Bevezetés a programozásba

10. Előadás: A `struct`

ISMÉTLÉS

Struct

```
#include<iostream>

using namespace std;

void rac_osszead(int& s1, int& n1, int s2, int n2)
{
    s1 = s1*n2 + s2*n1;
    n1 = n1 * n2;
}

int main()
{
    int sz1,ne1;
    int sz2,ne2;
    sz1 = 3; ne1 = 6;
    sz2 = 5; ne2 = 6;

    rac_osszead(sz1,ne1,sz2,ne2);
    //sz1==48; ne1==36; sz2==5; ne2==6

    return 0;
}
```

ISMÉTLÉS

Struct

```
#include<iostream>

using namespace std;

struct Rac
{
    int sz,ne;
};

void rac_osszead(Rac& _r1, Rac _r2)
{
    _r1.sz = _r1.sz*_r2.ne + _r2.sz*_r1.ne;
    _r1.ne = _r1.ne*_r2.ne;
}

int main()
{
    Rac r1 ={3,6};
    Rac r2 ={5,6};

    rac_osszead(r1,r2);
    //rT.sz==48; r1.ne==36; r2.sz==5; r2.ne==6

    return 0;
}
```

ISMÉTLÉS

Struct

```
#include<iostream>

using namespace std;

struct Rac
{
    int sz,ne;
};

Rac rac_osszead(Rac _r1, Rac _r2)
{
    _r1.sz = _r1.sz*_r2.ne + _r2.sz*_r1.ne;
    _r1.ne = _r1.ne*_r2.ne;
    return _r1;
}

int main()
{
    Rac r1 ={3,6};
    Rac r2 ={5,6};
    Rac r3;
    r3 = rac_osszead(r1,r2); //r3.sz==48; r3.ne==36

    return 0;
}
```

Operátorok

```
#include<iostream>

using namespace std;

struct Rac
{
    int sz,ne;
};

int main()
{
    Rac r1 ={3,6};
    Rac r2 ={5,6};

    r3 = r1 + r2;

    return 0;
}
```

Operátorok

- Két azonos struct típusú változó egymásnak értékül adható, de sok más, pl. az egyenlőség vizsgálat már nem működik
- **`error: no match for 'operator+' in 'r1 + r2'`**
- Ha szükségünk van erre az operátorra, akkor meg lehet írni
- Az operátorok valójában speciális nevű és használathatóságú függvények, amiket ki lehet kiterjeszteni új típusokra

Operátorok

```
#include<iostream>

using namespace std;

struct Rac
{
    int sz,ne;
};

Rac operator+ (Rac _r1, Rac _r2)
{
    _r1.sz = _r1.sz*_r2.ne + _r2.sz*_r1.ne;
    _r1.ne = _r1.ne*_r2.ne;
    return _r1;
}

int main()
{
    Rac r1 ={3,6};
    Rac r2 ={5,6};
    Rac r3;

    r3 = r1 + r2; //r3.sz==48; r3.ne==36

    return 0;
}
```

Operátorok

- Megvalósítható operátorok:
  - @a alakú: `+ - * & ! ~ ++ --`
    - **`operator@(A) {..}`**
  - a@ alakú: `++ --`
    - **`operator@(A, int) {..}`**
  - a@b alakú : `+ - * / % ^ & | < > == != <= >= << >> && || ,`
    - **`operator@(A,B) {..}`**
- Az operátorok szerepét illik a nevükhöz és a szokásos jelentésükhöz igazodva használni.
- Alaptípusokra nem bírálhatjuk felül az operátort

Operátorok

- Ami nem volt közte: értékadás, értékadó összetett műveletek (pl. +=)
- Néhány jellegzetes, gyakori operátorhasználat:
  - ostream & operator<<(ostream& ki, T t)
  - istream & operator>>(istream& be, T& t)

```
istream& operator>>(istream &be, pont &p) {
    be >> p.x >> p.y;
    return be;
}
...
pont a,b;
cin >> a >> b;
```

Operátorok: visszatérési típus

```
istream& operator>>(istream &be, pont &p) {
    be >> p.x >> p.y;
    return be;
}
...
pont a,b;
(cin >> a) >> b; //OK
```

```
void operator>>(istream &be, pont &p) {
    be >> p.x >> p.y;
}
...
pont a,b;
cin >> a; // ez még jó
cin >> a >> b; //hiba
```

Operátorok

- Ha olyan típust készítünk, amire természetes módon értelmezhetőek az operátorok (pl. racionális szám), valósítsuk meg! (természetesen minden más függvény mellett)
  - Ezzel a típusunk és a műveleteink egységet alkothatnak
  - A kód olvashatósága javul
  - A típus újrafelhasználhatóvá válik, más feladat megoldására változtatás nélkül átvihető
  - Csökken a kísértés, hogy a mezőket közvetlenül megváltoztassuk, ezzel esetleg inkonzisztens állapotot létrehozva

Tervezési kérdések

- A struct mezőinek változása egy viszonylag gyakori esemény, és az a cél, hogy ennek a hatása minimális legyen
- Ezért minden olyan függvényt, aminek a paraméterei egy struct mezői közül kerülnek ki, ne külön vegyük át, hanem egyben
  - Ezzel a szignatúra egyszerűsödik, de feladjuk azt a lehetőséget, hogy struct nélkül is használható legyen a függvény. Ez utóbbi viszont csak jól behatárolható helyzetekben lehet fontos, ritkán merül fel (pl. scriptnyelvek), és akkor is orvosolható

Tervezési kérdések

```
struct pont {
    double x,y;
};

double tav(pont a, pont b) {
    return sqrt((a.x-b.x)*(a.x-b.x)+
                (a.y-b.y)*(a.y-b.y));
}

int main()
{
    pont a={1.0,1.0}, b={0.0, 0.0};
    cout << tav(a,b);
    return 0;
}
```

Tervezési kérdések

```
struct pont {
    double x,y;
};

double tav(pont a, pont b) {
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));

}

pont legtavolabbi(pont a, vector<pont> v) {
    pont b=v[0];
    double max=tav(a,b);
    for (int i=1;i<v.size();i++) {
        if (tav(a,v[i])>max) {
            b=v[i];
            max=tav(a,b);
        }
    }
    return b;
}
```

Tervezési kérdések

```
struct pont {
    double x,y,z;
};

double tav(pont a, pont b) {
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y)+
           (a.z-b.z)*(a.z-b.z));
}

pont legtavolabbi(pont a, vector<pont> v) {
    pont b=v[0];
    double max=tav(a,b);
    for (int i=1;i<v.size();i++) {
        if (tav(a,v[i])>max) {
            b=v[i];
            max=tav(a,b);
        }
    }
    return b;
}
```

Tervezési kérdések

- Az előző példában egy teljes függvényben megspóroltuk a változtatási kényszert
- Előnyök:
  - Kevesebbet kell gépelni, ha változás van
    - Ez nem csak időbeli megtakarítás, de *biztonság*ot is ad: ha nem kell átírni, akkor nincs mit elfelejteni
  - A kód jobban tükrözi a célt
  - Körvonalazódik a típus és a típus műveleteinek a kapcsolata
    - Vannak függvények, amik „belelátnak”, és vannak, amik a „belelátó függvények”-et használják

Tervezési kérdések

- Absztrakció: egy entitás tulajdonságainak olyan szűkítése, ami egy adott szempont szerint csak a fontosakat tartja meg
- A struct mezőit tehát a felhasználás szabja meg
- Az entitás „fontos” tulajdonságai azok, amik alapján a feladat szempontjai szerint leírhatóak, egymástól megkülönböztethetőek az egyes objektumok
  - Tipikus példa a jó kitöltendő űrlap

Tervezési kérdések

- A struct akkor jó, ha
  - A neve és a szerepe jól illeszkedik egymáshoz
  - A szerepe és a mezői között fennáll a kölcsönös szükségesség
    - Csak a „fontos” adatok vannak a mezők között (nem kerül bele irreleváns (pl. ciklusváltozó) vagy mindenhol egyforma adat)
    - Minden „fontos” adat benne van
  - A mezők által meghatározott szerep és a felhasználás harmonikus
    - Minden mezőt felhasznál a feladat megoldása

Tervezési kérdések: gyakorlatban

- A feladat megértése után a lehetséges megoldásokhoz szükséges adatokat írjuk a mezőkbe
- Elkészítjük a szükséges műveleteket, ilyenkor a main()-ban csak teszteljük ezek helyességét
- Végül a már egyszerűen leírható feladatot implementáljuk a main()-ben
- Fontos, hogy folyamatosan, akár két-három programsoronként győződjünk meg az eddigiek működőképességéről

Tagfüggvények

- A string hosszát s.length() függvénnyel kérjük le
  - ez függvény, meg lehet hívni, és () van a végén
  - ez mező, a változó után '.'-al elválasztva van írva
- Mindkettő: tagfüggvény
- A legfontosabb tulajdonsága a tagfüggvénynek:
  - mindig egy struct típusú változóra hívjuk meg, ezt implicit paraméterként megkapja úgy, hogy a paraméterlistájában nem szerepel, de a mezőneveket, mint önálló változókat használhatja

Külső függvényként

```
struct koord {
    double x,y;
};

koord olvas(istream &be) {
    koord a;
    be >> a.x >> a.y;
    return a;
}

int main() {
    koord a;
    a = olvas(cin);
    cout << a.x << "," << a.y <<endl;
    return 0;
}
```

Tagfüggvényként

```
struct koord {
    double x,y;
    void olvas(istream &be) {
        be >> x >> y;
    }
};

int main() {
    koord a;
    a.olvas(cin);
    cout << a.x << "," << a.y <<endl;
    return 0;
}
```

Tagfüggvények

- Az igazán szép megoldások mindent amit lehet, tagfüggvényként adnak meg
  - Néhány dolgot nem érdemes tagfüggvényként megadni, tipikus az operator<<(T), amit ugyan meg lehet csinálni, de ez a „*mi structunk* << T” alakot jelenti
  - Végülis megy, de kicsit zavaró az eredmény:
    - `cout << a << b;` helyett `b >> (a >> cout);`
- Minden más esetben a tagfüggvény a jobb

Tagfüggvények

- A kimaradt operátorok is megvalósíthatóak, de kizárólag tagfüggvényként
- Tehát ha értékadást szeretnél a típusodhoz, tagfüggvényként kell leírnod
-
```
struct S{
  …
   void operator=(S masik) {
   .. mezo=masik.mezo;..
   }
  …
};
```

Értékadás operátor

```
struct koord {
    double x,y;
    void operator=(koord masik) {
        x=masik.x; y=masik.y;
    }
};


int main() {
    koord a,b;
    b=a;
    return 0;
}
```

Értékadás operátor

```
struct koord {
    double x,y;
    void operator=(koord masik) {
        x=masik.x; y=masik.y;
    }
};


int main() {
    koord a,b;
    c=b=a; //hiba
    return 0;
}
```

Értékadás operátor

```
struct koord {
    double x,y;
    koord operator=(koord masik) {
        x=masik.x; y=masik.y;
        return *this;
    }
};


int main() {
    koord a,b,c;
    c=b=a;
    return 0;
}
```

A *this jelentése: az implicit paraméter

Az összes értékadó tagfüggvény végén kell

Speciális tagfüggvények

- Három speciális tagfüggvényt nézünk, mindben közös, hogy a neve a típus neve
- struct S {
  - S(*paraméterek*) : konstruktor, minden deklarációnál lefut, így működik pl az ofstream f(”a.txt”);
  - ~S() : destruktor, élettartam végén lefut
  - S(const S& m) : másoló konstruktor, inicializáláskor ez hívódik meg, és nem az értékadás
- Nem kötelező kettesért, de roppant hasznos
  - Pl. kezdetiérték problémákat jól lehet kezelni

Láthatóság szabályozása

- A struct-on belül lehetséges az egyes mezők vagy tagfüggvények láthatóságának módosítása
- Láthatósági módosítók:
  - public: mindenki mindent lát, ez az alapértelmezett
  - private: ami innen az első „public:”-ig van, azt csak a tagfüggvények láthatják
- Ezzel megelőzhető, hogy a program „illetéktelen” részéről inkonzisztens állapotot idézzenek elő

A private mezők csak tagfüggvényekből érhetőek el

```
struct koord {
private:
    double x,y;
public:
    void olvas(istream &be) {
        be >> x >> y;
    }
};

int main() {
    koord a;
    a.olvas(cin); //OK
    cout << a.x << "," << a.y <<endl; //hiba
    return 0;
}
```

Láthatóság szabályozása

- „Átlátszatlan típus” fogalma: a típus reprezentációja nem ismert vagy nem használható közvetlenül, kizárólag a tagfüggvényeken keresztül lehet a típust használni
- Megvalósítás: minden mező private, minden tagfüggvény public
- Ennek a technikának az előnyeit főleg a csoportos programozásnál élvezhetjük

Kitekintés

- A struct fogalma, a műveletek és a láthatóság módosítása együtt messzire vezet
  - Az objektumorientált programozás előszobája
- Adat absztrakció: a feladat megoldását absztrakt fogalmakkal is fel lehet írni, ha megvan a kapocs az absztrakt fogalmak és az implementáció között
  - az absztrakt fogalmak a rekordok
  - a kapocs a megvalósított függvény

Összefoglalás

- A struct arra való, hogy adatokat összefogva új típust hozzunk létre
- A típusunkhoz műveleteket csinálhatunk
  - függvényekkel, amelyek paraméterként vagy visszatérési típusként használják
  - operátorokkal
  - tagfüggvényekkel
- A láthatóság szabályozásával esetleg kikényszeríthetjük, hogy a típust kizárólag a műveleteivel használják