C++ pointerek

szerző: Nikházy László
előadó: Németh Zsolt

2025. szeptember 12.

Rejtélyes példa

Mit ír ki az alábbi program?

```
int t[] = {1, 2};
cout << (*t)[t]*t[*t];
```

Rejtélyes példa – Megoldás

Az eredmény: **4** (hiszen $2 \cdot 2 = 4$).

Most nem részletezzük, a végén visszatérünk rá!

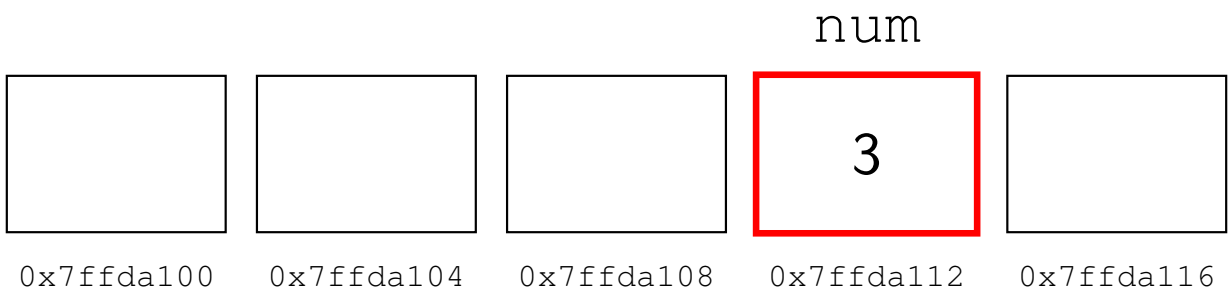
1. találós kérdés

Mit ír ki az alábbi program?

```
int num = 3;
cout << &num;
```

1. találós kérdés – Megoldás

Az & az ún. címképző (address-of) operátor.
Az eredmény: **a változó címe a memóriában** Például: `0x7ffda112`.
A számítógép memóriája elképzelhető dobozok sorozataként, ahol mindegyik doboznak van egy sorszáma (címe).



A `&num` nem a tartalom (3), hanem a pirossal keretezett memóriahely címe.

- Milyen szám ez a `0x7ffda112` ?
- Miért nem egyesével nőnek a dobozok sorszámai? Miért pont négyesével?

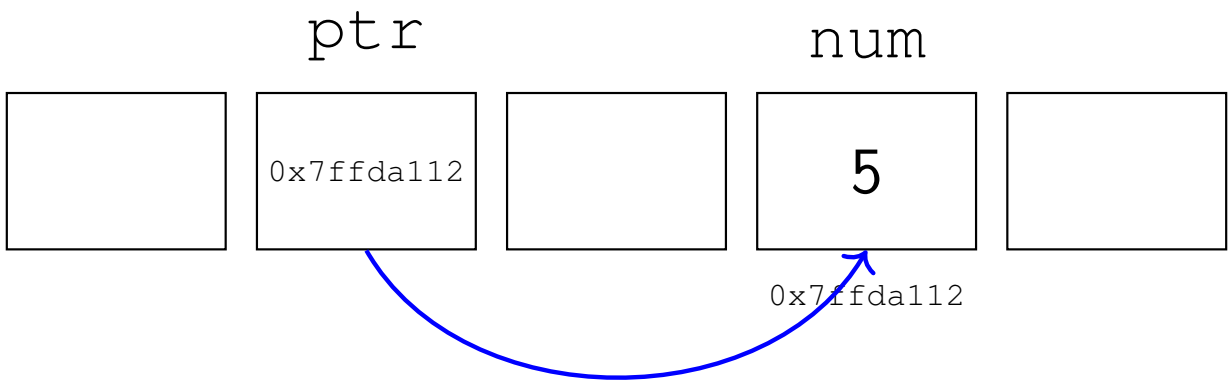
2. találós kérdés

Mit ír ki az alábbi program?

```
int num = 3;
int* ptr = &num;
num = 5;
cout << *ptr;
```

2. találós kérdés – megoldás

Az eredmény: `5`.
Az `int* ptr` egy úgynevezett **pointer** változó.
A **`*` a dereferálás operátor:** kiolvassa az adott címen lévő változó értékét.



Magyarázat: A `ptr` a `num` változó címét tárolja. A `*ptr` kiolvassa a `num` aktuális értékét, ami 5.

3. találós kérdés

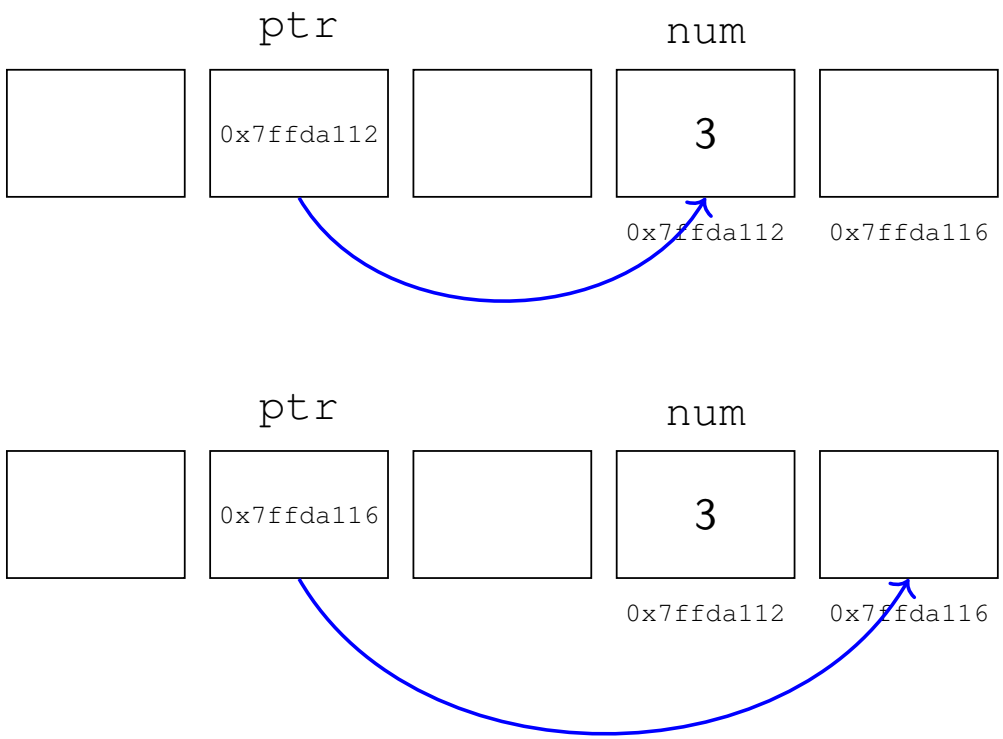
```
int num = 3;
int *ptr = &num;
cout << ptr++ << endl;
cout << ptr << endl;
```

3. találós kérdés – megoldás

A `ptr++` a pointert növeli: a jelenlegi értékét adja vissza, majd egy hellyel továbblép. A pointer növelése **nem bájtonként**, hanem a mutatott típus méretével történik (`int` → 4 bájt). Tehát a kimenet ilyesmi:

```
0x7ffda112
0x7ffda116
```

Valójában a mai gépeken kétszer ilyen hosszúak a pointerek, miért?



4. találós kérdés

```
int num = 3;
int *ptr = &num;
cout << sizeof(num) << "␣";
cout << sizeof(*ptr) << "␣";
cout << sizeof(ptr) << endl;
```

4. találós kérdés – Megoldás

Kimenet (64 bites gépen):

```
4 4 8
```

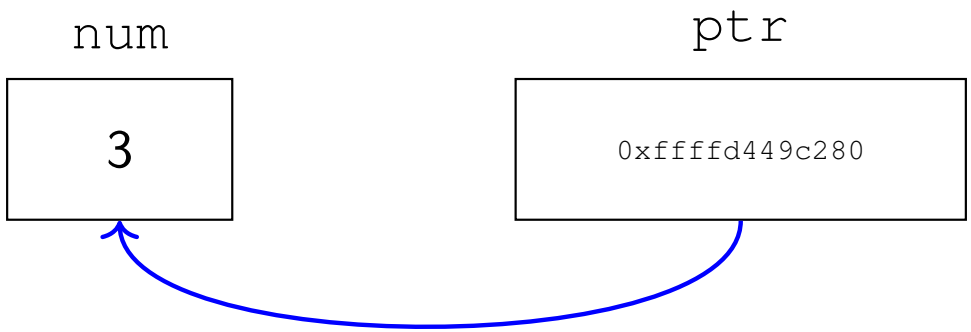
Magyarázat:

A `sizeof` operátor visszaadja egy típus vagy kifejezés tárhely-igényét bájtokban.

`sizeof(num)` = 4 bájt (egész mérete).

`sizeof(*ptr)` = 4 bájt, mert a `*ptr` típusa `int`.

`sizeof(ptr)` = 8 bájt, mert a pointer maga egy memóriacím, ami 64 biten, azaz 8 bájton tárolódik.



5. találós kérdés

```
int a[3] = {1,2,3};
cout << a << "␣" << &a[0] << "␣" << a[0];
```

5. találós kérdés – Megoldás

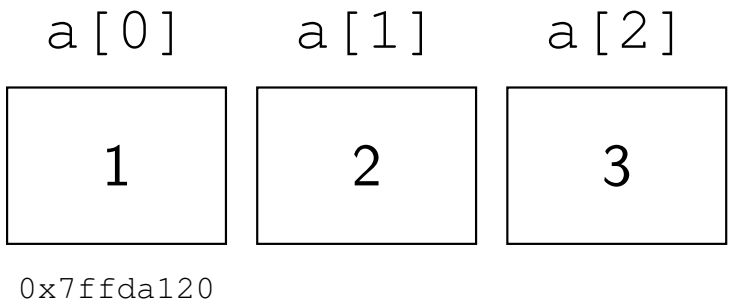
Kimenet: a tömb első elemének memóriacíme kétszer, majd az `1` érték. Például: `0x7ffda120 0x7ffda120 1`

Magyarázat:

A tömb neve (`a`) automatikusan az első elem címére konvertálódik.

`&a[0]` szintén az első elem címe.

`a[0]` az első elem értéke.



6. találós kérdés

```
int a[] = {1,2,3};
int* ptr = a;
cout << a << "␣" << ptr << "␣" << *ptr;
```

6. találós kérdés – Megoldás

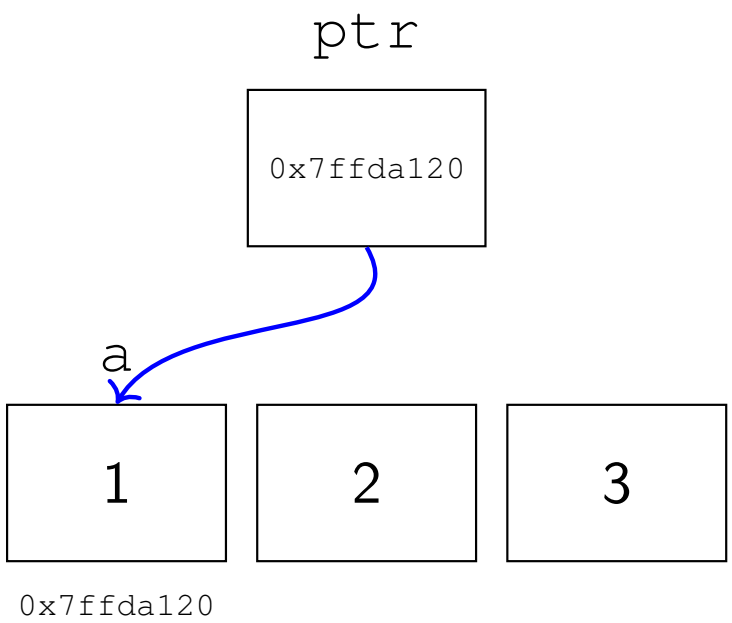
Kimenet: a tömb első elemének memóriacíme kétszer, majd az `1` érték. Például: `0x7ffda120 0x7ffda120 1`

Magyarázat:

A tömb neve (`a`) mutatóként viselkedik, az első elem címére mutat.

`ptr` ugyanoda mutat, mivel `ptr = a`.

`*ptr` a mutató dereferálása, vagyis az első elem értéke.



7. találós kérdés

```
int a[3] = {10,20,30};
cout << a[2] << "␣" << *(a+2);
```

7. találós kérdés – Megoldás

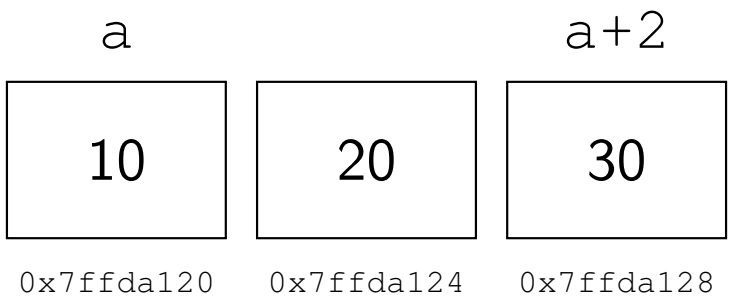
Kimenet: `30 30`

Magyarázat:

`a[2]` a tömb harmadik elemének értéke, vagyis `30`.

A `a` mutatóként az első elem címét adja. `a+2` = „két int mérettel arrébb” = harmadik elem címe.

A `*(a+2)` dereferálása a harmadik elem értékét adja: `30`.



Érdekesség: *Mivel az* `a[2]` *ténylegesen* `*(a+2)` *-re fordul le, ezért működik a* `2[a]` *jelölés is, mivel* `*(2+a)` *is ugyanaz az elem.*

8. találós kérdés

```
int a[5];
cout << sizeof(a) << "␣";
cout << sizeof(*a) << "␣";
cout << sizeof(&a);
```

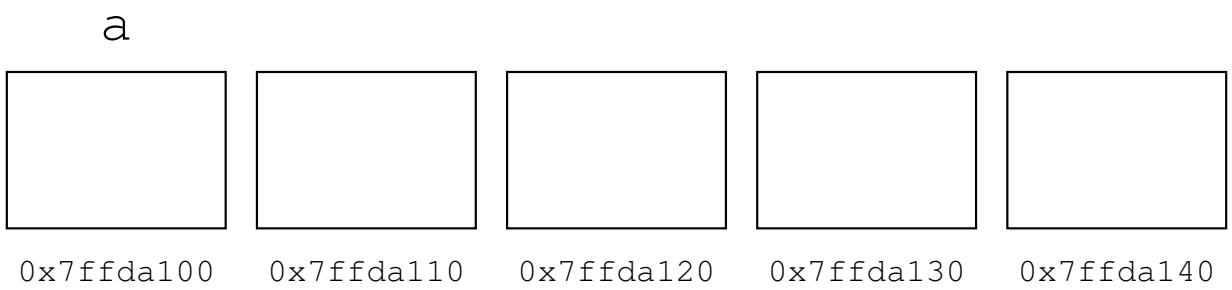
8. találós kérdés: megoldás

Kimenet példa (64-bites rendszer): `20 4 8`
Magyarázat: A **sizeof operátor** visszaadja a kifejezés típusának méretét byte-ban.

`sizeof(a)`: a teljes tömb mérete (5 $\times$ sizeof(int)), 20 byte.

`sizeof(*a)`: a tömb első elemének mérete (sizeof(int)), 4 byte.

`sizeof(a)`: a tömb címének mérete, azaz pointer mérete (8 byte 64-bites rendszeren).



9. találós kérdés

```
struct letters { char a, b; };
letters c = {'x', 'y'};
cout << sizeof(c);
```

9. találós kérdés: megoldás

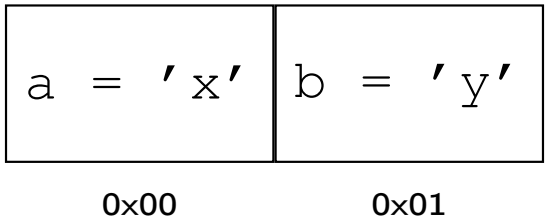
Kimenet: `2`
Magyarázat:

`struct letters` két `char`-ot tartalmaz.

Egy `char` általában 1 bájt.

A struct mérete a legtöbb rendszeren `2` egyes rendszereken **padding** miatt a `4`-re kerekítik.

A konkrét méret függ a compiler és az architektúra szabályaitól.



Tanulság: Struct-oknál előfordulhat, hogy figyelembe kell venni a padding-et és az igazítást (*alignment*) a memóriaméret számolásánál.

10. találós kérdés

```
int* p;
*p = 42;
cout << *p;
```

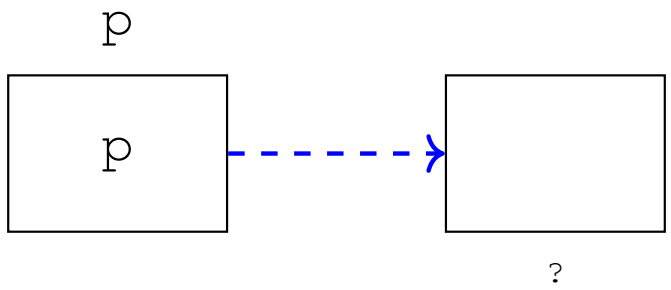
10. találós kérdés: megoldás

Válasz: Undefined behavior
Magyarázat:

`int* p;` – deklaráljuk a pointert, de még nem mutat érvényes memóriacímre.

`*p = 42;` – dereferálunk egy inicializálatlan pointert, ami **undefined behavior**-t okoz.

A program lefutása nem definiált: összeomolhat (segmentation fault), vagy „véletlenszerű” helyre írhat a memóriába.



Tanulság: Mielőtt dereferálunk egy pointert, mindig biztosítani kell, hogy érvényes memóriacímet tartalmazzon.

`new` operátor: egész szám lefoglalása

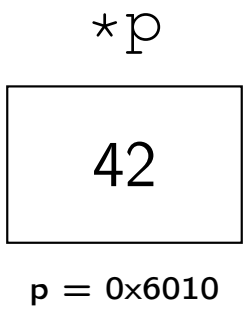
```
int* p = new int;
*p = 42;
cout << *p;
```

Magyarázat:

A `new int` lefoglal a memóriából egy egész számnak helyet.

`p` a lefoglalt memória címét tartalmazza.

A `*p` dereferálásával írhatjuk/olvashatjuk az értéket.



new operátor: tömb lefoglalása

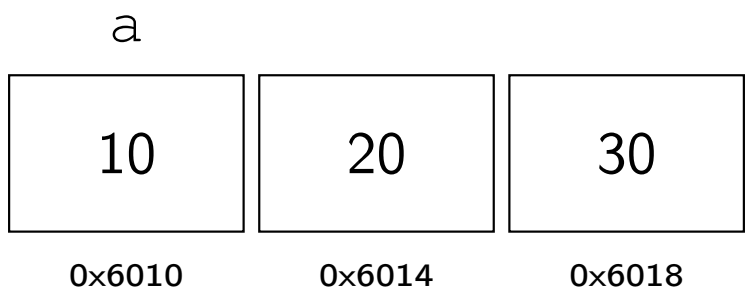
```
int* a = new int[3];
a[0] = 10; a[1] = 20; a[2] = 30;
cout << a[2] << " " << *(a+2);
```

Magyarázat:

`new int[3]` lefoglal 3 egymás utáni int-et.

A `a` mutató a tömb első elemére mutat.

Olvashatunk indexeléssel: `a[i]`
és pointer aritmetikával is: `*(a+i)`.



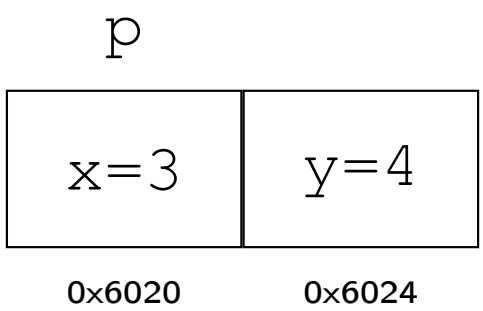
new operátor: struct lefoglalása

```
struct Point { int x, y; };
Point* p = new Point;
(*p).x = 3;
p->y = 4;
cout << p->x << " " << (*p).y;
```

Magyarázat:

`new Point` lefoglal egy `Point` struct-ot.

`p->x` és `(*p).x` ugyanazt az értéket érik el.



Memória modell

Kód szegmens

A program bináris (gépi) kódja.

Adat szegmens

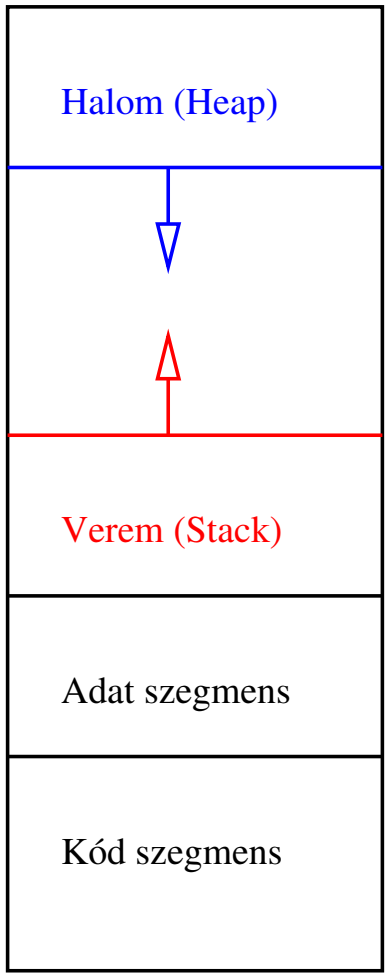
A program globális változóinak helye, azaz az olyan változóké, melyeket a main függvényen kívül deklaráltunk.

Verem

A függvények paramétereinek és lokális változóinak itt foglalódik memória. Függvény hívásakor foglalódik, terminálásakor felszabadul a memória.

Halom

A dinamikus változók számára itt foglalódik memória. A felszabadításról külön kell gondoskodni.



(Egyszerűsített) memória modell

`delete` operátor: memória felszabadítása

A `delete` operátorral a `new`-val lefoglalt memóriát lehet felszabadítani. Ha tömböt foglaltunk a `new[]` operátorral, akkor `delete[]` szükséges.

```
int* p = new int(42);
delete p;    // Egyetlen int felszabadítása

int* t = new int[5];
delete[] t; // Tömb felszabadítása

struct Point { int x, y; };
Point* q = new Point{1,2};
delete q;    // Struktúra felszabadítása
```

Nincs automatikus garbage collection!

A lefoglalt memória csak a `delete` hatására vagy a program végén szabadul fel, még akkor is, ha egyébként látható, hogy már nem fogjuk használni.

11. találós kérdés

Mit ír ki, illetve mi történik?

```
int* p = new int(7);
delete p;
cout << *p;
```

11. találós kérdés

Mit ír ki, illetve mi történik?

```
int* p = new int(7);
delete p;
cout << *p;
```

Magyarázat

`new int(7)`: lefoglal egy `int` típusú memóriát a heap-en, és inicializálja az értékét `7`-re.

`delete p`: felszabadítja a korábban lefoglalt memóriát.

Ez egy *dangling pointer* példa: a pointer még mindig mutat a felszabadított memóriára.

A `cout << *p;` *use after free*-t eredményez, ami **undefined behavior**, tehát bármi történhet: futási hiba, váratlan érték, vagy akár az is lehet, hogy nem jelez hibát.

12. találós kérdés

```
int* p = new int[3]{1,2,3};
cout << p[1];
delete p;
```

12. találós kérdés

```
int* p = new int[3]{1,2,3};
cout << p[1];
delete p;
```

Kimenet: `2` vagy futási hiba (pl. AddressSanitizer hibát dob)

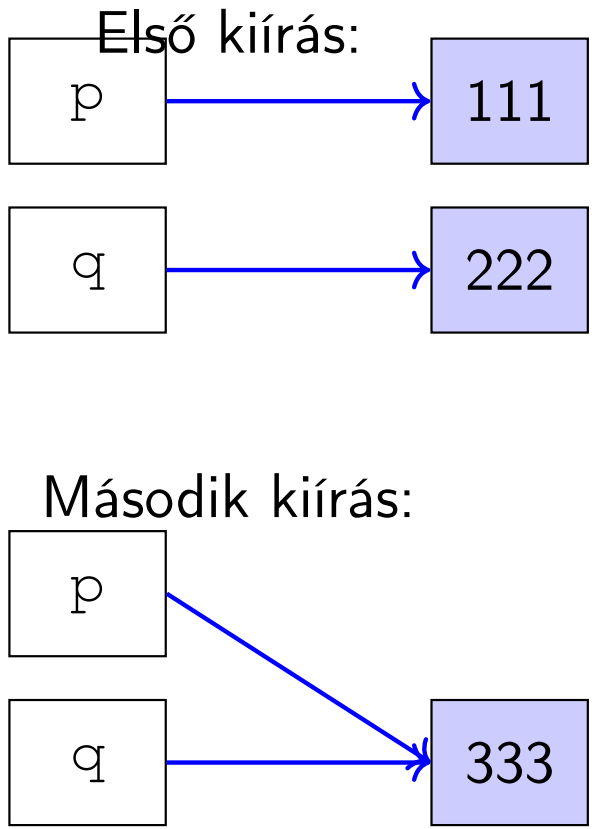
A hiba: tömböt `delete[]` -tel kell felszabadítani, nem sima `delete`-tel.

13. találós kérdés

```
int *p = new int;
int *q = new int;
*p = 111; *q = 222;
cout << *p << "␣" << *q << endl;
p = q;
*q = 333;
cout << *p << "␣" << *q << endl;
```

13. találós kérdés

```
int *p = new int;
int *q = new int;
*p = 111; *q = 222;
cout << *p << "␣" << *q << endl;
p = q;
*q = 333;
cout << *p << "␣" << *q << endl;
```



Kimenet: `111 222 333 333`

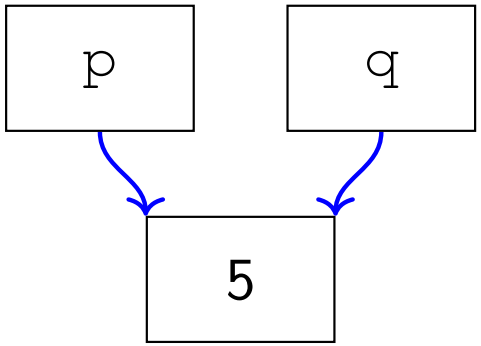
Magyarázat: A `p = q;` után a `p` pointer is ugyanarra a memóriacímre mutat, mint `q`. Az első `new int` által lefoglalt memória elveszett (memóriaszivárgás).

14. találós kérdés

```
int* p = new int(5);
int* q = p;
cout << *q;
delete p;
delete q;
```

14. találós kérdés

```
int* p = new int(5);
int* q = p;
cout << *q;
delete p;
delete q;
```



Kimenet: `5` vagy futási hiba (pl. AddressSanitizer hibát dob)

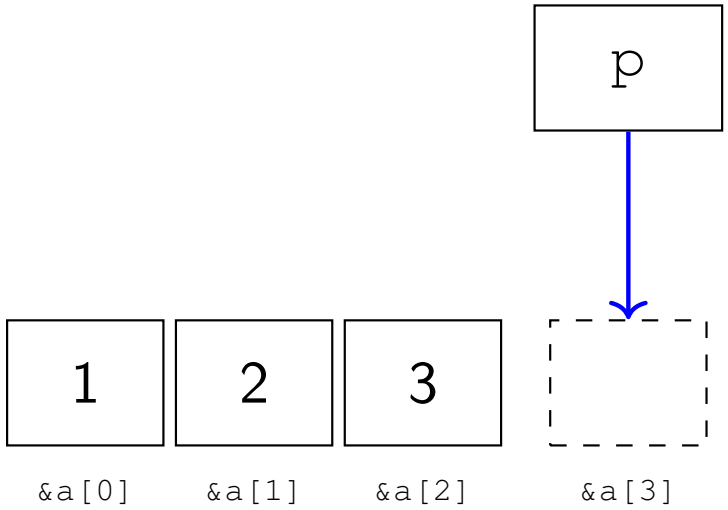
A hiba: kétszeres felszabadítás (*double free*) $\rightarrow$ undefined behavior.

15. találós kérdés

```
int a[3] = {1,2,3};
int* p = a+3;
cout << (p == &a[3]);
```

15. találós kérdés

```
int a[3] = {1,2,3};
int* p = a+3;
cout << (p == &a[3]);
```



Kimenet: `1` (igaz).

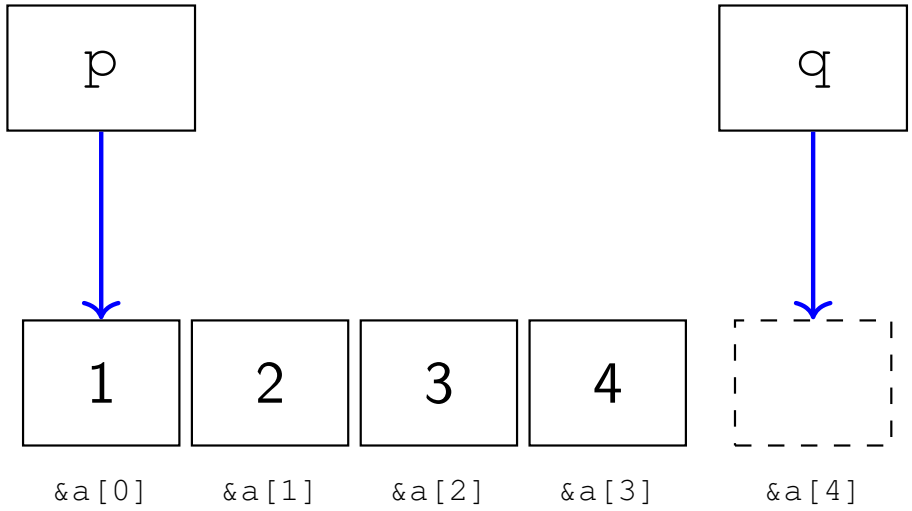
Magyarázat: az `a+3` a tömb utáni első címre mutat, ami ugyanaz, mint `&a[3]`. Ez **érvényes cím**, de a **dereferálása (`*p`) már hibás** lenne (*one-past-the-end szabály*).

16. találós kérdés

```
int a[4] = {1,2,3,4};
int* p = &a[0];
int* q = &a[4];
cout << q - p;
```

16. találós kérdés

```
int a[4] = {1,2,3,4};
int* p = &a[0];
int* q = &a[4];
cout << q - p;
```



Kimenet: `4`.

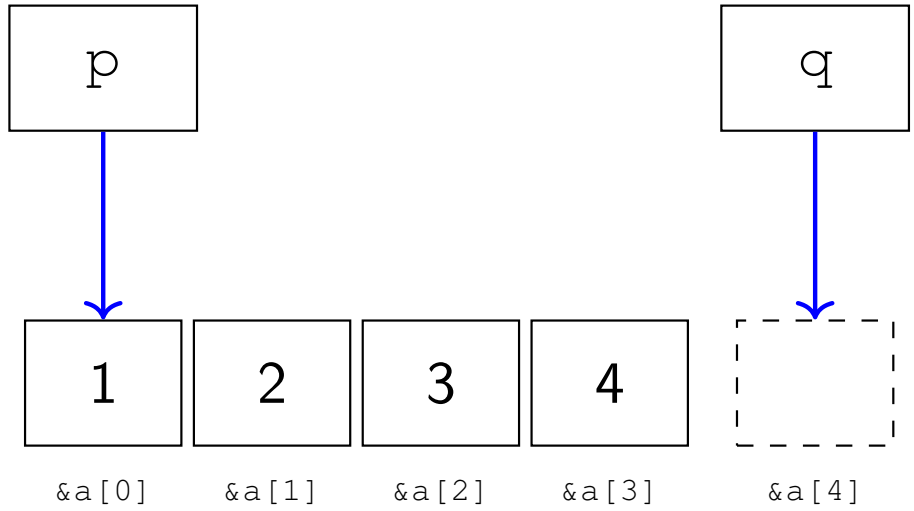
Magyarázat: pointerek kivonásakor a különbség az elemek számában értendő. Itt a két cím között pontosan 4 `int`-nyi távolság van, ezért `q - p = 4`.

17. találós kérdés

```
int a[4] = {1,2,3,4};
char* p = (char*)a;
char* q = (char*)(a + 4);
cout << q - p;
```

17. találós kérdés

```
int a[4] = {1,2,3,4};
char* p = (char*)a;
char* q = (char*)(a + 4);
cout << q - p;
```



Kimenet: `16`.

Magyarázat: `a+4` az int tömb utáni cím, ami 4 darab `int`-tel van arrébb. Mivel 1 int = 4 byte, ezért a két cím közötti különbség 16 byte. A `char*` pointereknél a távolság mindig byte-ban számítódik, így `q - p = 16`.

18. találós kérdés

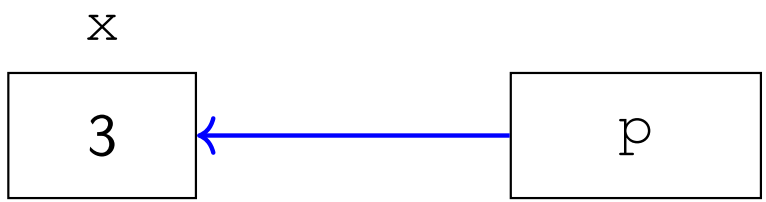
```
void inc(int* p) {
    *p++;
}

int main() {
    int x = 3;
    inc(x);
    cout << x << "\n";
}
```

18. találós kérdés

```
void inc(int* p) {
    *p++;
}

int main() {
    int x = 3;
    inc(x);
    cout << x << "\n";
}
```



Kimenet: fordítási hiba!

Magyarázat: Az `inc` függvény `int*`-ot vár paraméterként, de a híváskor az `x` nevű `int` változót adjuk át (nem a címét). Ezért a kód nem fordul le.

19. találós kérdés

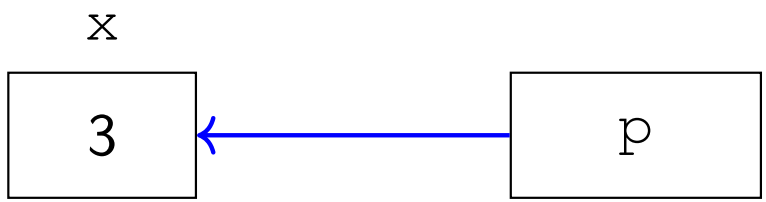
```
void inc(int* p) {
    *p++;
}

int main() {
    int x = 3;
    inc(&x);
    cout << x << "\n";
}
```

19. találós kérdés

```
void inc(int* p) {
    *p++;
}

int main() {
    int x = 3;
    inc(&x);
    cout << x << "\n";
}
```



Kimenet: `3`

Magyarázat: A `*p++` kifejezés nem az `*p`-t növeli, hanem a pointert (`p`) lépteti eggyel előrébb. Ezért az `x` változó értéke nem változik.

20. találós kérdés

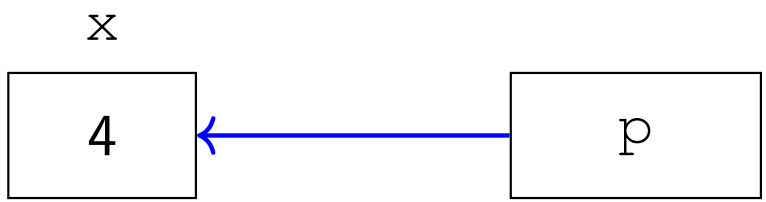
```
void inc(int* p) {
    ++*p;
}

int main() {
    int x = 3;
    inc(&x);
    cout << x << "\n";
}
```

20. találós kérdés

```
void inc(int* p) {
    ++*p;
}

int main() {
    int x = 3;
    inc(&x);
    cout << x << "\n";
}
```



Kimenet: `4`

Magyarázat: A `++*p` először dereferálja a pointert (`*p`), majd növeli az ott tárolt értéket. Így az `x` változó értéke eggyel nő.

21. találós kérdés

```
void inc(int* p) {
    ++*p;
}

int main() {
    long long x = 3;
    inc(x);
    cout << x << "\n";
}
```

21. találós kérdés

```
void inc(int* p) {
    ++*p;
}

int main() {
    long long x = 3;
    inc(x);
    cout << x << "\n";
}
```

Futásidőben: segmentation fault, de fordítási hiba is lehet (fordító verziótól függ).

Magyarázat:

`inc` `int*`-et vár, de `x` típusa `long long`.

Sok fordító csak warningot ad, a program lefut, de `p` értéke nem értelmezhető címet tartalmaz.

Emiatt a dereferálás (`++*p`) futásidőben *segmentation fault*-hoz vezet.

Modern, szigorú fordítók már hibát jeleznek fordításkor.

Pointer vs. referencia: növelő függvény

```cpp
// C-stílusú
void inc(int* p) {
    (*p)++;
}
int main() {
    int x = 3;
    inc(&x);
    cout << x << "\n";
}
```

```cpp
// C++ referencia
void inc(int& r) {
    r++;
}
int main() {
    int x = 3;
    inc(x);
    cout << x << "\n";
}
```

Magyarázat:

- Pointeres verzió: a függvény paramétere memóriacímet kap, a dereferálás szükséges (`*p`). Könnyű lehagyni a * -ot, vagy a zárójeleket, és meghíváskor az & jelet.
- Referencia: a változóra egy alternatív név. A dereferálás automatikus, a függvényhívás egyszerűbb (`inc(x)`).
- Mindkét esetben a kimenet: `4`.

Referencia: további példák

```
// Egyszerű referencia
int x = 5;
int& r = x;
r = 10;
// x értéke is 10 lesz
```

```
// Vektor referenciával
std::vector<int> v = {1,2,3};
for(int& val : v) {
    val *= 2;
// minden elem kétszerezve
}
```

Megjegyzések:

- A referencia mindig egy létező változóra mutat, nem lehet null.
- A referencia a létrejötte után nem változtatható, mindig ugyanarra a változóra mutat.
- Referencián keresztül közvetlenül olvasható és írható a célváltozó értéke.
- Függvényparaméterként a referencia lehetővé teszi a változó módosítását anélkül, hogy pointert kellene átadni.
- Iterálásnál `for (auto x : v)` közvetlenül módosíthatja a tömb vagy vektor elemeit.

22. találós kérdés

```
int* p;
void foo(int x) {
    p = &x;
}
int main() {
    foo(5);
    cout << *p << "\n";
}
```

22. találós kérdés

```
int* p;
void foo(int x) {
    p = &x;
}
int main() {
    foo(5);
    cout << *p << "\n";
}
```

Futásidőben: undefined behavior (dangling pointer).

Magyarázat:

Az `x` lokális változó a `foo` függvény verem-szegmensén van.

A `foo` visszatérésekor a verem ezen része felszabadul.

A globális `p` változó még az `x` címét tartalmazza, de az már nem érvényes.

A `*p` dereferálása futásidőben hibát okozhat, az eredmény nem definiált.

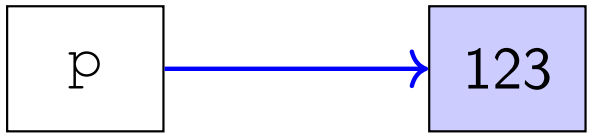
23. találós kérdés

```
int* p;
void f() {
    int x; x = 123;
    p = &x;
}
void g() {
    float a = 1.23;
}
int main() {
    int x = 1;
    f(); cout << *p << endl;
    g(); cout << *p << endl;
}
```

23. találós kérdés

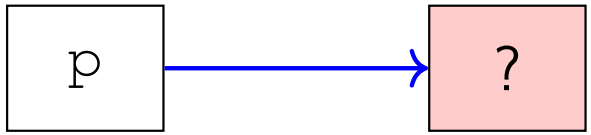
```
int* p;
void f() {
    int x; x = 123;
    p = &x;
}
void g() {
    float a = 1.23;
}
int main() {
    int x = 1;
    f(); cout << *p << endl;
    g(); cout << *p << endl;
}
```

f() után:



g() után:

lokális változó f()-ben



felszabadult terület

Kimenet (például): `123 32764` (nem definiált viselkedés)

Magyarázat:

Az `f()` lokális változója (`x`) a veremben jön létre, majd megszűnik.

A `p` pointer így érvénytelen (dangling pointer), már az `f()` visszatérése után is.

A `g()` hívásakor a verem újrahasználódik, felülírhatja a korábbi helyet.

24. találós kérdés

```
struct myvector {
  int len, *ptr;
  myvector(int n) { len = n; ptr = new int[n]; } // konstruktor
  int& operator[](int i) { return ptr[i]; }
  ~myvector() { delete[] ptr; } // destruktor
};

int main() {
  myvector v(10);
  v[9] = 42;
  cout << sizeof(v) << "␣" << v.len << "␣" << v[9] << "␣" << *v.ptr;
}
```

24. találós kérdés

```
struct myvector {
  int len, *ptr;
  myvector(int n) { len = n; ptr = new int[n]; } // konstruktor
  int& operator[](int i) { return ptr[i]; }
  ~myvector() { delete[] ptr; } // destruktor
};

int main() {
  myvector v(10);
  v[9] = 42;
  cout << sizeof(v) << "␣" << v.len << "␣" << v[9] << "␣" << *v.ptr;
}
```

Kimenet: platformtól függ, például: `16 10 42 -1094795586` (egy 64 bites linux gépen).

- A `v.len` = 10, `v[9]` = 42.
- A `*v.ptr` a tömb első elemére mutat (még nem inicializált, általában 0 vagy garbage érték).
- A sizeof(v) vajon miért 16?

vector memóriakép

A `myvector` struct két tagot tartalmaz: `len` (`int`) és `ptr` (`int*`).

A `sizeof(v)` csak a struct méretét adja (2 tag), a dinamikusan foglalt tömb mérete nem.

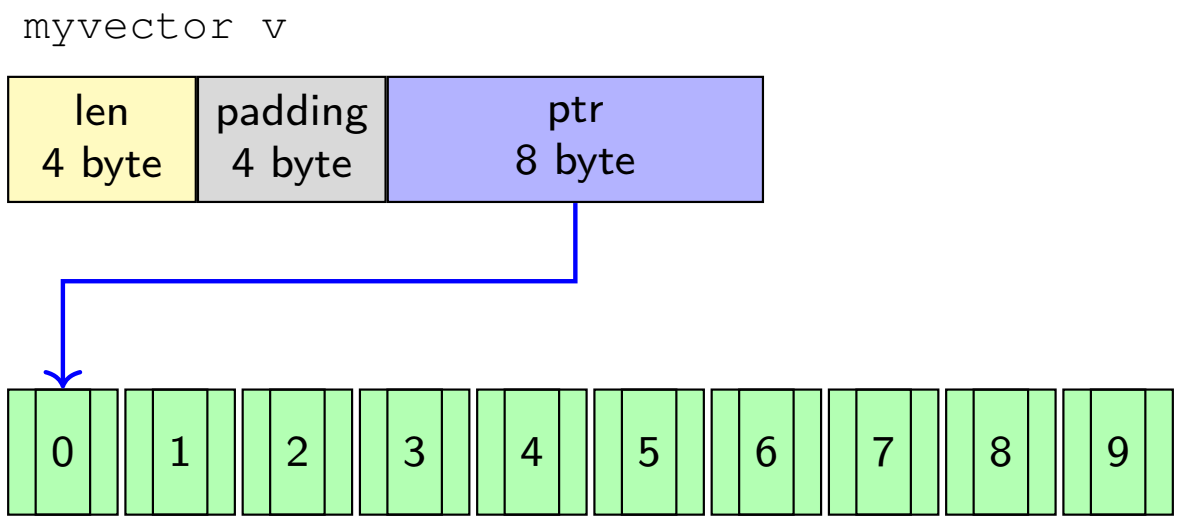
Miért 16 byte 64 bites rendszeren?

`len`: 4 byte

4 byte padding a pointer 8 byte-os alignmentjéhez

`ptr`: 8 byte

Összesen: $4 + 4 + 8 = 16$ byte



vector működése

Egy fokkal jobb verzió (paraméter nélküli konstruktorral, és nullptr vizsgálattal a destruktorban):

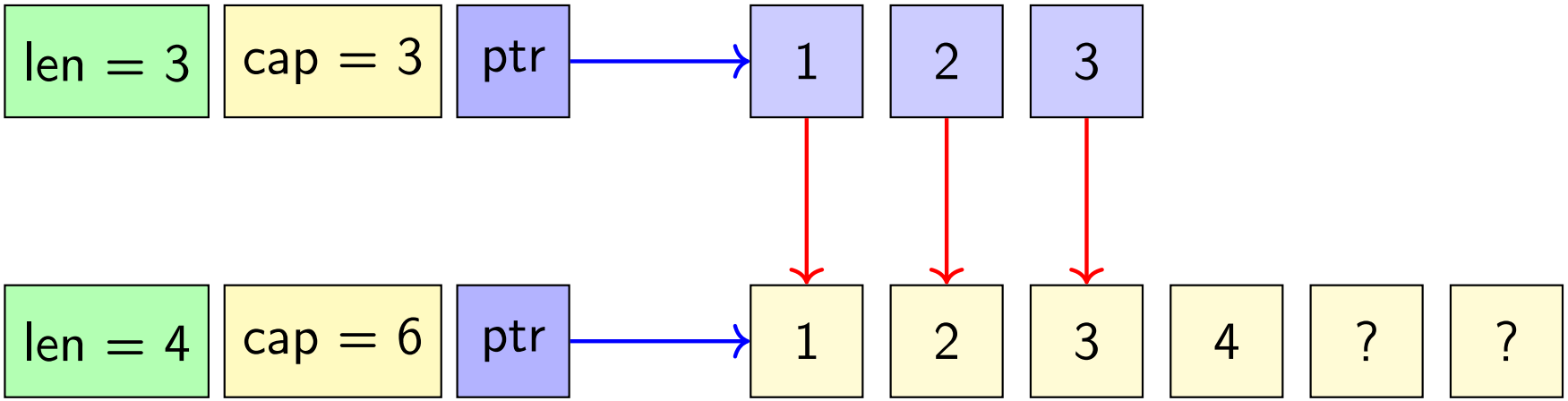
```
struct myvector {
    int len, *ptr;
    myvector() { len = 0; ptr = nullptr; } }
    myvector(int n) { len = n; ptr = new int[n]; } // konstruktor
    int& operator[](int i) { return ptr[i]; }
    ~myvector() {if (ptr != nullptr)} delete[] ptr; } // destruktor
};
```

Vajon hogy működik a `push_back` ?

push_back: újrafoglalás és átmásolás

Egy vector hossza és kapacitása is 3, majd egy új elemet teszünk bele (`push_back(4)`).

`push_back()` művelet esetén ha betelt a kapacitás, akkor az aktuális kapacitás kétszeresére foglal új memóriát, az első felére átmásolja az aktuális tartalmat. Az előzőleg lefoglalt memóriát felszabadítja.



push_back működése

```
struct myvector {
    int len, cap, *ptr;
    myvector() { len = 0; cap = 1; ptr = new int[1]; }
    myvector(int n) { len=cap=n; ptr = new int[cap]; }
    void push_back(int x) {
        if (len == cap) {
            cap *= 2;
            int* newptr = new int[cap];
            for(int i = 0; i < len; i++)
                newptr[i] = ptr[i];
            delete[] ptr;
            ptr = newptr;
        }
        ptr[len++] = x;
    }
    ~myvector() { delete[] ptr; }
};
```

25. találós kérdés

```
int n = ...;
myvector v;
for (int i = 0; i < n; i++) {
    v.push_back(i);
}
```

Kérdés: Legrosszabb esetben mennyi memóriát foglal el a `v` vector által fenntartott tömb?

25. találós kérdés

```
int n = ...;
myvector v;
for (int i = 0; i < n; i++) {
    v.push_back(i);
}
```

Kérdés: Legrosszabb esetben mennyi memóriát foglal el a `v` vector által fenntartott tömb?

Válasz: n elem helyett legfeljebb $2n - 2$ elemre is lefoglalhat.

Magyarázat:

Amikor betelik a vektor, új tömböt foglal (a `myvector` kétszer akkorát), és átmásolja az elemeket. Legrosszabb esetben az n-edik lépésben dupláz, amikor $n - 1$ elem van benne.

Ezért n elem után legfeljebb $2n - 2$ elemnyi memória lehet lefoglalva.

Rejtélyes példa - újra

Mit ír ki az alábbi program?

```
int t[] = {1, 2};
cout << (*t)[t]*t[*t];
```

Rejtélyes példa – Megoldás

```
int t[] = {1, 2};
cout << (*t)[t]*t[*t];
// *t másképp t[0], ami 1, vagyis:
cout << 1[t] * t[1];
// az 1[t] nem más, mint
// *(1+t) = *(t+1), azaz t[1]
cout << t[1] * t[1];
// A t[1] az 2.
cout << 2 * 2;
```

Tanács

Csak akkor használjunk pointereket, ha feltétlenül szükséges.

Mikor szükséges? Például dinamikus adatszerkezeteknél (következő előadás).