Bevezetés a programozásba 2

3. Előadás: Bevezetés az öröklődésbe

ISMÉTLÉS

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();
```

ISMÉTLÉS

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

ISMÉTLÉS

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;
}
```

ISMÉTLÉS

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;
}
```

ISMÉTLÉS

Speciális tagfüggvények

- Konstruktor
- Destruktor
- Másoló konstruktor
- Értékadó operátor
- Ezek mindegyike objektum létrejöttével, megszűnésével, vagy másolásával foglalkoznak
- Ha te nem írsz, akkor is van!

Programozási stratégia

- Milyen szerepű típusokat használjunk?
  - Ez a legnehezebb (olyan mint az ultiban a licit)
  - Beleértendő a teljes tagfüggvény készlet
- A kiválasztott típusokat hogyan reprezentáljuk?
- A tagfüggvények implementálása az adott reprezentációval
- Főprogram megírása a típusokkal
- Kritika: akkor sok hasonló szerepű típusnál sok hasonló tagfüggvényt kell leírni. Hogyan lehetne ezen spórolni?

Öröklődés

- A struct megfogalmazásánál azzal kezdjük, hogy a struct tartalmaz minden mezőt és tagfüggvényt, ami egy másik, már meglevő structban van, és ezt bővítjük
- Ha akarjuk, akár felülbírálhatunk néhány tagfüggvényt is, de az egyformákat nem kell újra megírni.
- Ilyet sima függvényekkel nem lehet csinálni, a tagfüggvényhasználat egyik legfontosabb oka ez

Példa öröklődésre: Ős

```
struct Particle {
    int x,y;
    void torol() {
        gout << move_to(x, y)
        << color (0, 0, 0) << dot;
     }
    void rajzol() {
        gout << move_to(x, y)
        << color (255, 255, 255)
        << dot;
    }
};
```

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;
    }
};
```

Példa öröklődésre: Ős és örökös

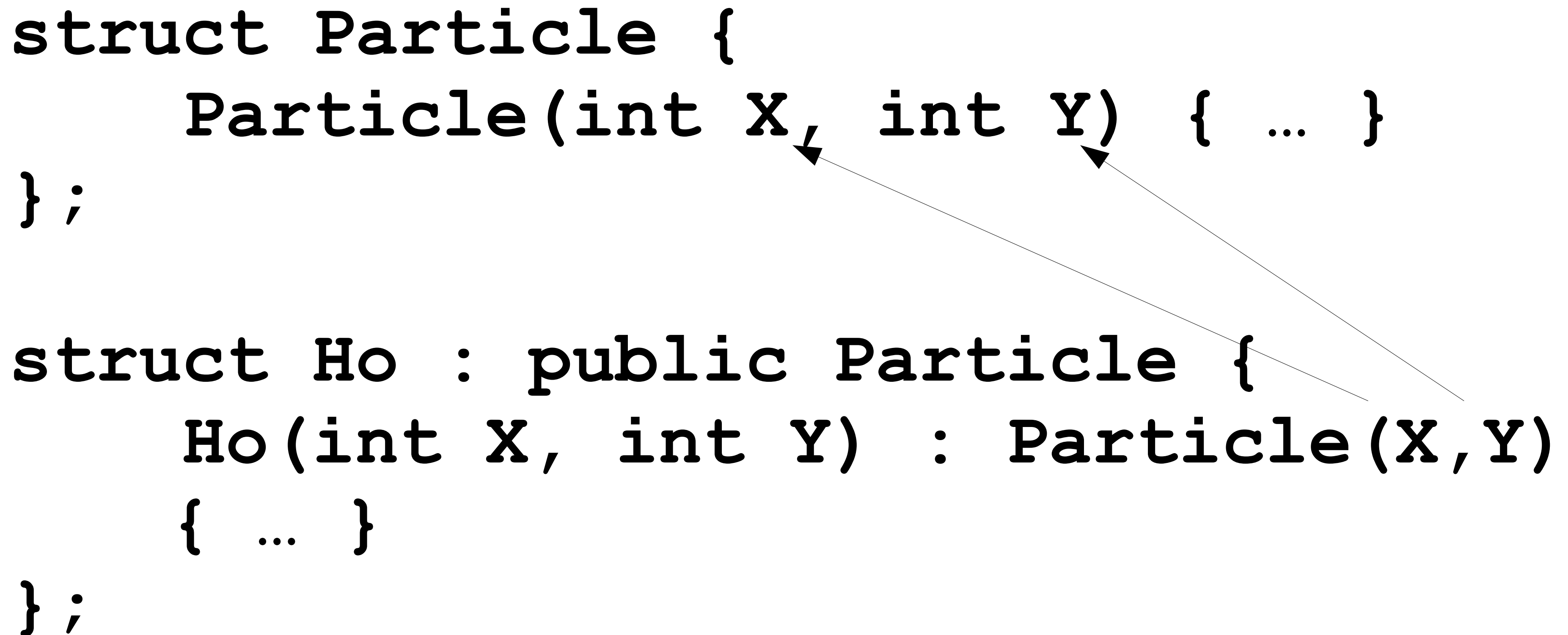
```
struct Particle {
    int x,y;
    void torol() {
        gout << move_to(x, y)
        << color (0, 0, 0) << dot;
     }
    void rajzol() {
```

```
    }
```

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

```
Colorparticle c;
c.x=c.y=100; c.r=255; c.g=c.b=128;
c.rajzol(); // … refresh …
c.torol();
c.x+=5;
c.rajzol(); // … refresh …
```

Öröklődés és a konstruktorok



Öröklődés

- Jelölése:
```
struct Os { .. };
struct Orokos : public Os { .. };
```
- Elnevezések:
  - Ős, ősosztály, bázis
  - Örökös, leszármazott
- A „:” az örökítés jele itt is, mint a konstruktornál
- A „public” azt jelenti, hogy az ős láthatósági viszonyait változtatás nélkül vesszük át.

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

- Ezzel a lehetőséggel megspórolhatunk hasonló tartalmú kódrészleteket, ami hasznos, mert
  - Adott funkció csak egyszer szerepel, ha változtatni kell, elég egy helyen változtatni
  - A forráskód rövidebb, tehát átláthatóbb, és kevesebb a hibalehetőség
  - Hamarabb készen van a program, mert nem kell sokat gépelni
- Ugyanakkor ez a lehetőség egy újfajta gondolkodást igényel: eleve érdemes úgy tervezni, hogy koncentrálunk a hasonlóságokra

Tervezés öröklődéssel

- Észrevettem, hogy szükség van hasonló dolgokra, mi a teendő?
  - Van átfedés a mezők között? (pl mindegyiknek koordinátái vannak)
  - Van átfedés a műveletek között? (pl mindegyiket ki lehet rajzolni)
  - Van különbség funkcióban? (pl. nincs, ha csak a színükben különböznek, de van, ha másképp mozognak)
- Ha ezekre a kérdésekre igen a válasz, érdemes megfontolni az öröklődést

Öröklődés

- Van tehát A és B típusom, amik között átfedés van, és funkcióbeli különbség
- Elkészítem a C nevű őst, amiben kizárólag a közös van benne
- Utána az A-t és a B-t úgy, hogy örökölnek C-től, és a különbségek vannak bennük leírva
- Végül használom A-t és B-t, és a közös tagfüggvények csak egyszer vannak megírva

Öröklődés

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

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

Öröklődés

```
struct Sakkbabu {
  int x,y;
  void lep(int cx, int cy) {
    x=cx;
    y=cy;
  }
  bool szabalyos(int cx, int cy) {
    //ez a bábu típusától függ
  }
};
```

Öröklődés

```
struct Sakkbabu {
  int x,y;
  void lep(int cx, int cy) {
    if (szabalyos(cx,cy)){
      x=cx;
      y=cy;
    }
  }
  bool szabalyos(int cx, int cy) {
    //ez a bábu típusától függ
  }
};
```

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;
    }
  }
}
```

Mi a technikai korlát?

- A típus fix, ha a vector Sakkbabut tartalmaz, akkor a Sakkbabu tagfüggvénye fog meghívódni, nem a leszármazottaké
- Egy közönséges változónak nem lehet egyszerre két típusa is.
- Kéne egy olyan konstrukció, ami lehetővé teszi, hogy
  - „két típusa legyen egy változónak”, egy amivel deklaráljuk, egy ami szerint tagfüggvénye hívódik
  - futás közben dőlhessen el ez utóbbi

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!

Mutatók

- A mutató veszélyes, mert
  - ha nem olyan memóriaterületre mutat, ami a mienk, akkor a program lefagy
  - ha nem oda mutat, ahová gondoljuk, nehezen magyarázható hibás működést kapunk
  - meglepően sok dolgot lehet mutatóval csinálni, pl. lehet hozzáadni számot, hogy mennyivel odébb mutasson. Ha elírsz valamit, jó eséllyel lefordul, és nem azt csinálja, amit vársz.
  - nem lehet eldönteni a mutatóról, hogy érvényes-e
    - kivétel a nullába mutató mutató: nullmutató. Az tuti nem.

Statikus és dinamikus típus

```
struct A {
};

struct B : public A {
};

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

Dinamikus

Statikus

- 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

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?

- Őstípusból 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
  - Csakhogy 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 akarod, 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 annyival 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 problématérbeli 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