# C++
Gyakorlat jegyzet
4. óra

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlata alapján. (2018. április 30.)

# 1. Literálok

## 1.1. Karakterláncok

Mi lesz a `"Hello"` karakterlánc literál típusa?

Egy konstans karakterekből álló 6 méretű tömb (`const char[6]`). Azért 6 elemű, mert a karakterlánc literál végén el van tárolva a végét jelző `\0` karaktert.

| H | E | L | L | O | \0 |
|---|---|---|---|---|---|

```
int main()
{
    char* hello= "Hello";
    hello[1] = 'o';
}
```

A fenti kódban megsértettük a konstans korrektséget, hisz egy nem konstansra mutató pointerrel mutatuk egy konstans karakterlánc literál első elemére. Ennek ellenére, a fenti kód lefordul. Ennek az az oka, hogy az eredeti C-ben nem volt const kulcsszó, a kompatibilitás végett ezért C++ban lehet konstans karakterlánc literál elemire nem konstansra mutató pointerrel mutatni.

**1.1.1. Megjegyzés.** Ezt a fajta kompatibilitás miatt meghagyott viselkedést kerülni kell. Lefordul, de kapunk rá warningot.

Ha módosítani próbáljuk a karakterlánc literál értékét, az nem definiált viselkedéshez vezet.

Futtatáskor linuxon futási idejű hibát kapunk, méghozzá szegmentálási hibát. Ennek az az oka, hogy a konstansok értékei readonly memóriában vannak tárolva, aminek a módosítását nem engedi az operációs rendszer.

Ez jól rámutat arra, hogy miért is nem jó az, ha a fenti konverziót megengedjük.

## 1.2. Szám literálok

Függően attól, hogy egy szám literált hogyan írunk C++ban, mást jelenthet:

| | |
|---|---|
| `5` | `int` |
| `5.` | `double` |
| `5.f` | `float` |
| `5e-4` | `double`, értéke 0.0005 |
| `5e-4f` | `float` |
| `0xFF` | 16-os számrendszerben ábrázolt `int` |
| `012` | 8-as számrendszerben ábrázolt `int` |
| `5l` | `long int` |
| `5u` | `unsigned int` |
| `5ul` | `unsigned long int` |

Létezik C++ban `signed` kulcsszó, mely a char miatt lett bevezetve. A char is egész számokat tartalmaz, de az implementáció függő, hogy a `char` singed vagy unsigned értéket tartalmaz-e.

Természetesen leggyakrabban egész számoknál szoktuk használni. Alapértelmezetten minden `int` egy `signed int`, amennyiben egy előjel nélküli egész számra van szükségünk, `unsigned int`-et vagy röviden

`unsigned`-t kell írnunk. Megállapítható, hogy az `unsigned` típusok ugyanannyi számjegyet tudnak tárolni, ám értékben kétszer akkorát.

Míg a túlcsordulás `signed` típusoknál nem definiált, addig `unsigned` típusoknál az, a maximális érték (mely implementációfüggő) utáni inkrementális 0-ra állítja a változót. Értelemszerűen, ez a determinisztikus viselkedés futási idejű költséggel jár.

Mint tudjuk, a nem egész számoknál a törtrészt magyarban vesszővel, C++ban ponttal választjuk el. Azonban mégis, az alábbi kód helyesen lefordul:

```
int pi = 3,14;
```

Ennek oka az, hogy a vessző operátor egy szekvenciapont is, így a fordító az egyenlőség bal oldalát amikor kiértékeli, először rendre kiértékeli a 3 számliterált, eldobja, utána kiértékeli a 14-et, és értékül adja `pi`-nek. Ez azt jelenti, hogy az ilyen jellegű hibát nem olyan könnyű megfogni, mint elsőre gondolnánk.

A lebegőpontos számok másik veszélye, az összehasonlítás. Jusson eszünkbe, ami Numerikus Módszereken hangzik el: minden lebegőpontos szám tartalmazhat egy kis pontatlanságot.

```
for (double d = 0; d != 1; d += 0.1)
{
    std::cout << d << ' ';
}
```

Itt bár azt várnánk, hogy a kimenet `0 0.1 0.2 ... 1.0` legyen, legnagyobb valószínűséggel végtelen ciklusba futunk. Ez amiatt van, hogy a 0.1 (várhatóan) tartalmaz egy kis pontatlanságot, és így hiába írja ki a programunk hogy `d` értéke 1, az várhatóan csak nagyon közel lesz hozzá.

**1.2.1. Megjegyzés.** Viszonylag kevés esetben éri meg `float`-ot használni `double` helyett. Modern CPU-k ugyanolyan hatékonyan dolgoznak mind a kettővel, így érdemesebb a pontosabbat választani. (Ha magát a GPU-t programozzuk, az lehet egy kivétel.)

**1.2.2. Megjegyzés.** Érdemes mindig `int`et használnunk, ha nincs jó okunk arra, hogy mást használjunk. Az `int`-el általában a leghatékonyabb a processzor.

# 2. Struktúrák mérete

## 2.1. Primitív típusok mérete

A `sizeof(char)` mindig 1-et ad vissza. A karakter mérete mindig az egység. Minden más típusra a `sizeof` függvény azt adja vissza, hogy paraméterül megadott objektum vagy típus mérete hányszorosa a `char`nak.

Attól, hogy `sizeof(char) == 1`, a `char` mérete byteokban még implementáció függő.

A lebegőpontos számok mindig rendelkeznek előjellel.

A `char` méretén túl minden másnak a mérete implementációfüggő, bár a szabvány kimond pár relációt:

$$\texttt{sizeof(X)} == \texttt{sizeof(signed X)} == \texttt{sizeof(unsigned X)}$$
$$\texttt{sizeof(float)} \leq \texttt{sizeof(double)} \leq \texttt{sizeof(long double)}$$
$$\texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$
$$\texttt{sizeof(char)} \leq \texttt{sizeof(bool)}$$

## 2.2. Nem primitív típusok mérete

Egy általunk megalkotott típus mérete több szabálytól is függhet.

```
#include <iostream>

struct Hallgato
{
    double atlag;
    int kor;
    int magassag;
```

```
}

int main()
{
    std::cout << sizeof(double) << std::endl;
    std::cout << sizeof(int) << std::endl;
    std::cout << sizeof(Hallgato) << std::endl;
}
```

A gyakorlaton használt gépen a `double` mérete 8, az `int` mérete 4, `Hallgato`-é 16. Ezen azt látjuk, hogy a `Hallgato` tiszta adat.

```
struct Hallgato
{
    int kor;
    double atlag;
    int magassag;
}
```

Miután átrendeztük a mezők sorrendjét, és újra kiírjuk a struktúra méretét, akkor a válasz 24. Ennek az oka az, hogy míg az első esetben így volt eltárolva a memóriában: (ne feledjük, ez még mindig implementációfüggő!)


Azaz, `atlag`, illetve `kor` és `magassag` pont efértek 1-1 gépi szóban. Viszont, ha megcseréljük a sorrendet, ez már nem lesz igaz:


Itt az `atlag` két fele két különböző gépi szóba kerülne. Ez a ma használt processzorok számára nem hatékony, hiszen az átlag értékének kiolvasásához vagy módosításához két gépi szót is olvasni vagy módosítani kéne (a legtöbb processzor csak szóhatárról tud hatékonyan olvasni).


A fenti elrendezés hatékonyabb, bár 3 gépi szót használ. Ebben az esetben a fordító *padding*et illeszt be a mező után. Ennek hatására hatékonyan olvasható és módosítható minden mező. Cserébe több memóriát foglal a struktúra.

A szabvány kimondja, hogy egy `struct` mérete az adottagok méreteinek összegénél nagyobb vagy egyenlő.

Az, hogy egy gépi szó mekkora, implementációfüggő.

Egy `struct` egyes adattagjaira a pont operátor segítségével hivatkozhatunk:

```
int main()
{
    //...
    Hallgato a;
    std::cout << a.kor << std::endl;
    Hallgato b = a;
    b.magassag = 3;
}
```