Objektum orientált programozás C++ nyelven 5. Osztályok II.

PEKÁRDY MILÁN – PANNON EGYETEM

PEKARDY@DCS.UNI-PANNON.HU

Speciális tagfüggvények (special member functions)

- Speciális tagfüggvények olyan függvények, amelyek bizonyos körülmények között implicit kerülnek definiálásra

Member function	typical form for class C:
Default constructor	`C::C();`
Destructor	`C::~C();`
Copy constructor	`C::C (const C&);`
Copy assignment	`C& operator= (const C&);`
Move constructor	`C::C (C&&);`
Move assignment	`C& operator= (C&&);`

Alapértelmezett konstruktor (default constructor)

- Az alapértelmezett konstruktor az objektum deklarálásakor hívódik meg, amikor nem adunk meg argumentumot
- Ha az osztály nem tartalmaz egy konstruktort sem, akkor a fordító implicit feltételezi, hogy van alapértelmezett konsturktor

```
class Example {
  public:
    int total;
    void accumulate (int x) { total += x; }
};

int main () {
  Example ex;
  return 0;
}
```

Alapértelmezett konstruktor (default constructor)

- Ha egy osztályban megadunk legalább egy konstruktort tetszőleges számú argumentummal, akkor a fordító már nem feltételezi az alapértelmezett konstruktor meglétét, így ha szükségünk van rá, akkor explicit kell definiálnunk

```
class Example2 {
  public:
    int total;
    Example2 (int initial_value) : total(initial_value) { }
    void accumulate (int x) { total += x; }
};

int main () {
  Example2 ex (100);   // ok: calls constructor
  Example2 ex;         // not valid: no default constructor
  return 0;
}
```

Alapértelmezett konstruktor (default constructor)

```
// classes and default constructors
#include <iostream>
#include <string>
using namespace std;

class Example3 {
    string data;
  public:
    Example3 (const string& str) : data(str) {}
    Example3() {}
    const string& content() const {return data;}
};

int main () {
  Example3 foo;
  Example3 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

Destruktor (destructor)

- A destruktorok a konstruktorok által foglalt erőforrások (pl. dinamikus memória) felszabadításáért felelnek, amikor az objektum élettartama lejár

```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example4 foo;
  Example4 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

Másoló konstruktor (copy constructor)

- Amikor egy objektumnak egy a típusával megegyező másik objektumot adunk át paraméterként, akkor az objektum másoló konstruktora hívódik meg, hogy egy másolat létrejöjjön
- A másoló konstruktor olyan konstruktor, aminek az első paramétere egy referencia egy az objektummal megegyező típusú másik objektumra: `MyClass::MyClass (const MyClass&);`
- Ha egy osztálynak nem adunk meg másoló konstruktort (se mozgató konstruktort, se értékadásokat), akkor a fordító implicit generál. Az implicit másoló konstruktor egyszerűen lemásolja az adattagokat (shallow copy)

```
MyClass::MyClass(const MyClass& x) : a(x.a), b(x.b), c(x.c) {}
```

Másoló konstruktor (copy constructor)

- Az adattagok egyszerű lemásolása (shallow copy) sokszor nem elegendő a megfelelő funkcionalitáshoz (pl. mutatók esetén a mutatót másoljuk nem pedig a mutatott adatot), ilyenkor explicit definiáljuk a másoló konstruktort és magunk gondoskodunk az adattagok megfelelő másolásáról (deep copy)

```
// copy constructor: deep copy
#include <iostream>
#include <string>
using namespace std;

class Example5 {
    string* ptr;
  public:
    Example5 (const string& str) : ptr(new string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) : ptr(new string(x.content())) {}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example5 foo ("Example");
  Example5 bar = foo;

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

Másoló értékadás (copy assignment)

- Az objektumok másolása nem csak konstruktorral lehetséges hanem értékadáskor is:

```
MyClass foo;
MyClass bar (foo);       // object initialization: copy constructor called
MyClass baz = foo;       // object initialization: copy constructor called
foo = bar;               // object already initialized: copy assignment called
```

- Deklaráláskor a konstruktorok hívódnak, értékadáskor az értékadó operátor: `MyClass& operator= (const MyClass&);`
- Másoló értékadáshoz az = operátort terheljük túl (operator overload) úgy, hogy az argumentuma egy érték vagy referencia egy megegyező típusú objektumra, a visszatérési értéke általában egy referencia a *this-re, de ez nem kötelező
- A másoló értékadó operátor implicit létrejön ha az osztálynak nincs explicit definiált másoló vagy mozgató konstruktora, illetve értékadó operátorok

Másoló értékadás (copy assignment)

- Az implicit másoló értékadás is csak egyszerű másolatokat (shallow copy) készít az adattagokról, így a bonyolultabb eseteket célszerű nekünk kezelni explicit felülírt operátorral
- Ebben az esetben figyelnünk kell arra, hogy az operátor bal oldalán lévő objektum már inicializálva lett korábban így a dinamikus adattagjai már be lehetnek állítva, ilyenkor a memória szivárgások (memory leak) elkerülése érdekében kezelnünk kell ezeket az adattagokat a másoló operátorban (pl. törölni a korábban dinamikusan foglalt memóriát)

```
Example5& operator= (const Example5& x) {
  delete ptr;                          // delete currently pointed string
  ptr = new string (x.content());  // allocate space for new string, and copy
  return *this;
}
```

Mozgató konstruktor és értékadás (move constructor and assignment)

- Mozgatás esetén a másoláshoz hasonlóan egy objektum értékét használjuk egy másik objektum értékének beállítására
- A másolással ellentétben a mozgatás során az adatot ténylegesen átmozgatjuk egyik objektumból (source) a másikba (destination): a forrás így elveszti a tartalmat és a cél veszi azt át
- Mozgatás csak akkor történik ha a forrás egy nem nevesített objektum (unnamed object)
- Nem nevesített objektumok olyan objektumok, amelyek természetüknél fogva ideiglenesek és így nem lettek elnevezve, pl. ilyen objektumok a függvények által visszaadott értékek
- Ilyenkor nem szükséges másolni, mert az ideiglenes objektumot úgyse használjuk másra, a tartalma közvetlenül átmozgatható a cél objektumba

Mozgató konstruktor és értékadás (move constructor and assignment)

- A mozgató konstruktor akkor hívódik meg, amikor egy objektumot létrehozáskor inicializálunk egy ideiglenes objektummal.
- A mozgató értékadó operátor pedig akkor aktiválódik, amikor egy már inicializált objektumhoz rendelem hozzá egy ideiglenes objektum értékét

```
MyClass fn();               // function returning a MyClass object
MyClass foo;                // default constructor
MyClass bar = foo;          // copy constructor
MyClass baz = fn();         // move constructor
foo = bar;                  // copy assignment
baz = MyClass();            // move assignment
```

Mozgató konstruktor és értékadás (move constructor and assignment)

- A mozgató konstruktor és értékadó operátor olyan adattagok, amelyek a saját osztályuk típusával megegyező típusú jobb érték (rvalue reference) referenciát kapnak paraméterként:

```
MyClass (MyClass&&);                // move-constructor
MyClass& operator= (MyClass&&);     // move-assignment
```

- A mozgatás művelet akkor hasznos, amikor egy objektum a dinamikus tárhelyét maga kezeli a new és delete operátorok megfelelő használatával. Ilyenkor a másolás és mozgatás két nagyon különböző művelet:
  - Másolás: A-ból B-be történő másolás esetén új memóriát foglalunk B-hez majd az A tartalmát teljes egészében átmásoljuk a B-be,
  - Mozgatás: A-ból B-be történő mozgatás esetén nem foglalunk új memóriát, hanem a már lefoglalt tárhelyet rendeljük hozzá a cél objektumhoz (mutatót másoljuk)

```cpp
// move constructor/assignment
#include <iostream>
#include <string>
using namespace std;

class Example6 {
    string* ptr;
  public:
    Example6 (const string& str) : ptr(new string(str)) {}
    ~Example6 () {delete ptr;}
    // move constructor
    Example6 (Example6&& x) : ptr(x.ptr) {x.ptr=nullptr;}
    // move assignment
    Example6& operator= (Example6&& x) {
      delete ptr;
      ptr = x.ptr;
      x.ptr=nullptr;
      return *this;
    }
    // access content:
    const string& content() const {return *ptr;}
    // addition:
    Example6 operator+(const Example6& rhs) {
      return Example6(content()+rhs.content());
    }
};


int main () {
  Example6 foo ("Exam");
  Example6 bar = Example6("ple");   // move-construction

  foo = foo + bar;                  // move-assignment

  cout << "foo's content: " << foo.content() << '\n';
  return 0;
}
```

Implicit tagok

- Az eddig bemutatott hat speciális tagfüggvény implicit deklarálva lesznek az alábbiaknak megfelelően:

Member function	implicitly defined:	default definition:
Default constructor	if no other constructors	does nothing
Destructor	if no destructor	does nothing
Copy constructor	if no move constructor and no move assignment	copies all members
Copy assignment	if no move constructor and no move assignment	copies all members
Move constructor	if no destructor, no copy constructor and no copy nor move assignment	moves all members
Move assignment	if no destructor, no copy constructor and no copy nor move assignment	moves all members

- Az osztályban explicit megadhatjuk, hogy szeretnénk-e alapértelmezett implicit definíciót a fenti tagokból: a default kulcsszó hatására létrejön, a delete kulcsszó hatására pedig törlődik az implicit definíció:

```
function_declaration = default;
function_declaration = delete;
```

Implicit tagok

```
// default and delete implicit members
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int x, int y) : width(x), height(y) {}
    Rectangle() = default;
    Rectangle (const Rectangle& other) = delete;
    int area() {return width*height;}
};

int main () {
  Rectangle foo;
  Rectangle bar (10,20);

  cout << "bar's area: " << bar.area() << '\n';
  return 0;
}
```

„Barát” függvények (friend functions)

- A private és protected adattagok csak a tartalmazó osztályból vagy a származtatott osztályokból érhetők el alap esetben.
- A friend kulcsszóval deklarált függvények vagy osztályok viszont elérhetik a tartalmazó osztály private és protected adattagjait is
- Ezek a függvények nem tagfüggvényei az osztálynak, csak elérik annak az adattagjait
- Tipikus használat: amikor olyan műveleteket végez egy függvény, ami több különböző osztályon végez egyszerre különféle műveleteket

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

„Barát” osztályok (friend classes)

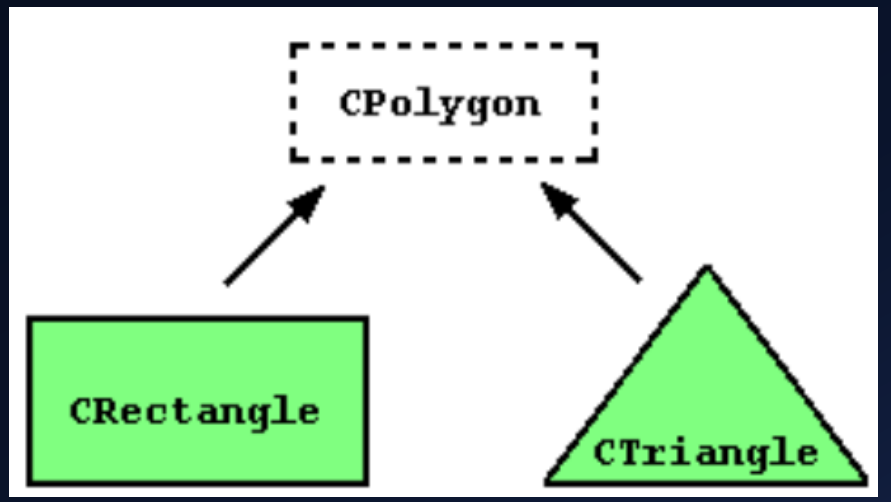
- A „barát” osztály olyan osztály, aminek a tagjai hozzáférnek a private és protected tagjaihoz egy másik osztálynak
- Mivel a példában mindkét osztály használja a másikat, ezért szükséges egy üres deklaráció a program elején
- A „barátság” mindig explicit definiált: a példában a Rectangle a Square „barátja” de fordítva ez nem igaz
- A „barátság” továbbá nem tranzitív: egy barát barátja nem barátja a kiindulási osztálynak

```
// friend class
#include <iostream>
using namespace std;

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

Osztályok közötti öröklés (inheritance)

- Az osztályok kibővíthetők új osztályok létrehozásával, amelyek megtartják a szülő osztály karakterisztikáit. Ezt öröklésnek nevezik: a szülő osztályból (base class) örököltetjük a származtatott osztályt (derived class)
- A származtatott osztály örökölni tudja a szülő tagjait (amelyek elérhetőek a származtatott osztályban), illetve saját új tagokat is deklarálhat

```
class derived_class_name: public base_class_name
{ /*...*/ };
```



Osztályok közötti öröklés (inheritance)

- Az öröklésnél használhatjuk a public, private, protected hozzáférés módosítókat:
  - Amennyiben a szülő tag hozzáférése megengedőbb mint az öröklés, akkor a származtatott osztályban az öröklésnek megfelelő hozzáféréssel fog szerepelni,
  - Amennyiben a szülő tag hozzáférése szigorúbb mint az öröklés, akkor a származtatott osztályban a szülőnek megfelelő hozzáféréssel fog szerepelni

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Osztályok közötti öröklés (inheritance)

- Egy publikusan örökölt osztály örökli a szülő osztály minden tagját, kivéve a következőket:
  - Konstruktorok és destruktor,
  - Értékadó operátorok (operator=)
  - Barátok
  - private tagok
- A konstruktorok és destruktor azonban automatikusan lefut, amikor a származtatott osztály létrejön, illetve törlődik
- Ha máshogy nem specifikáljuk akkor a szülő osztály alapértelmezett konstruktora hívódik, ha másik konstruktort szeretnénk hívni, akkor explicit kell azt megadnunk:

```
derived_constructor_name (parameters) : base_constructor_name (parameters) {...}
```

Osztályok közötti öröklés (inheritance)

```cpp
// constructors and derived classes
#include <iostream>
using namespace std;

class Mother {
  public:
    Mother ()
      { cout << "Mother: no parameters\n"; }
    Mother (int a)
      { cout << "Mother: int parameter\n"; }
};

class Daughter : public Mother {
  public:
    Daughter (int a)
      { cout << "Daughter: int parameter\n\n"; }
};

class Son : public Mother {
  public:
    Son (int a) : Mother (a)
      { cout << "Son: int parameter\n\n"; }
};

int main () {
  Daughter kelly(0);
  Son bud(0);

  return 0;
}
```

„Többes” öröklés (multiple inheritance)

- Egy osztály több másik szülő osztályból is örökölhet
- Vigyázni kell ennek a használatával: mi van ha mindegyik ősosztályban ugyanolyan nevű tagok vannak?

```
// multiple inheritance
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
};

class Output {
  public:
    static void print (int i);
};

void Output::print (int i) {
  cout << i << '\n';
}

class Rectangle: public Polygon, public Output {
  public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height; }
};

class Triangle: public Polygon, public Output {
  public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height/2; }
};

int main () {
  Rectangle rect (4,5);
  Triangle trgl (4,5);
  rect.print (rect.area());
  Triangle::print (trgl.area());
  return 0;
}
```