Bevezetés a programozásba

10. Előadás
Rekordok (folyt.)

Struct

```
#include<iostream>
using namespace std;

struct Rac {
    int sz, ne;
};

void rac_hozzaad( 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_hozzaad( r1, r2 );
    // r1.sz == 48; r1.ne == 36; r2.sz == 5; r2.ne == 6;
    return 0;
}
```

Struct

```cpp
#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;
}
```

Hiba: Ismeretlen művelet

```
#include<iostream>
using namespace std;

struct Rac {
    int sz, ne;
};

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

    r3 = r1 + r2;

    return 0;
}
```

Operátor

- Két azonos **struct** típusú változó egymásnak értékül adható, de sok más művelet (pl. az egyenlőség vizsgálat) már nem működik

```
error: no match for 'operator+' in 'r1 + r2‘
```

- Ha szükségünk van erre az operátorra, akkor meg kell írni
- Az operátorok valójában speciális nevű és használatú függvények, amelyeket ki lehet terjeszteni új típusokra

Operátor előtt

```
#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;
}
```

Operátor túlterheléssel

```
#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 valamint 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
```

```
(cin >> a) >> b;
   cin >> b;
```

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

```
(cin >> a) >> b;
      >> b;  //?!
```

Operátorok készítése

- Ha olyan típust készítünk, amelyre 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 viszonylag gyakori esemény és az a cél, hogy ennek a hatása minimális legyen
- Azért minden olyan függvényben, amelynek a paraméterei egy **struct** mezői közül kerülnek ki, ne külön vegyük át a mezőket, hanem egyben a teljes rekordot
  - 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

```
#include<iostream>
using namespace std;

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 };
    pont b = { 0.0, 0.0 };

    cout << tav( a, b );

    return 0;
}
```

Rekord és függvényei

```
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;
}
```

Rekord és függvényei

```
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;
}
```

Rekord és függvényei

```
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;
}
```

absztrakciós határ

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 *biztonságot* 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, amelyek „belelátnak”, és vannak, amelyek a „belelátó függvények”-et használják

Tervezési kérdések

- Absztrakció: egy entitás tulajdonságainak olyan szűkítése, amely egy adott szempont szerint csak a fontosakat tartja meg
- A **struct** mezőit tehát a felhasználás szabja meg
- Az entitások „fontos” tulajdonságai azok, amelyek 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 vele 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()`**-ben 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 s`** változó hosszát az **`s.length()`** függvénnyel kérjük le
  - biztos, hogy függvény: meg lehet hívni, és () van a végén
  - biztos, hogy mező: a változó után '.' –al elválasztva van írva
- Igazából mindkettő: **tagfüggvény**
- A legfontosabb tulajdonságai a tagfüggvénynek:
  - mindig egy **struct** típusú változóra hívjuk meg
  - ezt a változót a tagfüggvény implicit paraméterként megkapja (a paraméterlistájában nem fog szerepelni)
  - a **struct** típus mezőneveit, mint önálló változókat használhatja

Külső függvényként

```
#include<iostream>
using namespace std;

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

```
#include<iostream>
using namespace std;

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, tipikusan az `operator<<(T)`, amit ugyan el lehet készíteni, de ilyenkor külön össze kell hangolni a tagfüggvényt a rekorddal: `friend` kulcsszó (Ez most nem képezi részét a törzsanyagnak)
- Minden más esetben a tagfüggvény a jobb

= operátor tagfüggvény

- 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; …
    }
    …
};
```

= operátor szintaxisa

```
#include<iostream>
using namespace std;

struct koord {
    double x, y;

    void operator= ( koord masik ){
        x = masik.x;
        y = masik.y;

    }
};

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

= operátor használata

```
#include<iostream>
using namespace std;

struct koord {
    double x, y;

    void operator= ( koord masik ){
        x = masik.x;
        y = masik.y;

    }
};

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

= operátor használata

```
#include<iostream>
using namespace std;

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 az objektum, amelyen a tagfüggvényt meghívtuk.

Az összes értékadó tagfüggvény végén fontos

Speciális tagfüggvények

- Három speciális tagfüggvényt nézünk, mindben közös, hogy a nevük megegyezik a típus nevével
- `struct S {`
  - S(*paraméterek*): konstruktor, minden változó deklarációnál lefut
    - így működik pl.: **`az ofstream f( "a.txt" );`**
  - ~S( ): destruktor, a változó élettartamának végén fut le
  - S(const S& m): másolókonstruktor, inicializáláskor ez hívódik meg, és nem az értékadás

  `}`
- Ezek nem kötelezőek a kettesért, de roppant hasznosak.
  - Például kezdeti érték problémákat jól lehet kezelni

Láthatósági szabályok

- A **struct**-on belül lehetséges az egyes mezők vagy tagfüggvények láthatóságának módosítása
- Láthatósági módosítók:
  - **public:** mindenki mindent lát
    - ez az alapértelmezett **struct** esetén
  - **private:** ezeket a dolgokat csak a tagfüggvények láthatják
- A módosítók a rekord végéig vagy a következő módosítóig hatnak
- Ezekkel megelőzhető, hogy a program kritikus részéről inkonzisztens állapot jöjjön létre

A private mezők csak tagfüggvényekből érhetőek el

```
#include<iostream>
using namespace std;

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**
- Az absztrakciós határ beleírása ez a forráskódba, akinek nem dolga, ne foglalkozzon a reprezentációval
- 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 függvények

Összefoglalás

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