



**Pázmány Péter Katolikus Egyetem  
Információs Technológiai és Bionikai Kar**

## **Bevezetés a Programozásba II**

---

### **3. előadás**

## **Biztonságos adattípusok megvalósítása**

---

**© 2014.02.24. Giachetta Roberto**

**groberto@inf.elte.hu**

**<http://people.inf.elte.hu/groberto>**

# Biztonságos adattípusok megvalósítása

## Adattípusok

- Adattípusok létrehozása során definiálnunk kell az *érték*halmazt, illetve a *műveleth*almazt
  - összetett típusok esetén típuskonstrukciókat használunk, amelyek megadják az egyszerű adatok összekapcsolásának módját
  - pl.  $Rational = (int \times int \setminus \{0\}, \{+, -, \dots\})$
  - az adattípusokat mindig *újrafelhasználhatónak* kell megvalósítanunk, és lehetővé kell tenni annak *biztonságos* használatát
  - az adattípusok példányaira garantálnunk kell, hogy mindig olyan értékeket vegyenek fel, amelyeket az értékhalmaz megenged, azaz *konzisztens állapotban* legyen

# Biztonságos adattípusok megvalósítása

## Adattípusok

- A konzisztens állapot csak akkor tartható fent, ha korlátozzuk az értékek beállítását
  - a bárki által lekérdezhető, illetve módosítható értékek ezt nem tehetik lehetővé, pl.:

```
struct Rational { // racionális szám típusa
    int numerator; // számláló
    int denominator; // nevező
    ... // típusműveletek
};
...
Rational r;
r.Denominator = 0;
    // inkonzisztens állapotba hozza r-t
```

# Biztonságos adattípusok megvalósítása

## Láthatóság

- Annak érdekében, hogy a biztonságos kezelést lehetővé tegyük, *korlátoznunk kell a hozzáférést* a mezőkhöz
  - le kell tiltanunk a külső hozzáférést, csak a típuson belül (metódusokon keresztül) lehessen az értékeket módosítani
- A típusok tagjainak külső hozzáférését a *láthatóság* kezelésén keresztül szabályozhatjuk
  - az *elrejtendő* tagokat **private**: kulcsszó mögé tesszük, ezek csak a típuson belül lesznek láthatóak
  - a mindenki számára *elérhető* tagokat a **public**: kulcsszó mögé tesszük
  - alapértelmezetten (kulcsszó nélkül) minden látható lesz

# Biztonságos adattípusok megvalósítása

## Láthatóság

- A rejtett tagok külső hívása fordítási hibához vezet, ezért csak a publikus tagokon (metódusokon) keresztül, ellenőrzött környezetben kezelhetők

- Pl.:

```
struct MyType{
    private:
        int value; // rejtett mező
    public:
        MyType() { value = 0; }
        // publikus konstruktor
        void Add(int v) { value += v; }
        // publikus metódus
};
```

# Biztonságos adattípusok megvalósítása

## Láthatóság és hozzáférés

- A használat során csak a publikus tartalmat érhetjük el, pl.:

```
MyType mt;
```

```
    // példányosítás a publikus konstruktorral
```

```
mt.Add(10); // publikus művelet hívása
```

```
int v = mt.value;
```

```
    // rejtett mező elérése, fordítási hibás okoz!
```

- Az elrejtett tagokat elnevezhetjük más módon (pl. \_ jellel kezdve), így könnyen azonosíthatjuk őket
- Amennyiben biztonságos küldő hozzáférést szeretnénk biztosítani a rejtett mezőkhöz, készíthetünk publikus *beállító* („setter”), illetve *lekérdező* („getter”) *műveleteket*

# Biztonságos adattípusok megvalósítása

## Láthatóság és hozzáférés

- Pl.:

```
struct MyType {  
private:  
    int _value; // rejtett mező  
public:  
    ... // konstruktor, egyéb metódusok  
  
    void setValue(int v) { if (v > 0) _value = v; }  
        // ellenőrzött beállítás  
    int getValue() { return _value; }  
        // lekérdezés  
};  
  
...  
myType.setValue(-1); // nem rontja el az állapotot
```

# Biztonságos adattípusok megvalósítása

## Példa

*Feladat:* Valósítsuk meg a téglalapok típusát úgy, hogy a méretek sohase lehessenek negatívak.

- ehhez elrejtjük a mezőket, és ellenőrzött beállításokat végzünk (a konstruktorban és a beállító műveletben)

*Megoldás:*

```
struct Rectangle { // téglalap típusa
    private: // rejtett rész
        double _width; // szélesség
        double _height; // magasság

    public: // látható rész
        Rectangle() { ... }
```



# Biztonságos adattípusok megvalósítása

## Példa

*Megoldás:*

```
Rectangle(double w, double h) {  
    // 2 paraméteres konstruktor művelet  
    _width = w < 0 ? 0 : w;  
    _height = h < 0 ? 0 : h;  
}  
...  
double getWidth() { return _width; }  
    // lekérdező műveletek  
...  
void setWidth(double w) {  
    _width = w < 0 ? 0 : w;  
} // beállító műveletek  
...
```

# Biztonságos adattípusok megvalósítása

## Példa

*Feladat:* Valósítsuk meg a racionális szám típusát úgy, hogy a 0 nevezőt nem engedélyezzük.

- ehhez elrejtjük a mezőket, és ellenőrzött beállításokat végzünk (a konstruktorban és a beállító műveletben)

*Megoldás:*

```
struct Rational {  
private: // a mezők rejtettek  
    int _numerator;  
    int _denominator;  
  
public: // a műveletek publikusak  
    Rational();
```

# Biztonságos adattípusok megvalósítása

## Példa

*Megoldás:*

```
...
// lekérdező és beállító műveletek:
void setNumerator(int value);
void setDenominator(int value);
int numerator();
int denominator();
...
};
...
void Rational::setDenominator(int value) {
    _denominator = value == 0 ? 1 : value;
    // speciális beállítás
}
```

# Biztonságos adattípusok megvalósítása

## Példa

*Megoldás:*

...

```
// külső értékmódosítási operátorok:
```

```
Rational operator+=(Rational& r1, Rational r2) {  
    r1.setNumerator(r1.numerator() *  
                    r2.denominator() + r2.numerator() *  
                    r1.denominator());  
    // a műveletet a lekérdező/beállító  
    // műveleteken keresztül végezzük el  
    r1.setDenominator(r1.denominator() *  
                       r2.denominator());  
    return r1;  
}
```

# Biztonságos adattípusok megvalósítása

## Láthatóság és hozzáférés

- Általánosan elmondható, hogy az adattípusok mezőit mindig elrejtjük
  - csak megfelelő műveleteken keresztül biztosítjuk a hozzáférést és a módosítást (csak azokra a mezőkre, és csak azt a műveletet, amelyre kívül szükség lehet)
  - amennyiben a típus rendelkezik segédművelettel, amely csak az értékek kezelésében játszik szerepet, azt is elrejtjük
  - ha a konstruktort elrejtjük, akkor nem lehet példányosítani a típust, azaz nem tudunk belőle változót létrehozni (néha hasznos, ha a típust direkt nem akarjuk példányosítani)
  - ez nem vonatkozik a rekordokra (művelet nélküli típusokra)

# Biztonságos adattípusok megvalósítása

## Láthatósági kulcsszavak

- Az típusok tekintetében a C++ két fajtát különböztet meg
  - alapértelmezetten minden látható: **struct**
  - alapértelmezetten minden rejtett: **class**
  - konvenció szerint a **class** kulcsszót típusokra, a **struct** kulcsszót rekordokra használjuk

- Pl.:

```
class MyType {  
    ... // rejtett tagok  
public:  
    ... // publikus tagok  
private:  
    ... // rejtett tagok  
};
```

# Biztonságos adattípusok megvalósítása

## Példa

*Feladat:* Valósítsuk meg a racionális számok típusát úgy, hogy mindig egyszerűsítsük a tárolt értéket.

- minden értékbeállítás után le kell futtatni az egyszerűsítést, amit euklideszi algoritmussal valósítunk meg
  - ez egy segédművelet, ezért szintén elrejtjük
  - cím szerint adjuk át az értékeket, és adja vissza azok egyszerűsített értékét
- mivel az algoritmus csak pozitív számokra alkalmazható, módosítanunk kell a szerkezetén (ami a megjelenés miatt is fontos)
  - a nevezőnek és számlálónak tároljuk csak az abszolút értékét (unsigned int), és külön tároljuk az előjelet

# Biztonságos adattípusok megvalósítása

## Példa

*Megoldás:*

...

```
Rational::Rational(int num, int denom) {  
    _sign = num * denom < 0 ? -1 : 1;  
    // az előjelet megfelelően állítjuk be  
    _numerator = abs(num);  
    // csak az abszolút értékeket tároljuk  
    _denominator = denom != 0 ? abs(denom) : 1;  
    // nem engedélyezzük a 0 hányadost  
  
    simplify(_numerator, _denominator);  
}
```

...



# Biztonságos adattípusok megvalósítása

## Példa

*Megoldás:*

```
// racionális szám típusa:
class Rational {
private: // a mezők és a segédműveletek rejtettek
    int _sign;
        // az előjelet külön el kell mentenünk
    unsigned int _numerator;
        // csak az abszolút értéket tároljuk el
    unsigned int _denominator;

    void simplify(unsigned int& a,
                  unsigned int& b);
        // egyszerűsítés
    ...
}
```

# Biztonságos adattípusok megvalósítása

## Példa

*Megoldás:*

```
void Rational::simplify(int& a, int& b) {
    int c = a, d = b;
    // euklideszi algoritmus futtatása
    while (c != d)
    {
        if (c > d)
            c -= d;
        else
            d -= c;
    }
    a /= c;
    b /= d;
}
```

# Biztonságos adattípusok megvalósítása

## Barát műveletek

- Néha kényelmetlen, hogy a típushoz tartozó külső metódusok (pl. operátorok) nem láthatják a rejtett mezőket, holott igazából nem a külvilághoz tartoznak
- A típusban lehetőségünk van megadni *barát* (*friend*) műveletek halmazát, vagyis olyan külső műveleteket, amelyek láthatják a rejtett értékeket is
  - a műveletek szintaxisát a típusban kell leírni a **friend** kulcsszó mellett (ez nem számít deklarációnak, csak egy jelzésnek)
  - az azonos szintaxissal létrehozott külső műveletek láthatják a rejtett értékeket

# Biztonságos adattípusok megvalósítása

## Barát műveletek

- Pl.:

```
struct MyType {  
private:  
    int _value; // rejtett érték  
  
    friend void AddValue(MyType, int);  
        // barát művelet jelzése  
};  
...  
void AddValue(MyType m, int v){  
    // barát művelet definíciója  
    m._value += v;  
    // módosíthatjuk a rejtett értéket  
}
```

# Biztonságos adattípusok megvalósítása

## Példa

*Feladat:* Valósítsuk meg a téglalapok típusát úgy, hogy a kiíró operátorokat barát műveletként illesztjük.

*Megoldás:*

```
class Rational {  
    ...  
  
    friend ostream& operator>>(ostream& s,  
                                Rectangle& rec);  
    // barát műveletek  
    friend ostream& operator<<(ostream& s,  
                                Rectangle rec);  
    ...  
};
```

# Biztonságos adattípusok megvalósítása

## Példa

*Megoldás:*

```
istream& operator>>(istream& s, Rectangle& rec) {  
    // beolvasó operátor  
    double w, h;  
    s >> w >> h;  
  
    rec._width = w < 0 ? 0 : w;  
    // közvetlenül elérhetjük a mezőket, mivel  
    // barát műveletben vagyunk  
    rec._height = h < 0 ? 0 : h;  
  
    return s;  
}  
...
```

# Biztonságos adattípusok megvalósítása

## Konstansok

- A programjainkban sokszor használunk *konstansok*at, de ezek rendszerint név nélküli konstansok, amikre pusztán az értékük segítségével hivatkozunk
- Elnevezett konstansokat a **const** kulcsszóval tudunk létrehozni, ekkor a névhez rögzített értéket rendelhetünk, amely a későbbiekben nem változtatható, pl.:  

```
const double pi = 3.14159265358979323846;
```
- Lehetőségünk van konstansokat változónak, illetve változót konstansnak értékül adni, pl.:  

```
int u; u = 5;  
const int v = u; // v megkapja u értékét, azaz 5-t  
int w = v; w++; // w már változó
```

# Biztonságos adattípusok megvalósítása

## Konstansok

- Konstansokat alprogramoknál a paraméterátadásban, vagy a visszatérési értékben is használhatunk
  - konstans paraméter esetén nem módosíthatjuk a paraméterben kapott értéket
  - konstans visszatérési érték esetén az érték nem módosítható
  - pl.:

```
const int SomeFunction(const float param){ ... }
```
- Természetesen a saját típusainkból is létrehozhatunk konstansokat, pl.: `const MyType m;`
  - a konstruktor művelet beállítja a kezdőértékeket, de ezt követően nem módosíthatunk semmin



# Biztonságos adattípusok megvalósítása

## Konstans referenciák

- Nem csupán egyszerű konstansokat, de *konstans referenciákat* is létrehozhatunk
  - pl.:  
`MyType m; // változó`  
`const MyType& n = m; // konstans referencia`
  - ekkor nem jön létre új , de nem módosíthatunk az értékeken a későbbiek során a referencián keresztül (a változón keresztül igen)
  - ha paraméterátadásban használjuk, akkor elérjük, hogy csak bemenő irányú legyen a paraméter, ugyanakkor ne másolódjon a memóriában, ami *hatékonyabb, ugyanakkor biztonságos programműködést* tesz lehetővé

# Biztonságos adattípusok megvalósítása

## Konstans metódusok

- Amennyiben egy típuspéldányt konstansnak (vagy konstans referenciának) veszünk, csak olyan metódusai futtathatóak, amelyek nem módosítják az objektum attribútumait, ezeket nevezzük *konstans metódusoknak*
  - konstans metódusban nem módosíthatóak a mezők (ezt fordítási időben ellenőrzi)
  - a kódban a **const** kulcsszóval jelöljük a metódust:

```
class <típusnév>{  
    <típus> <metódusnév>(<paraméterek>) const {  
        <nem módosító műveletek>  
    }  
    ...  
};
```

# Biztonságos adattípusok megvalósítása

## Konstans metódusok

- Pl.:

```
class MyType {
    private:
        int _value;
    public:
        int getValue() const { // konstans művelet
            return _value;
        } // nem módosít semmilyen értéket
        void setValue(int v) { ... }
};
```

```
const MyType mt; // konstans objektum
cout << mt.getValue(); // megengedett
mt.setValue(10); // hiba, nem konstans művelet
```

# Biztonságos adattípusok megvalósítása

## Strukturált programozás

---

- A procedurális programozásban *felülről lefelé* tervezést követünk
  - a feladatot részfeladatokra bontjuk, ahhoz megoldó alprogramokat rendelünk
  - az adatok lehetnek globálisak, vagy lokálisak
  - hátránya, hogy:
    - a feladat módosítása könnyen újratervezést, illetve jelentős módosításokat igényel
    - ha az adatok felépítése változik, minden pontban változtatnunk kell a használatukon
    - a programrészek csak kis mértékben hordozhatóak

# Biztonságos adattípusok megvalósítása

## Strukturált programozás

- Strukturált programokban a program felépítését *alulról felfelé tervezéssel* végezzük, azaz azonosítjuk a feladatban részt vevő (összetett) *entitás*okat (változókat, konstansokat), és azok együttműködésével oldjuk meg a feladatot
  - az entitásokat adattípusok segítségével képezzük, majd azokból hozzuk létre a példányokat
  - az adattípusok megvalósításához korlátozzuk a hozzáférést, így a belső módosítások nem befolyásolják a felhasználást módját
  - előnye, hogy a feladat módosítása nem befolyásolja a program szerkezetét, a programegységek pedig könnyebben hordozhatóak

# Biztonságos adattípusok megvalósítása

## Példa

*Feladat:* Készítsünk grafikus felületű alkalmazást, amely a képernyő közepére kirajzol egy téglalapot, amelynek színét kattintással tudjuk változtatni.

- ha a procedurális megközelítést választjuk:
  - a feladat egy téglalap kirajzolása (alprogrammal), a színek (lokális változók) változtatása
- ha a strukturált megközelítést választjuk:
  - meg kell valósítanunk a téglalap típusát, ami a képernyő egy adott pontjába kirajzolható
  - tárolja a pozícióját, méreteit, színét
  - a színt akár külön típusként is megvalósíthatjuk

# Biztonságos adattípusok megvalósítása

## Példa

*Feladat:* Készítsünk grafikus felületű alkalmazást, amelyben a képernyő tetszőleges pontján el tudunk helyezni egy új téglalapot (bal kattintással), majd annak színét tudjuk módosítani (jobb kattintással), amelyiket elhelyezkedik az egér.

- ha a procedurális megközelítést választjuk, akkor:
  - az előző megoldást teljesen át kell dolgoznunk
  - be kell vezetni a pozíció kezelést, tárolást, keresést
- ha a strukturált megközelítést választjuk, akkor:
  - könnyen kiegészíthetjük a téglalap típusát egy tartalmazási művelettel, és minden más adott
  - egy téglalap helyett több téglalapot kezelünk