.c`

Függvények

- Megjegyzés
 - Ha egy függvény nem ad vissza semmit, akkor a visszatérési érték típusa `void` lesz.
 - Az ilyen „függvényt” (ami nem ad vissza értéket) *eljárásnak* nevezzük.
 - Azt mondjuk, hogy C-ben minden függvény. Vagyis az eljárás így tekinthető olyan speciális függvénynek is, ami nem ad vissza semmilyen értéket sem.
 - Erről még lesz szó, ill. a gyakorlaton is sokat találkozunk még vele.

Függvények

- Gyakorlás
 - Oldjuk meg a **Megrajzolható háromszögek** feladatot! ([link](#))

Tömbök

- A **tömb** egy alapvető adatszerkezet. Rendkívül hasznos, az egyik leggyakrabban használt adatszerkezet.
- **Homogén** adatszerkezet: az elemei azonos típusúak.
- Ábrázolása **folytonosan** történik: a tömb elemei a memóriában folytonos, összefüggő tárterületet alkotnak.
- Mivel az elemei azonos típusúak, ezért az elemek azonos méretűek a memóriában.
- **Statikus** adatszerkezet: miután létrehoztuk, a mérete nem változhat (pl. nem lehet új elemet beszúrni).

Tömbök

- C-ben a tömb elemeinek az indexelése 0-ról indul.
- Vagyis, egy n elemű tömb esetén a legelső elem indexe 0, míg a legutolsó elem indexe $(n-1)$.
- **Vigyázat!** Gyakori hiba, hogy egy olyan indexű elemre hivatkozunk, ami a tömbön kívül esik ("out of bounds"). A program lefordul, de lehet, hogy futás közben egy hibaüzenettel terminál.

Tömbök

- Tömb deklarációja

```
type name[size];
```

- **type**: a tömb elemeinek a típusa
- **name**: a tömb neve
- **size**: a tömb mérete, vagyis az elemeinek a száma

Tömbök

- Tömb deklarációja

```
int pontok[20];
```

- **type**: a tömb elemeinek a típusa
- **name**: a tömb neve
- **size**: a tömb mérete, vagyis az elemeinek a száma

Tömbök

- Tömb deklarációja

```
float havi_kiadasok[12];
```

- **type**: a tömb elemeinek a típusa
- **name**: a tömb neve
- **size**: a tömb mérete, vagyis az elemeinek a száma

Tömbök

- Egy **típus** típusú tömb minden eleme **típus** típusú. Ha úgy gondolunk rájuk, mint **típus** típusú változókra, akkor az összes eddigi művelet ismerős lesz.

```
int igazsagtabla[10];
```

```
...
```

```
igazsagtabla[2] = 0;           // hamis  
if (igazsagtabla[7] == 1)      // ha igaz  
{  
    puts( "True" );  
}  
igazsagtabla[10] = 1;          // igaz
```

Tömbök

- Egy **típus** típusú tömb minden eleme **típus** típusú. Ha úgy gondolunk rájuk, mint **típus** típusú változókra, akkor az összes eddigi művelet ismerős lesz.

```
int igazsagtabla[10];
```

```
...
```

```
igazsagtabla[2] = 0;
```

```
// hamis
```

```
if (igazsagtabla[7] == 1)
```

```
// ha igaz
```

```
{
```

```
    puts( "True" );
```

```
}
```

```
igazsagtabla[10] = 1;
```

```
// HIBA!
```

Tömbök

- A változókhoz hasonlóan egy tömböt is lehet **inicializálni**, vagyis a tömb létrehozásakor egy speciális szintaxissal meg lehet adni az elemek kezdőértékét.

- Előbb deklaráció, majd értékadás:

```
int pontok[3];  
pontok[0] = 12;  
pontok[1] = 8;  
pontok[2] = 15;
```

- Inicializálás:

```
int pontok[3] = { 12, 8, 15 };
```

ekvivalensek

Tömbök

- **Feladat:** készítsünk egy 100 elemű tömböt, amit töltünk fel a 0,...,99 értékekkel. Vagyis: a 0. indexű helyen a 0 elem szerepeljen, az 1-es indexű pozícióban legyen az 1-es érték, stb.
(Tipp: használjunk ciklust...)

Tömbök

- Az inicializálás során használható egy másik speciális szintaxis is.
- Inicializálás a méret explicit megadásával:

```
int pontok[3] = { 12, 8, 15 };
```

- Inicializálás a méret elhagyásával:

```
int pontok[] = { 12, 8, 15 };
```



Tömbök

- Főleg egydimenziós tömböket szoktunk használni, de egy tömb lehet több dimenziós is, pl.:

```
char tictactoe[3][3];  
...  
tictactoe[1][2] = 'X';
```

	0	1	2
0			
1			X
2			

lásd később

Tömbök

- Egy tömb elemeit úgy tudjuk kezelni, mintha azok önálló változók lennének.
- Egy teljes tömböt viszont nem tudunk ugyanúgy kezelni, mint egy hagyományos változót.
 - Például egy tömböt nem lehet értékül adni egy másik tömbnek!
 - Ezt a műveletet pl. egy ciklussal tudjuk megoldani: az egyik tömb minden egyes elemét átmásoljuk a másik tömbbe.

Tömbök

```
int t1[3] = { 1, 2, 3 };  
int t2[3];
```

```
t2 = t1;    // hiba
```

```
for (int i = 0; i < 3; ++i)  
{  
    t2[i] = t1[i];  
}
```

Tömbök

- C-ben **érték szerinti paraméterátadás** van. Egy függvény meghívásakor a formális paraméter megkapja az aktuális paraméter értékét. Ilyenkor egy értékmásolás történik. Az aktuális és a formális paraméter egymástól független. A formális paraméter nem tudja módosítani az aktuális paramétert.
- A tömbök átadása ezzel szemben **referencia szerint** történik (*). A hívott fél a tényleges tömböt kapja meg, nem pedig annak a másolatát! Mindkét fél ugyanazt a tömböt látja.
 - Ez azt fogja eredményezni, hogy ha a hívott fél módosítja a tömböt, akkor ezen módosításokat a hívó fél is látni fogja (hiszen mindkettő ugyanazt a tömböt látja)!

(*) Nem teljesen van így, de az igazságot kicsit később fedjük fel.

Tömbök

```
#include <stdio.h>

void set_int(int x)
{
    x = 42;
}

void set_array(int array[])
{
    array[0] = 22;
}

int main()
{
    int a = 10;
    int b[4] = { 0, 1, 2, 3 };
    set_int(a);
    set_array(b);
    printf("%d, %d\n", a, b[0]);
    return 0;
}
```

Tömbök



10, 22

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](#)).

Szorgalmi

- Haladjanak tovább a Linux operációs rendszerrel. Lejátszási lista: <http://bit.ly/31pRf7A> . Megtekintendő videók: 16-tól 21-ig. A Midnight Commander-t (16, 17, 18) mindenképp nézzék meg és kezdjék el használni!
(Windows alatt pedig használjanak Total Commander-t).