Bevezetés a programozásba 2

3. Előadás: Öröklődés 1

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, tervezés

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

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 a külvilág ismeri a rokoni viszonyt, és kihasználhatja.

private öröklődésnél kívülről csak a most bevezetett elemek érhetőek el, az örököltek nem

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

Az „is a” reláció

```
struct A {
   int a;
};
struct B : public A {
   int b;
};
int main() {
     A a;
     B b;
     a=b;   //a.a=b.a; OK
     b=a;   //b.a=a.a; b.b=??
}
```

A protected láthatóság

```
struct A {
private:
   int priv_mezo;
protected:
   int prot_mezo;
};
struct B : public A {
   void fv() {
      cout << prot_mezo; //OK
      cout << priv_mezo; //hiba
   }
};
```

A protected láthatóság

```
struct A {
private:
   int priv_mezo
protected:
   int prot_mezo
};
struct B : public A {
   void fv() {
      cout << prot_mezo; //OK
      cout << priv_mezo; //hiba
   }
};
```

```
int main() {
 A a;
 a.priv_mezo=0; //hiba
 a.prot_mezo=0; //hiba
}
```

Öröklődés

Ezzel a lehetőséggel megspórolhatunk hasonló tartalmú kódrészleteket, ami hasznos, mert

- Könnyen létrehozhatunk új típust, ami majdnem olyan mint egy már kipróbált meglevő
- Adott funkció csak egyszer szerepel, ha változtatni kell, elég egy helyen változtatni
- A forráskód rövidebb, tehát átláthatóbb, és kevesebb a hibalehetőség

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

ISMÉTLÉS

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;
}
```

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
}
```

Folytatjuk...

A következő epizód tartalmából:

- A **virtual** módosító tagfüggvényeknél
- Mire kell figyelni a konstruktor megírásakor
- Mi az a virtuális destruktor
- És egy teljes példa, amin az eddig látottakat mindenki megérti