C++ programozási nyelv
Konstruktorok-destruktorok

Nyugat-Magyarországi Egyetem
Faipari Mérnöki Kar
Informatikai Intézet

Soós Sándor
2004. szeptember

Tartalomjegyzék

Logikai típus

- **Új típus a C++ -ban: bool**
  - önálló típus, nem kell egész értékekkel kombinálni
  - Két lehetséges érték: **true, false**
  - Példa:

```
bool a, b;
a = true;
b = false;
int i = 10;
a = 10 == i;  // ==> def. szerint: a = (10 == i);  a = true
```

  - a bool és int típus között létezik konverzió oda-vissza:

```
int (true) = 1,    int(false) = 0
bool(0) = false,   bool(nem 0) = true
```

  - Például:

```
int i = true;      // i = 1
bool b = 8;        // b = true
```

A new és a delete operátorok

- **A new operátor:**
  - memória foglalás
  - inicializálás (objektumok esetén konstruktor meghívása)
- **A delete operátor:**
  - destruktor meghívása
  - memória felszabadítás
  - vannak olyan nyelvek (pl. a Java ilyen), amelyekben működik az ún. **garbage collection** (hulladékgyűjtés) mechanizmus. Ez automatikusan felszabadítja azokat a memória tartományokat, amelyekre már semmi nem "mutat". A C++ -ban ilyen nincs, nekünk kell gondoskodnunk arról, hogy amit new-val létrehoztunk, azt meg is semmisítsük.

Dinamikus tömbök kezelése

- **A new és delete használata tömbök esetében:**
- **Például:**

```
Nev = new char[30]; // létrehozunk egy 30 elemű tömböt

delete [] Nev;      // felszámoljuk a tömböt
```

- **Mi történne, ha elhagyjuk a [] jeleket a delete-ből?**
  - Nagy valószínűséggel elszállna a program, mert a tömböket másképpen adminisztrálja a memóriakezelő rendszer, mint a skalár változókat. Felszabadításkor meg kell mondani, hogyez a pointer tömbre, vagy skalárra mutat.

A class fogalma

- **A legtöbb nyelvben a class kulcsszó jelzi az osztályt. A C++ -ban megmaradt a struct is a C-vel való kompatibilitás kedvéért.**
- **Mi a különbség a class és a struct között?**
  - A **struct** esetében minden tagra a **public** hozzáférés az alapértelmezés
  - A **class** esetében viszont a **private**
  - Ettől eltekintve a két fogalom azonos. Objektum orientált programozás esetén általában a **class**-t használjuk.
  - Például:

```
struct osztaly {          // ugyanezt class-szal is írhatnám
  private:    int x;      // ugyanazt jelentené
  public:     void SetX( int ax );
              int GetX();
}
```

A konstruktor fogalma C++ -ban

- **A konstruktor az osztály, illetve az objektum inicializáló eljárása.**
- **Hogyan ismeri meg a rendszer a konstruktort?**
  - neve megegyezik a struktúra nevével
  - nincs visszatérő értéke (nem **`void`**!!!)
- **Egy osztálynak több konstruktora lehet, amelyek a szignaturájukban eltérnek egymástól.**
- **Szignatura: a függvény ill. a konstruktor neve + paramétereinek száma, típusa. A visszatérési érték nem része a szignaturának! Miért?**

Egy példa konstruktorokra

```
class Valami{
   int x;
   Valami(); // 1. konstruktor deklarációja
   Valami( int ax );  // 2. konstruktor dekl.
   };

// konstruktorok definíciója
Valami::Valami()
   { x = 0; }

Valami::Valami( int ax )
   { x = ax; }
```

Mikor fut le a konstruktor?

- A programozó soha nem hívja meg közvetlenül a konstruktort!
- Mikor fut le a konstruktor automatikusan?
  - amikor a program végrehajtása eljut egy objektum típusú változó deklarációjához (automatikus változó)

```
void main()
{
  Valami v;
```

  - a program a new operátor segítségével létrehoz egy objektumot (dinamikus változó)

```
Valami *vp;
vp = new Valami;
```

  - … vannak más esetek is, de ezekkel most nem foglalkozunk
- Melyik konstruktor fut le ilyenkor, ha több van?
  - a szignatura alapján egyértelműen csak egy jöhet szóba

A destruktor fogalma C++ -ban

- **A destruktor az osztály, illetve az objektum lezáró megsemmisítő eljárása.**
- **Hogyan ismeri meg a rendszer a destruktort?**
  - neve megegyezik a struktúra nevével, előtte egy tilde (~) karakter.
  - nincs visszatérő értéke (nem **`void`**!!!)
- **Egy osztálynak csak egy destruktora lehet.**

Példa destruktorra

- **Példa:**

```
class Valami{
  int x;
  Valami();                    // konstruktor deklarációja
  ~Valami();                   // destruktor deklarációja
};

Valami::Valami()               // konstruktor definíciója
{
  x = 0;
  printf("Konstruktor\n");     // demonstráció
}

Valami::~Valami()              // destruktor definíciója
{
  printf("Destruktor\n");      // demonstráció
}
```

Mikor fut le a destruktor?

- A programozó soha nem hívja meg közvetlenül a destruktort!
- Mikor fut le a destruktor automatikusan?
  - amikor a program végrehajtása eljut egy objektum típusú változó hatáskörének végére (automatikus változó)

```
void main()
{
  Valami v;
}
```

  - a program a delete operátor segítségével megsemmisít egy objektumot (dinamikus változó)

```
Valami *vp;
vp = new Valami;
delete vp;
```

  - … vannak más esetek is, de ezekkel most nem foglalkozunk

Alapértelmezett konstruktor

- **Mit nevezünk alapértelmezett (default) konstruktornak?**
  - A paraméterek nélküli konstruktort.
  - Ilyen konstruktora minden osztálynak van, ugyanis
    - általában mi definiálunk egyet
    - ha nem definiálunk egyetlen konstruktort sem az osztályban, akkor a fordító létrehoz egy üres, default konstruktort.
    - ha létrehozunk egy vagy több paraméteres (nem default) konstruktort, akkor a fordító nem hoz létre default konstruktort, de tőlünk megköveteli, hogy megírjuk.
  - Hasonlóképpen minden osztálynak van destruktora, ugyanis
    - vagy mi definiáltuk
    - vagy, ha mi nem tettük meg, akkor a fordító létrehoz egy üreset

Hogyan hozhatunk létre objektumokból álló tömböt?

- **Ennek egyetlen feltétele van: az objektumnak legyen default konstruktora, vagy mi definiáljunk, vagy engedjük, hogy a fordító készítsen. Ekkor a következő példa azonnal működik:**

```
void main()
{
Valami vt[3];
printf("----\n");
}
```

A kimenet:
Konstruktor
Konstruktor
Konstruktor

Destruktor
Destruktor
Destruktor

Hogyan hozhatunk létre objektumokból álló dinamikus tömböt?

- **Az előző példa a következőképpen módosul:**

```
void main()
{
Valami *vt;
vt = new Valami[3];
printf("----\n");
delete [ ] vt;
}
```

A kimenet:
Konstruktor
Konstruktor
Konstruktor

Destruktor
Destruktor
Destruktor

A hozzáférési hatáskörök osztályszintűek

- **Mi történik akkor, ha egy objektum metódusa egy másik objektum adataihoz szeretne hozzáférni?**
- **Vizsgáljuk meg a következő példát:**

```
#include<iostream.h>

class Valami{
  public:    void PublicFunc(Valami *Masik);
  private:   int  PrivateAdat;
};

void Valami::PublicFunc(Valami *Masik)
{
  cout << "A sajat PrivateAdat-om: " << PrivateAdat << endl;
  cout <<"Masik PrivateAdat-a: " << Masik->PrivateAdat << endl;
}
```

Tagfüggvény – szabad függvény

- **Mikor használjunk tagfüggvényt és mikor hagyományos "szabad" függvényt?**
- **Hasonló kérdés, hogy ha több osztállyal dolgozunk, akkor egy adott funkciót melyik osztályba tegyünk, illetve mikor legyen szabad függvény?**
  - Példák
  - Végső megoldás: Java - Csak tagfüggvény létezik!

Mikor írunk tagfüggvényt?

- **Az egyes adatmezőket beállító, illetve azok értékét visszaadó függvények mindig metódusok.**
- **A kizárólag adatmezőkkel operáló függvények mindig tagfüggvények.**
- **A két vagy több objektummal operáló függvények lehetnek az egyik osztály tagfüggvényei, míg a másik objektumot paraméterként kapják meg.**

Mikor írjunk szabad függvényt?

- **Nincs olyan eset, amit csak szabad függvényként írhatunk meg. (Kivétel a programot elindító main függvény!)**
- **Ezt bizonyítja a Java példája. Javaban minden függvény valamelyik osztályhoz tartozik, még az indító main függvény is.**
- **Persze ez nem jelenti azt, hogy ne lehetne érdemes időnként szabad függvényt írni!**

Példa: a múlt heti autós példa folytatása

- **Vizsgáljuk meg a következő kérdéseket!**
  - Hány éves az autó?
  - Mennyibe kerül egy út ezzel az autóval?
  - Mennyi az éves súlyadó erre az autóra?
  - Hány liter üzemanyagot használt el eddig az autó?
  - Melyik a legfiatalabb, legöregebb autó a flottában?
  - Ellenőrizzük, hogy csak helyes értékeket lehessen megadni az autóról!
- **Írjunk függvényeket a fenti kérdések megválaszolására! Írjuk meg őket tagfüggvényként és szabad függvényként egyaránt!**