

C++ Gyakorlat jegyzet 4. óra.

A jegyzetet UMANN Kristóf készítette HORVÁTH Gábor gyakorlatán. (2017. január 28.)

0.1. Tömbök átadása függvényparaméterként

Próbáljunk meg egy tömböt érték szerint átadni egy függvénynek!

```
#include <iostream>

void f(int t[])
{
    std::cout << sizeof(t) << std::endl;
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << sizeof(t) << std::endl;
    f(t);
}
```

Kimenet: 20 8 (implementáció függő)

Bár azt hittük, hogy `t` tömb méretét írtuk ki két alkalommal, valójában amikor azt érték szerint próbálunk meg átadni egy tömböt, az átkonvertálódik a tömb elejére mutató pointerre.

```
void f(int t[8])
{
    std::cout << sizeof(t) << std::endl;
}
```

Hiába adunk meg egy méretet a tömbnek a függvény fejlécében, még mindig egy pointer mérete lesz a második kiírt szám. Az a tanulság, hogy ha érték szerint akarunk átadni egy tömböt, az át fog konvertálódni pointerre. A legszebb az lenne, ha a fenti szintaxis nem fordulna le. Ennek azonban történelmi oka van, a C-vel való visszafelé kompatibilitás miatt fordul le.

0.1.1. Megjegyzés. Tömböt értékül adni a szabvány szerint nem is lehet: `int *t2[5] = t` nem helyes.

Korábban megismerkedtünk egy módszerrel, mely segítségével egy tömb méretét (elemszámát) paraméterátadás után is megőriztük:

```
#include <iostream>

void f(int *t, int size) // új paraméter!
{
    std::cout << sizeof(t) << std::endl;
}

int main()
{
    int t[] = {1,2,3,4,5};
    std::cout << sizeof(t) << std::endl;
    f(t, sizeof(t)/sizeof(t[0]));
}
```

0.1.2. Megjegyzés. Amennyiben C++11ben programozunk, érdemes az `std::array`-t használnunk, ami olyan, mint egy tömb, de nem tud pointerre konvertálódni és mindig tudja a méretét.

Ha szeretnénk egy tömböt egy darab paraméterként átadni, megpróbálhatunk egy tömbre mutató pointert létrehozni. Azonban figyelni kell a szintaktikára, ha `int *t[5]`-t írunk, egy öt elemű intre mutató pointereket tároló tömböt kapunk.

Ha tömbre mutató mutatót szeretnénk, így csinálhatjuk:

```

void g(int (*t)[5])
{
    std::cout << sizeof(t) << std::endl;
}

```

Azonban ez még mindig egy pointer méretét fogja kiírni, mert a `t` az egy sima mutató! Ahhoz, hogy megkapjuk, mire mutat, dereferálnunk kell, így a `sizeof` paraméterének `*t`-t kell megadni, ha a tömb méretére vagyunk kíváncsiak.

0.1.3. Megjegyzés. Ha referenciával vennénk át `t`-t, az is hasonlóan nézne ki: `int (&t)[5]`.

Ha eltérő méretű tömböt próbálunk meg átadni, akkor nem fordul le a kód, mert nem egy 5 elemű tömbre mutató mutató 6 elemű tömbre mutató mutatóvá konvertálódni.

```

int main()
{
    int a[6];
    g(&a); //forditasi hiba!
    int b[5];
    g(&b); //ok
}

```

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 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, még hozzá 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 írnak 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

1.2.1. Megjegyzés. Alapértelmezetten minden int egy signed int.

1.2.2. 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.)

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

1.2.3. Megjegyzés. Érdemes mindig **int**-et használnunk, hanincs jó okunk arra, hogy mást használjunk. Az **int**-el általában a leghatékonyabb a processzor.

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:

$$\begin{aligned} \text{sizeof}(X) &== \text{sizeof}(\text{signed } X) == \text{sizeof}(\text{unsigned } X) \\ \text{sizeof}(\text{float}) &\leq \text{sizeof}(\text{double}) \leq \text{sizeof}(\text{long double}) \\ \text{sizeof}(\text{short}) &\leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \\ \text{sizeof}(\text{char}) &\leq \text{sizeof}(\text{bool}) \end{aligned}$$

2. Struktúrák mérete

```
#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 érték 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 *paddinget* 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 adotttagok 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;
}
```