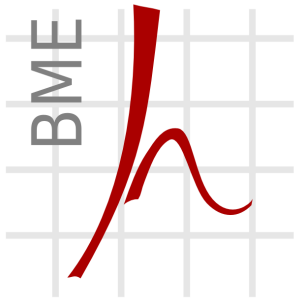
Mutatók – Sztringek

A programozás alapjai I.



Hálózati Rendszerek és Szolgáltatások Tanszék
Farkas Balázs, Fiala Péter, Vitéz András, Zsóka Zoltán

2018. október 8.

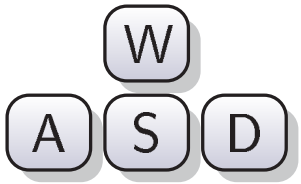
Tartalom

1. fejezet

A felsorolt típus

A felsorolt típus – Motiváció

- Mászkálós játékprogramot írunk, melyben a felhasználó a játékos mozgását négy billentyűvel vezérli.



- A felhasználói input beolvasására sokszor szükség van, ezért ezt a műveletet célszerűen egy `read_direction()` függvényre bízzuk
- A függvény a billentyűzetről olvas, majd visszaadja a haladási irányt a hívó programrésznek.
- Milyen típust adjon vissza a függvény?

A felsorolt típus – Motiváció

- 1. javaslat: Adja vissza a leütött karaktert (’a’,’s’,’w’,’d’):

```c
char read_direction(void)
{
  char ch;
  scanf("%c", &ch);
  return ch;
}
```

link

A felsorolt típus – Motiváció

- 1. javaslat: Adja vissza a leütött karaktert (’a’,’s’,’w’,’d’):

```c
char read_direction(void)
{
  char ch;
  scanf("%c", &ch);
  return ch;
}
```

link

- Problémák:
  - A program többi részén (sok helyen) kell dekódolnunk a karakterekből az irányokat.
  - Ha a programot átírjuk ← ↓ ↑ → vezérlésre, ezer helyen kell módosítanunk.

A felsorolt típus – Motiváció

- 1. javaslat: Adja vissza a leütött karaktert (’a’,’s’,’w’,’d’):

```
1  char read_direction(void)
2  {
3    char ch;
4    scanf("%c", &ch);
5    return ch;
6  }
```

link

- Problémák:
  - A program többi részén (sok helyen) kell dekódolnunk a karakterekből az irányokat.
  - Ha a programot átírjuk ← ↓ ↑ → vezérlésre, ezer helyen kell módosítanunk.
- Megoldás:
  - Helyben kell dekódolnunk, és csak az irányt kell visszaadnunk.
  - De azt milyen formában?

A felsorolt típus – Motiváció

- 2. javaslat: Adjon vissza 0,1,2,3 `int` értékeket:

'a'	0	←
'w'	1	↑
'd'	2	→
's'	3	↓

```c
int read_direction(void) {
  char ch;
  scanf("%c", &ch);
  switch (ch) {
  case 'a': return 0; /* bal */
  case 'w': return 1; /* fel */
  case 'd': return 2; /* jobb */
  case 's': return 3; /* le */
  }
  return 0; /* bal default :) */
}
```

A felsorolt típus – Motiváció

- 2. javaslat: Adjon vissza 0,1,2,3 `int` értékeket:

'a'	0	←
'w'	1	↑
'd'	2	→
's'	3	↓

```c
int read_direction(void) {
  char ch;
  scanf("%c", &ch);
  switch (ch) {
  case 'a': return 0; /* bal */
  case 'w': return 1; /* fel */
  case 'd': return 2; /* jobb */
  case 's': return 3; /* le */
  }
  return 0; /* bal default :) */
}
```

- Probléma:
  - A program többi részén a 0-3 számokat kell használnunk az irányokra, a programozónak emlékeznie kell a szám-irány összerendelésre.

A felsorolt típus – Megoldás

- Egy `direction` nevű típusra van szükségünk, amely a `LEFT`, `RIGHT`, `UP`, `DOWN` értékeket tudja tárolni.
- C-ben csinálhatunk ilyet!

  A megfelelő felsorolt típus (enumerated type, `enum`) deklarációja:

```
1  enum direction {LEFT, RIGHT, UP, DOWN};
```

A felsorolt típus – Megoldás

- Egy `direction` nevű típusra van szükségünk, amely a `LEFT`, `RIGHT`, `UP`, `DOWN` értékeket tudja tárolni.
- C-ben csinálhatunk ilyet!

  A megfelelő felsorolt típus (enumerated type, `enum`) deklarációja:

```
1 enum direction {LEFT, RIGHT, UP, DOWN};
```

- A típus használata

```
1 enum direction d;
2 d = LEFT;
```

A felsorolt típus – Megoldás

- A végleges megoldás az új típussal

```
1  enum direction {LEFT, RIGHT, UP, DOWN};
2  typedef  enum direction  direction; /* egyszerűsítés */
3
4  direction read_direction(void)
5  {
6    char ch;
7    scanf("%c", &ch);
8    switch (ch)
9    {
10   case 'a': return LEFT;
11   case 'w': return UP;
12   case 'd': return RIGHT;
13   case 's': return DOWN;
14   }
15   return LEFT;
16 }
```

link

A felsorolt típus – Megoldás

- És a függvény használata:

```
1  direction d = read_direction();
2  if (d == RIGHT)
3    printf("Megevett egy tigris\n");
```

link

A felsorolt típus – Megoldás

- És a függvény használata:

```
1 direction d = read_direction();
2 if (d == RIGHT)
3   printf("Megevett egy tigris\n");
```
link

- Ugyanez a felsorolt típus nélkül ilyen lenne:

```
1 int d = read_direction();
2 if (d == 2) /* "magic" konstans, mit is jelent? */
3   printf("Megevett egy tigris\n");
```
link

A felsorolt típus – Megoldás

- És a függvény használata:

```
1 direction d = read_direction();
2 if (d == RIGHT)
3   printf("Megevett egy tigris\n");
```
link

- Ugyanez a felsorolt típus nélkül ilyen lenne:

```
1 int d = read_direction();
2 if (d == 2) /* "magic" konstans, mit is jelent? */
3   printf("Megevett egy tigris\n");
```
link

- A felsorolt típus...
  - beszédes kóddal helyettesíti a „magic konstansokat”,
  - a tartalomra koncentrál az ábrázolás helyett,
  - magasabb szintű programozást tesz lehetővé.

A felsorolt típus – Definíció

A felsorolt (enum) típus

Szimbolikus néven hivatkozott egész típusú állandók összefogása egy típussá.

```
enum [<felsorolás címke>]opt
{ <felsorolás lista> }
[<változó azonosítók>]opt ;
```

A felsorolt típus – Definíció

A felsorolt (enum) típus

Szimbolikus néven hivatkozott egész típusú állandók összefogása egy típussá.

```
enum [<felsorolás címke>]opt
{ <felsorolás lista> }
[<változó azonosítók>]opt ;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

A felsorolt típus – Definíció

A felsorolt (enum) típus

Szimbolikus néven hivatkozott egész típusú állandók összefogása egy típussá.

```
enum [<felsorolás címke>]opt
{ <felsorolás lista> }
[<változó azonosítók>]opt ;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

A felsorolt típus – Definíció

A felsorolt (enum) típus

Szimbolikus néven hivatkozott egész típusú állandók összefogása egy típussá.

```
enum [<felsorolás címke>]opt
{ <felsorolás lista> }
[<változó azonosítók>]opt ;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

A felsorolt típus – Definíció

A felsorolt (enum) típus

Szimbolikus néven hivatkozott egész típusú állandók összefogása egy típussá.

```
enum [<felsorolás címke>]opt
{ <felsorolás lista> }
[<változó azonosítók>]opt ;
```

```
1 enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

enum példák

```
1  enum month {
2    JAN,   /*  0 */
3    FEB,   /*  1 */
4    MAR,   /*  2 */
5    APR,   /*  3 */
6    MAY,   /*  4 */
7    JUNE,  /*  5 */
8    JULY,  /*  6 */
9    AUG,   /*  7 */
10   SEPT,  /*  8 */
11   OCT,   /*  9 */
12   NOV,   /* 10 */
13   DEC    /* 11 */
14 };
15
16 enum month m=OCT; /*9*/
```

enum példák

```c
enum month {
  JAN,   /*  0 */
  FEB,   /*  1 */
  MAR,   /*  2 */
  APR,   /*  3 */
  MAY,   /*  4 */
  JUNE,  /*  5 */
  JULY,  /*  6 */
  AUG,   /*  7 */
  SEPT,  /*  8 */
  OCT,   /*  9 */
  NOV,   /* 10 */
  DEC    /* 11 */
};

enum month m=OCT; /*9*/
```

```c
enum {
  RED,         /*  0 */
  BLUE = 3,    /*  3 */
  GREEN,       /*  4 */
  YELLOW,      /*  5 */
  GRAY = 10    /* 10 */
} c;

c = GREEN;
printf("c: %d\n", c);
```

```
c:  4
```

2. fejezet

Mutatók

Fundamental Theorem of Software Engineering (FTSE)

„We can solve any problem
by introducing an extra level of indirection.”
Andrew Koenig

Hol vannak a változók?

Írjunk programot, mely kilistázza változók címét és értékét

```
1  int a = 2;
2  double b = 8.0;
3  printf("a címe: %p, értéke: %d\n", &a, a);
4  printf("b címe: %p, értéke: %f\n", &b, b);
```

```
a címe:  0x7fffa3a4225c, értéke:  2
b címe:  0x7fffa3a42250, értéke:  8.000000
```

[1] általánosabban balérték

Hol vannak a változók?

Írjunk programot, mely kilistázza változók címét és értékét

```
1 int a = 2;
2 double b = 8.0;
3 printf("a címe: %p, értéke: %d\n", &a, a);
4 printf("b címe: %p, értéke: %f\n", &b, b);
```

```
a címe:  0x7fffa3a4225c, értéke:  2
b címe:  0x7fffa3a42250, értéke:  8.000000
```

- változó címe: a változót tartalmazó „memóriarekesz” kezdőcíme bájtokban mérve
- a címképzés operátorával tetszőleges változó [1] címe képezhető &<balérték> formában

[1] általánosabban balérték

A mutató típus

memóriacímek tárolására való

Mutató (pointer) deklarációja

```
<mutatott típus> * <azonosító>;
```

```
1  int*    p;  /*   p egy int adat címét tárolja      */
2  double* q;  /*   q egy double adat címét tárolja   */
3  char*   r;  /*   r egy char adat címét tárolja     */
```

A mutató típus

memóriacímek tárolására való

Mutató (pointer) deklarációja

```
<mutatott típus> * <azonosító>;
```

```
1  int*    p;  /*   p egy int adat címét tárolja      */
2  double* q;  /*   q egy double adat címét tárolja   */
3  char*   r;  /*   r egy char adat címét tárolja     */
```

másként tördelve is ugyanaz

```
1  int    *p;  /*   p egy int adat címét tárolja      */
2  double *q;  /*   q egy double adat címét tárolja   */
3  char   *r;  /*   r egy char adat címét tárolja     */
```

Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”

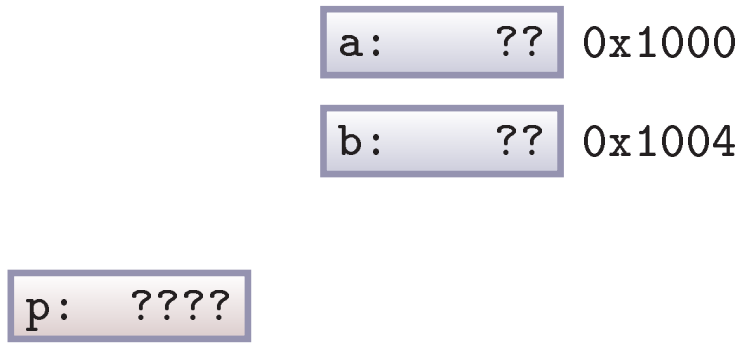
Az indirekció operátora

- Ha a `p` mutató az `a` változó címét tartalmazza, akkor `p` „a-ra mutat”
- Ha `p` a-ra mutat, akkor az `a` változó `*p`-ként elérhető. Itt `*` az indirekció operátora (dereferencia operátor).

Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

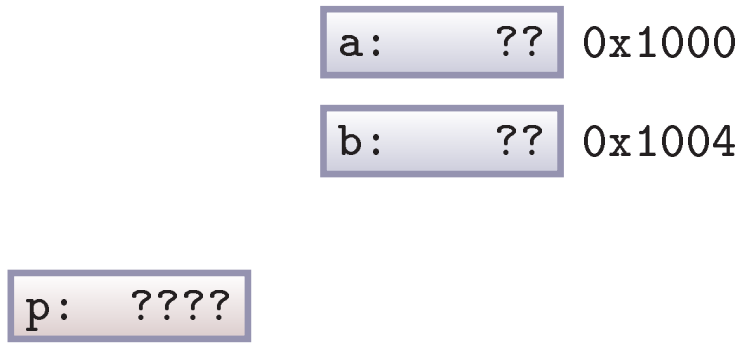
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

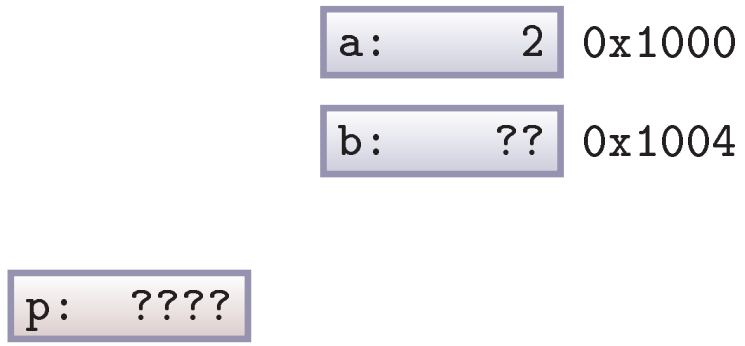
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

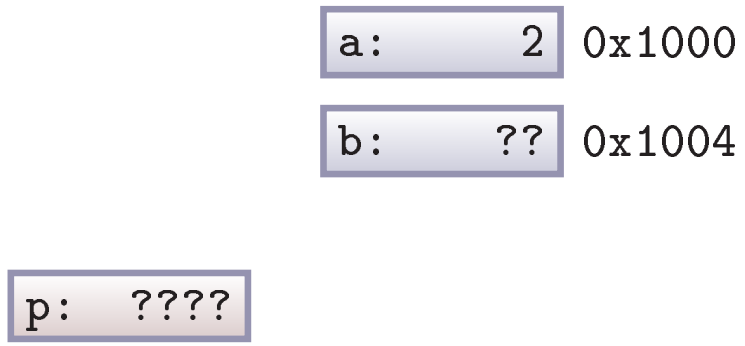
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

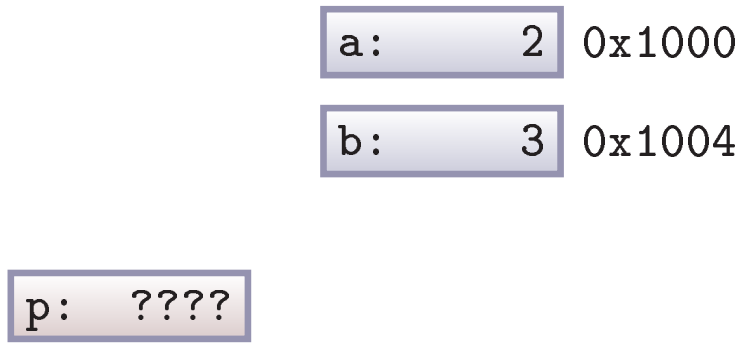
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

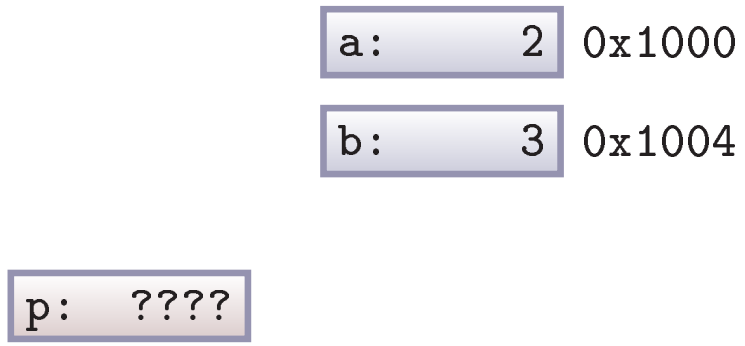
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

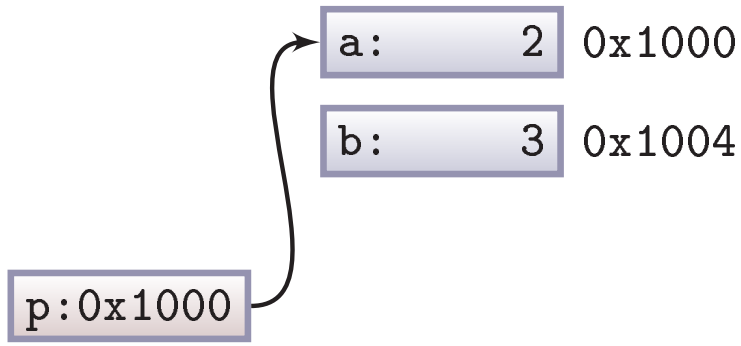
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

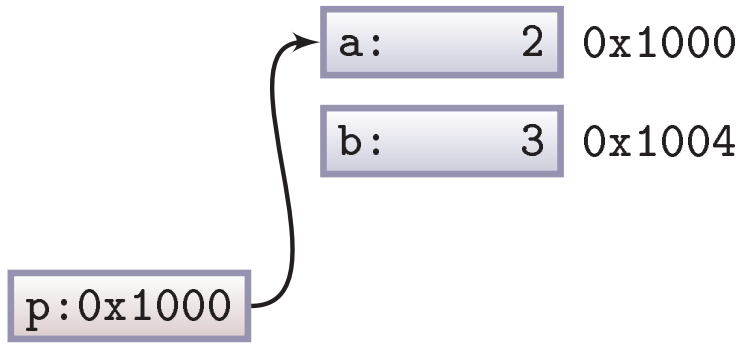
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

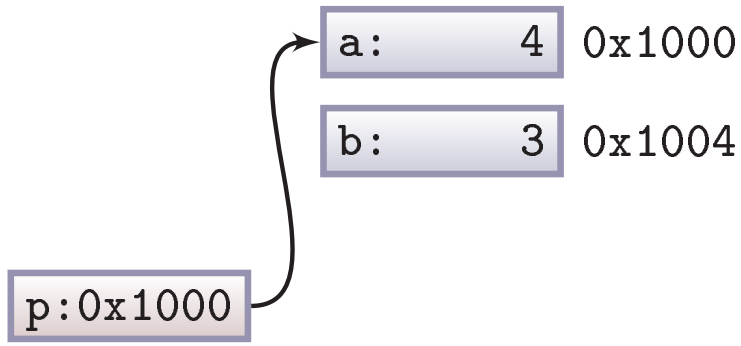
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

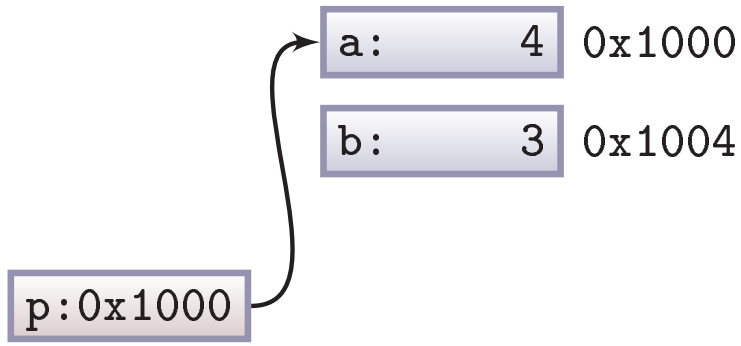
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

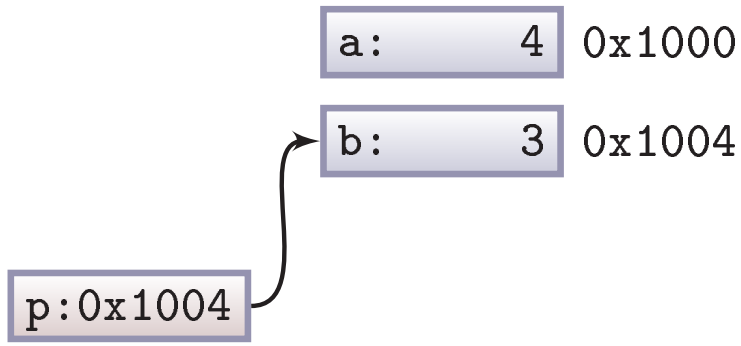
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

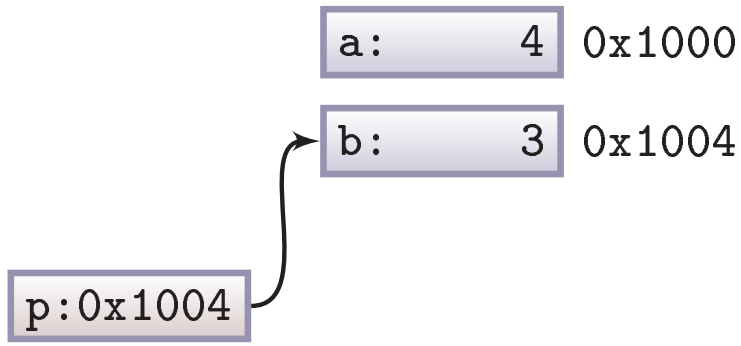
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

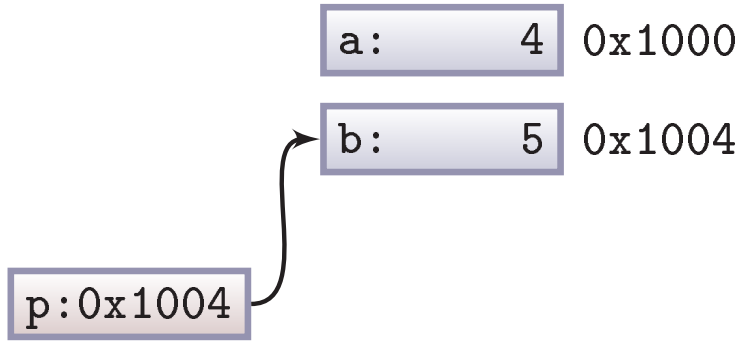
```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Az indirekció operátora

- Ha a p mutató az a változó címét tartalmazza, akkor p „a-ra mutat”
- Ha p a-ra mutat, akkor az a változó *p-ként elérhető. Itt * az indirekció operátora (dereferencia operátor).

```
1  int a, b;
2  int *p; /* int pointer */
3
4  a = 2;
5  b = 3;
6  p = &a; /* p a-ra mutat */
7  *p = 4; /* a = 4 */
8  p = &b; /* p b-re mutat */
9  *p = 5; /* b = 5 */
```



Címképzés és indirekció – összefoglalás

operátor	művelet	leírás
&	címképzés	változóhoz a címét rendeli
*	indirekció	címhez a változót rendeli

Címképzés és indirekció – összefoglalás

operátor	művelet	leírás
&	címképzés	változóhoz a címét rendeli
*	indirekció	címhez a változót rendeli

- Deklaráció értelmezése: `*p` `int` típusú

```
1  int *p;        /* ezt szokjuk meg */
```

Címképzés és indirekció – összefoglalás

operátor	művelet	leírás
&	címképzés	változóhoz a címét rendeli
*	indirekció	címhez a változót rendeli

- Deklaráció értelmezése: `*p` `int` típusú

```
1 int *p;       /* ezt szokjuk meg */
```

- Többszörös deklaráció: a, `*p` és `*q` `int` típusúak

```
1 int a, *p, *q; /* már csak ezért is */
```

Alkalmazás – Függvény két változó cseréjére

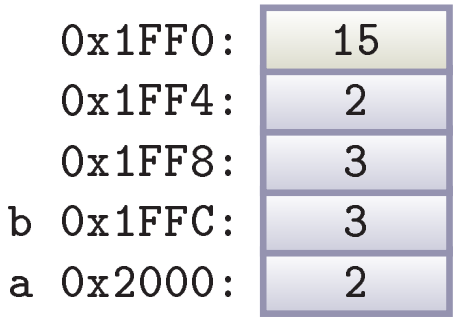
```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```

Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```

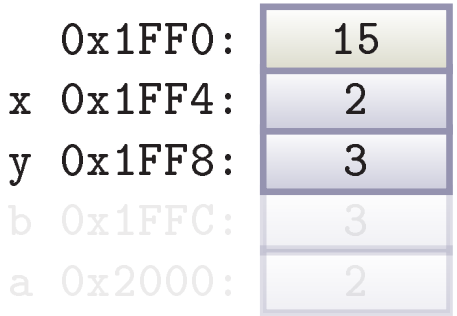
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



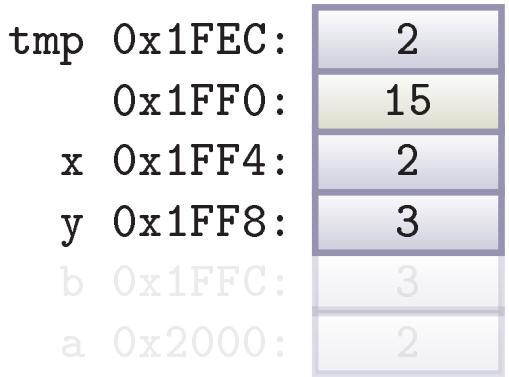
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



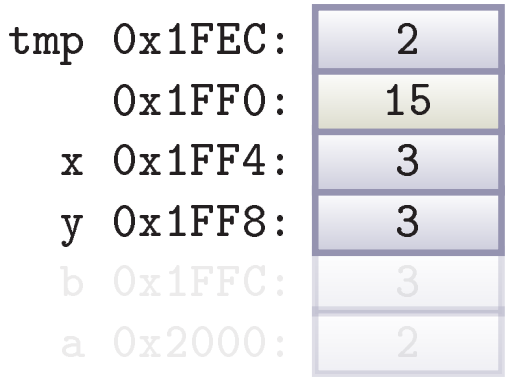
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



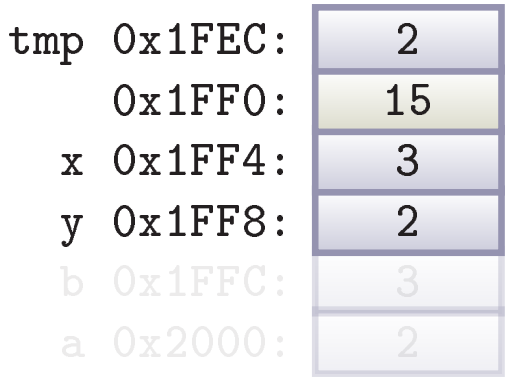
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



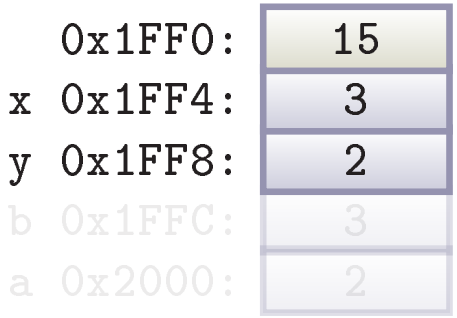
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



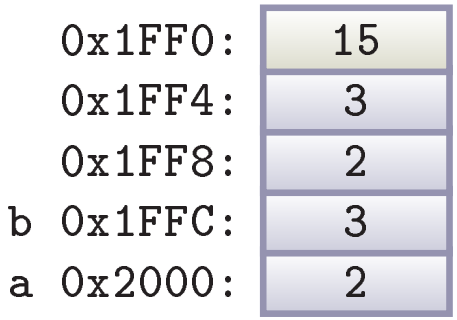
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



Alkalmazás – Függvény két változó cseréjére

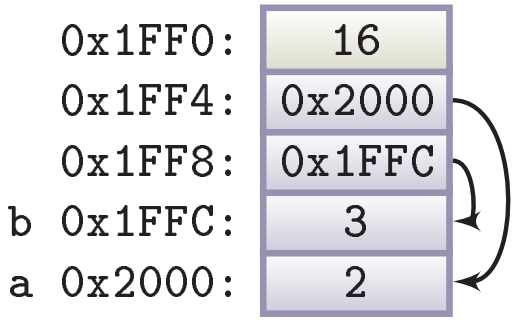
```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```

Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```

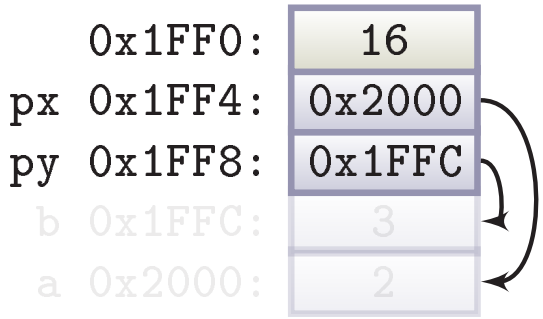
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



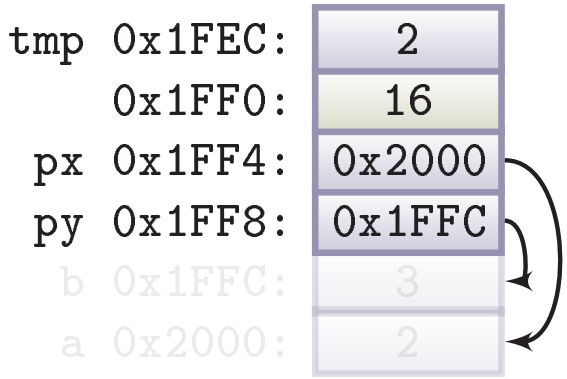
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



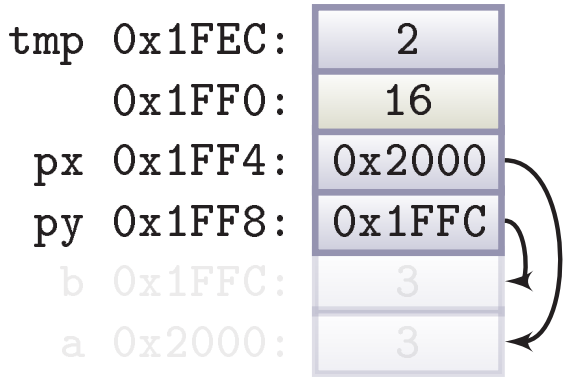
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



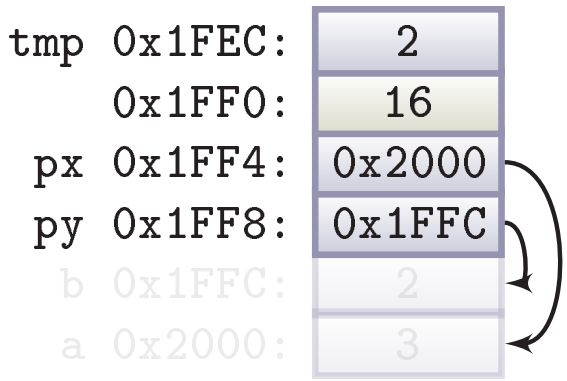
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



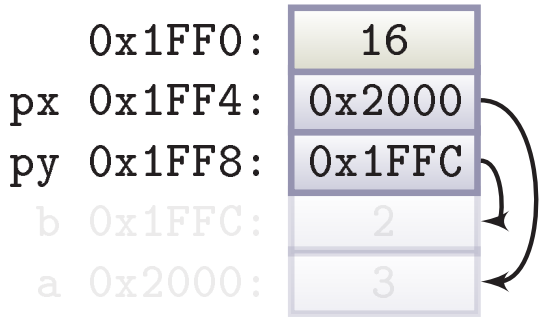
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



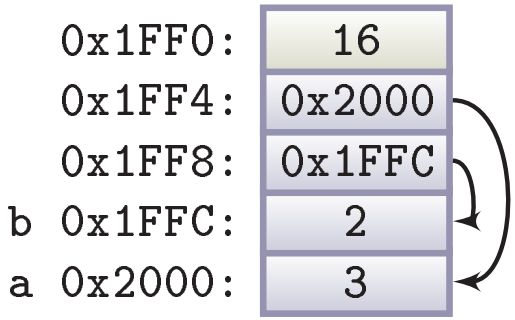
Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```



Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```

Alkalmazás – Függvény két változó cseréjére

```
1  void xchg(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6
7  void xchgp(int *px, int *py) {
8    int tmp = *px;
9    *px = *py;
10   *py = tmp;
11 }
12
13 int main(void) {
14   int a = 2, b = 3;
15   xchg(a, b);    /* nem cserél*/
16   xchgp(&a, &b);/* cserél */
17   return 0;
18 }
```

Alkalmazás – paraméterlistán visszaadott értékek

- Ha egy függvénynek több adatot kell kiszámolnia, akkor...
  ...alkalmazhatunk struktúrákat, de ez sokszor erőltetett.

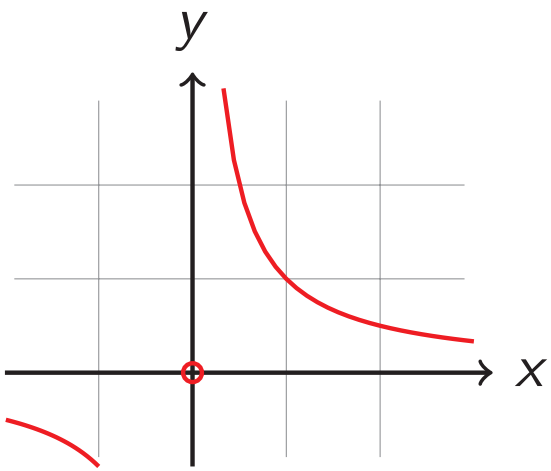
Alkalmazás – paraméterlistán visszaadott értékek

- Ha egy függvénynek több adatot kell kiszámolnia, akkor...

  ...alkalmazhatunk struktúrákat, de ez sokszor erőltetett. Inkább...

```
1  int inverse(double x, double *py)
2  {
3    if (abs(x) < 1e-10) return 0;
4    *py = 1.0 / x;
5    return 1;
6  }
```

link



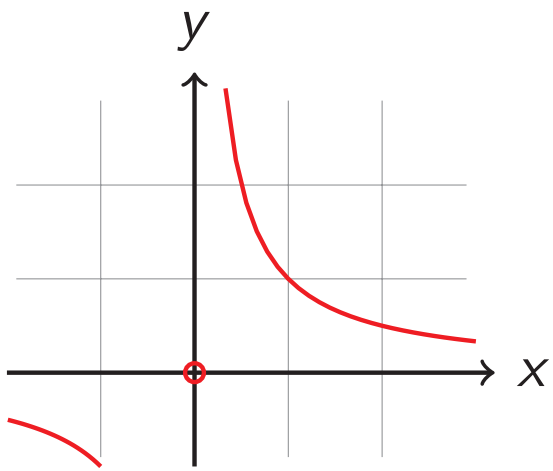
Alkalmazás – paraméterlistán visszaadott értékek

- Ha egy függvénynek több adatot kell kiszámolnia, akkor...

  ...alkalmazhatunk struktúrákat, de ez sokszor erőltetett. Inkább...

```
1  int inverse(double x, double *py)
2  {
3    if (abs(x) < 1e-10) return 0;
4    *py = 1.0 / x;
5    return 1;
6  }
```

link




```
1  double y;          /* helyfoglalás az eredménynek */
2  int success = inverse(5.0, &y);
3  if (success)
4    printf("%f reciproka %f\n", 5.0, y);
5  else
6    printf("Nem képezhető a reciprok");
```

link

Alkalmazás – paraméterlistán visszaadott értékek

- Most már értjük, mit jelent az, hogy

```
1 int n, p;
2 /* paraméterlistán visszaadott értékek */
3 scanf("%d%d", &n, &p); /* a címeket adjuk át */
```

Megjegyzések:

- Miért jó, hogy különböző típusok címei különböző típusúak?

Megjegyzések:

- Miért jó, hogy különböző típusok címei különböző típusúak?
- Típus = értékkészlet + műveletek
- Az értékkészlet nyilván minden mutatóra ugyanaz (előjel nélküli egész címek)
- A műveletek eltérőek!

Megjegyzések:

- Miért jó, hogy különböző típusok címei különböző típusúak?
- Típus = értékkészlet + műveletek
- Az értékkészlet nyilván minden mutatóra ugyanaz (előjel nélküli egész címek)
- A műveletek eltérőek!
- Az indirekció (*) operátor
  - `int` pointerből `int`-et
  - `char` pointerből `char`-t képez

Megjegyzések:

- Miért jó, hogy különböző típusok címei különböző típusúak?
- Típus = értékkészlet + műveletek
- Az értékkészlet nyilván minden mutatóra ugyanaz (előjel nélküli egész címek)
- A műveletek eltérőek!
- Az indirekció (*) operátor
  - `int` pointerből `int`-et
  - `char` pointerből `char`-t képez
- Egyéb műveletbeli különbségek a mutatóaritmetikában...

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

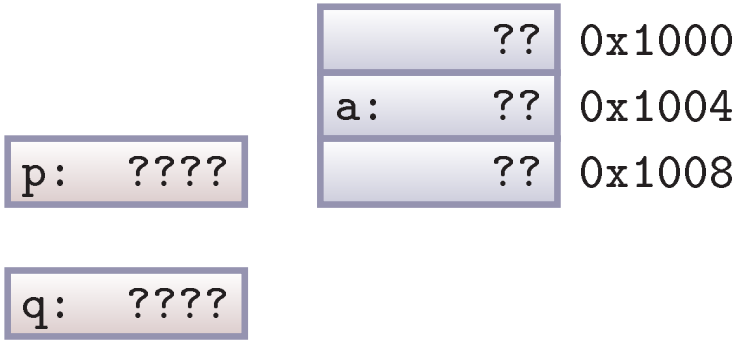
[2] A példában feltételezzük, hogy `int` 4 bájtos

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```



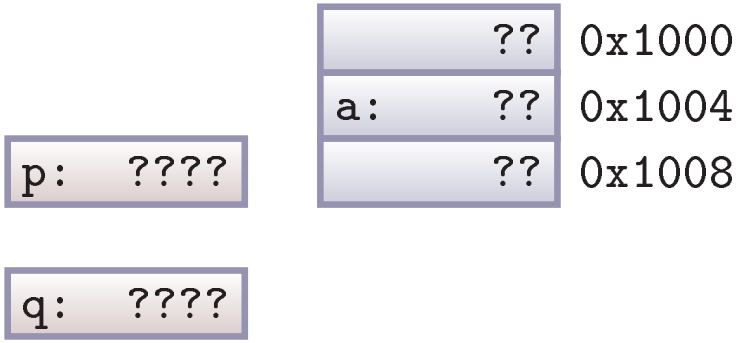
[2] A példában feltételezzük, hogy int 4 bájtos

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```



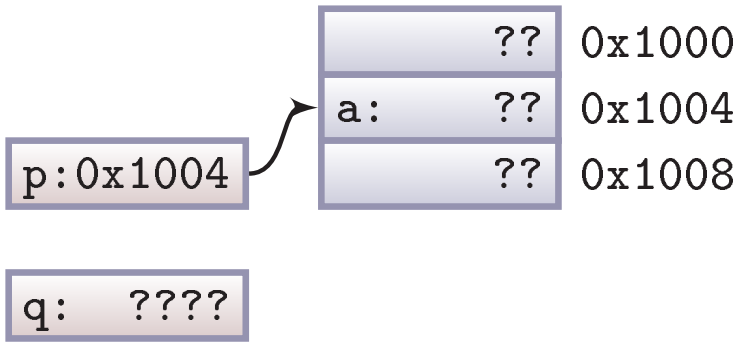
[2] A példában feltételezzük, hogy int 4 bájtos

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```



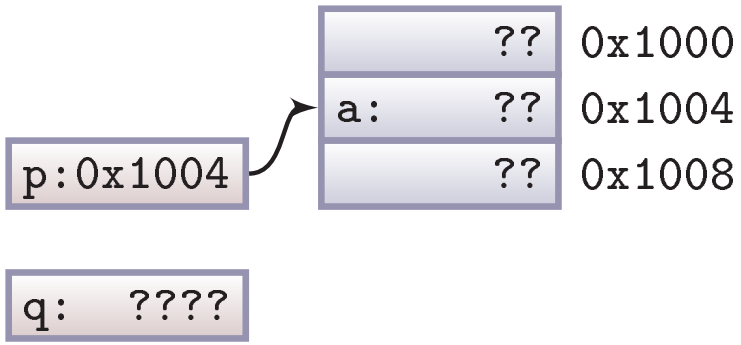
[2] A példában feltételezzük, hogy int 4 bájtos

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```



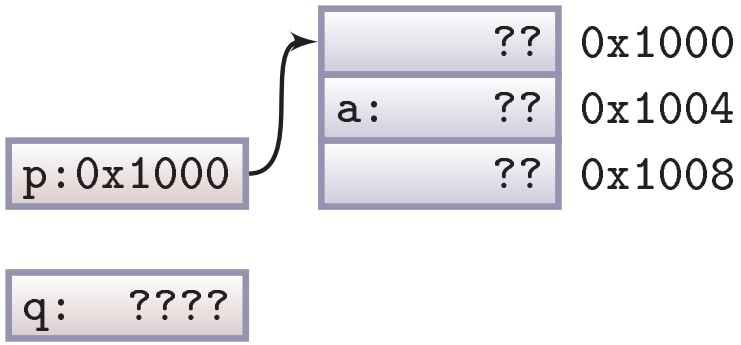
[2] A példában feltételezzük, hogy `int` 4 bájtos

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```



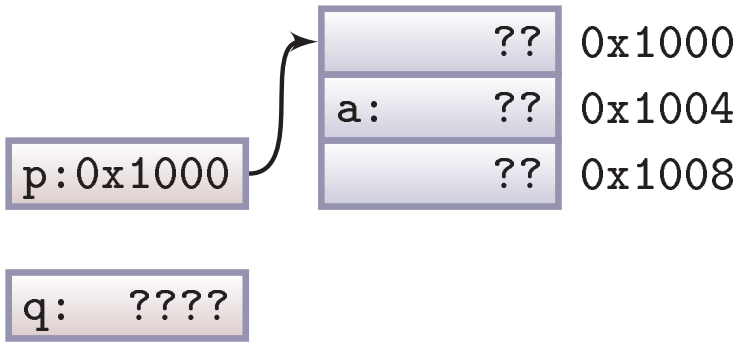
[2] A példában feltételezzük, hogy `int` 4 bájtos

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```



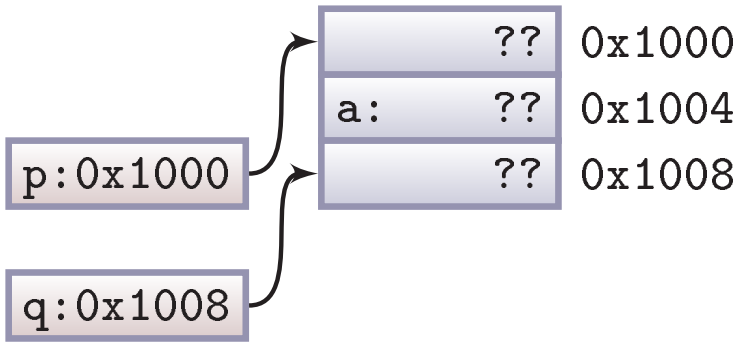
[2] A példában feltételezzük, hogy `int` 4 bájtos

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```



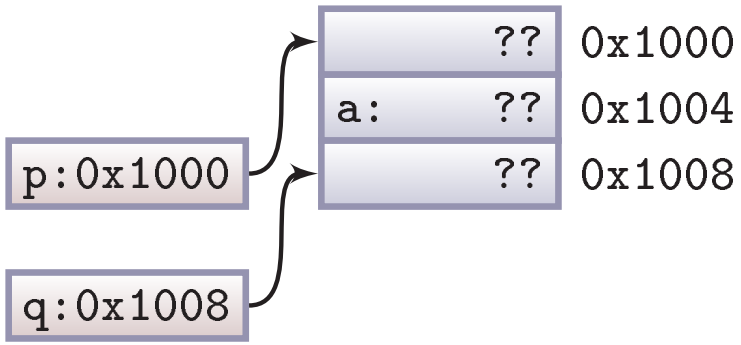
[2] A példában feltételezzük, hogy `int` 4 bájtos

Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```



[2] A példában feltételezzük, hogy int 4 bájtos

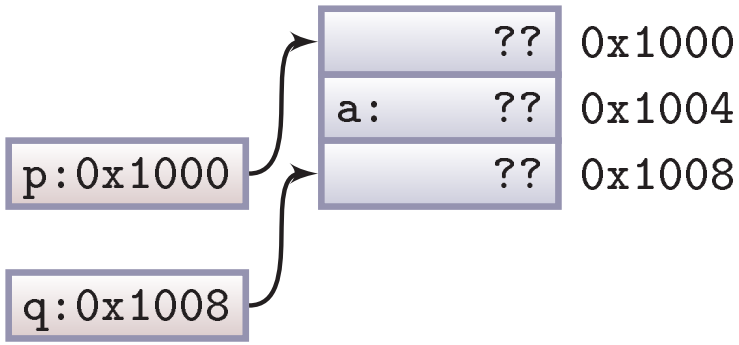
Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```

```
2
```



[2] A példában feltételezzük, hogy int 4 bájtos

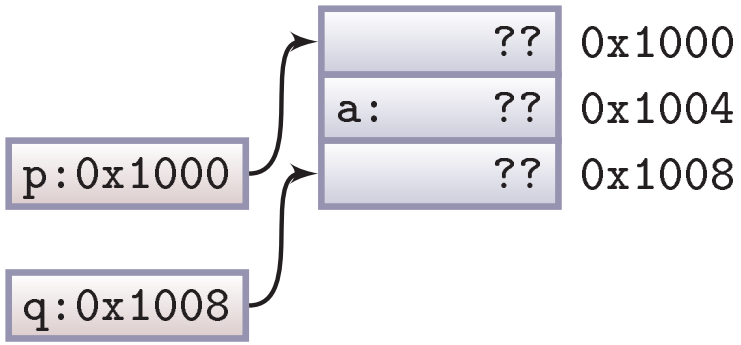
Mutatóaritmetika

Ha p és q azonos típusú mutatók, akkor

kif.	típus	jelentés
p+1	mutató	a következő elemre mutat
p-1	mutató	az előző elemre mutat
q-p	egész szám	két cím közötti elemek számát adja meg

```
1  int a, *p, *q;
2
3  p = &a;
4  p = p-1;
5  q = p+2;
6  printf("%d", q-p);
```

```
2
```



- Mutatóaritmetikai műveleteknél a címeket nem bájtban, hanem a mutatott típus ábrázolási méretében mérjük[2]

[2] A példában feltételezzük, hogy `int` 4 bájtos

Mutatóaritmetika

- A fenti példában a mutatóaritmetikának nincs sok értelme, hiszen nem tudhatjuk, mi áll az a változó előtt vagy mögött.
- A művelet ott nyer értelmet, ahol a memóriában egymást követő, azonos típusú változók helyezkednek el.
- Ezek a tömbök.

Mutatók és tömbök

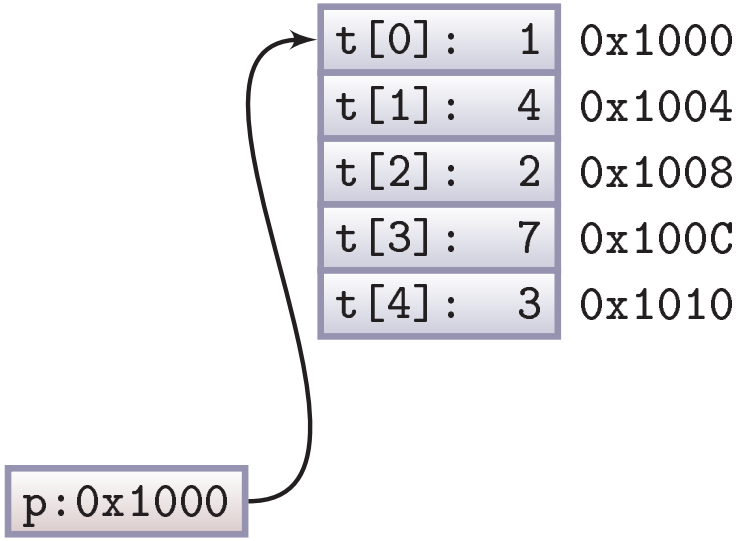
- Tömb bejárása lehetséges mutatóaritmetika alkalmazásával

Mutatók és tömbök

- Tömb bejárása lehetséges mutatóaritmetika alkalmazásával

```
1  int t[5] = {1,4,2,7,3};
2  int *p, i;
3
4  p = &t[0];
5  for (i = 0; i < 5; ++i)
6    printf("%d ", *(p+i));
```

```
1 4 2 7 3
```

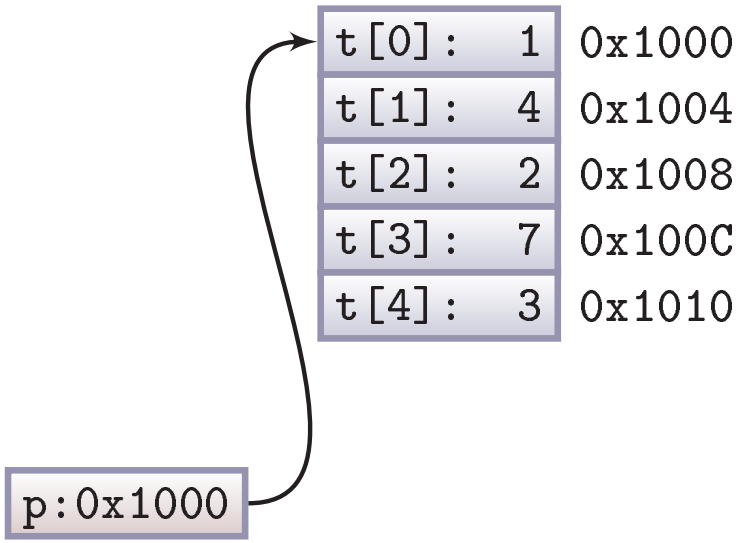


Mutatók és tömbök

- Tömb bejárása lehetséges mutatóaritmetika alkalmazásával

```
1  int t[5] = {1,4,2,7,3};
2  int *p, i;
3
4  p = &t[0];
5  for (i = 0; i < 5; ++i)
6    printf("%d ", *(p+i));
```

```
1 4 2 7 3
```



- Jelen példában *(p+i) megegyezik t[i]-vel, mert p a t tömb elejére mutat

Mutatók és tömbök

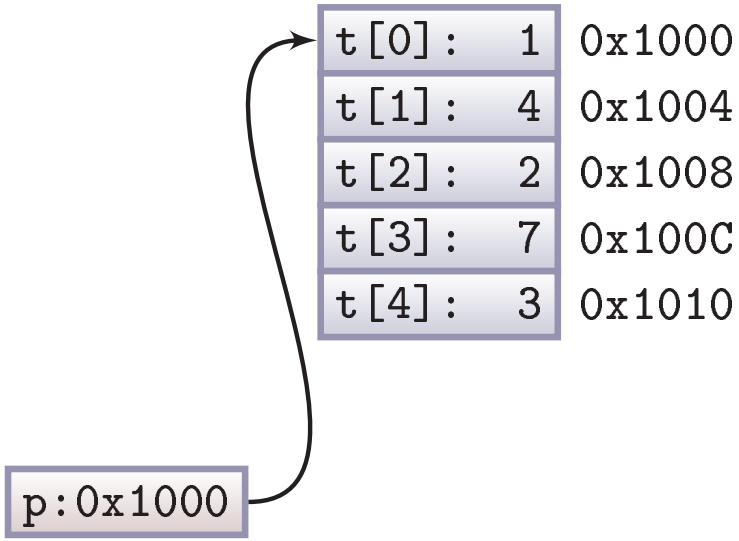
- Mutatók tömbként kezelhetőek, vagyis indexelhetőek. Definíció szerint
  `p[i]` azonos `*(p+i)` -vel

Mutatók és tömbök

- Mutatók tömbként kezelhetőek, vagyis indexelhetőek.
  Definíció szerint
  `p[i]` azonos `*(p+i)` -vel

```
1  int t[5] = {1,4,2,7,3};
2  int *p, i;
3
4  p = &t[0];
5  for (i = 0; i < 5; ++i)
6    printf("%d ", p[i]);
```

```
1 4 2 7 3
```

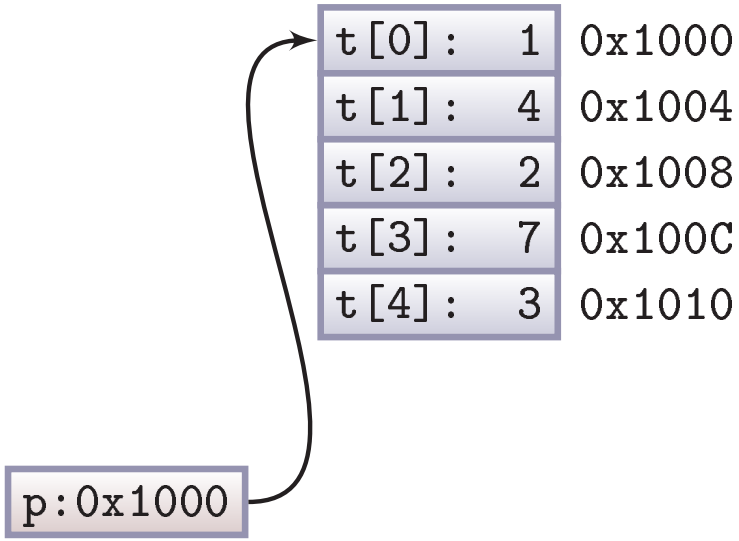


Mutatók és tömbök

- Mutatók tömbként kezelhetőek, vagyis indexelhetőek. Definíció szerint
  `p[i]` azonos `*(p+i)` -vel

```
1  int t[5] = {1,4,2,7,3};
2  int *p, i;
3
4  p = &t[0];
5  for (i = 0; i < 5; ++i)
6    printf("%d ", p[i]);
```

```
1 4 2 7 3
```



- Jelen példában `p[i]` megegyezik `t[i]`-vel, mert `p` a `t` tömb elejére mutat

Mutatók és tömbök

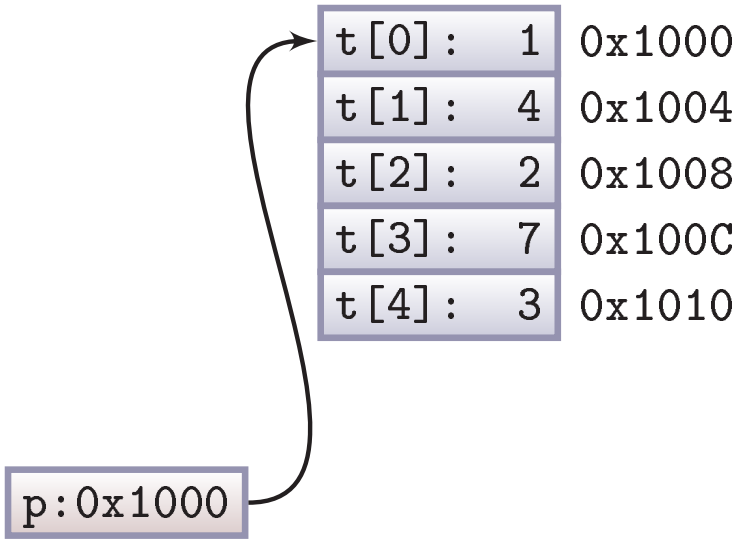
- Tömbök mutatóként kezelhetőek.
  Tömb nevét írva a tömb kezdőcímét kapjuk meg, vagyis a `t` kifejezés értéke `&t[0]`

Mutatók és tömbök

- Tömbök mutatóként kezelhetőek.
  Tömb nevét írva a tömb kezdőcímét kapjuk meg, vagyis a `t` kifejezés értéke `&t[0]`

```
1  int t[5] = {1,4,2,7,3};
2  int *p, i;
3
4  p = t; /* &t[0] */
5  for (i = 0; i < 5; ++i)
6    printf("%d ", p[i]);
```

```
1 4 2 7 3
```

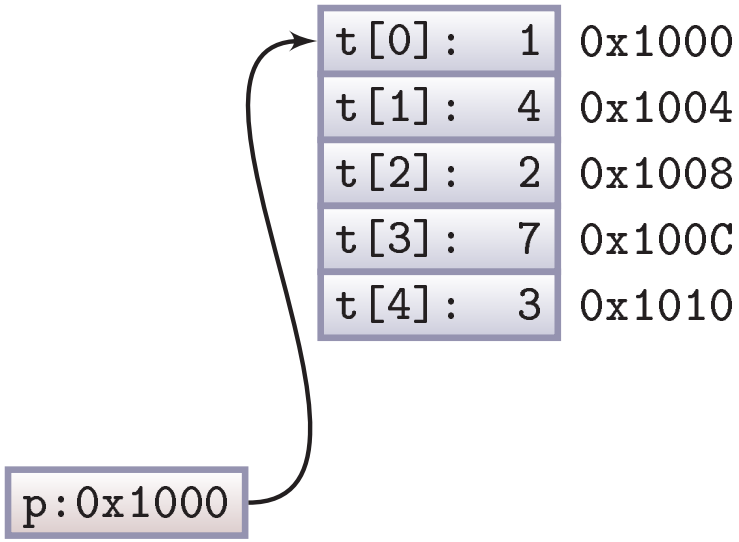


Mutatók és tömbök

- Tömbök mutatóként kezelhetőek.
  Tömb nevét írva a tömb kezdőcímét kapjuk meg, vagyis a `t` kifejezés értéke `&t[0]`

```
1  int t[5] = {1,4,2,7,3};
2  int *p, i;
3
4  p = t; /* &t[0] */
5  for (i = 0; i < 5; ++i)
6    printf("%d ", p[i]);
```

```
1 4 2 7 3
```



- A mutatóaritmetika tömbökre is működik:
  `t+i` azonos `&t[i]`-vel

Mutatók és tömbök – összefoglalás

- Mutató kezelhető tömbként, tömb kezelhető mutatóként.

Mutatók és tömbök – összefoglalás

- Mutató kezelhető tömbként, tömb kezelhető mutatóként.
- Az index operátor csak egy jelölés
  `a[i]`-t a fordító mindig `*(a+i)`-vel helyettesíti,
  akkor is, ha *a* mutató, akkor is, ha *a* tömb.

Mutatók és tömbök – összefoglalás

- Mutató kezelhető tömbként, tömb kezelhető mutatóként.
- Az index operátor csak egy jelölés
  `a[i]`-t a fordító mindig `*(a+i)`-vel helyettesíti,
  akkor is, ha *a* mutató, akkor is, ha *a* tömb.
- Különbségek:
  - A tömbelemeknek fenntartott tárhelyük van (változók).
    A mutatóhoz nem tartoznak foglalt elemek.

Mutatók és tömbök – összefoglalás

- Mutató kezelhető tömbként, tömb kezelhető mutatóként.
- Az index operátor csak egy jelölés
  `a[i]`-t a fordító mindig `*(a+i)`-vel helyettesíti,
  akkor is, ha *a* mutató, akkor is, ha *a* tömb.
- Különbségek:
  - A tömbelemeknek fenntartott tárhelyük van (változók).
    A mutatóhoz nem tartoznak foglalt elemek.
  - A tömb kezdőcíme konstans, nem változtatható.
    A mutató változó, a benne tárolt cím módosítható.

Mutatók és tömbök – összefoglalás

- Mutató kezelhető tömbként, tömb kezelhető mutatóként.
- Az index operátor csak egy jelölés
  a[i]-t a fordító mindig *(a+i)-vel helyettesíti,
  akkor is, ha *a* mutató, akkor is, ha *a* tömb.
- Különbségek:
  - A tömbelemeknek fenntartott tárhelyük van (változók).
    A mutatóhoz nem tartoznak foglalt elemek.
  - A tömb kezdőcíme konstans, nem változtatható.
    A mutató változó, a benne tárolt cím módosítható.

```
1  int array[5] = {1, 3, 2, 4, 7};
2  int *p = array;
3
4  /* az elemek p-n és a-n keresztül elérhetőek */
5  p[0] = 2;                array[0] = 2;
6  *p = 2;                  *array = 2;
7
8  /* p változtatható       array nem */
9  p = p+1; /* jó */        array = array + 1; /* HIBA */
```

Tömbök átadása függvénynek

- Határozzuk meg függvénnyel az `array` tömb első negatív elemét!

[3] az `stdio.h` definiálja

Tömbök átadása függvénynek

- Határozzuk meg függvénnyel az `array` tömb első negatív elemét!
- Tömb átadása:
  - Első elem címe `double*`
  - Tömb mérete `typedef unsigned int size_t`[3]

[3] az `stdio.h` definiálja

Tömbök átadása függvénynek

- Határozzuk meg függvénnyel az array tömb első negatív elemét!
- Tömb átadása:
  - Első elem címe `double*`
  - Tömb mérete `typedef unsigned int size_t`[3]

```
1  double first_negative(double *array, size_t size)
2  {
3    size_t i;
4    for (i = 0; i < size; ++i)  /* minden elemre */
5      if (array[i] < 0.0)
6        return array[i];
7
8    return 0; /* mind nemnegatív */
9  }
```

link

```
1  double myarray[3] = {3.0, 1.0, -2.0};
2  double neg = first_negative(myarray, 3);
```

link

[3] az `stdio.h` definiálja

Tömbök átadása függvénynek

- Hogy a paraméterlistán elkülönüljön a tömb és a mutató, tömbök átvételekor alkalmazhatjuk a tömbös jelölést is.

```
1  double first_negative(double array[], size_t size)
2                  /*    (double *array,  size_t size)  */
3  {
4     ...
5  }
```

Tömbök átadása függvénynek

- Hogy a paraméterlistán elkülönüljön a tömb és a mutató, tömbök átvételekor alkalmazhatjuk a tömbös jelölést is.

```
1  double first_negative(double array[], size_t size)
2                  /*    (double *array,  size_t size)   */
3  {
4     ...
5  }
```

- Formális paraméterlistán `double a[]` azonos `double *a`-val.
- Formális paraméterlistán csak az üres `[]` jelölés használható, a méretet mindig külön paraméterként kell átadni!

Tömbök átadása függvénynek

- Határozzuk meg függvénnyel az array tömb első negatív elemét!
- Visszatérési érték legyen a megtalált elem címe

```
1  double *first_negative(double *array, size_t size)
2  {
3    size_t i;
4    for (i = 0; i < size; ++i)  /* minden elemre */
5      if (array[i] < 0.0)
6        return &array[i];
7
8    return NULL; /* mind nemnegatív */
9  }
```

link

Tömbök átadása függvénynek

- Határozzuk meg függvénnyel az array tömb első negatív elemét!
- Visszatérési érték legyen a megtalált elem címe

```
1  double *first_negative(double *array, size_t size)
2  {
3    size_t i;
4    for (i = 0; i < size; ++i)  /* minden elemre */
5      if (array[i] < 0.0)
6        return &array[i];
7
8    return NULL; /* mind nemnegatív */
9  }
```

link

Nullpointer

- A nullpointer (`NULL`)

Nullpointer

- A nullpointer (`NULL`)
  - A `0x0000` memóriacímet tartalmazza

Nullpointer

- A nullpointer (`NULL`)
  - A `0x0000` memóriacímet tartalmazza
  - Megállapodás szerint „nem mutat sehova”

3. fejezet

Sztringek

Sztringek

- C-ben a szövegeket végjeles karaktertömbökben, ún. sztringekben (string, karakterfüzér) tároljuk.

Sztringek

- C-ben a szövegeket végjeles karaktertömbökben, ún. sztringekben (string, karakterfüzér) tároljuk.
- A végjel a 0-s ASCII-kódú `'\0'` nullkarakter.

'E'	'z'	' '	's'	'z'	'ö'	'v'	'e'	'g'	'\0'

Sztringek definiálása karaktertömbként

- Karaktertömb definiálása kezdetiérték-adással

```
1  char s[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Sztringek definiálása karaktertömbként

- Karaktertömb definiálása kezdetiérték-adással

```
1  char s[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- Ugyanaz egyszerűbben

```
1  char s[] = "Hello"; /* s tömb (konst.cím 0x1000) */
```

'H'	0x1000
'e'	0x1001
'l'	0x1002
'l'	0x1003
'o'	0x1004
'\0'	0x1005

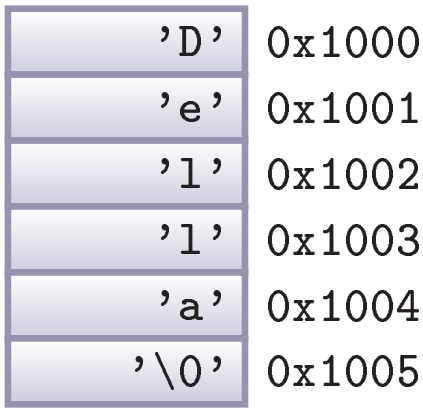
Sztringek definiálása karaktertömbként

- Karaktertömb definiálása kezdetiérték-adással

```
1 char s[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- Ugyanaz egyszerűbben

```
1 char s[] = "Hello"; /* s tömb (konst.cím 0x1000) */
```



- s elemei indexeléssel vagy mutatóaritmetikával elérhetőek

```
1 *s = 'D';    /* s-et mutatóként kezelem */
2 s[4] = 'a'; /* s-et tömbként kezelem */
```

Sztringek definiálása karaktertömbként

- Hosszabb sztringnek is helyet foglalhatunk későbbi felhasználás céljából

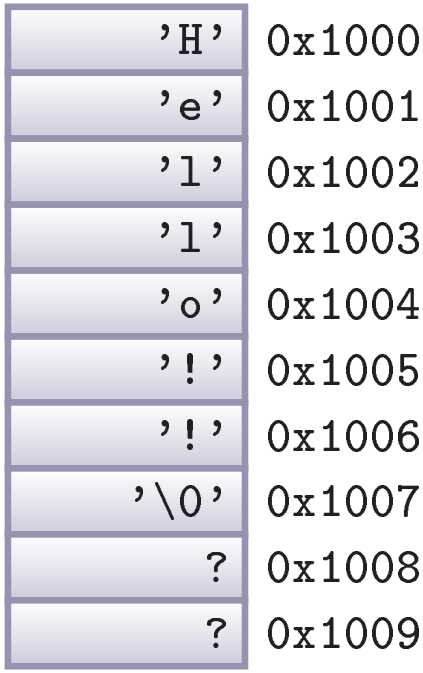
```
1  char s[10] = "Hello"; /* s tömb, (konst.cím 0x1000) */
```

'H'	0x1000
'e'	0x1001
'l'	0x1002
'l'	0x1003
'o'	0x1004
'\0'	0x1005
?	0x1006
?	0x1007
?	0x1008
?	0x1009

Sztringek definiálása karaktertömbként

- Hosszabb sztringnek is helyet foglalhatunk későbbi felhasználás céljából

```
1  char s[10] = "Hello"; /* s tömb, (konst.cím 0x1000) */
```



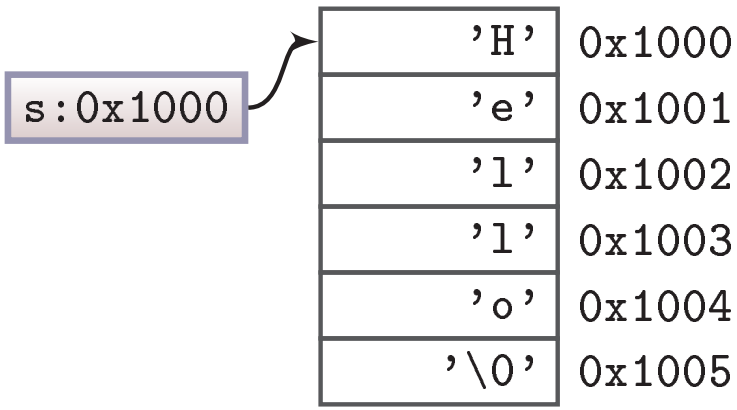
- Módosítás:

```
1  s[5] = s[6] = '!';
2  s[7] = '\0';          /* le kell zárni */
```

Sztringek definiálása karaktermutatóként

- Konstans karaktertömb és rá mutató pointer definiálása kezdetiérték-adással

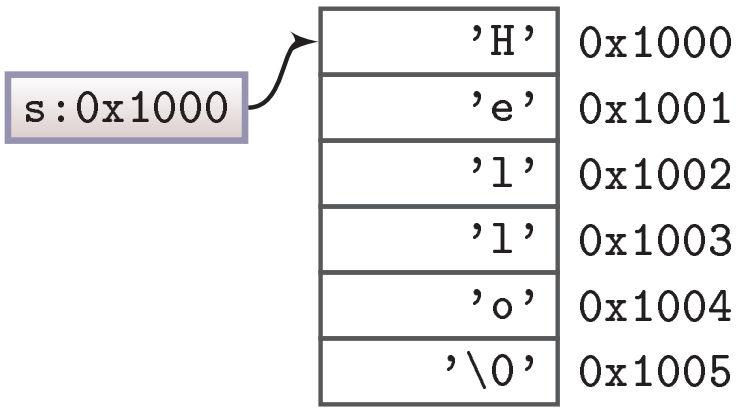
```
1  char *s = "Hello"; /* s mutató */
```



Sztringek definiálása karaktermutatóként

- Konstans karaktertömb és rá mutató pointer definiálása kezdetiérték-adással

```
1  char *s = "Hello"; /* s mutató */
```

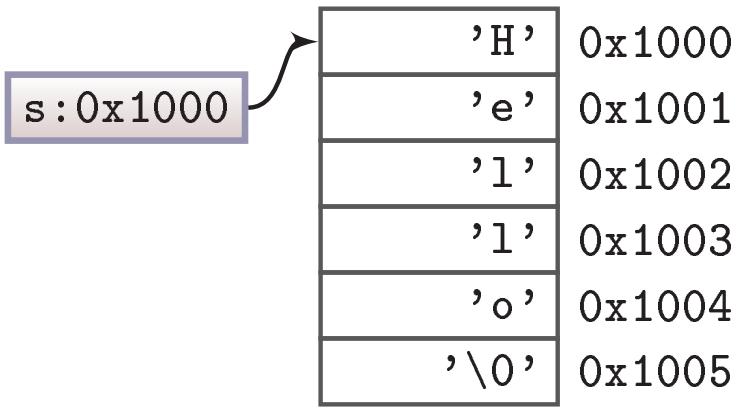


- Itt a karaktereknek az ún. statikus területen foglalunk helyet, és a sztring tartalma nem módosítható.

Sztringek definiálása karaktermutatóként

- Konstans karaktertömb és rá mutató pointer definiálása kezdetiérték-adással

```
1  char *s = "Hello"; /* s mutató */
```



- Itt a karaktereknek az ún. statikus területen foglalunk helyet, és a sztring tartalma nem módosítható.
- s értéke viszont felülírható, de ez nem ajánlott, mert a sztringnek lefoglalt területet csak s-en keresztül érjük el.

Megjegyzések

- Karakter vagy szöveg?

```
1  char s[] = "A"; /* két bájt: {'A', '\0'} */
2  char c = 'A';   /* egy bájt: 'A' */
```

Megjegyzések

- Karakter vagy szöveg?

```
1 char s[] = "A"; /* két bájt: {'A', '\0'} */
2 char c = 'A';   /* egy bájt: 'A' */
```

- Üres szöveg van, üres karakter nincs

```
1 char s[] = "";  /* egy bájt: {'\0'} */
2 char c = '';    /* HIBA, ilyen nincs */
```

Sztring beolvasása és kiírása

- sztringek kiírása-beolvasása a %s formátumkóddal

```
1  char s[100] = "Hello";
2  printf("%s\n", s);
3  printf("Adj meg egy max 99 karakter hosszú szót: ");
4  scanf("%s", s);
5  printf("%s\n", s);
```

```
Hello
Adj meg egy max 99 karakter hosszú szót:  csodalámpa
csodalámpa
```

Sztring beolvasása és kiírása

- sztringek kiírása-beolvasása a %s formátumkóddal

```
1 char s[100] = "Hello";
2 printf("%s\n", s);
3 printf("Adj meg egy max 99 karakter hosszú szót: ");
4 scanf("%s", s);
5 printf("%s\n", s);
```

```
Hello
Adj meg egy max 99 karakter hosszú szót:  csodalámpa
csodalámpa
```

- Miért nem kell a printf függvénynek átadni a méretet?
- Miért nem kell a scanf függvényben a &?

Sztring beolvasása és kiírása

- A scanf csak az első whitespace karakterig olvas. Több szóból álló szöveg beolvasása fgets függvénnyel:

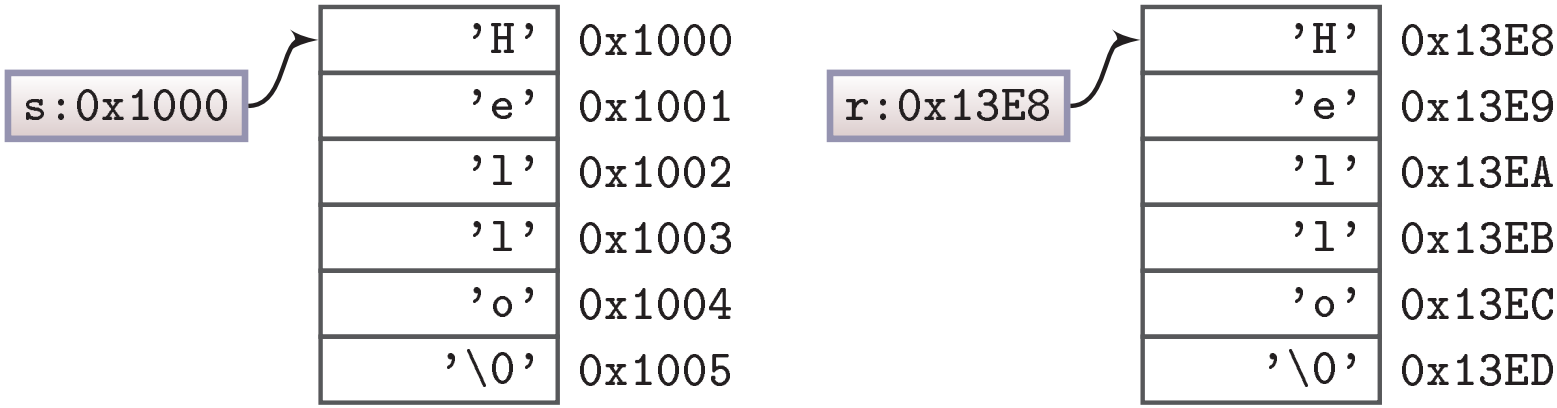
```
1 char s[100];
2 printf("Adj meg max. 99 karakter hosszú szöveget: ");
3 fgets(s, 100, stdin);
4 printf("%s\n", s);
```

```
Adj meg egy max.  99 karakter hosszú szöveget:  ez szöveg
ez szöveg
```

Sztringek – tipikus hibák

- Tipikus hiba: sztringek összehasonlítása

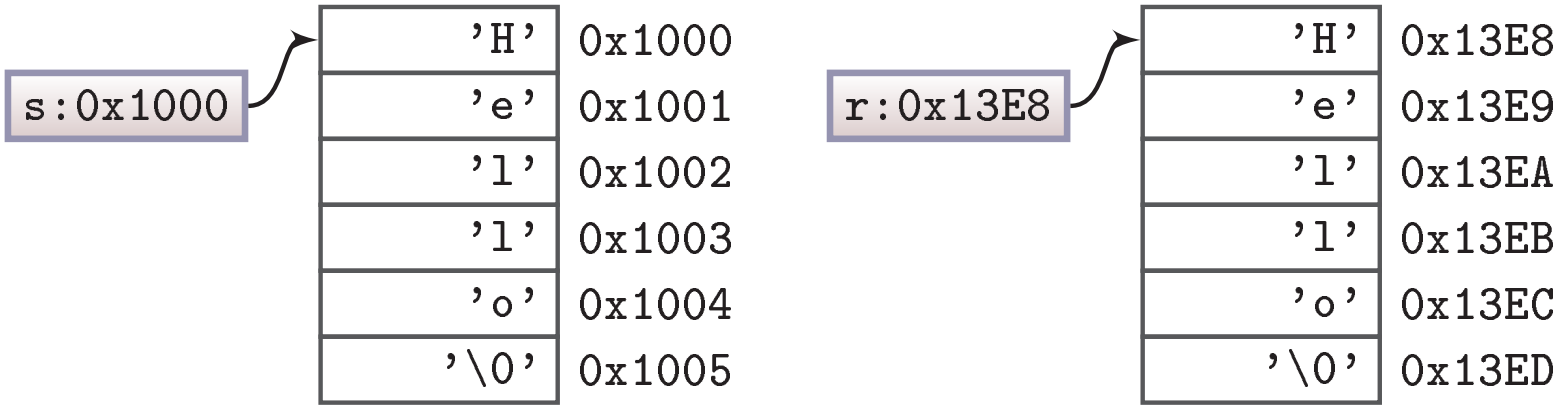
```
1  char *s = "Hello";
2  char *r = "Hello";
3  if (s == r) /* mit hasonlítunk össze? */
4    ...
```



Sztringek – tipikus hibák

- Tipikus hiba: sztringek összehasonlítása

```
1  char *s = "Hello";
2  char *r = "Hello";
3  if (s == r) /* mit hasonlítunk össze? */
4    ...
```



- Tömbös definíció esetén ugyanez a hiba

Sztringfüggvények

- Sztringek összehasonlítása
- az eredmény
  - pozitív, ha s1 a névsorban s2 után áll
  - 0, ha megegyeznek
  - negatív, ha s1 a névsorban s2 előtt áll

```
1  int strcmp(char *s1, char *s2) /* mutatós jelölés */
2  {
3    while (*s1 != '\0' && *s1 == *s2)
4    {
5      s1++;
6      s2++;
7    }
8    return *s1 - *s2;
9  }
```

Sztringfüggvények

- Sztringek összehasonlítása
- az eredmény
  - pozitív, ha s1 a névsorban s2 után áll
  - 0, ha megegyeznek
  - negatív, ha s1 a névsorban s2 előtt áll

```
1  int strcmp(char *s1, char *s2) /* mutatós jelölés */
2  {
3    while (*s1 != '\0' && *s1 == *s2)
4    {
5      s1++;
6      s2++;
7    }
8    return *s1 - *s2;
9  }
```

- Nem baj, hogy s1 és s2 megváltozott vizsgálat közben?

Sztringfüggvények

- Sztringek összehasonlítása
- az eredmény
  - pozitív, ha s1 a névsorban s2 után áll
  - 0, ha megegyeznek
  - negatív, ha s1 a névsorban s2 előtt áll

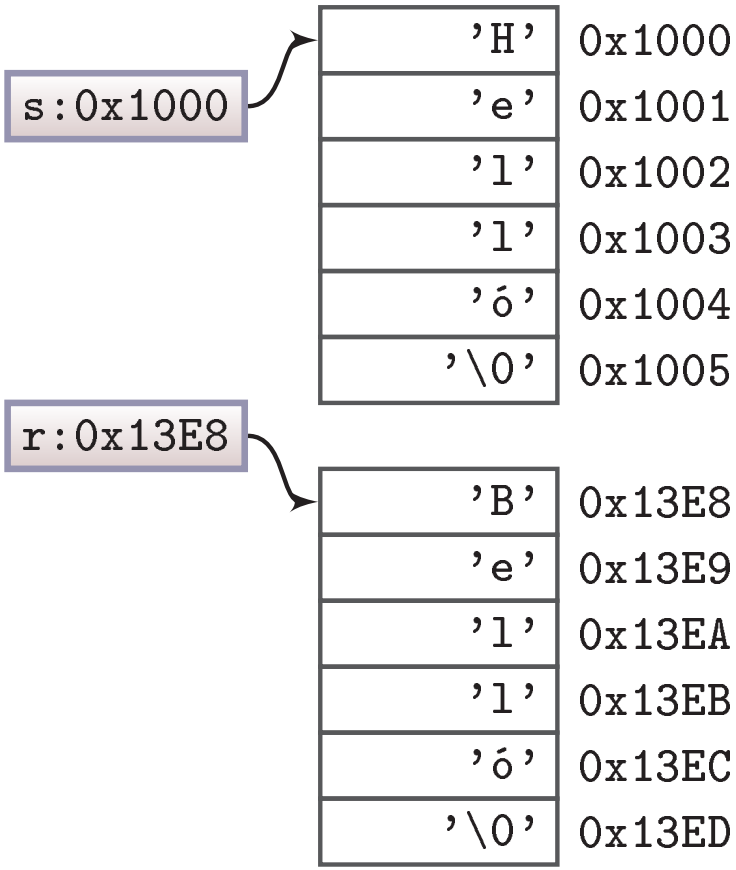
```
1  int strcmp(char *s1, char *s2) /* mutatós jelölés */
2  {
3    while (*s1 != '\0' && *s1 == *s2)
4    {
5      s1++;
6      s2++;
7    }
8    return *s1 - *s2;
9  }
```

- Nem baj, hogy s1 és s2 megváltozott vizsgálat közben?
- Gondoljuk meg: A megoldásban kihasználtuk, hogy \0 a 0 kódú karakter!

Sztringek – tipikus hibák

- Tipikus hiba: sztringek (képzelt) másolása

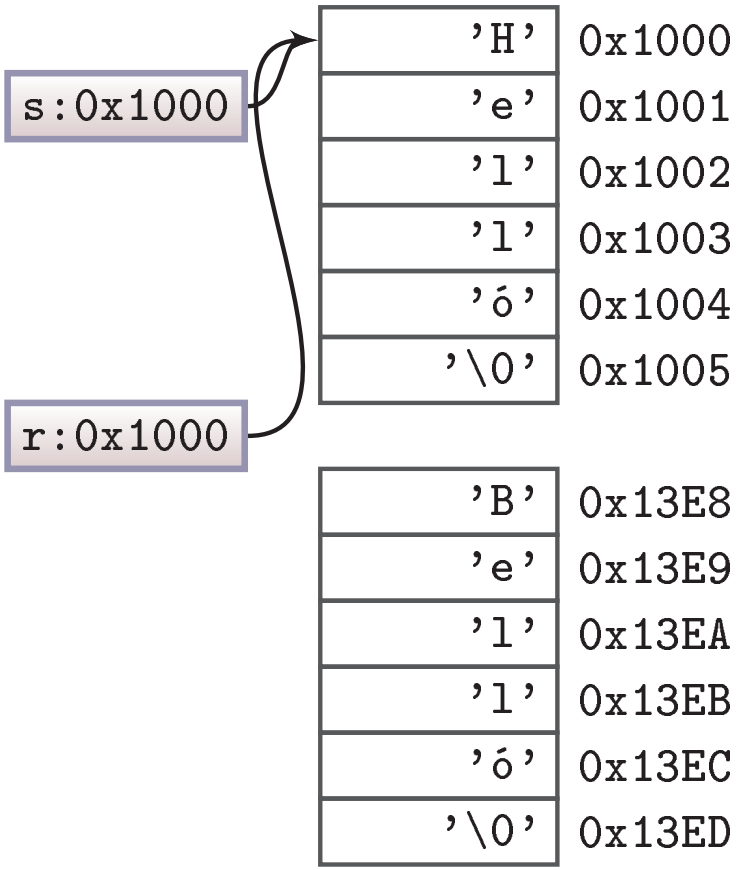
```
1  char *s = "Helló";
2  char *r = "Belló";
3  r = s; /* mit másolunk? */
```



Sztringek – tipikus hibák

- Tipikus hiba: sztringek (képzelt) másolása

```
1  char *s = "Helló";
2  char *r = "Belló";
3  r = s; /* mit másolunk? */
```



Egyéb sztringfüggvények

- `#include <string.h>`
  - `strlen` sztring hossza
  - `strcmp` sztringek összehasonlítása
  - `strcpy` sztring másolása
  - `strcat` sztring másik után fűzése
  - `strchr` karakter keresése sztringben
  - `strstr` sztring keresése sztringben
- a `strcpy` és `strcat` függvények ész nélkül másolnak, a felhasználónak kell gondoskodnia az eredménynek fenntartott helyről!

Köszönöm a figyelmet.