

# **Objektumorientált programozás**



## **Objektumalapú programozás a C++ programozási nyelvben**

A programozó által definiált típuskonverzió


**Darvay Zsolt**

# Túlterhelés, konverzió és alosztályok



- 14. Operátorok túlterhelése
- 15. A programozó által definiált típuskonverzió
- 16. Az objektumorientált programozási  
módszer

## 15. A programozó által definiált típuskonverzió



`operator int();`

# Áttekintés



- ▶ 15.1. A típuskonverzió megvalósításának körülményei
- ▶ 15.2. Implicit típuskonverzió
- ▶ 15.3. Explicit típuskonverzió

# 15.1. A típuskonverzió megvalósításának körülményei

- ▶ A rendszer a következő esetekben végez automatikus módon konverziót:
  - ▶ ha egy operátor különböző típusú operandusokra vonatkozik;
  - ▶ ha egy függvény aktuális paraméterének típusa különbözik a neki megfelelő formális paraméter típusától;
  - ▶ ha egy függvény által visszaadott érték típusa (a fejlécben szereplő típus) különbözik a return utasításbeli kifejezés típusától.

# Különböző típusú operandusok



- ▶ Ha egy operátor különböző típusú operandusokra vonatkozik, akkor:
  - ▶ értékadó operátor esetén a jobb oldali kifejezést a bal oldali típusára konvertálja a rendszer;
  - ▶ ellenkező esetben az alapértelmezett konverzió szabályait használjuk.

# Különböző típusú aktuális és formális paraméter



- ▶ Ha egy függvény aktuális paraméterének típusa különbözik a neki megfelelő formális paraméter típusától, akkor az aktuális paramétert a formális paraméter típusára konvertálja a rendszer.

# Különböző típusú visszaadott érték és *return*-beli kifejezés

- ▶ Ha egy függvény által visszaadott érték típusa (a fejlécben szereplő típus) különbözik a *return* utasításbeli kifejezés típusától, akkor a *return* utasításban megadott kifejezést a fejlécben szereplő típusra konvertálja a rendszer.



## 15.2. Implicit típuskonverzió



- ▶ Példa a tort osztályra
- ▶ A konverzió megvalósítása konstruktorral
  - ▶ Az explicit minősítő
  - ▶ A tort osztály továbbfejlesztése
- ▶ Osztályok közti konverzió konstruktorral

# Példa a tort osztályra



```
#include <iostream>
using namespace std;
int Inko(int a, int b); // legnagyobb közös
                        // osztó

class tort {
    // ...
}
```

# Az osztálydeklaráció

```
class tort {  
    int sz; // számláló  
    int n; // nevező  
public:  
    tort(int sz1, int n1);  
    tort& operator ~();           //egyszerűsítés  
    tort operator *(tort r);  
    void kiir();  
};
```

# A konstruktor



```
inline tort::tort(int sz1, int n1)
{
    sz = sz1;
    n = n1;
}
```

- ▶ A formális paraméterek nincsenek alapértelmezett értékkel ellátva.

# Az egyszerűsítés

```
tort& tort::operator ~()  
{  
    int d = Inko(sz, n);  
    sz /= d;  
    n /= d;  
    return *this;  
}
```

# A szorzás



```
inline tort tort::operator *(tort r)
{
    return ~tort(sz*r.sz, n*r.n);
}
```

egyszerűsítés

- ▶ A formális paraméter nem referenciaként van megadva.

# A kiírás



```
void tort::kiir()
{
    if (n)
        cout << sz << " / " << n << endl;
    else
        cerr << "hibas tort";
}
```

# A legnagyobb közös osztó

```
int Inko(int a, int b) {           // nem tagfüggvény!  
    while (b) {  
        int r = a % b;  
        a = b;  
        b = r;  
    }  
    return a;  
}
```



# A fő függvény

```
int main() {  
    tort x(21, 10);  
    tort y(5, 9);  
    tort z(0, 1);    // nincs alapértelmezett konstruktor  
    tort w(6, 1);  
    z = x * y;  
    z.kiir();        // 7 / 6  
    z = x * w;        // z = x * 6; hibás lenne  
    z.kiir();        // 63 / 5  
}
```

# A konverzió megvalósítása konstruktorral

- ▶ Az előző példa esetén a `z = x * 6;` utasítás egy fordítási hibát okoz.
- ▶ A hibát a következő két módon lehetne kiküszöbölni:
  - ▶ egy olyan `*` operátort definiálunk, amelynek az első operandusa egy tört, a második pedig egy egész szám.
  - ▶ megvalósítjuk `int` típusról `tort` típusra a konverziót. Ezt egy megfelelő konstruktorral tehetjük meg.

# A konverziót lehetővé tevő konstruktor



- ▶ A konstruktor formális paraméterének típusa meg kell egyezzen azzal a típussal, amelyről szeretnénk konvertálni. Az eredményként kapott típus az illető osztály által meghatározott típus lesz.
- ▶ Ha egy szabványos típusról egy absztrakt adattípusra szeretnénk konvertálni, akkor ezt a módszert kell használnunk.
- ▶ Ha a konstruktor még más formális paraméterekkel is rendelkezik, akkor ezek inicializálva kell legyenek.

# Lehetséges konstruktorok a `tort` osztály esetén

- ▶ A konverziót `int`-ről `tort`-re a következő három konstruktor közül az egyik valósíthatja meg.

`tort(int sz1, int n1 = 1);`

`tort(int sz1 = 0, int n1 = 1);`

`tort(int sz1);`

- ▶ A fenti konstruktorok közül egyszerre csak egy lehet jelen.

# Az explicit minősítő



- ▶ Ha azt szeretnénk, hogy egy konstruktor semmiképpen ne valósíthasson meg típuskonverziót, akkor az explicit minősítőt kell használni.
- ▶ Például abban az esetben, ha a tort osztály konstruktorát úgy deklaráljuk, hogy a formális paraméterek kezdeti értékkel vannak ellátva, és azt szeretnénk, hogy ez a konstruktor ne végezhesen típuskonverziót, akkor használhatjuk az explicit minősítőt.

# Példa explicit minősítőre

```
class tort {  
    // ...  
    explicit tort(int sz1 = 0, int n1 = 1);  
    // ...  
};
```

- ▶ Ebben az esetben a konstruktor nem végezhet típuskonverziót. A továbbiakban feltételezzük, hogy ezt a konstruktort használjuk, de az explicit minősítő nélkül.

# A tort osztály módosítása

```
class tort {  
    int sz; // számláló  
    int n; // nevező  
public:  
    tort(int sz1 = 0, int n1 = 1);  
    tort& operator ~();           //egyszerűsítés  
    tort operator *(tort r);  
    void kiir();  
};
```

# A fő függvény

```
int main() {  
    tort x(21, 10);  
    tort z(0, 1);  
    z = x * 6; // rendben: konverzió  
    //z = 6 * x; // hiba  
    z.kiir(); // 63 / 5  
}
```



# A fő függvény más formában

```
int main() {  
    tort x(21, 10);  
    tort z(0, 1);  
    z = x.operator*(6);    // ugyanaz, mint z = x * 6;  
    //z = 6.operator*(x); // ugyanaz, mint z = 6 * x;  
    // Ez hibát okoz, mivel a 6 nem objektum.  
    z.kiir();// 63 / 5  
}
```

# A tort osztály továbbfejlesztése



- ▶ A hiba kiküszöbölésére két lehetőség van.
- ▶ Barát függvénnnyel:
  - ▶ a \* operátort barát függvényként adjuk meg.
- ▶ Barát függvény nélkül:
  - ▶ a \* operátor nem lesz tagfüggvény, de egy „szoroz” nevű tagfüggvényt hív meg.

# A \* operátor deklarációja barát függvényként

```
class tort {  
    int sz; // számláló  
    int n; // nevező  
public:  
    tort(int sz1 = 0, int n1 = 1);  
    tort& operator ~(); //egyszerűsítés  
    friend tort operator *(tort p, tort q);  
    void kiir();  
};
```

# A \* operátor

```
tort operator *(tort p, tort q)
```

```
{
```

```
    return ~tort(p.sz*q.sz, p.n*q.n);
```

```
}
```

- ▶ Nem tagfüggvénye a **tort** osztálynak.

# A fő függvény

```
int main() {  
    tort x(21, 10);  
    tort z(0, 1);  
    z = x * 6;           // operator *(x, 6);  
    z.kiir();           // 63 / 5  
    z = 6 * x;          // operator *(6, x);  
    z.kiir();           // 63 / 5  
}
```

# Barát függvény nélkül

```
class tort {  
    int sz; // számláló  
    int n; // nevező  
public:  
    tort(int sz1 = 0, int n1 = 1);  
    tort& operator ~(); //egyszerűsítés  
    tort szoroz(tort r);  
    void kiir();  
};
```

# A „szoroz” tagfüggvény

```
inline tort tort::szoroz(tort r)
{
    return ~tort(sz*r.sz, n*r.n);
}
```

# A \* operátor



```
tort operator *(tort p, tort q)
{
    return p.szoroz(q);
}
```



# A fő függvény

```
int main() {  
    tort x(21, 10);  
    tort z(0, 1);  
    z = x * 6;           // operator *(x, 6);  
    z.kiir();           // 63 / 5  
    z = 6 * x;          // operator *(6, x);  
    z.kiir();           // 63 / 5  
}
```

# Osztályok közti konverzió konstruktorral



- ▶ Az ismertetett módszer akkor is alkalmazható, ha egy absztrakt adattípusról egy másik absztrakt adattípusra szeretnénk konvertálni.
- ▶ A továbbiakban egy olyan példát ismertetünk, amelynek a keretében egy bizonyos hosszúság mérését **inch**ben (line, inch, foot, yard), illetve **méter**ben (mm, cm, dm, m) valósíthatjuk meg. A konverzió átalakítást fog végezni az egyik mértékrendszerrel a másikkra.

# Átalakítás más mértékrendszerre

- ▶ A következő szabályok szerint történik:

<b><i>1 line</i></b>		=	2.54 mm
<b><i>1 inch</i></b>	= 10 lines	=	2.54 cm
<b><i>1 foot</i></b>	= 12 inches	=	30.48 cm
<b><i>1 yard</i></b>	= 3 feet	=	91.44 cm

# A hossz\_inch osztály

```
class hossz_inch {  
    int line;  
    int inch;  
    int foot;  
    int yard;  
public:  
    hossz_inch(int l = 0, int i = 0, int f = 0, int y = 0);  
    int line_hossz();  
    void kiir();  
};
```

# A hossz\_m osztály

```
class hossz_m {  
    int mm, cm, dm, m;  
    double marad;    // maradék, a konverzió után  
public:  
    hossz_m(int mm_1 = 0, int cm_1 = 0,  
            int dm_1 = 0, int m_1 = 0);  
    hossz_m(hossz_inch h);           // konverzió  
    void kiir();  
};
```

# A hossz\_inch konstruktora



```
hossz_inch::hossz_inch(int l, int i, int f, int y)
{
    line = l;
    inch = i;
    foot = f;
    yard = y;
}
```

# A line\_hossz tagfüggvény

```
int hossz_inch::line_hossz()  
{  
    return line + 10 * inch + 120 * foot + 360 * yard;  
}
```

- ▶ A teljes hosszúságot **line**ban fejezi ki.

# Kiírás **inch**ben



```
void hossz_inch::kiir()  
{  
    if (line) cout << line << " line ";  
    if (inch) cout << inch << " inch ";  
    if (foot) cout << foot << " foot ";  
    if (yard) cout << yard << " yard";  
    cout << endl;  
}
```



# A hossz\_m hagyományos konstruktora



```
hossz_m::hossz_m(int mm_1, int cm_1,  
int dm_1, int m_1)  
{  
    mm = mm_1;  
    cm = cm_1;  
    dm = dm_1;  
    m = m_1;  
    marad = 0;  
}
```

# A konverzió

```
hossz_m::hossz_m(hossz_inch h)
{
    marad = 2.54 * h.line_hossz();
    mm = static_cast<int>(marad);
    marad -= mm;
    m = mm / 1000; mm %= 1000;
    dm = mm / 100; mm %= 100;
    cm = mm / 10; mm %= 10;
}
```

# Kiírás méterben

```
void hossz_m::kiir()
{
    if (m) cout << m << " m ";
    if (dm) cout << dm << " dm ";
    if (cm) cout << cm << " cm ";
    if (mm + marad)
        cout << mm + marad << " mm ";
    cout << endl;
}
```

# A fő függvény

```
int main() {  
    hossz_inch y(6, 3, 2, 5);  
    y.kiir();  
    cout << y.line_hossz() << " line\n";  
    cout << 2.54 * y.line_hossz() << " mm\n";  
    hossz_m x;  
    x = y;  
    x.kiir();  
}
```

# A kimenet



6 line 3 inch 2 foot 5 yard

2076 line

5273.04 mm

5 m 2 dm 7 cm 3.04 mm

## 15.3. Explicit típuskonverzió



- ▶ Az implicit típuskonverziót konstruktorral valósítottuk meg.
- ▶ Például akkor használhatjuk, ha egy szabványos típusról egy absztrakt adattípusra szeretnénk konvertálni.
- ▶ A fordított irányú konverziót (absztrakt adattípusról standard típusra) nem valósíthatjuk meg konstruktorral.
- ▶ Ezért van szükség az explicit típuskonverzióra.

# Az explicit típuskonverzió megvalósítása

- ▶ Az explicit típuskonverziót a típusmódosító operátor túlterhelésével valósíthatjuk meg.
- ▶ Egy absztrakt adattípusról (osztályról) tetszőleges típusra konvertálhatunk (szabványos típusra, illetve más absztrakt adattípusra is).
- ▶ Legyen az eredeti osztály neve „**oszt**”, és az a típus, amire konvertálunk „**cél\_típus**”.

# A típusmódosító operátor túlterhelése

```
class oszt {  
    // ...  
public:  
    // ...  
    operator cél_típus(); // deklaráció  
    // ...  
};
```



# A típusmódosító operátor definiálása

```
oszt::operator cél_típus()
{
    // ...
    return kifejezés;    // a kifejezés típusa
}                        // cél_típus kell legyen.
```

- ▶ Sem a deklaráció, sem a definíció nem tartalmaz visszatérített értéket, mégis a **return** utasítással meg kell adni azt a kifejezést, amit a konverzióval kapunk.

# Példa explicit típuskonverzióra



```
#include <iostream>
```

```
using namespace std;
```

```
int Inko(int a, int b);
```

```
// legnagyobb közös osztó
```

# A tort osztály

```
class tort {  
    int sz;    // számláló  
    int n;    // nevező  
public:  
    tort(int sz1, int n1);  
    tort& operator ~();    //egyszerűsítés  
    tort operator *(tort r);  
    operator double();  
    void kiir();  
};
```

# A típusmódosító operátor



```
tort::operator double()  
{  
    return static_cast<double>(sz) / n;  
}
```

# Két függvény a kiírásra

```
void kiiras(tort t)// nem tagfüggvény
{
    t.kiir();
}
```

```
void kiiras(double x)
{
    cout << x << endl;
}
```

# A fő függvény

```
int main() {  
    tort x(2, 7);  
    kiiras(x * 2.1);  
    cout << typeid(x * 2.1).name() << endl;  
}
```

Kimenet:

0.6  
double

- ▶ A `kiiras(2.1 * x);` utasítás is ugyanazt az eredményt szolgáltatja.
- ▶ A kifejezés mindkét esetben `double` típusú.

# Implicit típuskonverzióval



- ▶ A tort osztály konstruktorát úgy deklaráltuk, hogy nem tartalmazott alapértelmezett értékeket.
- ▶ Ha ezt tennénk, akkor a fordító hibát jelzne, mivel az  $x * 2.1$  kifejezést kétféle képpen értékelhetné ki.

# Az $x * 2.1$ kifejezés kiértékelése

- ▶ Két lehetőség:
  - ▶ Az  $x$ -et `double` típusra konvertálja a rendszer, és a kifejezés `double` típusú lesz.
  - ▶ A  $2.1$ -et információvesztéssel `int` típusra konvertálja a rendszer (a kapott érték `2` lesz), és a konstruktorral egy törtet hoz létre. Ez által a kifejezés `tort` típusú lesz.
- ▶ Ha a típusmódosító operátort nem terheljük túl, de a konstruktornak vannak alapértelmezett értékei, akkor eredményként egy törtet kapunk.



# A tort osztály



```
class tort {  
    int sz;      // számláló  
    int n;      // nevező  
public:  
    tort(int sz1 = 0, int n1 = 1);  
    tort& operator ~();      //egyszerűsítés  
    tort operator *(tort r);  
    // operator double();  
    void kiir();  
};
```

# A fő függvény

```
int main() {  
    tort x(2, 7);  
    kiiras(x * 2.1);  
    //kiiras(2.1 * x);  
    cout << typeid(x * 2.1).name() << endl;  
}
```

Kimenet:

4 / 7

class tort

- ▶ A `kiiras(2.1 * x);` utasítás fordítási hibát okoz.

# Osztályok közti konverzió a típusmódosító operátorral

- ▶ Az osztályok közti konverziót a típusmódosító operátor túlterhelésével is végezhetjük.
- ▶ Ebben az esetben az implicit típuskonverziót megvalósító konstruktor nem lehet jelen.
- ▶ A továbbiakban a hosszúság meghatározására vonatkozó példát módosítjuk.
- ▶ Két osztály: `hossz_inch`, `hossz_m`.

# A hossz\_m osztály

```
class hossz_m {  
    int mm;    int cm;    int dm;    int m;  
    double marad;    // maradék, a konverzió után  
public:  
    hossz_m(int mm_1 = 0, int cm_1 = 0,  
            int dm_1 = 0, int m_1 = 0, double marad1 = 0.0);  
    void kiir();  
    // hossz_m(hossz_inch h);  
};
```

# A hossz\_m osztály konstruktora

```
hossz_m::hossz_m(int mm_1, int cm_1,  
                 int dm_1, int m_1, double marad1)  
{  
    mm = mm_1;  
    cm = cm_1;  
    dm = dm_1;  
    m = m_1;  
    marad = marad1;  
}
```

# A hossz\_inch osztály

```
class hossz_inch {  
    int line;    int inch;    int foot;    int yard;  
public:  
    hossz_inch(int l = 0, int i = 0, int f = 0, int y = 0);  
    int line_hossz();  
    void kiir();  
    operator hossz_m();  
};
```

# A típusmódosító operátor

```
hossz_inch::operator hossz_m() {  
    double marad1 = 2.54 * line_hossz();  
    int mm1 = static_cast<int>(marad1);  
    marad1 -= mm1;  
    int m1 = mm1 / 1000; mm1 %= 1000;  
    int dm1 = mm1 / 100; mm1 %= 100;  
    int cm1 = mm1 / 10; mm1 %= 10;  
    return hossz_m(mm1, cm1, dm1, m1, marad1);  
}
```

# A fő függvény

```
int main() {                                     // ugyanaz
    hossz_inch y(6, 3, 2, 5);
    y.kiir();
    cout << y.line_hossz() << " line\n";
    cout << 2.54 * y.line_hossz() << " mm\n";
    hossz_m x;
    x = y;
    x.kiir();
}
```



# A kimenet



6 line 3 inch 2 foot 5 yard

2076 line

5273.04 mm

5 m 2 dm 7 cm 3.04 mm

▶ A kimenet is ugyanaz.