



# Bevezetés a programozásba 2

## 4. Előadás: Öröklődés 2

# 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();  
};  
  
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;  
}
```

# Speciális tagfüggvények

Konstruktor

Destruktor

Másoló konstruktor

Értékadó operátor

Ezek mindegyike objektum létrejöttével,  
megszűnésével, vagy másolásával  
foglalkoznak

Ha te nem írsz, akkor is van!

# Példa öröklődésre: Ős és örökös

```
struct Particle {  
    int x,y;  
    void torol() {  
        gout << move_to(x, y)  
        << color (0, 0, 0) << dot;  
    }  
    void rajzol() {  
};
```

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

# Az „is a” reláció

```
struct A {  
};
```

```
struct B : public A {  
};
```

```
int main() {  
    A a;  
    B b;  
    a=b;  
    b=a;  
}
```

Garantálva van,  
hogy minden  
mező létezik



# Öröklődés

```
struct Futo : public Sakkbabu {  
    bool szabalyos(int cx, int cy);  
};
```

```
struct Bastya : public Sakkbabu {  
    bool szabalyos(int cx, int cy);  
};
```

# Ez lenne kényelmes

```
int main() {  
    vector<Sakkbabu> babuk(16);  
    babuk[0] = valahogy Futo  
    babuk[1] = valahogy Bastya  
    ...  
    bool sakkban=false;  
    for (int i=0;i<babuk.size();++i) {  
        if (babuk[i].szabalyos(kirx, kiry) {  
            sakkban=true;  
        }  
    }  
}
```

# Dinamikus változó

```
int main() {  
    int a=0;  
    int b(0);  
  
    int *m = new int(0);  
  
    cout << a;  
    cout << b;  
    cout << *m;  
  
    delete m;  
  
}
```

Típus\* : mutató

\*mutató : mutatott érték

new : kérünk memóriát most

delete : felszabadítás

veszélyes!

# Statikus és dinamikus típus

```
struct A {  
};  
  
struct B : public A {  
};  
  
int main() {  
    A *m = new B;  
}
```

Statikus

Dinamikus

Statikus típus: a deklaráció típusa

Dinamikus típus: a példányosítás típusa

Ez utóbbi menet közben dől el

# A dinamikus típus fordításkor ismeretlen

```
struct A { ... };
struct B : public A { ... };
struct C : public A { ... };
int main() {
    A *m;
    if (rand()%2) {
        m = new B;
    } else {
        m = new C;
    }
    // m dinamikus típusa fordítási időben nem ismert
}
```

# Mutatók, jelölések

`T v;`

`v=érték`

`v.mező=érték`

`vector<T> v;`

`v[i]=érték`

`v[i].mező=érték`

`T *m=new T;`

`*m=érték`

`m->mező=érték`

`vector<T *> mv;`

`*mv[i]=érték`

`mv[i]->mező=érték`

`delete m`

# Öröklődés és a konstruktorok

Példányosításkor az ősök konstruktorai is lefutnak

Ha nem csinálsz semmit, akkor az alapértelmezett konstruktort próbálja meg

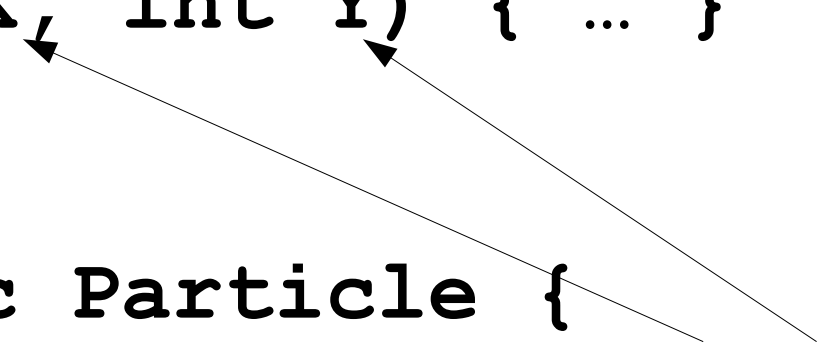
Ha az ősnek nincs paraméter nélküli konstruktora, akkor gondoskodni kell a paramétereiről

Ezt a konstruktornál kettősponttal tehetjük meg

Először a legősibb konstruktor fut le, és sorban az öröklődési lánc lépései

# Öröklődés és a konstruktorok

```
struct Particle {  
    Particle(int X, int Y) { ... }  
};  
  
struct Ho : public Particle {  
    Ho(int X, int Y) : Particle(X,Y)  
    { ... }  
};
```

The diagram consists of two thin black arrows. One arrow starts from the 'int X' parameter in the 'Ho' constructor and points to the 'int X' parameter in the 'Particle' constructor. The other arrow starts from the 'int Y' parameter in the 'Ho' constructor and points to the 'int Y' parameter in the 'Particle' constructor. This illustrates how the 'Ho' constructor delegates the initialization of its parameters to the 'Particle' constructor.



# Dinamikus típus alkalmazása

```
struct Particle {  
    ...  
    void mozog( ... );  
    ...  
};  
  
struct Ho : public Particle {  
    ...  
    void mozog( ... );  
    ...  
};
```

# Dinamikus típus alkalmazása

```
int main() {  
    vector<Particle *> v;  
    Particle *m1 = new Particle(X,Y);  
    Particle *m2 = new Ho(X,Y);  
    v.push_back(m1);  
    v.push_back(m2);  
  
    ...  
    for (int i=0;i<v.size();i++) {  
        v[i]->mozog( ... );  
    }  
  
    ...  
}
```

# Hol is tartunk?

Östípusból leszármazottat készítettünk

A közös részeket csak egyszer kellett megírni

A különbségeket bővítés mellett felüldefiniálással is megadhatjuk

Mutatóval példányosítva közös vektorba fűzhettük a rokonokat

Lefordul az a program, ami a közös szignatúrájú, de különböző implementációjú függvényeket hívja

Csak hogy egyformán viselkednek

# Kulcsszó: virtual

```
struct Particle {  
    ...  
    virtual void mozog( ... );  
    ...  
};  
  
struct Ho : public Particle {  
    ...  
    void mozog( ... );  
    ...  
};
```

A felhasználó  
rész változatlan

# virtual

Ha egy tagfüggvény `virtual`, az azt jelzi, hogy függvényhíváskor a dinamikus típus szerint dől el, hogy melyik tagfüggvény hívódik meg

Ha nincs `virtual`, mindig a statikus típus szerint hívódik meg a tagfüggvény

```
int main() {  
    vector<Particle *> v;  
    Particle *m1 = new Particle(X,Y);  
    Particle *m2 = new Ho(X,Y);  
  
    ...  
    for (int i=0;i<v.size();i++)  
    {  
        v[i]->mozog( ... );  
    }  
  
    ...  
}
```

```
struct Particle {  
    ...  
    virtual void mozog( ... );  
    ...  
};  
  
struct Ho : public Particle {  
    ...  
    void mozog( ... );  
  
    ...  
};
```

# A virtual használata

Ha egy tagfüggvényt arra tervezel, hogy az örökösök majd intézik a konkrét teendőt, az legyen virtual

Ha egy tagfüggvény szerepét rögzíteni akarsz, amit minden leszármazottnak egyformán kell csinálnia, akkor az nem virtual

Ha bizonytalan vagy, tervezd újra!

# Az osztály

A tagfüggvények mint típusműveletek

A láthatóság szabályozása

És az öröklődés lehetősége együtt a struct hagyományos fogalmánál annnyival gazdagabb, hogy **class**-nak hívjuk

Technikailag a különbség kicsi

Fogalmilag a különbség nagy

Illik jelezni a programokban

# Osztály

```
class Particle {  
public:  
    Particle(int X, int Y);  
    virtual void mozog( ... );  
    virtual void rajzol( ... );  
protected:  
    double x,y;  
    unsigned char r,g,b;  
};
```



# Mikortól class a struct?

Néhány tagfüggvény, és teljes láthatóság még struct

Láthatóság, vagy öröklődés bevezetésekor illik class-ra váltani

Egy rendes class-nak nincs publikus adatmezője  
... és nincs minden mezőhöz „szetter” tagfüggvénye

# Objektumorientált programozás

alias objektumelvű programozás

A problématerületi fogalmakból osztályokat képzünk (nehéz, rutin meg tapasztalat kell)

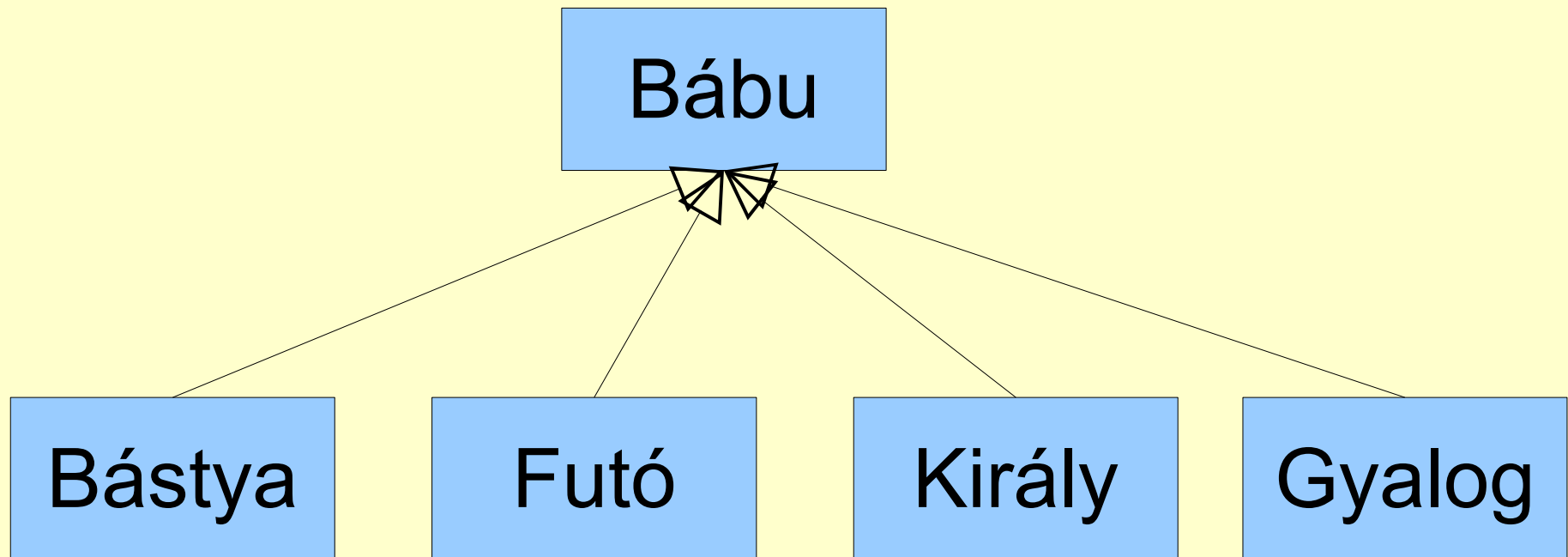
Örökösödési hálózat figyelembe vételével

A reprezentációt elrejtjük, a típust lényegében a tagfüggvényeivel jellemezzük (pl össze lehet adni őket)

A kész tervet akár csoportmunkában implementáljuk

# Osztályhierarchia

- Egy rendszer osztályainak örökösödési térképe gyakran hasznos vizualizáció (→ UML)



# Absztrakt osztály

- Ha egy osztályból szándékaink szerint nem készül objektum, ezt jelezhetjük

```
class Particle {  
public:  
    Particle(int X, int Y);  
    virtual void mozog( ... ) = 0;  
    virtual void rajzol( ... ) = 0;  
protected:  
    double x,y;  
    unsigned char r,g,b;  
};
```