



BME
Budapesti Műszaki és Gazdaságtudományi Egyetem

HAUT
Közlekedésautomatikai Tanszék

Járműfedélzeti rendszerek II.

3. előadás

Dr. Aradi Szilárd

Függvények (ismétlés)

Rekurzív hívás magyarázat

Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedésautomatikai Tanszék

```
int fakt (int i)
{
    return (i<2)?1:i*fakt(i-1);
}
```

Példa: 3!

Ugyanannak a függvénynek hívja meg egy újabb példányát az aktuális argumentummal.

fakt(3): 3*fakt(2)

fakt(2): 2*fakt(1)

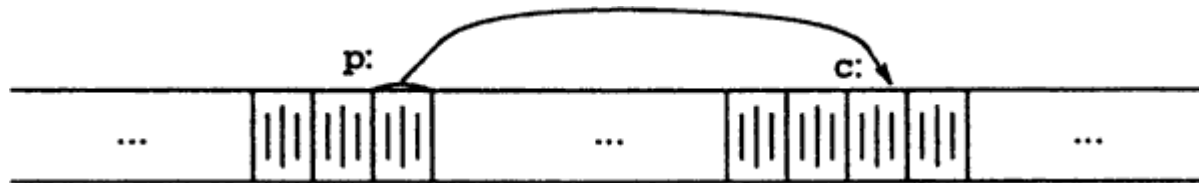
fakt(1): 1

Mutatók, tömbök

- A mutató vagy pointer olyan változó, amely egy másik változó címét tartalmazza. A C nyelvű programokban gyakran használják a mutatókat, egyrészt mert bizonyos feladatokat csak velük lehet megoldani, másrészt mert alkalmazásukkal sokkal tömörebb és hatékonyabb program hozható létre.

Mutatók és címek

- Az **&** unáris (egyoperandusú) operátor megadja egy operandus címét, ezért a $p = \&c;$ utasítás **c** változó címét hozzárendeli a **p** változóhoz és ilyenkor azt mondjuk, hogy **p** **c**-re „mutat”.



- A ***** unáris operátor neve *indirekció*, és ha egy mutatóra alkalmazzuk, akkor a mutató által megcímzett változóhoz férhetünk hozzá.

Mutatók és címek

Operátorok példa

Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedésautomatikai Tanszék

```
int x = 1, y = 2, z[10];
int *ip, *anotherp;
/* ip és anotherp int típusú mutatók */

ip = &x; /* ip x-re mutat */
anotherp = &x; /* anotherp is x-re mutat */
y = *ip; /* y most 1 lesz */
*ip = 0; /* most x nulla lesz */
*anotherp = 1; /* most x egy lesz */
ip = &z[0]; /* ip most z[0]-ra mutat */
```

Mutatók és címek

- Az indirekció alapján látható, hogy **egy mutató mindig meghatározott objektumra mutat**, azaz minden mutató meghatározott adattípust jelöl ki. (Ez alól csak egy **kivétel** van, a **void** típushoz tartozó mutató, ami egy olyan adat, amely bármilyen mutatót tartalmazhat. Erre az a megszorítás érvényes, hogy önmagára nem alkalmazhatja az indirekciót.

Void mutató

```
int x = 1;
void *ptr;
ptr = &x;
printf("%d", *ptr); /* FORDÍTÁSI HIBA!!! */

printf("%d", *(int *)ptr); /* OK! */
```

Mutatók és címek

Precedencia példa

Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedésautomatikai Tanszék

```
int *ip
```

```
*ip = *ip + 10; // *ip-t tízzel növeli
```

Az `&` és `*` unáris operátorok szorosabban kötnek, mint az aritmetikai operátorok, ezért az

```
y = *ip + 1;
```

kifejezés kiértékelésekor a gép először veszi azt az adatot, amire `ip` mutat, hozzáad egyet, majd az eredményt hozzárendeli `y`-hoz.

Mutatók és címek

Precedencia példa

Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedésautomatikai Tanszék

Az inkrementáló és dekrementáló operátorokkal egy szinten vannak és jobbról balra értékelődnek ki.

Az `*ip += 1;` inkrementálja azt a változót, amire `ip` mutat, csakúgy, mint a `++*ip;`.

Viszont `*ip++` a címet növeli!

Megoldás: `(*ip)++;`

Mutatók és Függvényargumentumok

Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedésautomatikai Tanszék

Mivel a C nyelv a függvényeknek érték szerint adja át az argumentumokat, így a hívott függvény nem tudja megváltoztatni a hívó függvény változóit.

Az ilyen argumentumok lokális változóként működnek.

```
int a=1,b=2;  
swap(a, b);
```

```
void swap(int x, int y)    /* Hibás!!! */  
{ int temp= x;  
  x = y;  
  y = temp;  
}
```

Mutatók és Függvényargumentumok

Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedésautomatikai Tanszék

Mivel a függvényt érték szerint hívjuk, a swap nem képes a hívásban szereplő a és b argumentumokat befolyásolni (azoknak csak egy helyi másolatával dolgozik, ezek cseréje pedig nem befolyásolná az eredeti argumentumok értékét).

```
int a=1,b=2;  
swap(&a, &b);
```

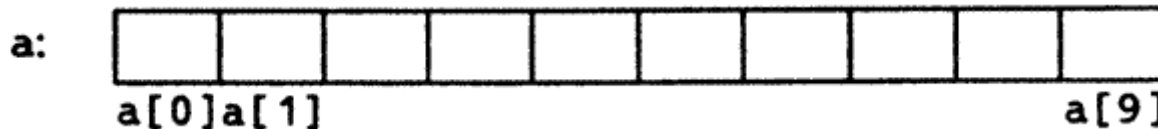
```
void swap(int *px, int *py) /* Helyes */  
{ int temp=*px;  
  *px = *py;  
  *py = temp;  
}
```

Mutatók és tömbök

A C nyelvben a mutatók és a tömbök között szoros kapcsolat van. Az

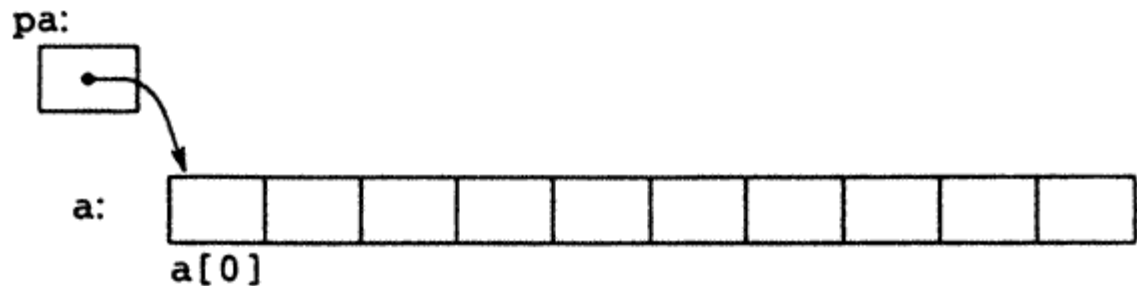
```
int a[10];
```

deklaráció egy tízelemű tömböt jelöl ki, azaz tíz egymást követő, $a[0] \dots a[9]$ névvel ellátott objektumot.



Mutatók és tömbök

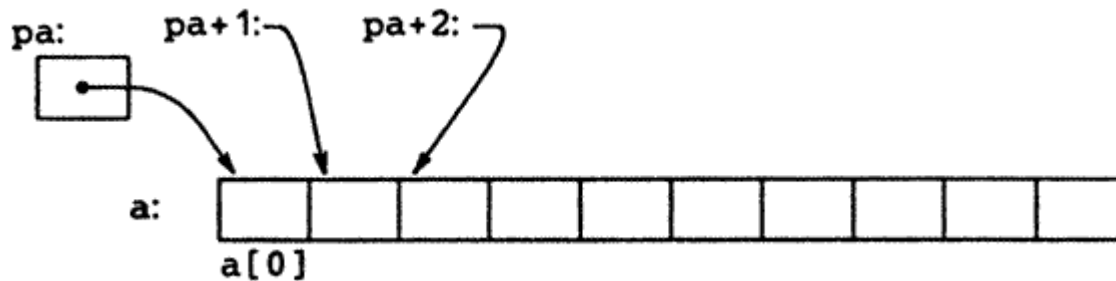
- Az `a[i]` jelölés a tömb *i*-edik elemére hivatkozik. Ha `pa` egy egész típushoz tartozó mutató, amit `int *pa;` módon deklaráltunk, akkor a `pa = &a[0];` értékadás hatására `pa` az `a` tömb nulladik elemére fog mutatni, vagyis `pa` az `a[0]` címét fogja tartalmazni.



Mutatók és tömbök

Ennek megfelelően az $x = *pa$; értékeadás $a[0]$ értékét másolja x -be.

A $*(pa+1)$ $a[i]$ -ik elemére mutat



$a[0]$ és a ugyanaz a mutató, ezért:

$pa = \&a[0]$; ekvivalens a $pa = a$; kifejezéssel.

$a[i]$ ekvivalens $*(a+i)$ hivatkozással.

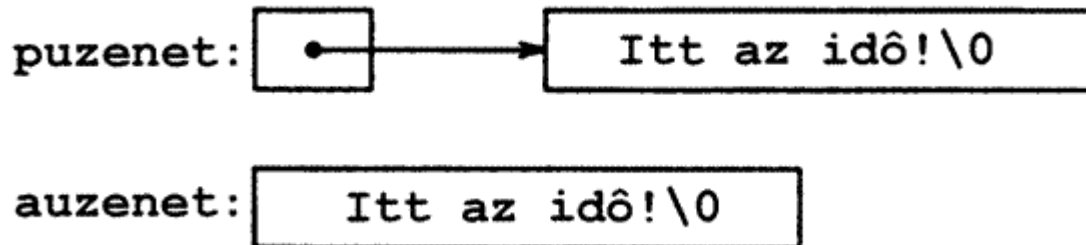
Címaritmetika

- Ha **p** egy tömb valamelyik elemének mutatója, akkor **p++** inkrementálja a **p** mutatót, hogy az a tömb következő elemére mutasson és **p+=i** pedig úgy növeli **p**-t, hogy az az aktuális elem utáni *i*-edik elemre mutasson.
Mutatók esetén általában igaz, hogy az **int** típusra alkalmazható operátorok alkalmazhatóak rá. (Bár bizonyos eseteknek nem nagyon van értelmük (pld.: /, %))

Karaktermutatók és függvények

Az "Ez egy karaktersorozat" alakban írt karaktersorozat állandók valójában karakterekből álló tömbök, amelyeket a belső ábrázolásban egy null-karakter ('`\0`') zár.

```
char auzenet[] = "Itt az ido"; /* ez egy tömb */  
char *puzenet = "Itt az ido"; /* ez egy mutató */
```



Ennek megfelelően „stringek” esetén az `s=t`; utasítás csak a pointert másolja a karaktereket nem helyezi el új tömbben, ez külön ciklussal kell megoldani.

Karaktermutatók és függvények

```
void strcpy(char *s, char *t)
{
    int i = 0;
    while ((s[i] = t[i]) != '\0') i++;
}
```

vagy egyszerűbben:

```
void strcpy(char *s, char *t)
{
    while(*s++ = *t++ != '\0') ;
}
```

Karaktermutatók és függvények

- Az `strcpy` függvény a standard könyvtárban (a `<string.h>` headerben) található, és visszatérési értéke az átmásolt karaktersorozat.
- A fent ismertetett megoldás nem foglalkozik azonban a különböző méretű tömbök problémakörével.
- A karaktersorozat kezelésére egy későbbi előadásban részletesen kitérünk.

Dinamikus memóriafoglalás

Adott méretű memóriaterület lefoglalása futásidőben.

- Előnye a hatékonyabb memóriakezelés
- Hátránya, hogy több figyelmet igényel
- A méret (size) bájtban kerül megadásra

```
void * malloc ( size_t size );
```

A lefoglalt memóriaterület nincs inicializálva.

```
void * calloc ( size_t num, size_t size );
```

A lefoglalt memóriaterület nullázásra kerül.

```
void * realloc ( void * ptr, size_t size );
```

Módosítja egy már lefoglalt memóriaterület hely méretét

```
void free ( void * ptr );
```

Felszabadítja a lefoglalt memóriaterületet.

Példa dinamikus memórfoglalásra

Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedésautomatikai Tanszék

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{ unsigned m,i;
  double *t;
  printf("Mekkora tomb kell? ");
  scanf("%u",&m); // memórfoglalás
  t=(double*)malloc(m*sizeof(double));
  if(t==NULL)
  {
    printf("Sikertelen allokálás.\n"); exit(1);
  } // Innentől úgy használjuk, mint egy közönséges tömböt.
  printf("\nt merete: %d \n",sizeof t);
  for(i=0;i<m;i++) t[i]=i;
  for(i=m-1;i+1!=0;i--) printf("%g ",t[i]);
  t=(double*)realloc(t,2*m*sizeof(double)); // 2x akkora helyet foglalunk
  free(t); // felszabadítás, ha már nem kell
}
```

```

#include <stdio.h>
#include <stdlib.h>
void kiir0(int a[]);
int main(int argc, char *argv[])
{
    int a[10],i;
    for (i=0;i<sizeof(a)/sizeof(i);i++)
        a[i]=i*2;
    for (i=0;i<sizeof(a)/sizeof(i);i++)
        printf("%d ",a[i]);
    printf("\n");
    kiir0(a);
    return 0;
}
void kiir0(int a[]) // HIBÁS!!!!
{
    int i;
    for (i=0;i<sizeof(a)/sizeof(i);i++)
        printf("%d ",a[i]);
}

```

```

#include <stdio.h>
#include <stdlib.h>
void kiir1(int a[],int n);
int main(int argc, char *argv[])
{
    int a[10],i;
    for (i=0;i<sizeof(a)/sizeof(i);i++)
        a[i]=i*2;
    for (i=0;i<sizeof(a)/sizeof(i);i++)
        printf("%d ",a[i]);
    printf("\n");
    kiir1(a, sizeof(a)/sizeof(i));
    return 0;
}
void kiir1(int a[],int n)
{
    int i;
    for (i=0;i<n;i++)
        printf("%d ",a[i]);
}

```

```

#include <stdio.h>
#include <stdlib.h>
void kiir2(int *a, int n);
int main(int argc, char *argv[])
{
    int a[10],i;
    for (i=0;i<sizeof(a)/sizeof(i);i++)
        a[i]=i*2;
    for (i=0;i<sizeof(a)/sizeof(i);i++)
        printf("%d ",a[i]);
    printf("\n");
    kiir2(a, sizeof(a)/sizeof(i));
    return 0;
}
void kiir2(int *a,int n)
{
    int i;
    for (i=0;i<n;i++)
        printf("%d ",*(a+i));
}

```

Vége

Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedésautomatikai Tanszék

Köszönöm a figyelmet!