

4. Gyakorlat

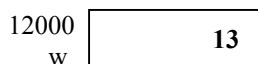
Rövid elméleti összefoglaló

- **Mutató:**

```
int w = 13, érték;  
int *poin; // a poin int-re mutató pointer (címet)tárol.  
           // a poin átveszi a w címét, ezért ugyanarra a memória-területre hivatkoznak  
  
poin = &w; // a *poin a poin által mutatott memória-terület tartalmát,  
           // azaz az adatot jelenti, így az érték változó tartalma 13 lesz.  
érték = *poin;
```

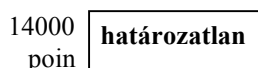
Bővebb magyarázat:

Az `int w = 13;` definíció fordításakor a fordító elhelyezi a `w` változót pl. a 12000 memóriacímre, amely mögött 4 bájt szabad memória-terület van egy egész szám tárolására. Mivel a változó már a definíció során kapott kezdőértéket, a 12000 memóriacímen található tároló tartalma a 13 egész szám lesz.

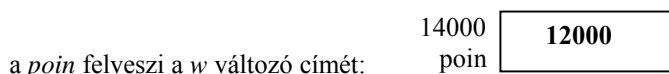


Hozzáférhetünk a `w` címéhez, az `&w` kifejezés a `w` címét jelenti, a példánkban ez 12000.

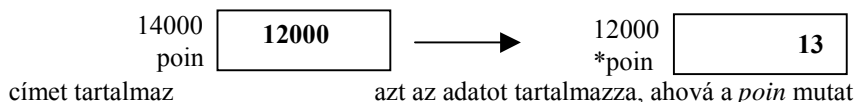
Az `int *poin;` definíció fordításakor a fordító elhelyezi a `poin` változót pl. a 14000 memóriacímre, amely mögött 4 bájt szabad memória-terület van, hogy egy memóriacím elférjen benne. Ebben az esetben a `poin` egy `int` típusra mutató címet fogad el továbbiakban, azonban kezdetben csak egy véletlenszerű értéket tartalmaz. Hibázik a program, ha nem gondoskodunk a megfelelő értékadásról.



`poin = &w;` utasítás végrehajtásának hatására



A 12000 cím valóban `int` adatra mutat (a példa szerint a 13 egészre).



Összefoglalva: a `poin` változó ebben a példában `int`-re, azaz egész számra mutató címet tárol, a `*poin` pedig az adatot tartalmazza (jelen példában a 13-at), ahová a `poin` mutat.

- **Referencia** (hivatkozási) típus felhasználásával már létező változóra hivatkozva *alternatív nevet* definiálhatunk:

típus & azonosító = változó

```
float b = 5.6;
```

// a referencia definiálásakor kötelező kezdőértéket adnunk.

```
float &rb = b;
```

A referencia tárolására általában nem jön létre külön változó!

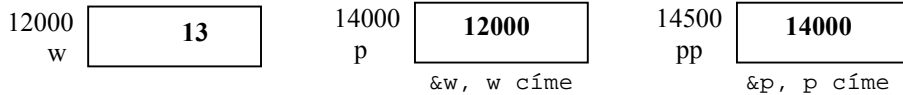
- **void*** // a C++ nyelv típus nélküli általános mutatója
`float w;` // float típusú változóra mutató címet vesz át
`void *poin = &w;`

// értékadásnál felhasználói típuskonverzióval (cast) típust kell a cím mellé rendelni
`*(float*)poin = 12.56;`

- **Többszörös indirektségű mutatók**

```
int w = 13, w1;  
int *p; // p egy int változóra mutató pointer  
int **pp; // int* változóra mutató pointer
```

`p = &w; pp = &p; w1 = *p + **pp; // w1 értéke 26`



A példában a `*p` és a `**pp` végül a címzések által a `w` tartalmát érik el, mindkettő értéke 13 lesz, ezért a `w1` a 13+13 összegét, azaz 26-ot kap értékül.

- Típuskonverziók:**

A típuskonverziók egy része automatikusan, a programozó beavatkozása nélkül megy végbe, a C++ nyelv definíciójában rögzített szabályok alapján. Ezeket a konverziókat *implicit* vagy *automatikus* konverzióknak nevezzük.

<pre>//MINTA4_01 #include <iostream> using namespace std; main() { int i = 12, j; float a = 3.43, ered1= 0; ered1 = i + a; cout << " eredmény1: " << ered1 << endl; j = i + a; cout << " eredmény2: " << j << endl; cin.get(); }</pre>	<p><i>A program futásának eredménye:</i></p> <p>eredmeny1: 15.43 eredmeny2: 15</p>
<pre>//MINTA4_02 #include <iostream> using namespace std; int main() { int i = 12; // tetszőleges típusú mutató átalakítása // void* mutatótípussá void *ip = &i; int *a = 0; //explicit konverzióval kell visszaalakítanunk a = static_cast<int*>(ip); cout << "*a tartalma: " << *a << endl; cin.get(); return 0; }</pre>	<p><i>A program futásának eredménye:</i></p> <p>*a tartalma: 12</p>

- Futásidejű típusazonosítás**

A C++ nyelvben a `typeid` operátor egy `type_info` típusú objektum referenciáját adja vissza:

`typeid` (kifejezés)
`typeid` (típusazonosító)

<pre>//MINTA4_03 #include <iostream> #include <typeinfo> using namespace std; int main() { int a = 2, b = 3; if (typeid (a) == typeid (b)) cout << "a es a b tipusa egyezik: " << typeid(a).name() << endl; cin.get(); return 0; }</pre>	<p><i>A program futásának eredménye:</i></p> <p>a es a b tipusa egyezik: i</p>
--	--

- **Tömbváltozó deklarálása**

A tömb azonos típusú értékeket tud tárolni:

```
int c;
// a w tömb egész elemeket tárol, indexe 0-9-ig változhat
int w[10];
// a z tömb valós elemeket tárol, indexe 0-49-ig változhat
double z[50];
w[0] = 12.5; // a w tömb első eleme 12.5 értéket kap.
```

- **A new operátor**

A **new** operátor az operandusában megadott típusnak megfelelő méretű területet foglal a szabad memóriában, és a területre mutató pointert adja vissza:

<code>int *ip;</code>	<i>ip</i> int típusú területre mutathat, de még nem tartalmaz címet.
<code>ip = new int;</code>	A new operátor int adatra mutató címet ad át az <i>ip</i> számára.
<code>*ip = 12;</code>	<i>*ip</i> értékadás a mutatóval kijelölt tárolónak.
<code>double *tp;</code>	A <i>tp</i> double típusú pointer, amely double típusú változóra mutathat.
<code>tp = new double[20];</code>	A new operátor mögött a double[20] azt jelenti, hogy 20 darab double típusú adat számára foglaltunk helyet a memóriában, így egy 20-elemű tömböt hoztunk létre. A tömb indexelése 0-19 között lehetséges, mivel a C++ nyelvben minden tömb 0 indexű elemmel kezdődik.
<code>tp[0] = 1.8;</code>	Értékadás a tömb első, 0. indexű elemének.
<code>tp[19] = 100.4;</code>	Értékadás a tömb utolsó, 19. indexű elemének.

- **A delete operátor**

A **delete** operátor a **new** operátor által lefoglalt memória-területet szabadítja fel.

```
delete ip; // az ip pointer által mutatott terület felszabadítása
delete[] tp; // a tp tömb területének felszabadítása
```

A tömb helyfoglalásának vizsgálata:

<pre>//MINTA4_04 #include <iostream> #include <new> using namespace std; int main() { double *tomb; long int meret; cout << "meret: "; cin >> meret; try { tomb = new double [meret]; } catch(bad_alloc) { cerr << "Nincs eleg memoria" << endl; cin.get(); cin.get(); return -1; } cout << "Sikeres a helyfoglalas" << endl; // memória felszabadítása delete[] tomb; cout << "A memoria felszabaditasa\n"; cin.get(); cin.get(); return 0; }</pre>	<p>Kivételkezelés lehetőségei:</p> <ul style="list-style-type: none"> – A try blokk a próbálkozás. – A kivételek elfogása és kezelése (catch). <p>A program futásának eredményei:</p> <pre>meret: 250 Sikeres a helyfoglalas A memoria felszabaditasa meret: 4000000000000 Nincs eleg memoria</pre>
--	--

Feladatok

1. Használjuk az alábbi definíciókat!

```
int d[40];

int db, osszeg, poz, neg, zero;
double atlag;
```

Olvassuk be az adatok számát a *db* változóba, majd töltsük fel a *d* tömböt egész adatokkal! Számítsuk ki a tömb elemeinek összegét, átlagát! Számláljuk meg a zérus, a pozitív és a negatív adatok darabszámát! (GYAK4_1)

A *d* tömb feltöltése:

```
do
{
    cout << "Adatok szama (max. 40): ";
    cin >> db;
}while ( 0 >= db || db > 40);
for (i = 0; i < db ; i++)
{
    cout << i << ". adat: "; cin >> d[i];
}
. . .
```

A program futásának eredménye:

```
Adatok szama (max. 40): 6
0. adat: 2
1. adat: -4
2. adat: 8
3. adat: 0
4. adat: 12
5. adat: 0

Atlag: 3
Osszeg: 18
pozitiv elemek szama: 3
Negativ elemek szama: 1
Zerus elemek szama : 2
```

2. Használjuk az alábbi definíciókat!

```
double a[] = {1, 2, 3, 4};
double b[] = {4, 3, 2, 1};
double Skszorzat = 0;
```

$$\text{skalarszorzat} = \sum_{i=0}^{i < M} a[i] \cdot b[i]$$

Számítsuk ki a két tömb skalárszorzatát. (GYAK4_2)

A program futásának eredménye:

```
Skalarszorzat: 20
```

3. Használjuk az alábbi definíciókat!

```
int n;
double *x, max, min, atlag;
```

Olvassuk be az adatok számát az *n* változóba! Foglaljuk le a memóriát az *x* tömb számára! Töltsük fel a tömböt adatokkal! Keressük meg a tömb legnagyobb és legkisebb elemét! A tömb elemeit osszuk el a legnagyobb és a legkisebb adatok átlagával, és jelenítsük meg a megváltozott értékeket! (GYAK4_3)

A *x* tömb feltöltése és a legkisebb, valamint a legnagyobb adat keresése:

```
cout << "Adatok szama: "; cin >> n;
x = new double[n];
if (!x)
{
    cerr << "Nincs eleg memoria" << endl;
}

for (i = 0; i < n; i++)
{
    cout << i << ". adat: ";
    cin >> x[i];
}
```

```
// legkisebb és a legnagyobb adat keresése
min = x[0]; max = x[0];
for (i = 1; i < n; i++)
{
    if (x[i] < min) min = x[i];
    if (x[i] > max) max = x[i];
}
atlag = (min+max)/2;
. . .
```

A program futásának eredménye:

Adatok szama: 4	Min: -6
0. adat: 12	Max: 12
1. adat: -6	Normalo tenyezo: 3
2. adat: 4	Normalt adatok
3. adat: 7	4
	-2
	1.33333
	2.33333

4. Használjuk az alábbi definíciókat!

```
int db;
int x[15];
```

Olvassuk be a rendezni kívánt (max.15) adatok számát (*db*), majd olvassuk be az egész számokat az *x* tömbbe! Rendezzük a tömböt fordított sorrendbe, azaz cseréljük fel az első elemet az utolsóval, majd a másodikat az utolsó előttivel stb.! (GYAK4_4)

Az *x* tömb fordított sorrendbe rendezése:

```
if (db % 2 == 0) db1 = db/2;
else db1 = db/2 +1;
for (i = 0; i < db1; i++)
{
    seged = x[i];
    x[i] = x[db-1 - i];
    x[db-1 - i] = seged;
}
. . .
```

A program futásának eredménye:

Elemek szama (max. 15): 4	A fordított sorrendu tomb
0. elem = 1	2
1. elem = 5	3
2. elem = 3	5
3. elem = 2	1