# C++
Gyakorlat jegyzet
3. óra

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

## 1. Statikus változók/függvények

A `static` kulcsszónak számos jelentése van, annak függvényében, hogy milyen kontextusban írjuk egy változó vagy függvény elé.

### 1.1. Fordítási egységre lokális változók

A függvényeken és osztályokon kívül deklarált statikus változók az adott fordítási egységre lokálisak – élettartamuk a futás elejétől végigtartamig tart, és kizárólagosan az adott fordítási egységben láthatóak.

**main.cpp**

```
#include <iostream>

static int x;

int main()
{
    x = 2;
}
```

**other.cpp**

```
#include <iostream>

static int x;

void f()
{
    x = 0;
}
```

Ha ezt a két fájlt együtt fordítjuk, nem kapunk linkelési hibát, ugyanis a `main.cpp`-ben lévő `x` egy teljesen más változó, mint ami az `other.cpp`-ben van.

Csak úgy mint a globális változókra, fordítási egységen belül bármikor hivatkozhatunk egy statikusra, és hasonló módon inicializálódnak.

```
static int x;

int main()
{
    int x = 4;
    std::cout << ::x << std::endl; // 0
}
```

### 1.2. Függvényen belüli statikus változók

Azokat a változókat, melyek függvényen belül vannak a `static` kulcsszóval definiálva, függvényszintű változónak is szokás hívni. Élettartamuk a függvény első hívásától a program futásának végéig tart, míg láthatóságuk csak az adott függvényen belül van. A hagyományos lokális változókkal ellenben tehát nem semmisülnek meg, amikor az adott függvény futása befejeződik. A következő kódrészlet szemlélteti ezt a viselkedést.

```
int f()
{
    static int x = 0;
    ++x;
    return x;
}

int main()
{
    for(int i = 0; i<5; i++)
        std::cout << f() << ' '; // 1 2 3 4 5
}
```

Ahogy az megfigyelhető fent, `x` csak egyszer inicializálódik, majd a későbbi függvényhívások után egyre növekszik az értéke.

### 1.3. Fordítási egységre lokális függvények

Nem csak változókat, függvényeket is deklarálhatunk statikusnak, melyek a fordítási egységre lokálisak.

```
static int f()
{
    return 0;
}

int main() {std::cout << f();} // 0
```

Ezek a függvények csak az adott fordítási egységen belül érhetőek el.

**1.3.1. Megjegyzés.** Figyelem! A később szóba kerülő metódusok esetében mást jelent a `static` kulcsszó, mint amit itt leírtunk.

### 1.4. Névtelen/anonim névterek

Fordítási egységre lokális változókat és függvényeket tudunk deklarálni névtelen névterek (*unnamed namespaces*, vagy *anonymous namespaces*) segítségével. Egy név nélküli névteren belül deklarált változók és függvények hasonlóan viselkednek, mintha eléjük lenne írva a `static` kulcsszó.

```
namespace
{
    int x;
    std::string y;
    void f() {}
}
```

**1.4.1. Megjegyzés.** A `static` osztályon belüli jelentéséről később lesz szó.

## 2. Függvény túlterhelés

Térjünk vissza a korábban megírt swap függvényünkhöz.

```
void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Ez a függvény addig jó, amíg csak `int`-eket szeretnénk megcserélni. Mi van, ha `std::string`-eket kéne? A megoldás egyszerű, **túlterheljük** (*overload*) a `swap` függvényt.

```
void swap(std::string &a, std::string &b)
{
    std::string tmp = a;
    a = b;
    b = tmp;
}
```

Túlterhelésnek azt nevezzük, amikor két vagy több függvénynek a neve azonos, de a paramétereik különböznek. Tagfüggvényeket konstansság alapján is túl lehet terhelni.

**2.0.1. Megjegyzés.** A később elhangzó osztályok tagfüggvényeinél a függvény konstanssága is számít (azonos nevű és paraméter listájú függvény különöző konstansséggal ugyanúgy túlterhelésnek számít).

## 2.1. Operátor túlterhelés

Ha példaként vesszük a lineáris algebrából tanult rendezett valós számhármasokat ($\mathbb{R}^3$), lehetőségünk van arra, hogy a tanultak alapján definiáljuk a köztük értelmezett összeadást.

```
struct LinAlgVector
{
    double x1, x2, x3;
};

LinAlgVector operator+(const LinAlgVector &lhs, const LinAlgVector &rhs)
{
    LinAlgVector ret;
    ret.x1 = lhs.x1 + rhs.x1;
    ret.x2 = lhs.x2 + rhs.x2;
    ret.x3 = lhs.x3 + rhs.x3;
    return ret;
}

int main()
{
    LinAlgVector a, b;
    a.x1 = 1; a.x2 = 2; a.x3 = 3;
    b.x1 = 1; b.x2 = 1; b.x3 = 1;

    LinAlgVector c = a + b;
}
```

A `main` függvényben lévő értékadás ezzel ekvivalens: `c = operator+(a, b)`, így láthatjuk, hogy az operátorok túlterhelése gyakorlatilag a függvénytúlterhelés speciális esete.

Írjuk meg a `print` függvényt 3D vektorokra! A gyakran kiíratáshoz használt jobb shift operátor (*right shift operator*), a `<<` is túlterhelhető. Mivel mi az `std::cout` változóval szeretnénk majd kiíratni, melynek típusa `std::ostream`, így a függvényünk első paramétere egy ilyen típus lesz, a második meg egy `LinAlgVector` típus.

```
/* ... */

std::ostream& operator<<(std::ostream& os, const LinAlgVector &l)
{
    os << l.x1 << ' ' << l.x2 << ' ' << l.x3;
    return os;
}

int main()
{
    /* ... */
    std::cout << c << std::endl; // 2 3 4
```

```
|| }
```

Feltűnhet, hogy a stream objektumra mutató referenciát a függvény végén vissza is adjuk, hogy tudjuk a kiíratást láncolni.