# 5. gyakorlat

## Tudnunk kell, pointer konstans területre mutat-e

### Pointer konstansra

const char**\*** pc **=** "abc"**;**

Jobbról balra érdemes kiolvasni: A $pc$ változó egy pointer ($\*$) karakterre ($char$), ami konstans ($const$).

Ez ekvivalens ezzel:

char**\*** pc **=** **{**'a'**,** 'b'**,** 'c'**,** '\0'**};**

Pointernek adom értékül a tömböt, akkor átalakul, az első elemre mutat majd.

pc**[**0**]=**'x'**;**//Hiba

char**\*** p**;**

pc**=**p**;**//Ez megengedett. Nem a pointer a konstans, hanem a mutatott char\*

### Konstans pointer

char**\*** const p**=**"abc"**;**

p**[**0**]=**'y'**;**//Ez megengedett

Ugyanis a deklaráció azt mondja, hogy $p$ egy konstans pointer karakterekre. És ennek a szemantikája pedig azt mondja, hogy magát a pointert nem állíthatuk el máshova.

p**=**pc**;**//Hiba, a konstans pointer nem állítható át.

### Konstans pointer konstansra

const char**\*** const cpc **=** "hi"**;**

cpc**=**p**;** //Hiba

cpc**[**0**]=**'n'**;**//Hiba

Itt se a pointert se a mutatott elemet nem lehet megváltoztatni. Legalábbis ezen a pointeren keresztül…

### Keverve lehet

char c**=**'a'**;**//Lehet módosítani, de a lenti pointeren keresztül nem.

const char**\*** pc**=&**c**;**//Nem konstans változó címét is odaadhatom konstansra mutató pointernek.

### Visszafelé nem

const char cc**=**'b'**;**//Konstansra nem mutathat nem konstans pointer

char**\*** p**=&**c**;**//Hiba

### Állásinterjún kínzás

char const**\*** s**;** //≡char const (\*s)≡

const char**\*** s2**;**

Mert**:**

**\***const//van

const**\***//Ilyen deklarátor operátor nincs.

### Csillagok háborúja

Sok a vita arról, hogy hova tesszük a csillagot…

int **\***z**;**

int**\*** z**;**

Mindkettő helyes. De a változó neve $z$, nem $\*z$.

int a**,** **\***b**,** c**();**//Három helyes deklaráció:

int a**;**

int**\*** b**;**

int c**();**

//De ilyet nem írunk le!

Mindenki úgy írja, ahogy akarja, csak össze ne zavarjuk.

### Trükk régi c stílusú karakterláncokkal

Azért, hogy többmilliónyi c kódsor érvényes maradjon…

Konstans karakter tömbre "abc" mégis mutathat nem konstans pointer.

## String hossza

**sizeof(**c**)/sizeof(**char**)** csak tömbre működik: char c**[]=**"Hello"**;** Ilyen stílusú stringekre viszont nem: char**\*** c**=**"Hello"**;**.

És azért 6 nem 5, mert a karakter végén van egy '\0'.

Függvénynek átadott tömb esetén viszont csak az első karakter marad meg. A pointer méretét adja vissza. A tömb mérete a függvény számára elvész.

Egy karakter hosszát úgy határozzuk meg, hogy elindulunk az elejéről és megyünk, megyünk, amíg nem érünk egy '\0'-hoz:

int strlen**(**char**\*** p**){**

 int n**=**0**;**

 **while(\***p **!=** '\0'**){**

 **++**p**;**

 **++**n**;**

 **}**

 **return** n**;**

**}**

char**\*** c**=**"Hello"**;**

std**::**cout**<<**"Length of c is: "**<<**strlen**(**c**)<<**std**::**endl**;**

Mi az, hogy $++p$? Pointer arithmetika része:

### Pointer arithmetika

p**+**4

A pointert mozgassuk át Ennyiszer a mutatott típus méretével. (4 **\*** **sizeof(**int**)**)

Kivonás ugyan így.

Itt semmilyen ellenőrzés nincs. Nincs "out of bounds" exception. A memóriát szabadon kezelhetjük.

Pointerekkel olyan helyekre mutathatunk, írhatunk és ovalshatunk, ahove nam lenne szabad.

Két pointert kis is lehet vonni egymásból. A két memóriacím közti különbség osztva a mutatott típus méretével.

Összeadni két pointert nincs sok értelme, de lehet.

Most optimalizáljuk a kódot:

### Minek ilyen hosszú ciklusfeltétel?

Nem elég a **while(\***p**){**? Pont ugyan azt csinálja.

### Operátor precedencia trükk

**while(\***p**++){**

 **++**n**;**

**}**

A $++$ hatására megnő a $p$, de még a megnövelés előtti értéket adja vissza.

### Túloptimalizált kód

int strlen**(**char**\*** p**){**

 char**\*** current**=**p**;**

 **while(\***current**++);**

 **return** current**-**p**-**1**;**

**}**

Readability is terrible!

## Dinamikus memóriahasználat, foglaljunk tömböt!

Szokásos tömb mérete fordításidőben ismert kell, legyen. Ez szabvány. De van rá megoldás.

int size**;**

cin**>>**size**;**

int v**[**size**];**//Hiba

Heap-en (nem kupac, hanem memóriaterület) magunknak foglalhatunk le memóriaterületet.

Erre van a void**\*** malloc**(**size /\*in bytes\*/**);** függvény c-ben.

C++-ban pedig a **new** és a **new[]** operátor.

### Példa

int**\*** p **=** **new** int**(**4**);**

cout**<<\***p**;**

Heap-en létrehoztunk egy integert, beleírtuk a 4-et. Ennek a $new$ operátor vissza fogja adni memóriacímét.

Enneks egítségével tudunk dinamikus tömböket és mátrixokat foglalni.

### Dinamikus tömb

int**\*** array **=** **new** int**[**size**];**

Ez már legális kód.

Le fogja foglalni a megfelelő méretű tömböt és visszaadja az első elem címét.

Ezt a szokásos módon be lehet járni:

**for(**int i**=**0**;**i**<**size**;++**i**){**

 array**[**i**]=**i**;**

**}**

Hasonlóképpen mátrix, több dimenziós tömb.

### Mátrix

Pointer olyan pointerre, ami integerre mutat

int**\*\*** matrix**;**

int m**=**/\*...\*/**;**

int n**=**/\*...\*/**;**

matrix **=** **new** int**\*[**n**];**

**for(**int i**=**0**;**i**<**n**;++**i**){**

 matrix**[**i**]** **=** **new** int**[**m**];**

**}**

Van egy tömbünk, aminek minden eleme tömb (-re mutató pointer).

### Mátrix indexelése

Mivel a $matrix$ pointer, ezért indexelhető. Mivel a megkapott elem ($matrix\left[3\right]$) megint csak pointer, az is indexelhető. Így lesz jó:

matrix**[**3**][**4**];**

### Memory leak

Ez így nagyon szép, de ez a memóriaterület a mi gondozásunk alatt áll.

Mindig fel kell szabadítani, hogy ne fogyjon el.

Memóriazabáló:

**for(;;){**

 int**\*** p**=new** int**(**4**);**

**}**

Nekünk kell takarítani: $delete p$

Hogy mikor, az ránk van bízva.

Ha tömb, akkor tömbös $delete[]$-t kell hívni.

## Pluszminusz, szünet

Pluszminusz lesz (valami gyakorlati, 2 soros kód vagy ilyesmi)

Utána 2x szünet

Gyakorlat vége.