



# Bevezetés a programozásba 2

1. Előadás: Tagfüggvények, osztály, objektum

# Ismétlés

```
int main() {  
    string s;  
    s="bla";  
    cout << s.length();  
}
```

# Tagfüggvény hívása

Kell hozzá változó

„objektum”

Mező jellegű, „ . ” alakú hozzáférés, az objektumnak valami sajátjáról van szó

Függvény jellegű szintaxis, zárójelek, esetleg paraméterek

Első félévben sok ilyet láttunk, előadáson is szerepelt

# Hasonló megoldás

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
};
```

```
void rajzol(Particle p) {  
    gout << move_to(p.x, p.y)  
    << color (p.r, p.g, p.b)  
    << dot;  
}
```

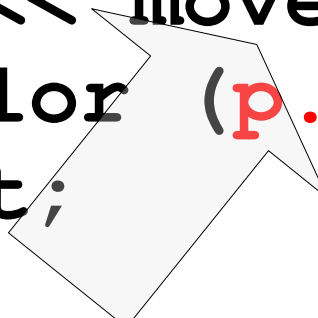
```
Particle p;
```

```
...
```

```
rajzol(p);
```

# „Tagfüggvényesítés”

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    void rajzol(Particle p) {  
        gout << move_to(p.x, p.y)  
        << color (p.r, p.g, p.b)  
        << dot;  
    }  
};
```



# Tagfüggvény

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    void rajzol() {  
        gout << move_to(x, y)  
        << color (r, g, b)  
        << dot;  
    }  
};
```

```
Particle p;  
...  
p.rajzol();
```

# Tagfüggvényhasználat

Elsődleges szerep: a típus saját műveleteinek nyelvi egysége az adatokkal

A típus: adat és művelet

Jótékony hatása:

Az adatmezőkre hivatkozás feleslegessé válik

Ezért funkció változtatáskor sokszor elég a tagfüggvényekhez nyúlni

Ezek a programkód jól meghatározható részét alkotják, nem lesz kifelejtve semmi

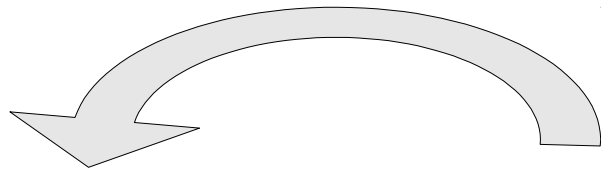
# Másik szintaxis

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
};  
  
void rajzol(Particle p) {  
    gout << move_to(p.x, p.y)  
    << color (p.r, p.g, p.b)  
    << dot;  
}
```



# Másik szintaxis

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
};
```



```
void Particle::rajzol() {  
    gout << move_to(p.x, p.y)  
    << color (p.r, p.g, p.b)  
    << dot;  
}
```

# Másik szintaxis

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    void rajzol();  
};  
  
void Particle::rajzol() {  
    gout << move_to(x, y)  
    << color (r, g, b)  
    << dot;  
}
```

# Interface - Implementation

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    void rajzol();  
};
```

---

```
void Particle::rajzol() {  
    gout << move_to(x, y)  
    << color (r, g, b)  
    << dot;  
}
```

# Kétféle szintaxis

Egyelőre bármelyik használható

Nincs jobb vagy rosszabb

Ha egy típusunknak csupa egyszerű tagfüggvénye van, vagy fontosnak ítéljük a forráskód megtartását, úgy az első írásforma a jobb

Ha könyvtárat akarunk fordítani, el kell választani a felületet a megvalósítástól, a bonyolultabb, vagy épp ki nem adandó kódot előre lefordíthatjuk

# Láthatóság szabályozása

```
struct Particle {  
private:  
    int x,y;  
    unsigned char r,g,b;  
public:  
    void rajzol();  
};  
void Particle::rajzol() {  
    gout << move_to(x, y)  
    << color (r, g, b)  
    << dot;  
}
```

# Láthatóság szabályozása

- private: csak a tagfüggvények számára látható
- public: a külvilág számára is látható
- cél: reprezentáció változása ne jelentsen problémát
- ökölszabály: mezők private, tagfüggvények public
  - van sok kivétel
- getter/setter: csupán mező lekérdezésére és beállítására használt tagfüggvény. Ha sok van mindkettőből, az rossz jel.

# Osztály, objektum

- Egy olyan struct, ami megvalósítja a láthatóság szabályozásával és a tagfüggvényekkel, hogy a reprezentációjától függetlenül hasznos, szokás osztálynak nevezni
- Azt a változót, aminek a típusa osztály, szokás objektumnak nevezni
- Objektum orientált programozás, OOP
- „Egységbezárás”

# Speciális tagfüggvények

Konstruktor

Destruktor

Másoló konstruktor

Értékadó operátor

Ezek mindegyike objektum létrejöttével,  
megszűnésével, vagy másolásával  
foglalkoznak

Ha te nem írsz, akkor is van!



# Konstruktor

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    Particle() {  
        x=y=r=g=b=0;  
    }  
};
```

```
Particle p;
```

# Konstruktor

Az objektum létrejöttekor fut le

A neve a típus neve

Kezdeti érték adható a mezőknek

Ha nem írsz konstruktort, az alapértelmezett konstruktor paraméter nélküli, és nem csinál semmit

Ha írsz konstruktort, nem készül alapértelmezett konstruktor

Több konstruktor is lehet, ha eltérnek paraméterezésben

# Paraméteres konstruktor

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    Particle(int X,int Y) {  
        x=X;  
        y=Y;  
        r=g=b=0;  
    }  
};  
Particle p(100,100);
```

# Paraméteres konstruktor

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    Particle(int X,int Y) {  
        x=X;  
        y=Y;  
        r=g=b=0;  
    }  
};
```

```
Particle p(100,100);  
Particle q;
```

# Paraméteres konstruktor

Nagyon hasznos: csak úgy lehet példányt csinálni, hogy rákényszerülünk a kezdeti értékről való gondoskodásra

Ugyanakkor néhány kényelmetlenség felmerül

Csak úgy nem lehet mező típusa

Csak úgy nem lehet `vector<IDE>` -ba tenni

Ezek megoldhatóak

```
vector<Particle> v(10, Particle(10,10));
```

# Paraméteres konstruktor mint mező

```
struct Particle {  
    Particle(int X,int Y) {  
        x=X;
```

```
    Particle p;  
    string s;  
};
```

```
}; TextParticle t;
```

# Paraméteres konstruktor mint mező

```
struct Particle {  
    Particle(int X,int Y) {  
        x=X;
```

```
struct TextParticle {  
    Particle p;  
    string s;  
    TextParticle() : p(0,0) {  
        s="bla";  
};
```

```
};
```

```
TextParticle t;
```

# Konstruktor örökítés

Mezők kezdeti értékét úgy is meg lehet adni, ha a konstruktor mögé kettősponttal felsoroljuk, és konstruktorparaméterként adjuk meg a kezdetiértéket

Ha nincs paraméter nélküli konstruktora a mezőnek, ez az egyetlen mód a példányosításra

Ez mellesleg kicsit hatékonyabb az értékadásnál



# Destruktor

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    ~Particle() {  
        cout << "kampec";  
    }  
};
```

```
{  
    Particle p;  
}
```

# Destruktor

Az objektum megszűnésekor fut le

A neve a típus neve, egy ~ jellel

Lehet takarítani az objektum után

hasznos példányszámláláskor

Nincs paramétere

Ha nem csinálsz, lesz alapértelmezett, ami nem csinál semmit

# Másoló konstruktor

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    Particle(const Particle &a)  
    {  
        x=a.x; y=a.y; ...  
    }  
};
```

```
Particle p;  
...  
Particle q(p);  
Particle w=p;
```

# Másoló konstruktor

Copy constructor

A neve a típus neve, paramétere fix

Lefut többek között

érték szerinti paraméterátadásnál

kifejezés kiértékeléskor átmeneti érték tárolására

vectorban: egy konstruktor fut le és lemásolódik sok példányban

Ha nem írsz másoló konstruktort, akkor is lesz egy, ami minden mező másoló konstruktorát hívja meg

# Másoló konstruktor

```
struct A {
    int a,b;
    int &r;
    A(int pa, int pb) :r(a) {
        a=pa; b=pb;
    }
};

int main() {
    A x(1,2);
    cout << x.a << x.b <<x.r << endl;
    A y(x);
    y.a=3; y.b=4;
    cout << x.a << x.b << x.r << endl
        << y.a << y.b << y.r << endl;
}
```

# Másoló konstruktor

```
struct A {  
    int a,b;  
    int &r;  
    A(int pa, int pb) :r(a) {  
        a=pa; b=pb;  
    }  
};
```

referencia

kezdeti érték

```
int main() {  
    A x(1,2);  
    cout << x.a << x.b <<x.r << endl;  
    A y(x);  
    y.a=3; y.b=4;  
    cout << x.a << x.b << x.r << endl  
        << y.a << y.b << y.r << endl;  
}
```

másolás

átmutat  
x-be!

# Másoló konstruktor

Akkor szokás megírni, ha a mezőnkénti másolás nem elég

- tartalmaz referenciát a típus
- bizonyos értékeknek különbözniük kell

Szerencsés esetben nincs rá szükség

- nincs referencia típus a mezőnkben
- aminek mégis van, annak már van saját másoló konstruktora (ilyen pl. a vector)

# Értékadó operátor

```
struct Particle {  
    int x,y;  
    unsigned char r,g,b;  
    Particle & operator=(const Particle &a)  
    {  
        x=a.x; y=a.y; ...  
        return *this;  
    }  
};
```

```
Particle p;  
...  
Particle q;  
q=p;
```



# Értékadó operátor

Értékadáskor hívódik meg

A neve operator=, visszatérési típusa a saját típusra referencia, paramétere fix

Mindig visszaadja saját magát: \*this

Emiatt lehet a=b=c; értékadást írni

Ha nem írsz értékadást, az alapértelmezett minden mezőre értékadást hív

# Összefoglalás

Minden struct kibővíthető tagfüggvényekkel

Ezek a mezőket változóként kezelhetik, hiszen meghíváskor szükség volt az implicit paraméterre

A :: jelzi, hogy egy típus tagfüggvényét írjuk le

A \*this jelenti az implicit paramétert „belülről”

A speciális tagfüggvények akkor is léteznek, ha nem írod meg őket, érdemes tudni, mit csinálnak