

ADATSZERKEZETEK ÉS ALGORITMUSOK

C++ ismétlés

C++ alapok

- Rendelkezésre álló eszköztár

- Változók
- Referenciák
- Pointerek
- Ezekből képezett tömbök
- Függvények
- Paraméterek átadása
- Eredmény visszaadása
- Vezérlési szerkezetek
 - Szekvencia
 - Ciklus
 - Elágazás
 - Többszörös elágazás

- Példák

```
int i;  
int & r = i;  
int * p = & i;  
int t[] = {1, 2, 3, 4};  
int fv() { return 1;}  
int square(int x) {  
    return x*x; }  
  
{ int i = 1; int j = 2;}  
while (feltétel) utasítás;  
for( ; ; ) utasítás;  
if (feltétel) utasítás;  
else utasítás;  
switch (feltétel) {  
case <eset1>: utasítások; <break>  
default:      utasítások;
```

Változók

- A változó
 - Lefoglalt memóriaterület, amelyben értéket lehet tárolni
 - Egész, valós, logikai, memóriaterület címe, karakter, ...
 - A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
 - Ez a változónév
- A változóhoz rendelt memóriaterület lefoglalása a deklarációkor történik:

- Változó deklaráció: típus és név meghatározása

- `int i;`

- Értékadás

- `i = 4;`

Ebben a példában 32 bites rendszert, valamint a könnyebb érthetőség kedvéért 4 bájtos rekeszeket feltételezünk.

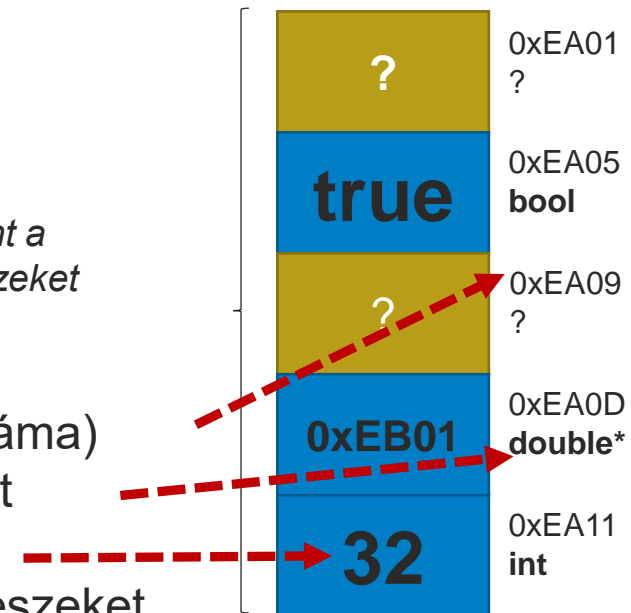
Memória – rekeszek sorozata

Minden rekesznek van címe (sorszáma)

Ismerjük a benne tárolt érték típusát

Ismerjük a benne tárolt értéket

Kék színnel jelezzük a lefoglalt rekeszeket



Változók

- A változó
 - Lefoglalt memóriaterület, amelyben értéket lehet tárolni
 - Egész, valós, logikai, memóriaterület címe, karakter, ...
 - A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
 - Ez a változónév
- A változóhoz rendelt memóriaterület lefoglalása a deklarációkor történik:
 - Változó deklaráció
 - `int i;`
 - Értékadás
 - `i = 4;`

Vegyük észre, hogy a memória bizonyos részein általunk nem ismert érték van. Ha nem adunk értéket egy változónak, attól még a mögöttes memóriaterület fog tartalmazni egy értéket.

?	0xEA01 ?
true	0xEA05 bool
?	0xEA09 ?
0xEB01	0xEA0D double*
32	0xEA11 int

Változók

- A változó

- Lefoglalt memóriaterület, amelyben értéket lehet tárolni
 - Egész, valós, logikai, memóriaterület címe, karakter, ...
- A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
 - Ez a változónév

- A változóhoz rendelt memóriaterület lefoglalása a deklarációkor történik:

- Változó deklaráció

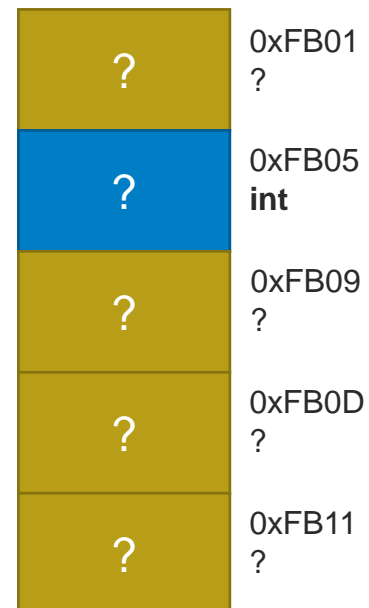


- `int i;`

- Értékadás

- `i = 4;`

i



Változók

- A változó

- Lefoglalt memóriaterület, amelyben értéket lehet tárolni
 - Egész, valós, logikai, memóriaterület címe, karakter, ...
- A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
 - Ez a változónév

- A változóhoz rendelt memóriaterület lefoglalása a deklarációkor történik:

- Változó deklaráció

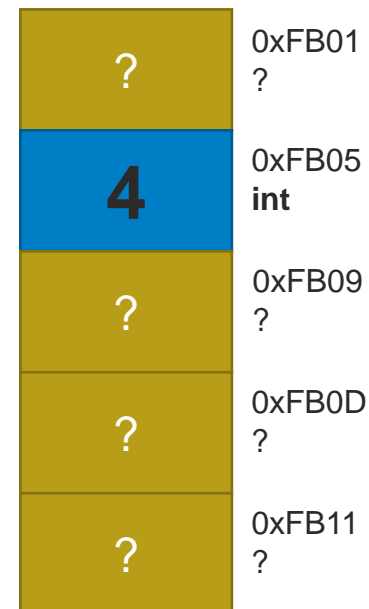


- `int i;`

- Értékadás

- `i = 4;`

i

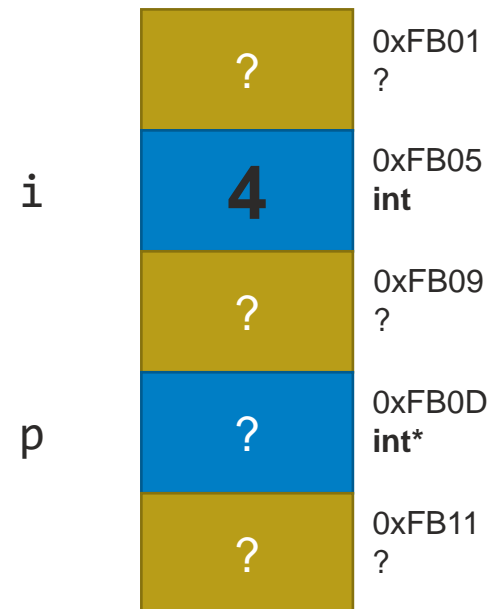


Mutató, pointer

- A mutató
 - Új memóriaterület kerül lefoglalása, mint közösleges változó esetén
 - Azonban a memóriaterületen memóriacímet tárolunk
 - Természetesen a mutatónak is van neve
 - Sőt típusa is, amivel meghatározzuk, hogy milyen típusú tároló memóriára tud hivatkozni
- A pointer létrehozása és használata a közösleges változóéhoz hasonló, a viselkedés a referenciához hasonló
 - Deklaráció
 - Típus és név megadása
 - `int * p;`
 - Értékadás – itt memóriacímet kell értékül adni
 - `p = & i;`
 - A mutatott memóriaterület elérése (**dereferencing**)
 - `*p = 6;`

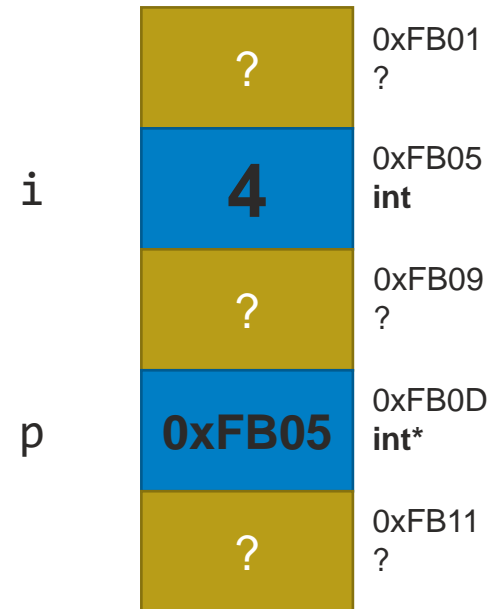
Mutató, pointer

- A mutató
 - Új memóriaterület kerül lefoglalása, mint közösleges változó esetén
 - Azonban a memóriaterületen memóriacímet tárolunk
 - Természetesen a mutatónak is van neve
 - Sőt típusa is, amivel meghatározzuk, hogy milyen típusú tároló memóriára tud hivatkozni
- A pointer létrehozása és használata a közösleges változóéhoz hasonló, a viselkedés a referenciához hasonló
 - Deklaráció
 - Típus és név megadása
 - ➔ • `int * p;`
 - Értékadás – itt memóriacímet kell értékül adni
 - `p = &i;`
 - A mutatott memóriaterület elérése (**dereferencing**)
 - `*p = 6;`



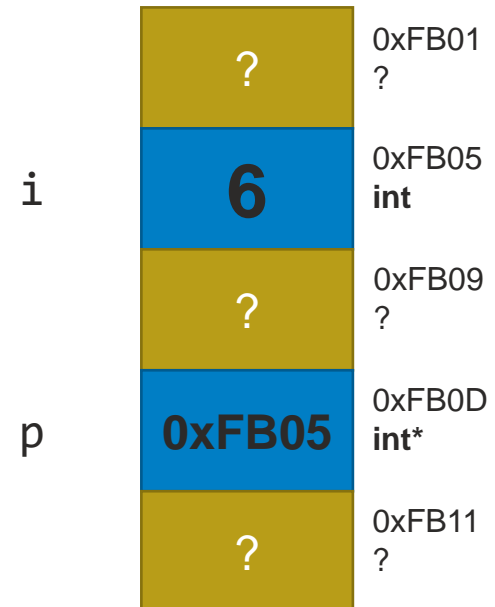
Mutató, pointer

- A mutató
 - Új memóriaterület kerül lefoglalása, mint közösleges változó esetén
 - Azonban a memóriaterületen memóriacímet tárolunk
 - Természetesen a mutatónak is van neve
 - Sőt típusa is, amivel meghatározzuk, hogy milyen típusú tároló memóriára tud hivatkozni
- A pointer létrehozása és használata a közösleges változóéhoz hasonló, a viselkedés a referenciához hasonló
 - Deklaráció
 - Típus és név megadása
 - `int * p;`
 - Értékadás – itt memóriacímet kell értékül adni
 - ➔ • `p = &i;`
 - A mutatott memóriaterület elérése (**dereferencing**)
 - `*p = 6;`



Mutató, pointer

- A mutató
 - Új memóriaterület kerül lefoglalása, mint közöséges változó esetén
 - Azonban a memóriaterületen memóriacímet tárolunk
 - Természetesen a mutatónak is van neve
 - Sőt típusa is, amivel meghatározzuk, hogy milyen típusú tároló memóriára tud hivatkozni
- A pointer létrehozása és használata a közöséges változóéhoz hasonló, a viselkedés a referenciához hasonló
 - Deklaráció
 - Típus és név megadása
 - `int * p;`
 - Értékadás – itt memóriacímet kell értékül adni
 - ➔ `p = &i;`
 - A mutatott memóriaterület elérése (**dereferencing**)
 - `*p = 6;`

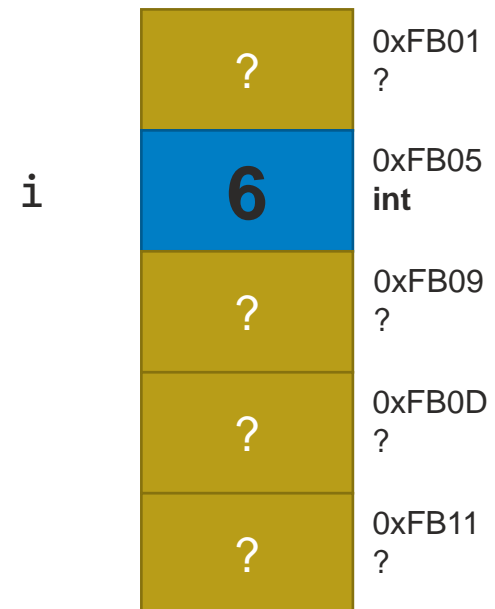


Referencia

- A referencia
 - Egy már lefoglalt területre új címke létrehozása
 - Ezután két (vagy több) névvel hivatkozhatunk ugyanoda
 - A megvalósítása általában pointer, technikailag konstans, **nemnull**, automatikusan dereferálódó mutató.
- Referencia esetén csak címkét hozunk létre, ezért meg kell adni a már lefoglalt területet, amire az új címkét tesszük
 - Deklaráció és inicializálás
 - Típus,név és létező változó megadása
 - `int & r = i;`
 - Értékadás
 - `r = 5;`

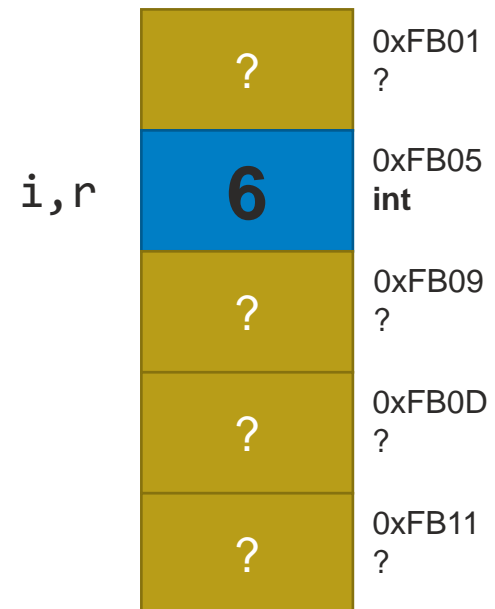
Referencia

- A referencia
 - Egy már lefoglalt területre új címke létrehozása
 - Ezután két (vagy több) névvel hivatkozhatunk ugyanoda
 - A megvalósítása általában pointer, technikailag konstans, **nemnull**, automatikusan dereferálódó mutató.
- Referencia esetén csak címkét hozunk létre, ezért meg kell adni a már lefoglalt területet, amire az új címkét tesszük
 - Deklaráció és inicializálás
 - Típus,név és létező változó megadása
 - `int & r = i;`
 - Értékadás
 - `r = 5;`



Referencia

- A referencia
 - Egy már lefoglalt területre új címke létrehozása
 - Ezután két (vagy több) névvel hivatkozhatunk ugyanoda
 - A megvalósítása általában pointer, technikailag konstans, **nemnull**, automatikusan dereferálódó mutató.
- Referencia esetén csak címkét hozunk létre, ezért meg kell adni a már lefoglalt területet, amire az új címkét tesszük
 - Deklaráció és inicializálás
 - Típus,név és létező változó megadása
 - ➔ • `int & r = i;`
 - Értékadás
 - `r = 5;`



Referencia

- A referencia

- Egy már lefoglalt területre új címke létrehozása
 - Ezután két (vagy több) névvel hivatkozhatunk ugyanoda
- A megvalósítása általában pointer, technikailag konstans, **nemnull**, automatikusan dereferálódó mutató.

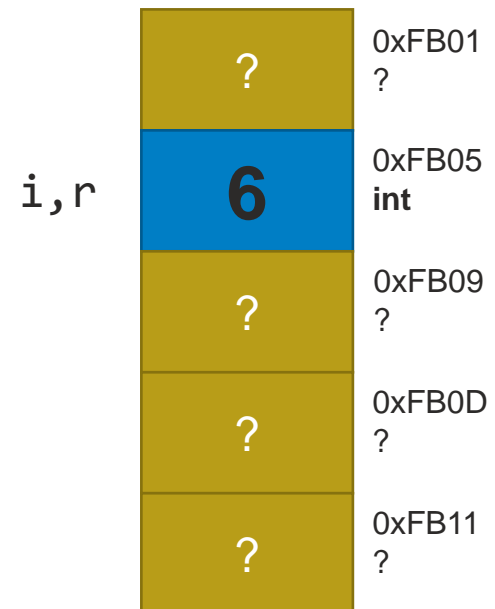
- Referencia esetén csak címkét hozunk létre, ezért meg kell adni a már lefoglalt területet, amire az új címkét tesszük

- Deklaráció és inicializálás
 - Típus,név és létező változó megadása

➔ `int & r = i;`

- Értékadás

- `r = 5;`



Változó, referencia, pointer

- Példák

```
1.  int    i  = 5;
2.  double d  = 4.0;
3.  int &   r  = i;
4.  r++;
5.  int *   p  = &i;
6.  (*p)++;
7.  int* & r2 = p;
8.  (*r2)++;
9.  int &   r3 = *p;
10. const int & k = 6;
11. int **  p2 = &p;
12. (**p2)++;
```

- Jelentése

```
1. Változó deklarálása és értékadás
2. Változó deklarálása és értékadás
3. Referencia létrehozása
4. Hozzáférés a változóhoz a referencián keresztül (az i értéke 6)
5. Új pointer, a változó címét tárolja
6. Hozzáférés a pointeren keresztül (az i értéke 7)
7. Másik referencia, a pointerre
8. Hozzáférés a pointerhez referencián keresztül (az i értéke 8)
9. Referencia a pointer által mutatott változóra (ez ugyanoda referál, ahova az r)
10. Konstans referencia konstans értékre
11. Pointer-re mutató pointer
12. Hozzáférés a pointer pointerén keresztül (az i értéke 9)
```

Próbáljuk ki!

- A 01_Valtozok projekt megnyitása után kipróbálhatod az eddigieket.
- Egészítsük ki a tesztprogramot és próbáljuk ki a konstans referenciát és a pointerre mutató referenciát!
- A & operátor mindig egy változó mögötti terület címét adja vissza.
- A * operátor mindig a memóriacímet követi és a mögöttes területet érjük el vele.
 - `int i = 5;`
 - `int * p = & i;`
 - `&>(*p) == p;`
- Az utóbbi sornak mindig igaznak kell lennie, mivel a mutató által mutatott terület címe a mutató értéke kell, hogy legyen.



auto típus

- Általában minden változó típusát pontosan meg kellett adni
 - Ez nagyon nehézkesé válhat template-ek használatakor
- Ennek könnyítésére van az auto kulcsszó
 - Olyan esetekben, amikor a kontextusból egyértelműen következik a típus, azt nem kell kiírni, az auto automatikusan kiválasztja.

```
auto a = 5;           // ez int lesz
auto b = 5.0f;        // ez float lesz
auto c = fuggvenyem(...); // ez az lesz, amit a függvény visszaad
```

- Kényelmes, de nehezen olvashatóvá is tudja tenni a kódot
 - Használni csak indokolt esetben tessék...



auto típus – structured binding

- Hivatkozás rész objektumokra nevekkal

- Pl. hivatkozás tömb elemeire:

```
int arr[2] = {1, 2};
```

```
auto [a, b] = arr;
```

// készül egy e[2] másolat arr-ról, a e[0]-ra b e[1]-re hivatkozik

```
auto& [c, d] = arr; // c arr[0]-ra d arr[1]-re hivatkozik
```

- Tuple like típusokra is működik és adatmezőkre:

```
struct S{ int a = 5; float b = 4.0f; };
```

```
auto [c, d] = S(); // c egy int, értéke 5; d egy float, értéke 4.0
```

- Referencia

- https://en.cppreference.com/w/cpp/language/structured_binding



Move szemantika

- Másolás vagy referencia készítése mellett lehetőségünk van arra is, hogy egy A változót B-be *moveoljunk*.
 - Ilyenkor B kvázi „ellopja” A tartalmát.

```
std::string a = "abc";  
std::string b = std::move(a);  
std::cout << a << std::endl;    // semmi  
std::cout << b << std::endl;    // abc
```
- Moveolni *rvalue referenceket* lehet.
 - Ilyenek az átmeneti objektumok (nincsenek változóhoz rendelve) és az `std::move`-val explicite annak jelölt kifejezések.
 - Követelmény még, hogy a típusnak legyen move konstruktora vagy move értékadó operátora.
- Egy alapos (és hosszú) cikk:
 - http://thbecker.net/articles/rvalue_references/section_01.html

Változók láthatósága

```
#include <iostream>
using namespace std;
```

```
int i;
int j;
```

```
int main()
{
```

```
    i = 1;
```

```
    j = 10;
```

```
    int k = 100;
```

```
    int j = 1000;
```

```
    {
```

```
        j++;
```

```
        int j = 2000;
```

```
        j++;
```

```
    }
```

```
}
```

• Globális:

- Függvényeken kívüli deklaráció
- Mindenhol hivatkozhatunk rájuk (ha nincs elfedés)

• Lokális:


- Blokkon belül deklarált
- Deklarációkor tárterület foglalódik le
- A blokk végén a tárterület felszabadul, a változó megszűnik.

Globális, lokális változók

```
#include <iostream>
using namespace std;
```

```
int i;
int j;
```

```
int main()
{
    i = 1;
    j = 10;
    int k = 100;
    int j = 1000;
    {
        j++;
        int j = 2000;
        j++;
    }
}
```



j	10	0xFB01 int
i	1	0xFB05 int
j	1001	0xFB09 int
j	2001	0xFB0D int
k	100	0xFB11 int

Globális, lokális változók

```
#include <iostream>
using namespace std;
```

```
int i;
int j;
```

```
int main()
{
```

```
    i = 1;
```

```
    j = 10;
```

```
    int k = 100;
```

```
    int j = 1000;
```

```
    {
```

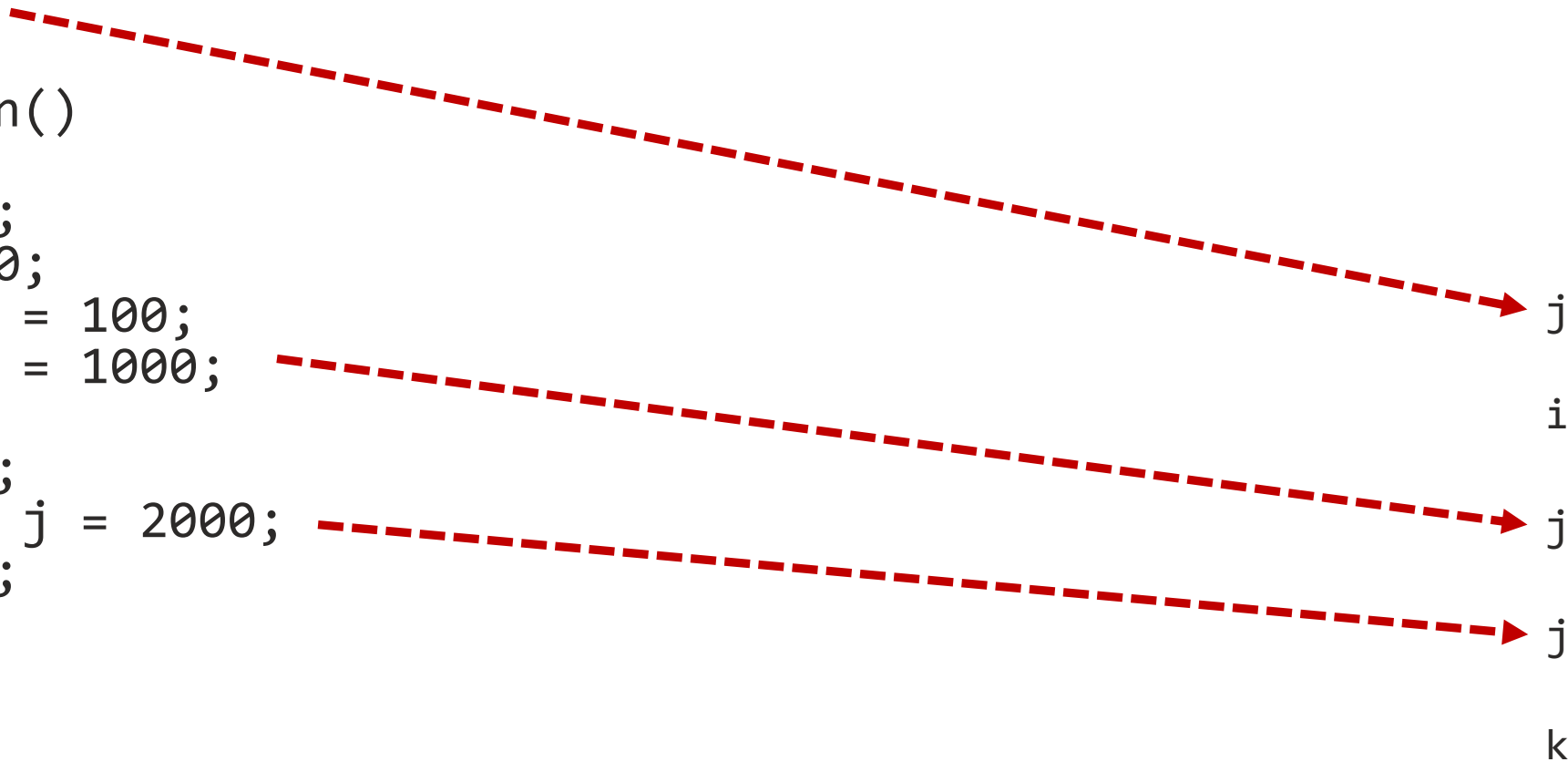
```
        j++;
```

```
        int j = 2000;
```

```
        j++;
```

```
    }
```

```
}
```



10	0xFB01 int
1	0xFB05 int
1001	0xFB09 int
2001	0xFB0D int
100	0xFB11 int

Eljárások és függvények

FÜGGVÉNY

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}

int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```

ELJÁRÁS

```
#include <iostream>
using namespace std;
void kiir ()
{
    cout << "Ezt írom ki!";
}

int main ()
{
    kiir ();
    return 0;
}
```

Paraméterátadás

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}
```

Formális paraméterek a és b.

addition (int a, int b)

```
int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```

z=addition (5,3);

Aktuális paraméterek értékei 5 és 3.

Paraméterátadás

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}
```

Formális paraméterek a és b.

addition (int a, int b)

```
int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```

z=addition (5,3);

Aktuális paraméterek értékei 5 és 3.

Paraméterátadás szabályai

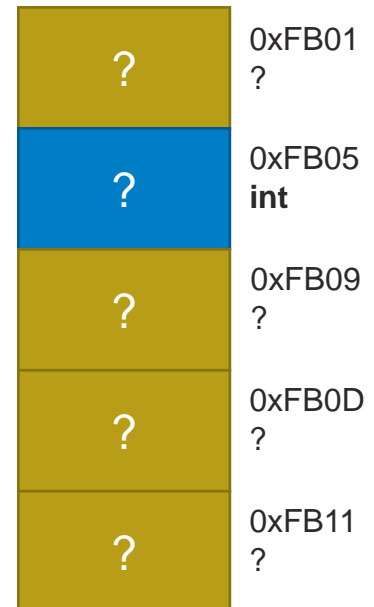
- C++-ban a **közönséges változó és pointer** esetén **érték szerinti paraméterátadás** történik.
 - Függvényhíváskor a formális paraméterek lokális változóként deklarálásra kerülnek
 - Ezt követően inicializálódnak az aktuális paraméterek értékével
 - Ez közönséges változó esetén az az érték, amit beletettünk
 - Ez pointer esetén a memóriacím
 - Ennek megfelelően az eredeti változó tartalmáról egy másolat készül egy másik memóriaterületre
 - Összefoglalva a formális paraméterek másik memóriaterületre hivatkoznak, mint az aktuális paraméterek!
- C++-ban a **referencia** formális paraméterek **esetén az aktuális paraméterre** (változóra) **egy (új) referencia kerül deklarálásra**.
 - Ennek megfelelően az eredetileg lefoglalt memóriaterületet címkézzük fel újra
 - Minden a formális paraméteren (lokális változón) keresztüli változtatás az eredeti memóriaterületet változtatja meg

Paraméterátadás bemutatása

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}
```

```
int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```

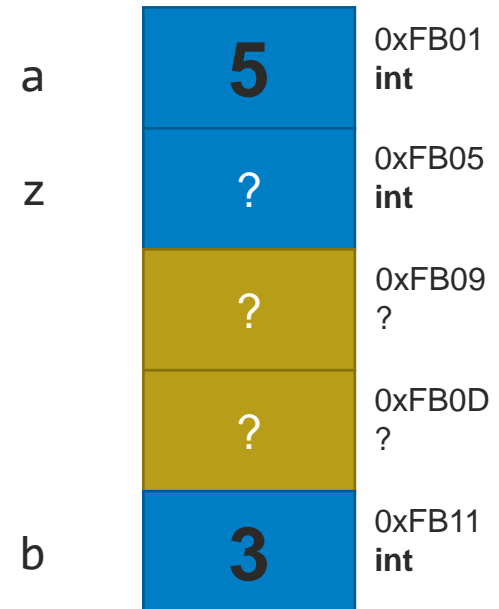
z



Paraméterátadás bemutatása

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}
```

```
int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```



Paraméterátadás bemutatása

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}
```

```
int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```

a	5	0xFB01 int
z	?	0xFB05 int
	?	0xFB09 ?
r	8	0xFB0D int
b	3	0xFB11 int

Visszatérési érték bemutatása

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}
```

```
int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```

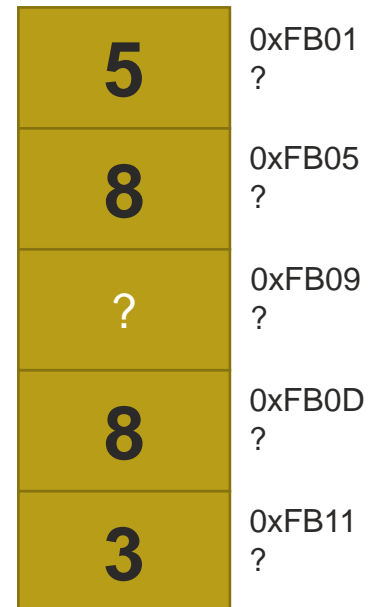
z

5	0xFB01 ?
8	0xFB05 int
?	0xFB09 ?
8	0xFB0D ?
3	0xFB11 ?

Paraméterátadás és visszatérési érték

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}
```

```
int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```



5	0xFB01 ?
8	0xFB05 ?
?	0xFB09 ?
8	0xFB0D ?
3	0xFB11 ?

Paraméterátadás és visszatérési érték

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;
    r = a + b;
    return r;
}
```

```
int main ()
{
    int z;
    z = addition (5,3);
    cout << "Eredmeny " << z;
    return 0;
}
```

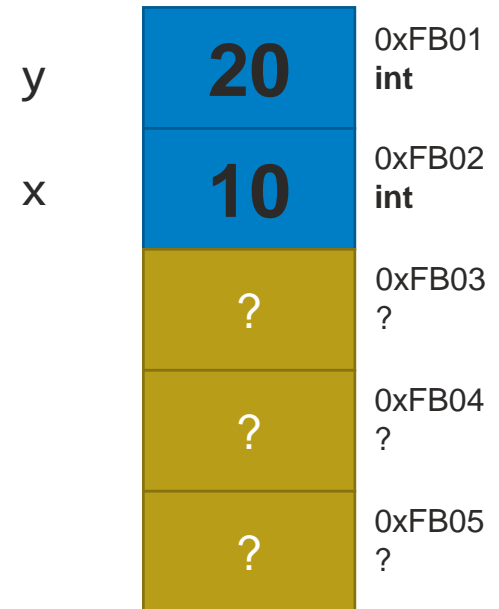
Vegyük észre, hogy a memóriában a korábban tárolt értékek továbbra is ott vannak, csupán egyikre sincs érvényes deklarált változó! Később azonban felülíródhatnak!

5	0xFB01 ?
8	0xFB05 ?
?	0xFB09 ?
8	0xFB0D ?
3	0xFB11 ?

Ugyanez referenciával és pointerrel

```
#include <iostream>
using namespace std;
void doSomething(int & a, int * b)
{
    a++;
    (*b)++;
}
```

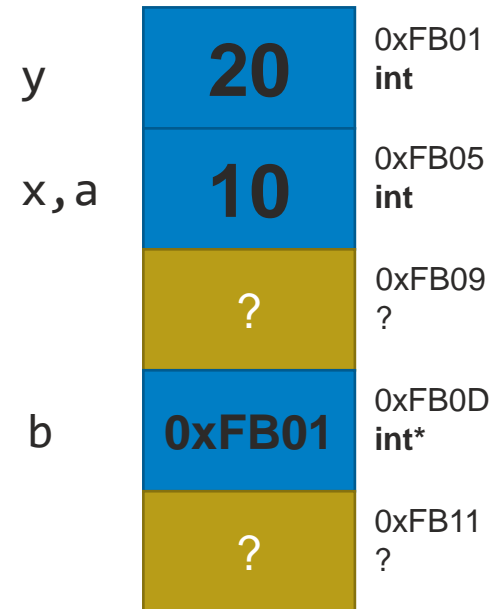
```
int main ()
{
    int x = 10;
    int y = 20;
    doSomething (x, &y);
    return 0;
}
```



Ugyanez referenciával és pointerrel

```
#include <iostream>
using namespace std;
void doSomething(int & a, int * b)
{
    a++;
    (*b)++;
}
```

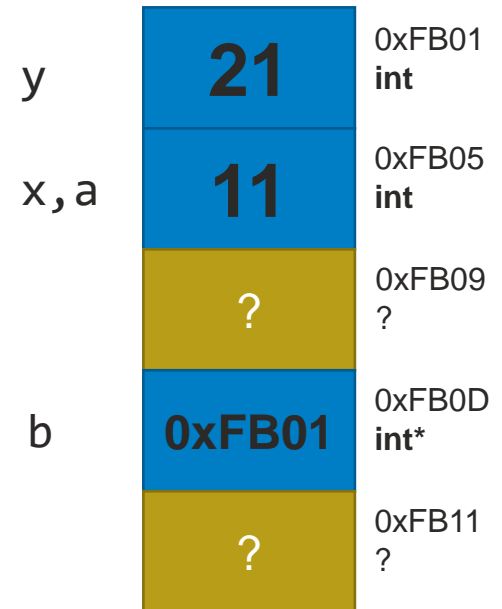
```
int main ()
{
    int x = 10;
    int y = 20;
    doSomething (x, &y);
    return 0;
}
```



Ugyanez referenciával és pointerrel

```
#include <iostream>
using namespace std;
void doSomething(int & a, int * b)
{
    a++;
    (*b)++;
}
```

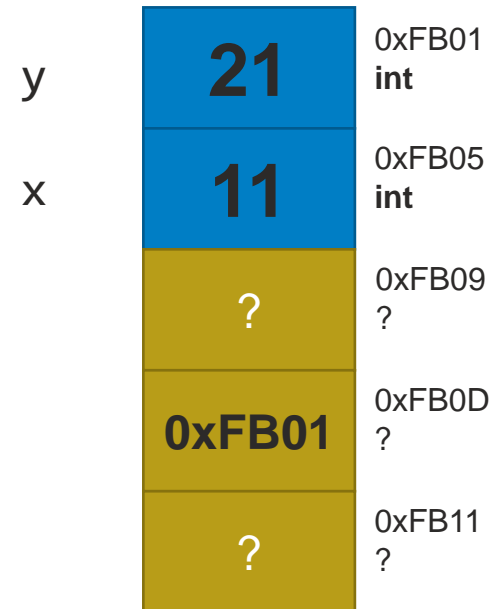

```
int main ()
{
    int x = 10;
    int y = 20;
    doSomething (x, &y);
    return 0;
}
```



Ugyanez referenciával és pointerrel

```
#include <iostream>
using namespace std;
void doSomething(int & a, int * b)
{
    a++;
    (*b)++;
}
```

```
int main ()
{
    int x = 10;
    int y = 20;
    doSomething (x, &y);
    return 0;
}
```



Gondolkodjunk

- A következő programot nézzük át figyelmesen – mi a probléma vele?

```
#include <iostream>
using namespace std;
int & create()
{
    int i = 5;
    return i;
}

int main ()
{
    int & x = create();
    x++;
    return 0;
}
```

Gondolkodjunk

- A következő programot gondoljuk át figyelmesen – nézzük meg

```
#include <iostream>
using namespace std;
int & create()
{
    int i = 5;
    return i;
}
```

```
int main ()
{
    int & x = create();
    x++;
    return 0;
}
```

i

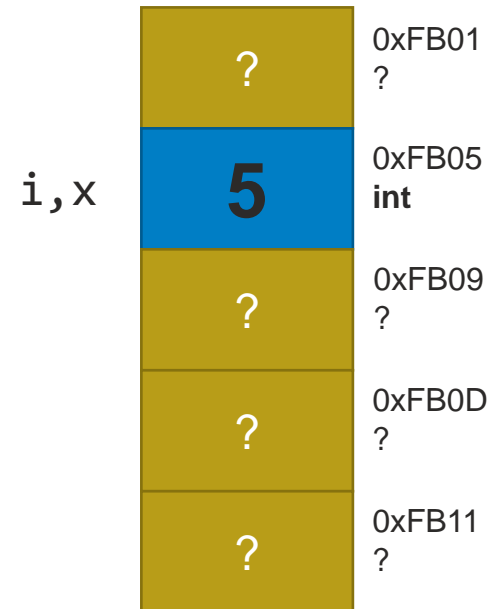
?	0xFB01 ?
5	0xFB05 int
?	0xFB09 ?
?	0xFB0D ?
?	0xFB11 ?

Gondolkodjunk

- A következő programot gondoljuk át figyelmesen – nézzük meg

```
#include <iostream>
using namespace std;
int & create()
{
    int i = 5;
    return i;
}
```

```
int main ()
{
    int & x = create();
    x++;
    return 0;
}
```



Gondolkodjunk

- A következő programot gondoljuk át figyelmesen:

```
#include <iostream>
using namespace std;
int & create()
{
    int i = 5;
    return i;
}
```

```
int main ()
{
    int & x = create();
    x++;
    return 0;
}
```

A referencia egy lokális változó memória-területére hivatkozik, azonban a lokális változóhoz tartozó memóriaterület a függvény befejeztével felszabadításra kerül. A felszabadult memóriához történő hozzáférés helytelen!

?	0xFB01 ?
5	0xFB05 ?
?	0xFB09 ?
?	0xFB0D ?
?	0xFB11 ?

Null pointer: **nullptr**

- Null pointer olyan pointer, aminél az érték (cím) azt jelzi, hogy a pointer nem mutat valós lefoglalt memóriaterületre (objektumra).

- Szokásos, **elavult** jelölés a

~~p = 0;~~

~~p = NULL;~~

- Ezekkel a számos probléma van
 - Általánosságban arra vezethetők vissza, hogy a 0 érték típusa **int** (mivel az egy int literál)
 - Ami nem azonos típusú a pointer változóval.

- A **nullptr** kulcsszó reprezentálja a null pointert, egyedi típussal

- Visszafelé kompatibilitási okokból:

```
char *pc = nullptr;    // OK
int *pi = nullptr;     // OK
bool b = nullptr;      // OK. b = false.
int i = nullptr;       // hiba
```

Pointer paraméterek és visszatérési értékek

- Pointert paraméterként átadva a referenciához hasonló viselkedést kapunk
 - Fontos ellenőrizni, hogy a pointerben érvényes címet kaptunk-e
 - Ha `nullptr` az érték, akkor az természetesen probléma
- Visszatérési érték gyanánt:
 - Leggyakrabban valaminek a legyártására szoktuk használni
 - **Borzasztó fontos, hogy a lefoglalt memóriaterületeket mindig fel kell szabadítani**
 - Nemsokára lesz erről még szó

const

- Kulcsszó annak jelzésére, hogy az adott változó értéke nem változtatható.
 - A kulcsszótól balra található típusra vonatkozik
 - Kivéve ha nincs balra semmi, ez esetben a jobbra levő legszűkebb típusra.
 - Mire jó?
 - Elsősorban programtervezés eszköze
 - const-ból nem csinálhatok nem const-ot, pl függvényhívás során

• Példák

- `const int i = 2;`
- `const int* j = &i;`
- `int * const j = &i;`

- `float tombom[i];`
- `const int a = 2 + 3;`
- `float tombom2[a];`

• Jelentése

- Az i változó értéke nem módosítható
- const int-re mutató pointer
- A j változó (int-re mutató pointer) értéke nem módosítható (nem mutathat másra)

- Csak konstans kifejezéssel megengedett (szabvány szerint)
- Konstans kifejezést már a fordító kiértékel

• Típus része

Konstans paraméterek

- A (referencia) paraméterátadás során garantálható, hogy a hívott függvény az eredeti értéket ne tudja megváltoztatni
 - Ekkor a paraméter konstans
 - **const int & i**
 - A módszer lényege, hogy az esetlegesen nagyméretű paraméter (nagy memóriaterületet elfoglaló) nincs lemásolva (a referencia paraméterátadás miatt), emellett nem megváltoztatható, mintha érték szerinti paraméterátadás lenne
- A konstans paraméterátadás pointerekkel is működik
 - **void f(int * const p)**
 - Ebben az esetben a cím konstans, mivel a pointer értéke a cím
 - **void f(const int * p)**
 - Itt a cím által mutatott memóriaterület nem megváltoztatható
 - Ezzel ekvivalens a következő írásmód is **void f(int const * p)**
 - **void f(int const * const p)**
 - Itt a pointer értéke (cím) és a mutatott memóriaterület egyaránt konstans

Konstans paraméterek?

- A cím konstans – `void f(int * const p)`
 - A függvényben nem tehetem meg a következőt
 - `p = new int;`
 - `p = q;`
- A címzett terület konstans – `void f(const int * p)`
 - A függvényben nem tehetem meg a következőt
 - `*p = 5;`
 - `*p = *q;`

Próbáljuk ki!

- A következő programot hozd létre:
 - Legyen egy függvény, ami paraméterként vár egy valós számokból álló vektort
 - A függvény a vektorban tárolt számok átlagát számolja ki.
 - A paraméterátadás során ne másold le a vektort!
- Ügyelj arra, hogy csak olyan memóriaterületet tudjon megváltoztatni az eljárás, amely feltétlen szükséges a feladat kiírás szerinti megvalósításához.



constexpr

const „kiterjesztése” fordítási idejű számításokra
Jelentése: a kifejezés értéke kiszámítható fordítási időben

- `constexpr int a = 5;`
- `constexpr int get_five() {
 return 5;
}`
- `int tombom3[get_five() + 7];`
- `constexpr double gravity_earth
 = 9.8;`
- `constexpr double gravity_moon
 = gravity_earth/6.0;`
- Változók esetén semmi különbség viselkedésben
- Függvény és konstruktor is lehet
- Csak constexpr-el érvényes
- constexpr inicializálásának minden tagja szintén constexpr



constexpr

```
// Pass by value
constexpr float exp(float x, int n) {
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}

// Pass by reference
constexpr float exp2(const float& x, const int& n) {
    return n == 0 ? 1 :
        n % 2 == 0 ? exp2(x * x, n / 2) :
        exp2(x * x, (n - 1) / 2) * x;
}

// Compile time computation of array length
template<typename T, int N>
constexpr int length(const T(&ary)[N]) {
    return N;
}

// Recursive constexpr function
constexpr int fac(int n) {
    return n == 1 ? 1 : n*fac(n - 1);
}
```

```
// User-defined type
class Foo
{
public:
    constexpr explicit Foo(int i) : _i(i) {}
    constexpr int GetValue() const {
        return _i;
    }
private:
    int _i;
};

int main() {
    //foo is const:
    constexpr Foo foo(5);
    // foo = Foo(6); //Error!
    //Compile time:
    constexpr float x = exp(5, 3);
    constexpr float y { exp(2, 5) };
    constexpr int val = foo.GetValue();
    constexpr int f5 = fac(5);
    const int nums[] { 1, 2, 3, 4 };
    const int
        nums2[length(nums) * 2] { 1, 2, 3, 4, 5, 6, 7, 8 };
    //Run time:
    cout << "The value of foo is " <<
        foo.GetValue() << endl;
}
```


Tömbök

- A tömb:
 - Azonos típusú elemek sorozata
 - A memóriában folytonosan helyezkedik el
 - Indexelhető
 - Az index 0-val kezdődik
 - Ennek megfelelően egy n méretű tömb esetén az érvényes indexek halmaza $[0..n-1]$
 - A mérete fordítási időben ismert érték kell, hogy legyen
 - Deklarációja:
 - `int tomb[5];`
 - Használata:
 - `tomb[4] = 5;`
 - A tömbre hivatkozó változó tömb első elemének a memóriacímét tartalmazza
 - De ez nem azt jelenti, hogy az egy pointer
 - A változót felhasználva pointer-aritmetikával lehetséges a tömb elemeinek manipulálása
 - A `[]` operátor használatával a pointerhez kapcsolódó dereferencing is megtörténik
- Figyeljük meg a következő példakódot a pointer-aritmetikára

Tömbök és pointer kapcsolata

- Nézzük meg a következő kódrészletet

- ```
int main ()
{
 int tomb[5];
 int* p;
 p = tomb; *p = 10;
 p++; *p = 20;
 p = &tomb[2]; *p = 30;
 p = tomb + 3; *p = 40;
 p = tomb; *(p+4) = 50;
 return 0;
}
```

# Tömbök és pointer kapcsolata

- Nézzük meg a következő kódrészletet

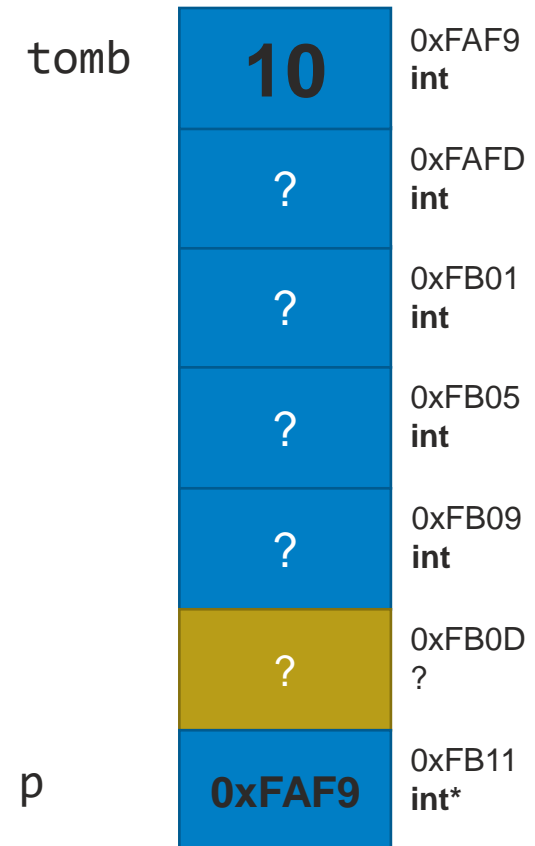
```
• int main ()
{
 int tomb[5];
 int* p;
 p = tomb; *p = 10;
 p++; *p = 20;
 p = &tomb[2]; *p = 30;
 p = tomb + 3; *p = 40;
 p = tomb; *(p+4) = 50;
 return 0;
}
```

|                                                                                                                                                                |   |               |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---------------|
| Egybefüggő tomb<br>memóriaterület<br>kerül lefoglalásra<br>a tömbhöz.<br>A tömbhöz tartozó<br>változó<br>lényegében a<br>tömb első<br>elemének<br>memóriacíme. | ? | 0xFAF9<br>int |
|                                                                                                                                                                | ? | 0xFAFD<br>int |
|                                                                                                                                                                | ? | 0xFB01<br>int |
|                                                                                                                                                                | ? | 0xFB05<br>int |
|                                                                                                                                                                | ? | 0xFB09<br>int |
|                                                                                                                                                                | ? | 0xFB0D<br>?   |
|                                                                                                                                                                | ? | 0xFB11<br>?   |

# Tömbök és pointer kapcsolata

- Nézzük meg a következő kódrészletet

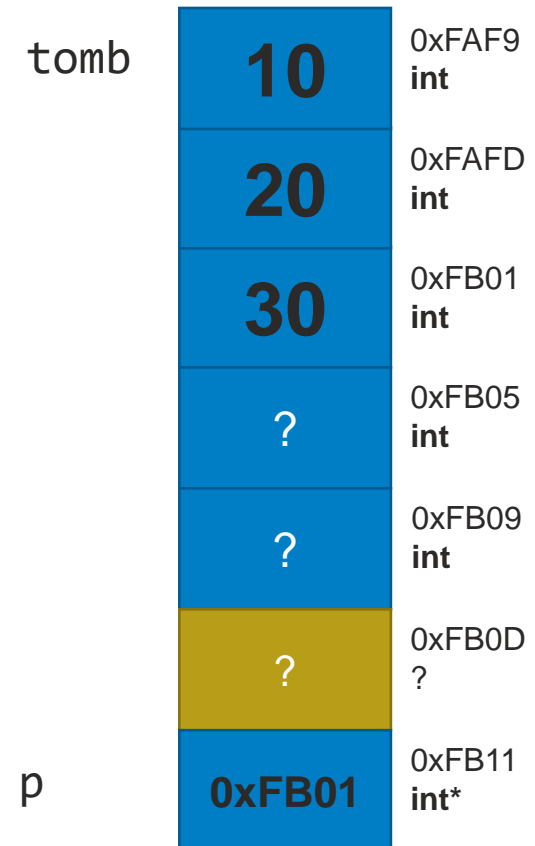
```
• int main ()
{
 int tomb[5];
 int* p;
 p = tomb; *p = 10;
 p++; *p = 20;
 p = &tomb[2]; *p = 30;
 p = tomb + 3; *p = 40;
 p = tomb; *(p+4) = 50;
 return 0;
}
```



# Tömbök és pointer kapcsolata

- Nézzük meg a következő kódrészletet

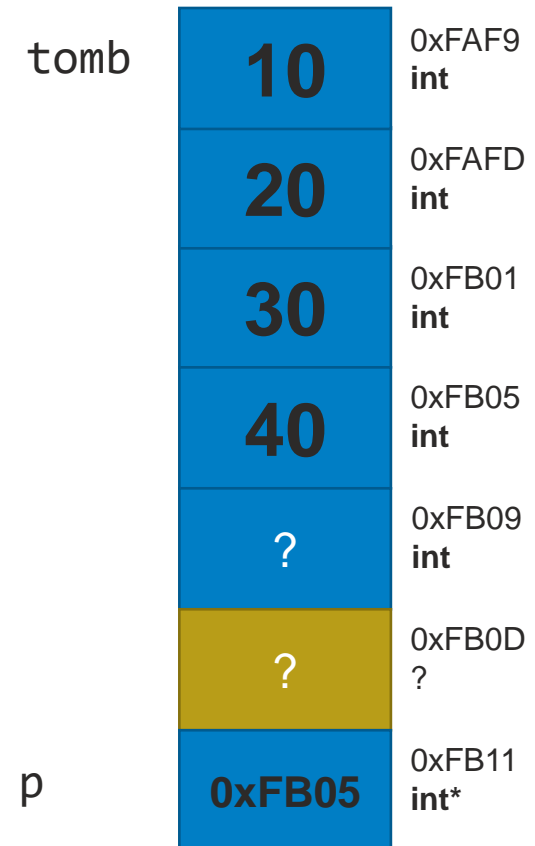
```
• int main ()
{
 int tomb[5];
 int* p;
 p = tomb; *p = 10;
 p++; *p = 20;
 p = &tomb[2]; *p = 30;
 p = tomb + 3; *p = 40;
 p = tomb; *(p+4) = 50;
 return 0;
}
```



# Tömbök és pointer kapcsolata

- Nézzük meg a következő kódrészletet

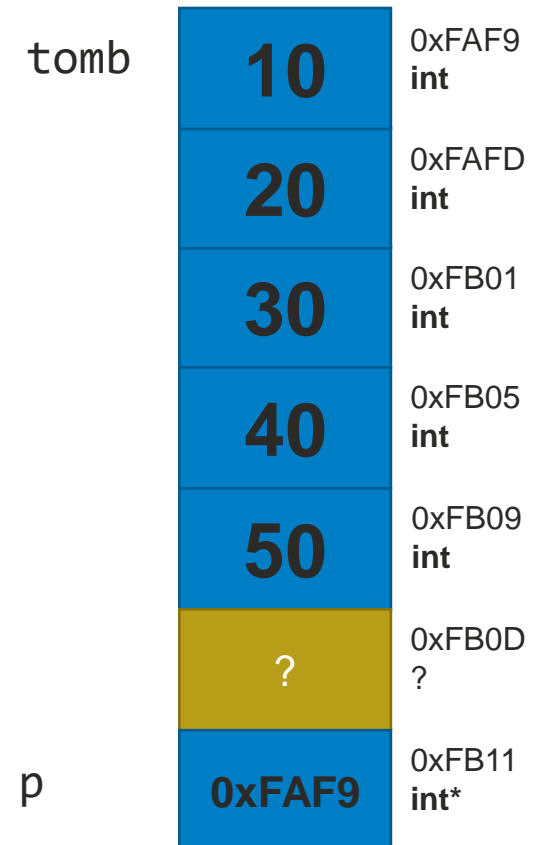
```
• int main ()
{
 int tomb[5];
 int* p;
 p = tomb; *p = 10;
 p++; *p = 20;
 p = &tomb[2]; *p = 30;
 p = tomb + 3; *p = 40;
 p = tomb; *(p+4) = 50;
 return 0;
}
```



# Tömbök és pointer kapcsolata

- Nézzük meg a következő kódrészletet

```
• int main ()
{
 int tomb[5];
 int* p;
 p = tomb; *p = 10;
 p++; *p = 20;
 p = &tomb[2]; *p = 30;
 p = tomb + 3; *p = 40;
 p = tomb; *(p+4) = 50;
 return 0;
}
```



# Tömb-pointer – következmények

- A tömbhöz tartozó változó nagyjából megfeleltethető a tömb első elemének a címével
  - De nem pointer az első elemre!
  - A tömbhöz tartozó változó – jelen esetünkben – `int[]` típusú
  - De láttuk, hogy kezelhetjük úgy is, mintha egy mutató lenne.
- Ennek következményei a következők:
  - Paraméterátadáskor a pointerekkel egyező mechanizmus működik

```
void modifyArray(int t[], int length)
{
 if (length > 0) t[0]++;
}
```

Ennek a függvénynek a hívásakor az átadott tömb címe másolódik és az eredeti tömbön kerül végrehajtásra bármilyen módosítás
  - A tömb méretét paraméterátadáskor át kell adni
    - A méret futási időben ugyan nem mindig határozható meg, illetve csak pointer aritmetikával számolható ki.
  - Semmilyen ellenőrzés nincsen a túlindexelést illetően
    - Nem csak pointer aritmetika használata esetén



# Undefined Behaviour - UB

- Túlindexelés Undefined Behaviour (UB)!

- Pl.: `int* p; p[0] = 10;`

UB egy mondatban:

- „Renders the entire program meaningless if certain rules of the language are violated.” <https://en.cppreference.com/w/cpp/language/ub>
- UB esetén semmilyen feltételezéssel nem élhetünk a program eredményét illetően.

- UB példák:

- Túlindexelés, vagy nem birtokolt memória dereferálása
  - Nem inicializált változó értékének használata
  - Null pointerek dereferálása
  - stb



# Range-based for loop

```
int tomb[10];
for (int i = 0; i < 10; i++)
 tomb[i] = i;
```

```
for (int &v : tomb) {
 v = 2 * v;
 cout << v;
}
// vagy
for (int a = 2;
 int &v : tomb) {
 v = a * v;
 cout << v;
}
```

- Klasszikus tömbön iteráció
- for-each iteráció
  - Módosítani is tudom az elemeket
- Csak olyan tömbökön ami fordítási időben ismert méretű, vagy olyan típus, aminek van begin-end függvénye (pl. `std::vector`)

Van fordító, ami engedi a következőt:

```
cin >> i;
int tomb[i];
```

# Próbáljuk ki!

- Teszteljük egy programmal a tömbök és pointerek kapcsolatát
  - Próbáld ki a tömb értékeinek feltöltését a korábban ismertetett pointer-aritmetikai módszerekkel
    - Találj minél több, különböző elvű módot
  - Írj egy programot, amiben egy eljárás tömböt vesz át paraméterként
  - Próbáld ki, hogy mi történik, ha az eljárásban megváltozik a tömb értéke
  - Nézd meg, hogy mi történik, ha túlindexelsz egy tömböt
    - Lehetséges-e megváltoztatni a tömb értékét?
  - Nézd meg, hogy mi történik, ha egy nem tömbre vonatkozó pointert tömbként kezelsz

# Dinamikus memóriakezelés

- Eddig amikor egy változó bevezetésre került (deklaráltuk), akkor az ahhoz tartozó memóriaterület automatikus lefoglalásra és a blokk végén felszabadításra került
  - Így történt ez a tömbök esetén is
- Azonban gyakran nem szeretnénk erre az automatizmusra bízni a memória és a változók kezelését
  - A programnak egy futásidőben kapott érték szerinti memóriamennyiségre van szükség (felhasználói inputtól függő tömbméret)
  - A függvény végén ne kerüljön felszabadításra a lefoglalt memóriaterület
    - Például mert a memóriában tárolt értékre továbbra is szükség van
- A dinamikus memóriakezelést a pointerok segítségével lehet megtenni
  - Két művelet:
    - `new` – lefoglal egy memóriaterületet és a terület címét adja vissza
    - `delete` – egy lefoglalt memóriaterületet szabadít fel
  - Mivel a memóriakezelést kivesszük az automatizmus kezéből ezért kritikus feladat a memória felszabadítás elvégzése

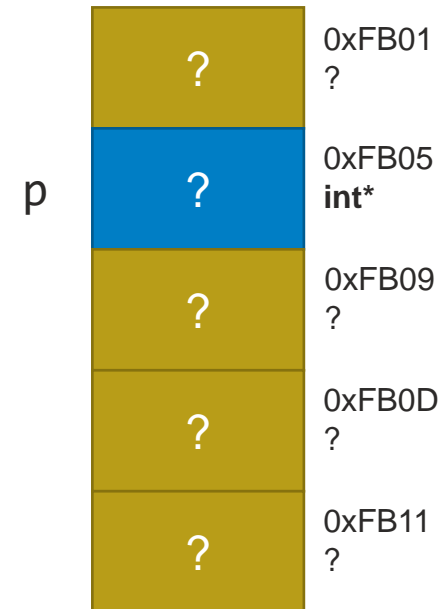
# Dinamikus memóriakezelés – hibaforrások

- Dinamikus változó, memóriaterület lefoglalása és felszabadítása:

```
{
```

```
→ int* p;
 p = new int;
 *p = 10;
 delete p;
 *p = 20;
```

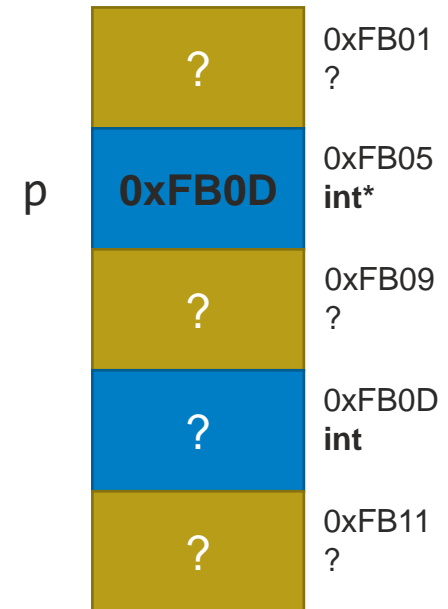
```
}
```



# Dinamikus memóriakezelés – hibaforrások

- Dinamikus változó, memóriaterület lefoglalása és felszabadítása:

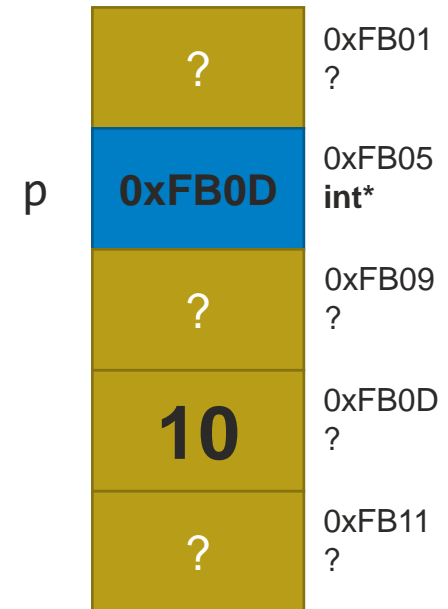
```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 *p = 20;
}
```



# Dinamikus memóriakezelés – hibaforrások

- Dinamikus változó, memóriaterület lefoglalása és felszabadítása:

```
{
 int* p;
 p = new int;
 *p = 10;
→ delete p;
 *p = 20;
}
```



# Dinamikus memóriakezelés – hibaforrások

- Dinamikus változó, memóriaterület lefoglalása és felszabadítása:

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 → *p = 20;
}
```

Vegyük észre, hogy a pointer létezik, az előzőleg lefoglalt és már **felszabadított** memóriaterületre hivatkozik. A memóriaterület *nem biztos*, hogy azonnal kerül újrahasznosításra, azaz le fog futni a teljes kódrészlet.

|        |                  |
|--------|------------------|
| ?      | 0xFB01<br>?      |
| 0xFB0D | 0xFB05<br>int* p |
| ?      | 0xFB09<br>?      |
| 20     | 0xFB0D<br>?      |
| ?      | 0xFB11<br>?      |

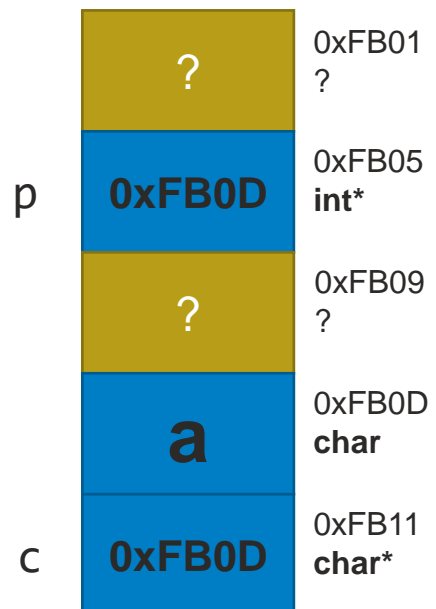


# Dinamikus memóriakezelés – hibaforrások

- Nézzük tovább, a hibás értékadó utasítás kihagyásával!

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 char* c = new char;
 *c='a';
}
```

Most pont ugyanaz a memóriaterület került lefoglalásra, amire a pointerünk mutat. Ez a foglálás az automatizmusra van bízva.



# Dinamikus memóriakezelés – hibaforrások

- Nézzük tovább!

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 char* c = new char;
 *c='a';
 → *p = 30;
}
```

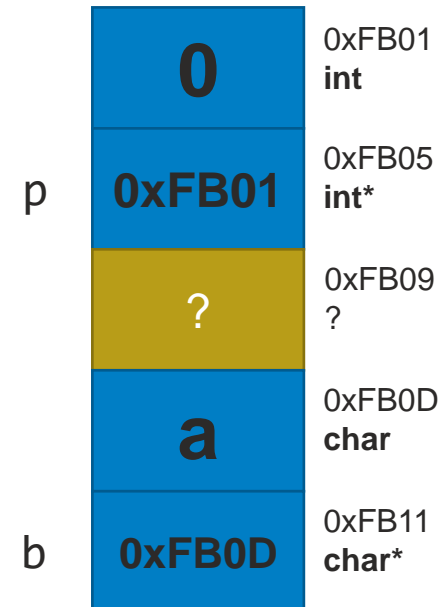
Túl azon, hogy felülírtunk egy értéket, amit korábban tároltunk még típusprobléma is felmerül. Mivel a bitek szintjén nincs az int és a char megkülönböztetve, az értékadás le fog futni.

|   |        |                 |
|---|--------|-----------------|
| p | ?      | 0xFB01<br>?     |
|   | 0xFB0D | 0xFB05<br>int*  |
|   | ?      | 0xFB09<br>?     |
|   | 30     | 0xFB0D<br>char  |
| c | 0xFB0D | 0xFB11<br>char* |

# Dinamikus memóriakezelés – hibaforrások

- Folytassuk az előző, hibás értékadást figyelmen kívül hagyva!

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 char* c = new char;
 *c='a';
 p = new int;
 *p = 0;
}
```



# Dinamikus memóriakezelés – hibaforrások

- Folytassuk!

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 char* c = new char;
 *c='a';
 p = new int;
 *p = 0;
}
```

Befejeződött a blokk.  
Az automatizmusra  
bízott változókhoz  
kapcsolódó  
memóriaterületek  
felszabadításra kerültek.  
Vegyük észre, hogy az  
0xFB01 címen levő  
lefoglalt terület nincs  
felszabadítva és nincs  
érvényes pointer hozzá!

|        |                |
|--------|----------------|
| 0      | 0xFB01<br>int  |
| 0xFB01 | 0xFB05<br>?    |
| ?      | 0xFB09<br>?    |
| a      | 0xFB0D<br>char |
| 0xFB0D | 0xFB11<br>?    |

# Dinamikus memóriakezelés – hibaforrások

- Tanulságok
  - Egy már felszabadított memóriaterülethez nem szabad a továbbiakban hozzáférnünk – hiszen felszabadítottuk
  - A felszabadított memóriaterülethez a pointeren keresztül továbbra is hozzá lehet férni – mint minden egyéb memóriaterülethez, ezt semmi nem ellenőrzi
    - Legfeljebb az operációs rendszer szól közbe, illegális művelet miatt
    - Ennek ellenére meglehetősen rossz ötlet ilyen kódot írni, mivel ugyanazt a memóriaterületet így fel tudja használni a kód egy más része is
  - Ha egy lefoglalt memóriaterületet nem szabadítunk fel és az egyetlen érvényes hivatkozást elveszítjük akkor
    - A memóriában szemetet hagyunk
    - Ezt a memóriaterületet az automatizmus nem találja meg, tehát a program lefutásának végéig nem szabadul fel és hozzáférni sem tudunk
- Ökölszabály
  - Amit lefoglalunk azt fel kell szabadítanunk: praktikusán a **new** és **delete** párban álljon valamilyen módon
  - **delete** után tilos hozzáférni a memóriaterülethez, amit felszabadítottunk
- Megoldás:
  - A **delete** p; hívás után a p értékét nullptr-re állítjuk: p=nullptr;
  - *smart pointers – későbbiekben.*

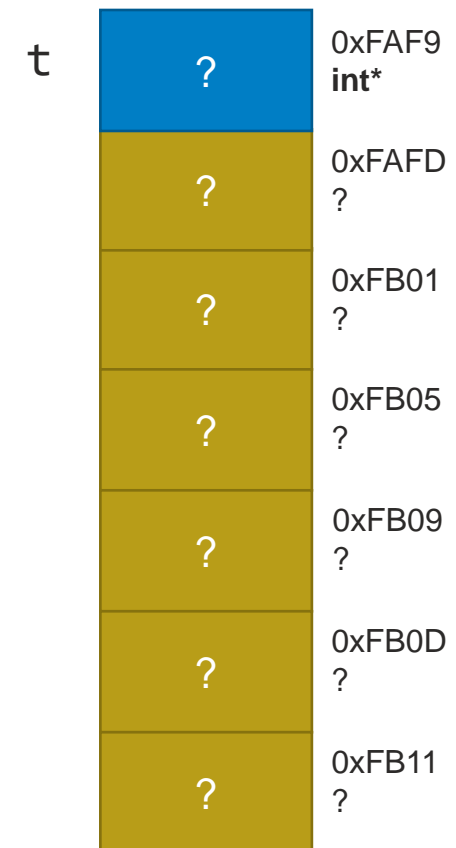
# Dinamikus méretű tömbök

- Láttuk, hogy
  - a tömb változó ~ a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
  - a dinamikus memóriakezelés pointerok segítségével történik
- A kettőt kapcsoljuk össze:

→ 

```
int* t;
int size;
cin >> size;
t = new int[size];

delete[] t;
```

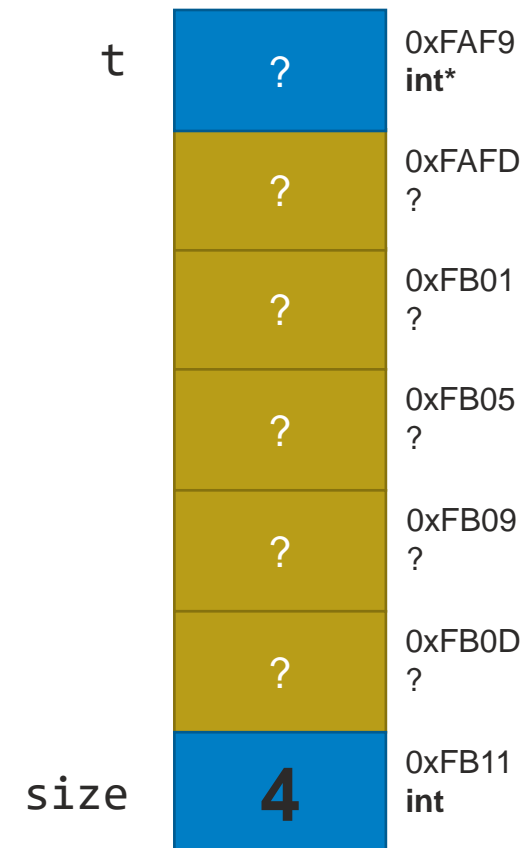


# Dinamikus méretű tömbök

- Láttuk, hogy
  - a tömb változó ~ a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
  - a dinamikus memóriakezelés pointerok segítségével történik
- A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
t = new int[size];

delete[] t;
```

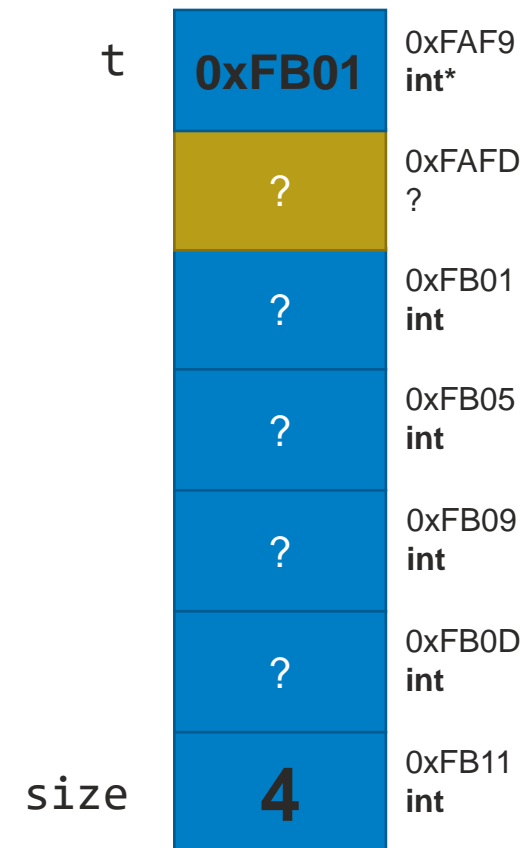


# Dinamikus méretű tömbök

- Láttuk, hogy
  - a tömb változó ~ a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
  - a dinamikus memóriakezelés pointerok segítségével történik
- A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
→ t = new int[size];

delete[] t;
```





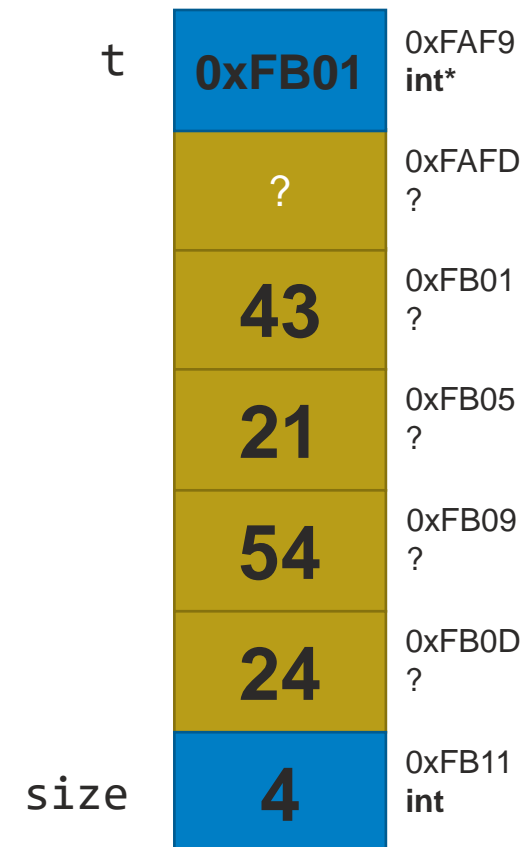
# Dinamikus méretű tömbök

- Láttuk, hogy
  - a tömb változó ~ a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
  - a dinamikus memóriakezelés pointerok segítségével történik
- A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
t = new int[size];
```



```
delete[] t;
```



# Dinamikus méretű tömbök

- Láttuk, hogy
  - a tömb változó ~ a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
  - a dinamikus memóriakezelés pointerok segítségével történik
- A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
t = new int[size];
```



```
delete t;
```

Változtassuk meg ezt a sort!  
Ha (tévedésből) kihagyjuk a `[]`-t a `delete` kulcsszó mögül, akkor nem a tömb által elfoglalt memóriaterület kerül felszabadításra, csak az első elemhez tartozó memóriaterület. Ez szintén memóriaszivárgáshoz vezet!

|      |        |                |
|------|--------|----------------|
| t    | 0xFB01 | 0xFAF9<br>int* |
|      | ?      | 0xFAFD<br>?    |
|      | 43     | 0xFB01<br>?    |
|      | 21     | 0xFB52<br>?    |
|      | 54     | 0xFB09<br>?    |
|      | 24     | 0xFB0D<br>?    |
| size | 4      | 0xFB11<br>int  |



# Stack és Heap

- Memória foglалás

- Az operációs rendszer garantálja, hogy a különböző programok ne tudják egymás memóriáját írni-olvasni
- A saját memóriaterületén viszont a programnak kell megoldani a konzisztenciát, hogy például ugyanahhoz a címhez nem rendel két változónevet
- Sosem írjuk le explicit módon, hogy melyik változó pontosan hova kerüljön
  - Ez a fordító dolga

- Stack – egy verem (az adatszerkezet részletesen lesz még)

- A lokális változók, és az „állapot” helye
- Fix az alja, az aktuális tetejét a „stack pointer” jelöli
- Mindig a tetejéhez fűzünk hozzá és a tetejéből veszünk el
- Ide kerülnek a lokális változók, függvényparaméterek, visszatérési értékek és a "return address" is - ez tárolja, hogy hol kell folytatni a programot, ha a függvény lefutott
- A stack maximális mérete erősen korlátos - egy mély rekurzióval könnyen kifuthatunk belőle.



# Stack

```
int d(int x, int y) {
 if (x < y) return 0;
 else return 1 + d(x-y, y);
}
void f(int x) {
 int r = d(x, 11);
 cout << r;
}
int main() {
 int i = 37;
 f(i);
 return 0;
}
```

Stack állapota

```
main(): +i
f(37): +x, +r
d(37, 11): +x, +y
d(26, 11): +x, +y
d(15, 11): +x, +y
d(4, 11): +x, +y
d(4, 11): -x, -y
d(15, 11): -x, -y
d(26, 11): -x, -y
d(37, 11): -x, -y
f(37): -x, -r
main(): -i
```

|    |   |
|----|---|
| 11 | y |
| 4  | x |
| 11 | y |
| 15 | x |
| 11 | y |
| 26 | x |
| 11 | y |
| 37 | x |
| 3  | r |
| 37 | x |
| 37 | i |



# Heap

- Heap – „kupac” (szintén részletesen lesz később az adatszerkezet)
  - Ide kerülnek a dinamikusan allokált dolgok
  - Tetszőleges sorrendben deallokálhatunk
  - Emiatt sokkal nagyobb az adminisztrációs overhead, lassabb, mint a stackre pakolni
  - Természetesen ez is korlátos, de ritkán szoktunk vele foglalkozni :)
    - 32-bites rendszereken 4 GB
    - Lehet több mint a fizikai RAM mennyisége

# Próbáljuk ki!

- Írjunk egy példaprogramot, amely
  - Bekér egy számot a felhasználótól és ennek megfelelően létrehoz egy tömböt
  - A tömbbe feltölti azokat a számokat, amelyeket a felhasználó megad
  - Ezt követően a tömböt paraméterként át kell adni egy függvénynek, amely
    - Az átvett tömb értékeit lemásolja egy új tömbbe
    - A másolatba minden érték kétszerese kerüljön bele, kivéve, ha a kétszeres érték osztható 10-zel
    - A függvény gondoskodjon arról, hogy ne hagyjon maga után memóriaszemetet
  - A visszakapott új tömb egy másik függvény inputja legyen, amely
    - Meghatározza a tömbben található számok átlagát
    - Az átlagértékkel térjen vissza a függvény
  - A program ne hagyjon maga után memóriaszemetet
- Debugoljuk a kódot

# Memóriaszivárgás

- Példát láttunk memóriaszivárgásra
  - Amikor egy blokkból hiányzik a **new** és **delete** párosból a **delete**, tehát nem kerül felszabadításra a memória
  - Amikor a **delete []** helyett **delete** utasítás kerül kiadásra a tömb által elfoglalt memóriaterületből csak az első elem kerül felszabadításra
- Ennél bonyolultabb esetek is előfordulhatnak
  - Függvényhívások között is lehetséges, hogy elveszítjük az összes tárolt memóriacímet, amivel felszabadíthatjuk a lefoglalt memóriaterületet
- Figyeljük meg a következő kódot ...

# Memóriaszivárgás

- Tekintsük az alábbi kódot

```
int createInt()
{
 int* i = new int(5);
 return *i;
}
```

```
int* createInt2()
{
 return new int(3);
}
```

```
int main()
{
 int* imut = new int;
 *imut = createInt();
 imut = createInt2();
 delete imut;
 return 0;
}
```




# Memóriaszivárgás

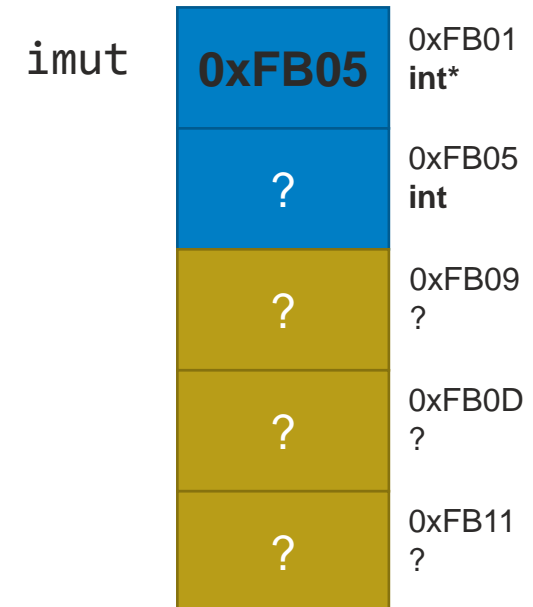
- Tekintsük az alábbi kódot

```
int createInt()
{
 int* i = new int(5);
 return *i;
}
```

```
int* createInt2()
{
 return new int(3);
}
```




```
int main()
{
 int* imut = new int;
 *imut = createInt();
 imut = createInt2();
 delete imut;
 return 0;
}
```




# Memóriaszivárgás

- Tekintsük az alábbi kódot

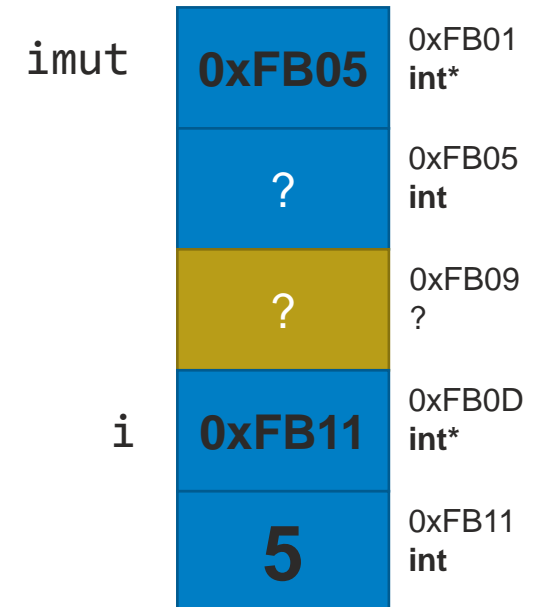


```
int createInt()
{
 int* i = new int(5);
 return *i;
}
```

```
int* createInt2()
{
 return new int(3);
}
```



```
int main()
{
 int* imut = new int;
 *imut = createInt();
 imut = createInt2();
 delete imut;
 return 0;
}
```



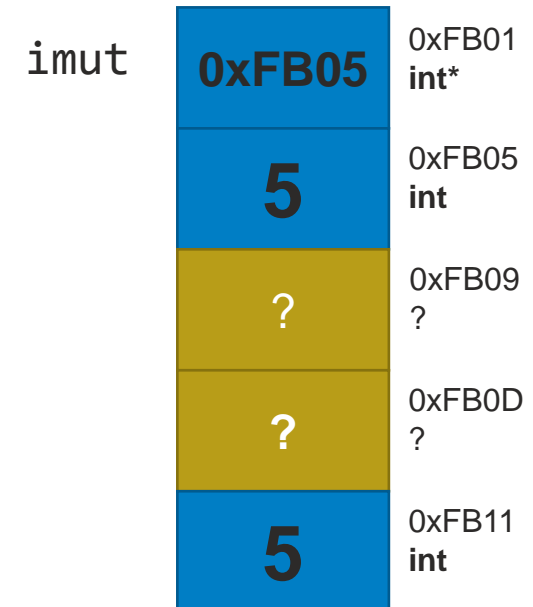
# Memóriaszivárgás

- Tekintsük az alábbi kódot

```
int createInt()
{
 int* i = new int(5);
 return *i;
}
```

```
int* createInt2()
{
 return new int(3);
}
```

```
int main()
{
 int* imut = new int;
 *imut = createInt();
 imut = createInt2();
 delete imut;
 return 0;
}
```



# Memóriaszivárgás

- Tekintsük az alábbi kódot

```
int createInt()
{
 int* i = new int(5);
 return *i;
}
```

```
int* createInt2()
{
 return new int(3);
}
```

```
int main()
{
 int* imut = new int;
 *imut = createInt();
 imut = createInt2();
 delete imut;
 return 0;
}
```


Befejeződött a függvény.  
Lefut a függvény és a `imut` létrehozott dinamikus változóhoz tartozó pointer automatikusan felszabadításra kerül. Vegyük észre, hogy a visszatérési típus nem `int*`, hanem `int`. Tehát az értéket adjuk vissza és nem a memóriacímet.

|        |                |
|--------|----------------|
| 0xFB05 | 0xFB01<br>int* |
| 5      | 0xFB05<br>int  |
| ?      | 0xFB09<br>?    |
| ?      | 0xFB0D<br>?    |
| 5      | 0xFB11<br>int  |


# Memóriaszivárgás

- Tekintsük az alábbi kódot

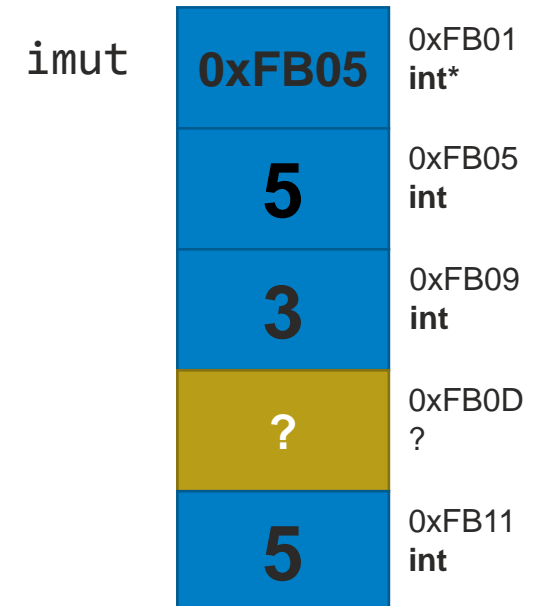
```
int createInt()
{
 int* i = new int(5);
 return *i;
}
```



```
int* createInt2()
{
 return new int(3);
}
```



```
int main()
{
 int* imut = new int;
 *imut = createInt();
 imut = createInt2();
 delete imut;
 return 0;
}
```



# Memóriaszivárgás

- Tekintsük az alábbi kódot

```
int createInt()
{
 int* i = new int(5);
 return *i;
}
```

```
int* createInt2()
{
 return new int(3);
}
```

```
int main()
{
 int* imut = new int;
 *imut = createInt();
 imut = createInt2();
 delete imut;
 return 0;
}
```



Befejeződött a második `imut` függvény.

Lefut a függvény és a létrehozott dinamikus változó címe visszaadásra kerül. Vegyük észre, hogy ez felülírja a korábban tárolt címet, azaz a `main()` elején lefoglalt memóriára vonatkozó cím elveszik.

|        |                |
|--------|----------------|
| 0xFB09 | 0xFB01<br>int* |
| 5      | 0xFB05<br>int  |
| 3      | 0xFB09<br>int  |
| ?      | 0xFB0D<br>?    |
| 5      | 0xFB11<br>int  |

# Memóriaszivárgás

- Tekintsük az alábbi kódot

```
int createInt()
{
 int* i = new int(5);
 return *i;
}
```

```
int* createInt2()
{
 return new int(3);
}
```

```
int main()
{
 int* imut = new int;
 *imut = createInt();
 imut = createInt2();
 delete imut;
 return 0;
}
```



A teljes lefutás után két helyen memóriacella sem került felszabadításra.

Mivel ez egy viszonylag egyszerű kód, három függvénnnyel, ezért különösen fontos ügyelni a megfelelő visszatérési értékekre és a pointerek újrahasznosítására!

|        |               |
|--------|---------------|
| 0xFB09 | 0xFB01<br>?   |
| 5      | 0xFB05<br>int |
| ?      | 0xFB09<br>int |
| ?      | 0xFB0D<br>?   |
| 5      | 0xFB11<br>int |

# Próbáljuk ki!

- Próbáljuk ki, hogy valójában mi történik:
  - Ha egy inicializálatlan pointer mögötti memóriaterületet piszkálunk meg
  - Ha egy törölt pointer mögötti memóriaterületet piszkálgatunk
  - Ha egy (dinamikus) tömböt túlindexelünk
  - Ha egy nagy (dinamikus) tömböt indexelünk túl
  - Ha konstans értéket próbálunk megváltoztatni
  - Konstans érték címe létezik-e?
  - Ha az `int &i=5;` utasítást adjuk ki.



# Gyakorló feladat G01F01

- Írj egy olyan programot, amely egy mátrixot tárol
  - A mátrix méretét és az elemeket előre nem ismerjük
  - Az adatokat egy „in.txt” fájlból kell beolvasni. A fájlban az első sorban a sorok száma ( $n$ ) van, a második sorban az oszlopok száma ( $m$ ), a fájl többi sorában (összesen még  $n \times m$  sor van) pedig a beolvasandó értékek, sorfolytonosan.
  - A mátrixot egy tömbben tároljuk el oszlopfolytonosan
    - Figyelem! Megváltozik a sorrend, az indexekre gondolni kell
  - Írd meg az összeadás és szorzás műveleteket!
  - Írj egy transzponálást végrehajtó függvényt!
  - Írj egy kiíró műveletet, ami az elemeket áttekinthető formában megjeleníti!
  - A főprogramon belül készíts menüt!
    - Egy ciklusban kérdezd meg a felhasználótól, hogy akar-e még műveletet végezni!
    - Ha igen, kérdezd meg, hogy hogy milyen műveletet!
    - Kérj be két mátrixot elemenként, majd végezd el a kért műveletet!
    - Az eredményt jelenítsd meg a képernyőn a kiíró műveletével!

# Gyakorló feladat G01F02

- Hozz létre egy  $n$  hosszú tömböt – dinamikus memóiafoglalással
  - Töltsd bele az első  $n$  prímszámot
- Másold le a tömböt (tényleges másolat)
  - Emeld négyzetre a másolat elemeit, kivéve ha a szám háromjegyű
  - Ha háromjegyű, akkor legyen a tömbben az új érték nulla
- A másolatot ki kell írni a képernyőre.
- Ezt követően írd ki a Pascal háromszög  $n$ -edik sorát
- A program a futás során kezelje a memóriát helyesen!

# Gyakorló feladat G01F03

- Készíts egy olyan programot, amely
  - Négyzetek és téglalapok adatainak tárolására alkalmas
    - Ehhez két tömböt kell használni, ahol a négyszögek két oldalának hosszát tároljuk el
  - A két tömb méretét a program indulásakor a felhasználó adja meg, amelyet követően azokat dinamikusán kell létrehozni
  - A tömbök értékekkel történő feltöltése után meg kell keresni:
    - A legnagyobb területű téglalapot
    - A legkisebb területű négyzetet
    - A megtalált síkidomok oldalainak hosszát és méretét ki kell írni
    - A két keresésre két függvényt kell írni, amelyek paraméterben kapják meg a tömböket
  - Ügyelni kell arra, hogy ne legyen memóriaszivárgás és ne legyen felesleges memóriahasználat
  - Legyen még egy függvény, amely paraméterként átveszi a tömböt és meghatározza a területértékek átlagát
    - Ezt ki is kell írni

# Gyakorló feladat G01F04

- Készíts egy programot, ami
  - Képes tárolni egy összefüggő irányítatlan gráfot
  - A gráf leírását tömbök tömbjével kell megoldani, dinamikus memóriakezeléssel
    - Az  $i$ -edik tömb  $j$ -edik értéke 1 vagy 0.
    - Ha 1, akkor az  $i$  és  $j$  pont között van él, ha 0, akkor nincsen
    - Ha az  $i$ -edik tömb  $j$ -edik értéke 1, akkor a  $j$ -edik tömb  $i$ -edik értéke is 1.
  - A program indításkor kérdezze meg, hogy mennyi csúcsot szeretne tárolni a felhasználó
    - Ennek megfelelően hozza létre a tömböket és töltsse fel 0 értékekkel
  - Ezt követően legyen lehetőség élek felvételére
    - Természetesen ügyeljen a program, hogy érvényesek legyenek a bevitt értékek
  - A program legyen képes eldönteni, hogy két tetszőleges csomópont között húzódik-e pontosan kettő hosszú út!

# Gyakorló feladat G01F05

- Dinamikus memóriakezelés segítségével hozz létre egy kétdimenziós tömböt.
  - Majd random módon töltsd fel (0-255) közötti értékekkel (egy szürkeárnyaltos kép).
- Ezután a „képet” mossuk el egy tetszőleges gaussian kernellel, melyet lehessen a felhasználótól bekérni a konzol segítségével.
- Tehát az adott pixel új értéke a példa kernel szerint:
  - A pixel értékének  $1/4$ -e + a szomszédos pixelek  $1/8$ -a + az átlós pixelek  $1/16$ -a.
  - A „kép” sarkainál és a széleknél vegyük észre, hogy kevesebb, mint 8 szomszéd van. Kezeljük le ezeket az eseteket.
- Írjuk ki a konzolra az új „kép” értékeit. Vigyázzunk, hogy ne legyen memória szivárgás.

|        |       |        |
|--------|-------|--------|
| $1/16$ | $1/8$ | $1/16$ |
| $1/8$  | $1/4$ | $1/8$  |
| $1/16$ | $1/8$ | $1/16$ |

# Házi feladat beadási konvenciók

- Projektek elnevezése:
  - <shibboleth>\_<hf\_jele>
- Kapcsolók legyenek beállítva
  - -Werror -Wall -Wextra -pedantic
- A repoba **ne** kerüljön fel a .idea mappa illetve a fordításhoz használt cmake-build-debug és hasonló mappák.
- Beadás SVN repositoryba:
  - [https://repo.itk.ppke.hu/adatszerk/<shibboleth\\_név>](https://repo.itk.ppke.hu/adatszerk/<shibboleth_név>)
- Mindegyik házit a megfelelő mappába kell tenni
  - <shibboleth\_név>/<hf\_jele>, ahol a <hf\_jele>: khf0[1-6], hf0[1-3]
- Például
  - 1. kisházi esetén az adatszerk/balga9/khf01 mappába töltöm fel a balga9\_khf01 projekt mappát és tartalmát.