

Kvíz

- <http://www.menti.com>
- mindenki a saját azonosítójával (**bbbbnnnn**) lépjön be

!

aki **nem** az azonosítóját használja,
annak nem fogjuk tudni beírni
a pontokat a kvízre

6508 8950



Pointerek I

KURZUS

Érdekességek

Érdekességek

extern lokális változó

- az **extern** tárolási osztállyal deklarált változó látható más forráskódokból is, élettartama a teljes program futásával megegyezik
- lokális változó kizárolag csak blokkon belül látható és érhető el, a blokkból való kilépéskor megszűnik létezni
- lokális változókat is lehet **extern** tárolási osztállyal deklarálni

ELLENTMONDÁS?

NEM

viszont első sorban nem a lokális változók korlátozásain enyhít,
hanem a globális változókra vezet be korlátozásokat

Érdekességek

extern lokális változó

ha a lokális változó nem volt korábban deklarálva

DE az **i** csak az **f** függvényben látható,
linkelése külső, akár a globális változónak

ha a lokális változó korábban deklarálva volt

DE az **i** csak az **f** függvényben látható,
linkeléssel nem rendelkezik

ha a lokális változó korábban **extern** módban volt
deklarálva

mivel korábban **extern** módban volt
deklarálva

```
void f()
{
    extern int i;
}
```

```
void f()
{
    int i;
    extern int i;
}
```

```
extern int i;
void f()
{
    extern int i;
}
```



```
extern int i;
void f()
{
}
```

```
void f()
{
    extern int i;
}
```

```
extern int i;
void f()
{
}
```

Érdekességek

extern lokális változó

ha a lokális változó korábban **static** módban volt deklarálva

mivel korábban **static** módban volt deklarálva

helytelen kódolás

mert a blokkban történik az **extern** típusú változó inicializálás

```
static int i;  
void f()  
{  
    extern int i;  
}
```



```
static int i;  
void f()  
{  
}
```

```
void f()  
{  
    extern int i = 0; — helyesen →  
}
```

```
extern int i = 0;  
void f()  
{  
}
```

|| de fura stílus

```
int i = 0;  
void f()  
{  
}
```

Érdekességek

inline függvények hátrányai

- megnövelheti a függvény méretét, így már nem fér be a cache-be, rengeteg **cache miss** hibát eredményezve
- ha olyan változóink vannak, amelyek intenzív regiszter használtot feltételeznek, akkor túlterhelheti a regisztert
- túlterhelheti a fordítóprogramot
- ha header állományban használjuk, akkor túlzott állományméret növekedést eredményezhet, ezáltal olvashatatlaná teszi azt
- ha több **inline** függvényt használunk, akkor nagyon könnyen szemetelhetünk a memóriába, amely több **page fault** hibát és teljesítmény csökkenést eredményez
- nem hasznos beágyazott rendszerek esetén, mert megnöveli a futtatható, bináris állomány méretét, viszont véges a memória mennyisége az ilyen rendszerekben
- ha ismétlő ciklusban hívunk meg **inline** függvényt, akkor nagyon elbúrjánzik a kód

Érdekességek

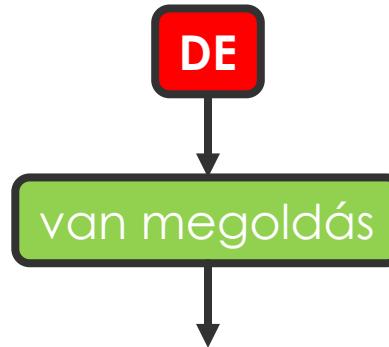
inline függvények hátrányai

- **DE** akkor mikor használjunk **inline** függvényeket?
 - ha biztos vagy abban, hogy futási teljesítmény növekedéssel jár
 - minden **inline** függvényt használunk **makro** függvények helyett
 - **NE** használunk nagy függvényeket **inline** módban, minden csak kódszegmenseket használunk **inline** módban, hogy teljesítmény növekedést érjünk el
 - rekurzív függvények esetén a fordítóprogram nem végzi el a beágyazást, attól függetlenül, hogy odaírtuk az **inline** kulcsszót

Érdekességek

#define

- moduláris programozás esetén technikailag nem megoldható, hogy csak egy bizonyos forráskódban legyen látható az így definiált szimbolikus változó



- az ilyen szimbolikus változót definiáljuk egy külön header állományban és azt csak abban a forráskódban **include**-oljuk, amelyikben szeretnénk, hogy látható legyen az a változó

Érdekességek

#error #warning #line

- az **#error** és a **#warning** direktívák lehetőséget biztosít egy üzenet megjelenítésére hiba, illetve figyelmeztetés esetén

```
#error ÜZENET  
#warning ÜZENET
```

példa

```
#ifndef VAR  
#error VAR is not defined  
#endif
```

- a **#line** direktíva lehetővé teszi a **__LINE__** és **__FILE__** makrók értékének megváltoztatását

```
#line SZAM  
#line SZAM NEV
```

példa

```
#line 10  
#line 20 ujNev.c
```

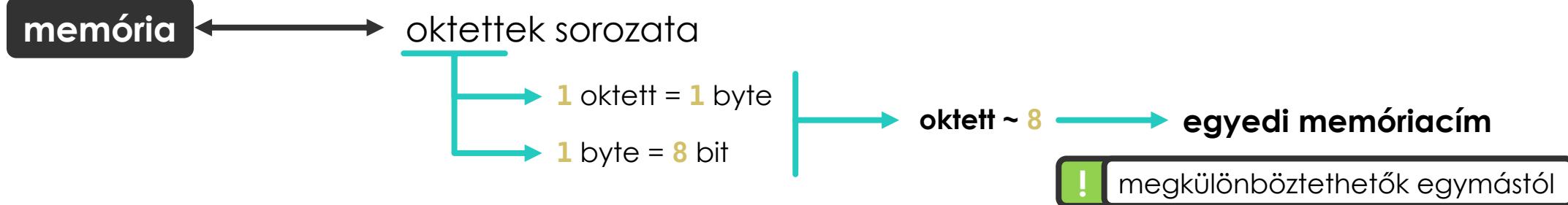
Érdekességek

#pragma

- a **#pragma** direktíva megengedi, hogy a fordítóprogrammal plusz információkat közöljünk, azon felül amiket a programozási nyelv alapból megtesz
- több opciónja létezik
 - **#pragma GCC dependency "állománynév.kiterjesztés"** dependenciát ellenőriz, megvizsgálva a megadott forráskód dátumát
 - **#pragma GCC poison utasítás** lehetővé teszi különböző utasítások használatának a megtiltását
 - **#pragma GCC system_header** a hátralevő kódot úgy kezeli, mintha a rendszer headerből származna, nincs egyéb argumentuma
 - **#pragma GCC warning ÜZENET** figyelmeztetés esetén egy üzenetet ír ki
 - **#pragma GCC error ÜZENET** hiba esetén egy üzenetet ír ki

Pointerek I

Klasszikus változók



klasszikus változó

→ egy hely a memóriában amely értéket tárolhat

6508 8950



példa

int i;

kvíz 1. kérdés

- lefoglalja a helyet a memóriában a változó tárolására
- az **i** egész típusú változónak **4** oktett kerül lefoglalásra
- az **i** változó címét az első elfoglalt oktett címe adja

Példa

`int i;` → `59180 59181 59182 59183`
az **i** változó címe **59180**

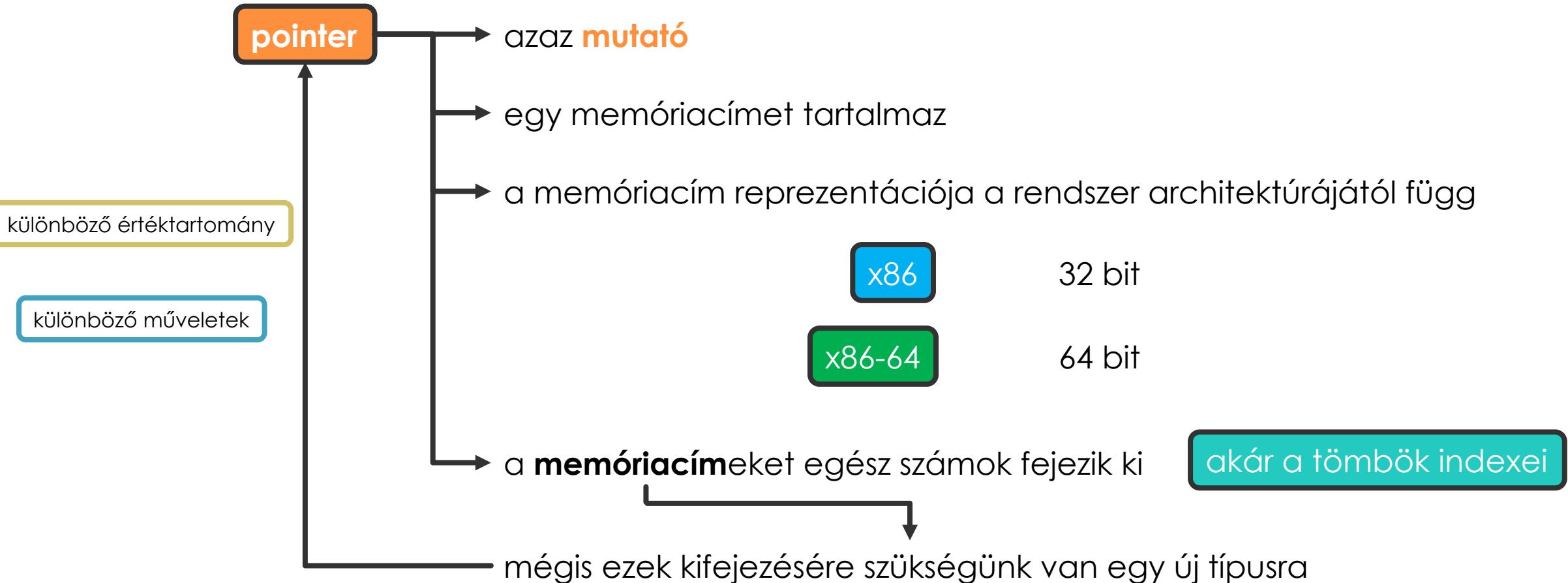
↓
4 oktett → `sizeof(int)`

cím	tárolt érték
n - 1	1 1 1 0 1 0 1 1
n - 2	1 0 1 0 1 0 1 0
n - 3	1 0 1 0 1 0 1 1
59183	0 0 1 0 1 0 1 1
59182	0 0 0 0 1 0 1 1
59181	1 1 1 0 1 1 1 1
59180	1 0 0 0 0 1 1 0
2	0 1 1 1 0 1 1 0
1	1 0 1 1 0 0 0 1
0	0 0 0 1 0 0 0 0

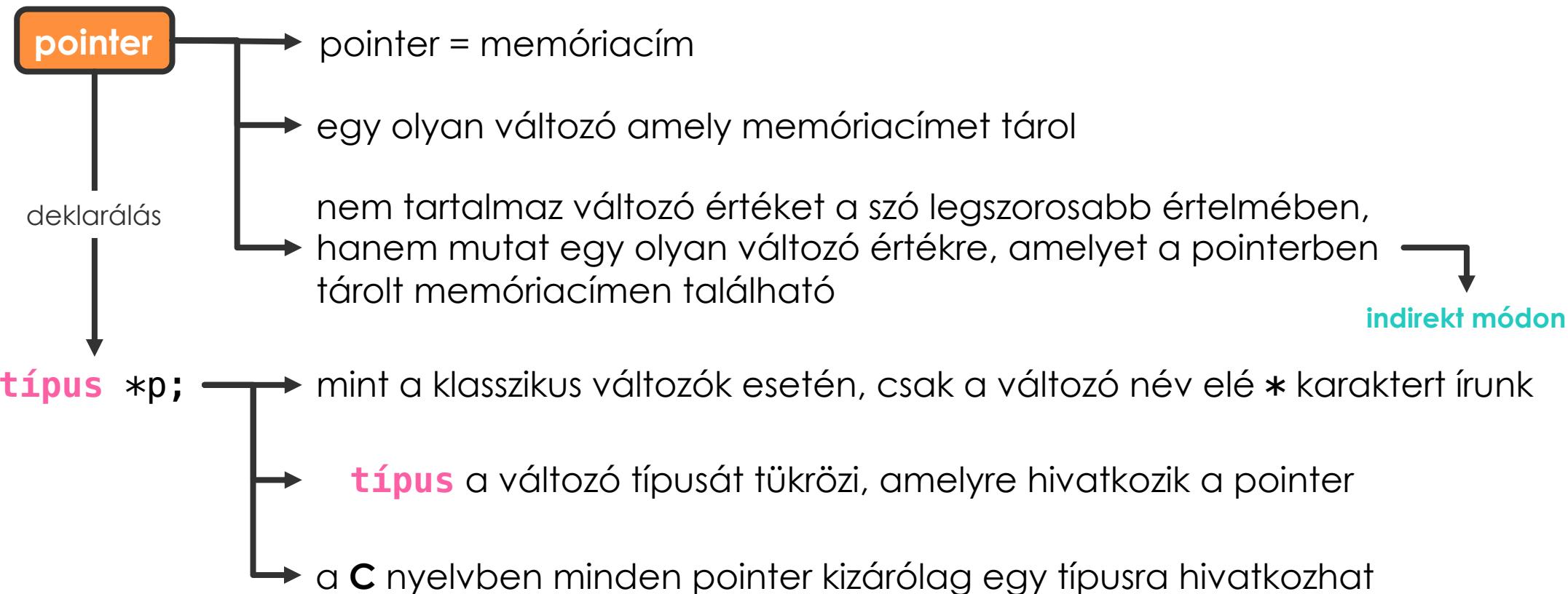
memória

ha a címezhető memória **n** oktettből áll, akkor a címek **0** és **n - 1** között vesznek fel értékeket

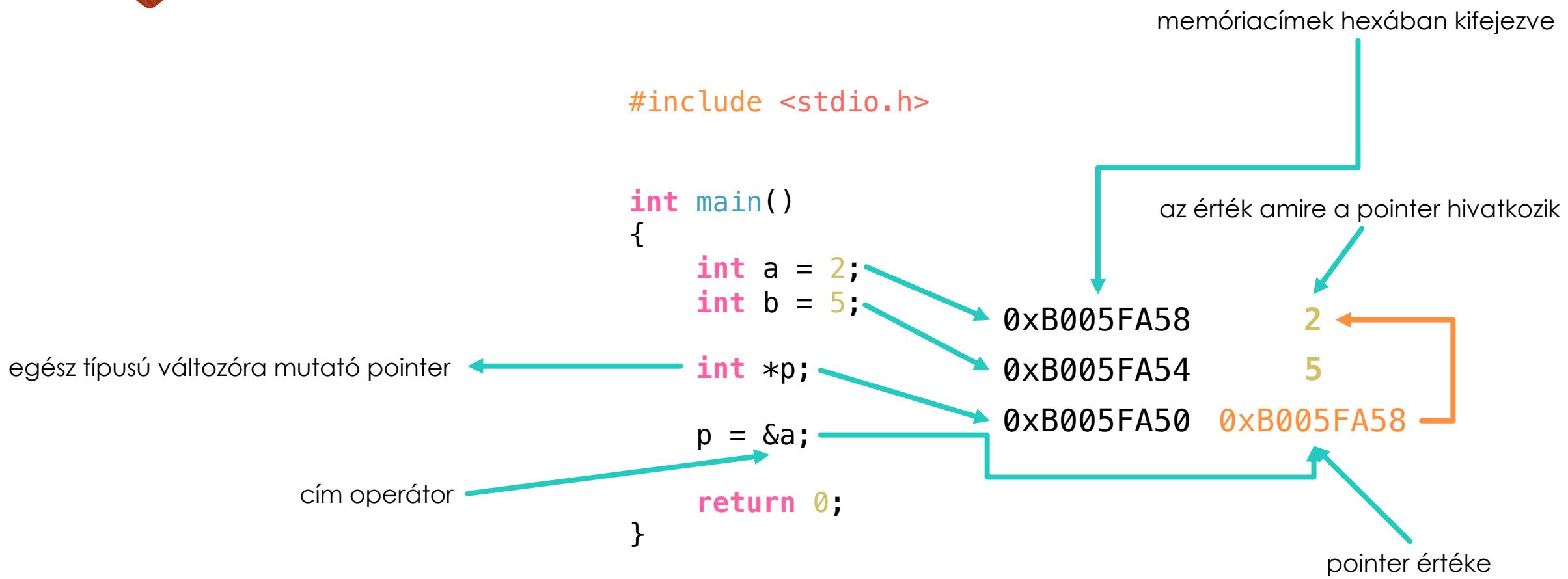
Pointerek



Pointerek



Példa



A cím operátor &

típus változó; → a változó azonosítója

&változó → a változó memóriacíme

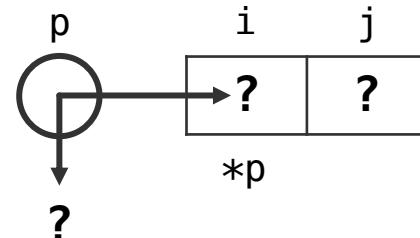
példa 0xB005FA58 2
0xB005FA54 5
0xB005FA50 0xB005FA58

példa &a = 0xB005FA58 &b = 0xB005FA54 &p = 0xB005FA50

! a pointert használat előtt inicializálni kell !

példa

```
int main()
{
    int i, j;
    int *p;
    p = &i;
    return 0;
}
```



→ deklarációkor a pointer egy véletlenszerű címre mutat

→ az inicializálás egy változó címének a hozzárendelése

→ akárcsak a változók, inicializálható deklarációkor

példa int *p = &i;

A dereferencia operátor

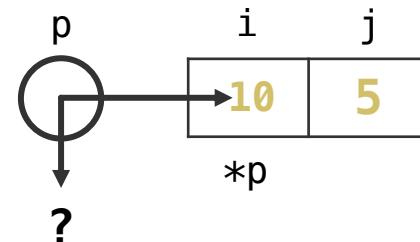
*

- hozzáférést biztosít ahhoz az értékhez, amelyet a pointer által hivatkozott memóriacímen tárolunk
 - lehetővé teszi az érték írását és olvasását

példa

```
int main()
{
    int i = 2, j = 5;
    int *p;
    p = &i;
    printf("> %d\n", *p);
    *p = 10;

    return 0;
}
```



→ **p** ekvivalens **&i** kifejezéssel
→ ***p** ekvivalens **i** értékével

→ a ***** operátor a **&** operátor inverze

példa **b = *&a** ekvivalens **b = a** kifejezéssel

!

SOHA nem használjuk a ***** operátort olyan pointerre ami nem volt korábban inicializálva

meghatározatlan viselkedés

Pointerek aritmetikája I

- ha **p** egy pointer, akkor **p + 1** a következő azonos típusú pointer memóriacímét jelöli

p → 0xB005FA54

ha **p** egy **int** típusú pointer **p + 1** → 0xB005FA58

ha **p** egy **char** típusú pointer **p + 1** → 0xB005FA55

ha **p** egy **double** típusú pointer **p + 1** → 0xB005FA5C

ÁLTALÁNOSAN

ha **p** egy típusú pointer, akkor **p + n** a **p + n * sizeof(típus)** címet jelöli

Tömbök és pointerek

- a cím operátor a tömbök esetén is használható

példa

```
int t[10];  
int *p;  
p = &t[0];  
p = &t[1];
```

 **p** a **t** tömb első elemére mutat
 **p** a **t** tömb második elemére mutat

egyszerűbb megoldás

C nyelvben a tömb neve egyben pointer is a tömb elejére

$$\begin{array}{rcl} t[i] & = & *(t + i); \\ p[i] & = & *(p + i); \end{array} \quad \begin{array}{l} p = t; \\ p = t + 1; \end{array} \quad \begin{array}{l} p = \&t[0]; \\ p = \&t[1]; \end{array}$$



DE azért van különbség a **t** és a **p** között: **t** egy konstans, fix tömb, tehát **t - p** törvénytelen művelet

Pointerek aritmetikája II

- ha **p** és **q** két pointer, amely ugyan annak a tömbnek 2 külön elemére mutat, akkor **p - q** egy egész érték, amely a tömbben **p** és **q** között szereplő elemek számát jelöli

példa

```
int t[15];
int *p, *q;
p = &t[4];
q = &t[9];
printf("> %d\n", p - q);
printf("> %d\n", (int)p - (int)q);
```

a **t[4]** címére mutat
a **t[9]** címére mutat
kiírja, hogy > 5
kiírja, hogy > 20 → `sizeof(int) * 5`

- legyen **a** és **b** két valós szám, amely egymást követő címekre van eltárolva

- pl. két lokális változó, amelyek a veremben vannak eltárolva, a verem címezése a kisebb értékek felé halad, tehát a **b** egy kisebb címértéken van eltárolva, mint az **a**

példa

```
double a, b;
double *p, *q;
p = &b;
q = p + 1;
printf("> %d\n", p - q);
printf("> %d\n", (double)p - (double)q);
```

a **b** címére mutat
azzal ekvivalens, hogy **q = &a**;
kiírja, hogy > 1
kiírja, hogy > 8 → `sizeof(double) * 1`

Konstans pointer

- egy olyan pointer, amely konstansul egy memóriacímre mutat, viszont az ott szereplő érték változhat

típus *const azonosító = érték;

példa

```
char str1[10] = "ez egy string";
char *str3;
char *const str2 = str1;
```



példa

```
int *const constP = 9;
int *q;
```



Pointer, ami egy konstansra mutat

- egy olyan pointer, amelynek változhat a címe amire hivatkozik, de a címen tárolt érték az konstans

típus const *azonosító = konstans_érték;
const típus *azonosító = konstans_érték;

példa

```
char const *str1 = "ez egy string";  
char *str2;
```

változtatható a cím amelyre mutat

```
str1 = str2;
```

NEM változtatható az érték, amire hivatkozik

```
*str1 = 's';  
*(str1 + 1) = 't';  
*(str1 + 2) = 'r';
```

módosítható az érték egy másik, nem konstansra mutató pointer segítségével

```
str2 = str1;  
*str2 = 's';  
*(str2 + 1) = 't';  
*(str2 + 2) = 'r';
```

A void pointer

- a pointerek a hivatkozott érték típusával rendelkeznek
- kivéve **általános pointerek**

példa

```
int *p;  
void *v;
```

`p = (int *)v;` → figyeljünk, hogy a típuskonverzió során a típusok egyezzenek

void *azonosító;

általános pointer

bármilyen típushoz hozzárendelhető,
explicit típuskonverzió segítségével

példa

```
int main()  
{  
    int i = 5;  
    double d = 3.1415;  
    void *v;  
  
    v = &i;  
    printf("> %d\n", *(int *)v);  
  
    v = &d;  
    printf("> %lf\n", *(double *)v);  
  
    return 0;  
}
```

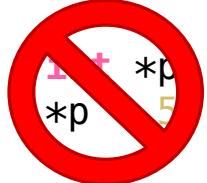


a tárolt értékre való hivatkozás (dereferencia) sem lehetséges megfelelő típuskonverzió nélkül

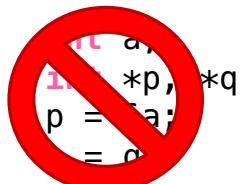
Pointerek

gyakori hibák

- egy inicializálatlan pointer értékére való hivatkozás vagy annak módosítása



- helytelen referencia



- **NULL** referencia



Pointer, mint argumentum



paraméterek átadása

6508 8950



kvíz 2. kérdés

Paraméterek átadása

- C nyelvben a paraméterek átadása kizárolag **érték szerint** történik
 - az aktuális paraméter értéke **bemásolásra kerül** a meghívott függvény megfelelő formális paraméterébe

```
#include <stdio.h>

int summa1N(int n);

int main()
{
    int n, sum;

    printf("> adj meg egy egész szamot: ");
    scanf("%d", &n);

    sum = summa1N(n);

    printf("> az else %d szam osszege: %d\n", n, sum);

    return 0;
}

int summa1N(int n)
{
    int s = 0;

    for(; n >= 0; s += n, n--)
        ;
    return s;
}
```

függvény prototípusa

beolvasásra kerül az **n** változó értéke

az **n** változó értéke átmásolásra kerül, a másolat értéke teljesen független a főprogramban levő **n** változó értékétől

az **n** változó értéke változatlan marad

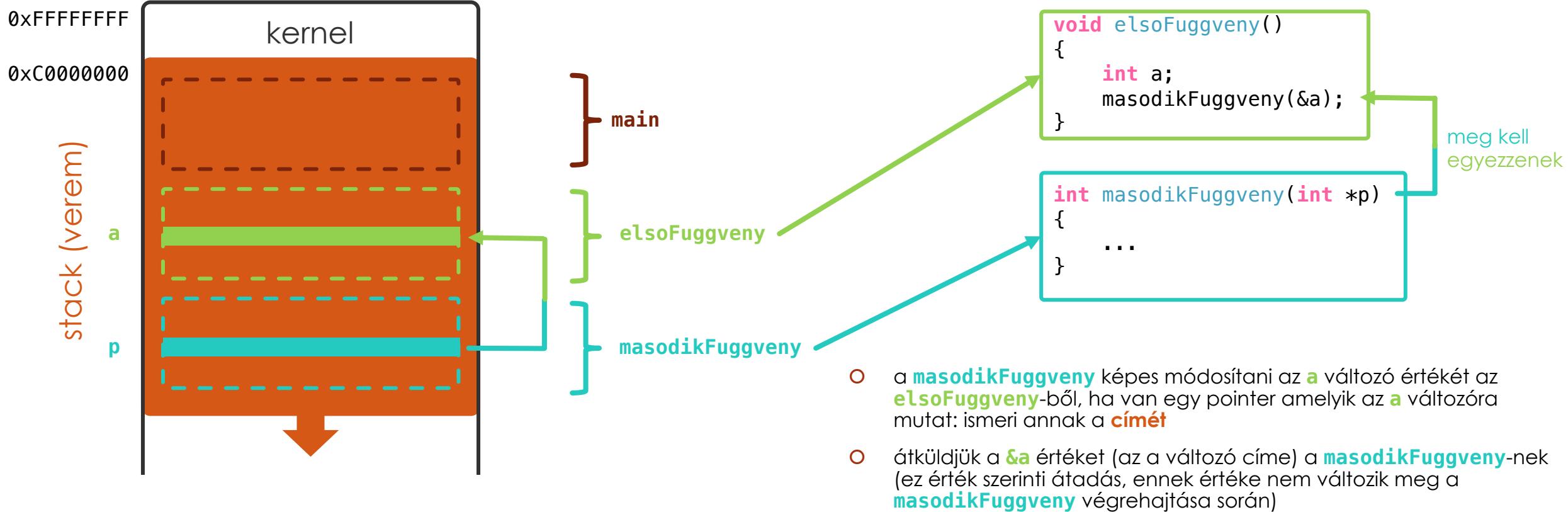
előny: megóvja a meghívó függvény változóit

hátrány: a másolás nem hatékony művelet nagy adatszegmensek esetén



Paraméterek átadása

- és az örökös kérdés: hogyan módosíthatná egy függvény a kapott paraméter értékét úgy, hogy a módosítás a meghívó függvényben is látható maradjon? ➔ pointerek



Paraméterek átadása

érték szerinti átadás

```
#include <stdio.h>

int fuggveny(int a);

int main()
{
    int a = 1, b;
    b = fuggveny(a);

    return 0;
}
```

- az **a** változó értéke bemásolásra kerül egy lokális változóba a **fuggveny** meghívásakor
- az **a** értéke a **main** programrészben nem változik

```
#include <stdio.h>

int main()
{
    int a;
    scanf("%d", &a);

    return 0;
}
```

- abban az esetben, ha szeretnénk az **a** változó értékét megváltoztatni, meg kell mondjuk a **scanf** függvénynek, hogy hol találja az **a** változót: a **&a** címen

pointeren keresztüli történő átadás

Pointer átadása, mint argumentum

példa

```
osszeg.c
#include <stdio.h>

int osszeg(int *t, int hossz)
{
    int *p;
    int s = 0;

    for(p = t; p - t < hossz; ++p)
    {
        s += *p;
    }

    return s;
}

int main()
{
    int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    printf("> a tomb elemeinek osszege: %.2f\n", osszeg(t, 10));

    return 0;
}
```

osszeg.c

tömb átadása

6508 8950



kvíz 3. kérdés

Paraméterek átadása

példa

Swap.c

```
#include <stdio.h>

void swap(int *, int *);

int main()
{
    int a = 2;
    int b = 5;

    printf("> a = %d\ tb = %d\n", a, b);
    swap(&a, &b);
    printf("> a = %d\ tb = %d\n", a, b);

    return 0;
}

void swap(int *p, int *q)
{
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

a formális paraméterek **int** típusú pointerek

a függvény hívásakor átadjuk a változók **címét**

a függvény a ***** operátort használjuk, hogy hozzáférjünk a változók értékéhez, amelyet a továbbított címen tárolnak

- ***p** ekvivalens a főprogram **a** változójával
- ***q** ekvivalens a főprogram **b** változójával

Pointer, mint visszatérített érték

- egy függvény visszatérhet egy pointerrel

extern típusú változó

static típusú lokális változó

példa

```
int *max(int *p, int *q)
{
    return (*p > *q) ? p : q;
}
```

a függvény kap két pointert paraméterként és visszatéríti a nagyobb értékre mutató pointert

példa

```
int *kozepsoElem(int *t, int hossz)
{
    return &t[hossz/2];
}
```

a függvény visszatérít egy pointert, amely a tömb középső elemére mutat

példa

```
char *getBuffer()
{
    static char *buff = 0;
    if(buff == 0)
    {
        buff = malloc(BUFFER_SIZE);
    }
    return buff;
}
```

a függvény visszatérít egy pointert egy **static** lokális változóra
a függvény minden meghívásakor ugyan azon értékét téríti vissza
a buff inicializálása és a memória foglalás csak a függvény első meghívásakor kerül végrehajtásra

Pointer, mint visszatérített érték

○ FONTOS

- nem téritünk vissza pointert ami **auto** változóra mutat → **ÉRVÉNYTELEN REFERENCIA**
- az **auto** változó csak azon a blokkon belül látható, amelyben definiálva volt
- az **auto** változó megszűnik létezni a blokkból való kilépéskor

```
int *function()
{
    int a = 5;
    return &a;
}
```



Pointer ami pointerre mutat

- egy olyan pointer, ami a másik pointer címét tartalmazza **pointerre mutató pointer**

```
int a, *p, **q; → egy int típusú értékre mutató pointerre mutató pointer  
p = &a;  
q = &p;
```

- korlátlan számban végezhetünk pointer láncolásokat

- értékre mutató pointerre mutató pointerre mutató pointerre ... mutató pointer

```
int ***...*p;
```

```
int a, *p, **q, ***r;  
p = &a;  
q = &p;  
r = &q;  
***r = 5;
```

→ hivatkozás az r pointer hivatkozott értékének a hivatkozott értékére

!

a & operátor csak egyszer alkalmazható a változó címének visszatérítésére

p = &a;

q = &p;

Példa

pointer.c

bejárjuk a többi elemet a **p** változó pointer segítségével

6508 8950

kvíz 4. kérdés



```
#include <stdio.h>

int main()
{
    int t[20] = {1, 2, 3, 4, 5}; ← konstans pointer
    int *p;
    int i; ← az első 5 elem implicit inicializálva van, a többi értéke 0

    for(i = 5; i < 10; i++)
    {
        printf("> t[%d] : ", i);
        scanf("%d", t + i); ← pointerek aritmetikája
    }

    for(p = t + 10; p - t < 20; p++)
    {
        printf("> t[%d] : ", ((int)p - (int)t)/(int)sizeof(int));
        scanf("%d", p);
    }

    p = t; ← az inkrementálás nagyobb prioritással bír
    i = *p++; ← i = *++p;
    i = ++*p; ← i = p - t;
    i = p - t; ← *p = *p + 1; i = *p;
    *p = *p + 1; i = *p;

    return 0; ← hány egész érték fér be p és t közé?
}
```

További részletek

bibliográfia

- K. N. King **C programming – A modern approach**, 2nd edition, W. W. Norton & Co., 2008
 - 11., 12. és 17. fejezet
- Deitel & Deitel **C How to Program**, 6th edition, Pearson, 2009
 - 7. fejezet

6508 8950



kvíz 5. kérdés