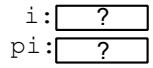
1. Pointer

1.1. Pointer definició

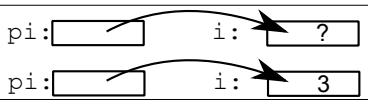
```
int i;
int *pi;
```



1.2. Operátorok

Pointer indirekció: $*$
Címképző operátor: &

```
ip=&i;
*pi=3;
```



1.3. Pointer aritmetika

```
double *p1;
double *p2;
int i;
.....
p2 = p1+i;
p2 = p1-i;
p2++; p2--; ++p2; --p2;
i=p2-p1;
```

1.4. Pointerek összehasonlítása

```
double *p1;
double *p2;

if(p1==p2) {...}
if(p1<=p2) {...}
```

1.5. Tömb

```
char str[120];
```

A C nyelvben csak egydimenziós tömbök vannak, de a tömb elemei lehetnek tömbök.

A tömbökkel csak két műveletet tudunk végezni:

1. megállapíthatjuk a `sizeof` operátorral a méretét
2. lekérdezhetjük az első elemének a címét

Kifejezésben a tömb neve a tömb első elemének a címét adja vissza. Típusa azonos a tömb egy elemére mutató pointer típusával.

```
double array[50];
double* pa;
pa=array;
```

1.6. Syntactic sugar

Egyszerűbb írásmód érdekében olyan jelölést vezetünk be, amelyre már van nyelvi elem.

http://en.wikipedia.org/wiki/Syntactic_sugar

1.7. Index operátor

Az index operátor a C nyelvben egy *syntactic sugar*.

```
int tomb[50];
tomb[5] ≡ *(tomb+5) ≡ *(5+tomb)     ≡ 5[tomb]
```

A fent leírt kifejezések ekvivalensek egymással, a fordító minden index kifejezést a második formába konvertálja, ezért a negyedik alak is hibátlanul lefordul.

A következő két sorban a pointerek azonos értéket vesznek fel.

```
int tomb[50];
int *pt1 = &tomb[0];
int *pt2 = tomb;
```

A tömbhivatkozás, pointer hivatkozás átalakítás miatt nincs mód az indexhatárok ellenőrzésére.

1.8. A tömb miért nem azonos a pointerrel?

Látszólag a tömbök és a pointerek használata között nincs különbség.
Például:

```
int tomb[50];
int *pt = tomb;

tomb[0] = 5;
pt[0] = 5;
```

Azonban már a következő kifejezésekre ez nem igaz:

```
pt++;                   /* OK   */
tomb++;                 /* HIBA */
```

Kifejezésben a tömb nevét felfoghatnánk konstans pointernek, azaz az értékét nem lehet megváltoztatni, de ez sem teljesen igaz.

A következő két utasításból különböző gépi kód keletkezi.

```
tomb[0]=5;              /* mov dword ptr[tomb],2   */

pt[0]=5;                /* mov ecx, dword ptr[pt]  */
                        /* mov dword ptr[ecx],2    */
```

```
a.c                                  b.c
=============                        ================
int tomb[50];                        extern int tomb[];   /* OK   */
int t[50];                           extern int* t;       /* HIBA */
```

Külön fordítási egységekben a tömbként definiált változót a másik fordítási egységben nem deklarálhatjuk pointerként.

1.9. Következmények

Nincs tömb értékadás!

```
char s1[120];
char s2[120];
......
s1=s2;              /* HIBA */
```

Két tömböt értelmetlen összehasonlítani!

```
char s1[120];
char s2[120];
......
if( s1==s2){ ... }          /* Valóban ezt akarjuk? */
```

1.10. Függvény deklaráció

Függvényhívásnál a paraméterek értéke a stack-re kerül, azaz érték szerinti paraméter átadás van. Tömb esetén a tömb elemei nem másolódnak le, hanem a tömb címe adódik át. Ez persze az előzőekben leírtakból is következik, mivel a függvényt hívó kifejezésben a tömb neve a tömb első elemének a címét adja vissza. Paraméter listán a tömb jelölés *syntatctic sugar*, igazából ilyenkor a tömb kezdőcíme adódik át. A következő függvény deklarációk azonosak:

```
int sum( int* t, int db );
int sum( int t[], int db);
int sum( int t[50], int db);
```

Ezzel az egyszerű programmal jól szemléltethetjük a fent leírtakat:

```
#include <stdio.h>

void f( int t[50] )
{
   printf("%d\n", sizeof(t) );
}

int main()
{
   int tomb[50];
   f(tomb);
   printf("%d\n", sizeof(tomb) );
   return 0;
}
```

output:

```
4
200
```

32 bites rendszerben az `f` függvény a paraméterként kapott pointer méretét, a `main` függvény a tömb méretét (50×4) írja ki.

Felmerül a kérdés, hogy akkor miért jó függvényparamétereknél a tömb jelölést bevezetni. Egyrészt olvashatóbbá teszi a programot, másrészt megkíméli a programozót a bonyolultabb pointerhivatkozás megfogalmazásától. A következő példában az `init` függvénynek egy sakktábla állást tartalmazó mátrixot szeretnénk átadni.

```
int sakk[8][8];
```

Definiáljunk egy `sakk` változóra mutató pointert!

```
int *p[8]; /* Nem jó, mert ez egy pointereket tartalmazó 8 elemű tömb */
```

```
int (*psakk)[8]; /*Így már jó */
```

Tehát az `init` függvényt a következőképpen deklarálhatjuk:

```
int init( int s[8][8] );
int init( int s[][8] );
int init( int (*p)[8]);
```

A fordítónak ismerni kell a tömb elemeinek a méretét, ezért csak a tömböket tartalmazó vektor mérete hagyható el, a vektor elemeit alkotó vektor mérete már nem. Hibás a következő deklaráció:

```
int init( int s[][] );      /* HIBA */
```

1.11. Miért okozhat hibát, ha tömbnél használjuk a cím operátort

```
char line[256];
.....
scanf("%s", line );      /* tökéletes */
scanf("%s", &line );     /* működik, de nem szép */
```

A két kifejezésben a `line` és a `&line` ugyanazt a memóriacímet adja vissza, de első esetben a cím típusa `char*`, a második esetben `char (*)[256]`. Változó paraméterszámú függvénynél a fordító a paraméterek típusát nem ellenőrzi. `scanf` függvény esetén a formátum mező határozza meg, hogy milyen típusú paramétert olvasunk be.
Mi történik, ha a `line` string első elemét olvasásnál változatlanul akarjuk hagyni?

```
.....
scanf("%s", line+1 );    /* OK */
scanf("%s", &line+1 );   /* nagyon rossz */
```

Második esetben nem eggyel, hanem 256 byte-tal odébb címzünk!