Objektum orientált programozás C++ nyelven
4. Osztályok

PEKÁRDY MILÁN – PANNON EGYETEM

PEKARDY@DCS.UNI-PANNON.HU

Osztályok I. (class)

- Osztályok az eddigi adat struktúrák kiegészített változata: adattagokon kívül tartalmazhatnak függvényeket is
  - Adattagok = állapot
  - Függvények = viselkedés
- Egy objektum (object) az osztály példányosításával jön létre (hasonlat: az eddigi változók analógiájára az osztály a típus és az objektum a változó)
- Definiálás a `class` vagy `struct` kulcsszóval:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

- class_name: érvényes azonosító, az osztály neve
- members: adattagok vagy függvények
- access_specifiers: az egyes tagok elérhetőségét szabályzó módosítók

Osztályok I.

- hozzáférési szintek (access specifiers)
  - private: ezek a tagok csak az osztály más tagjaiból érhetők el vagy az osztály „barátaiból” (friend classes)
  - protected: ezek a tagok elérhetők az osztály más tagjaiból, az osztály „barátaiból” valamint a származtatott osztályokból
  - public: ezek a tagok bárhonnan elérhetők ahol az objektum látható
- alapértelmezetten minden tag private eléréssel rendelkezik

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

- osztály: Rectangle
- objektum: rect
- tagok (members):
  - adattagok: width, height
  - függvények: set_values, area (csak deklaráció)
- tagok elérése:

```
rect.set_values (3,4);
myarea = rect.area();
```

Osztályok I.

- a függvényeket csak deklaráltuk, a definíciókat is meg kell adni:
  - osztályon belül is meg lehet adni egyből a deklarálásnál (ilyenkor a fv. automatikusan inline)
  - osztályon kívüli definíciónál: class_name::method_name a szintaktika (ugyanabba a fájlba vagy külön fájlba is lehet)
- a scope (::) operátor használatával ugyanúgy elérhetjük a tagokat mintha az osztályon belül lennénk

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

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

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

Osztályok I.

- az osztályok az objektum-orientált programozás alapját adják
- egy osztályból tetszőleges példányt hozhatunk létre, minden példánynak megvannak a saját adattagjai és a metódusaik ezeken a saját adattagokon dolgoznak

```
// example: one class, two objects
#include <iostream>
using namespace std;

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

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Osztályok I. - konstruktorok

- a konstruktorok segítségével értéket adhatunk az osztály adattagjainak az osztály használata előtt
- a konstruktor automatikusan meghívódik az objektum létrehozásakor
- a konstruktoroknak nincs visszatérési értékük, a nevük megegyezik az osztály nevével
- a konstruktorokat explicit nem hívhatjuk, az objektum létrehozásakor egyszer hívódik meg

```
// example: class constructor
#include <iostream>
using namespace std;

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

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Osztályok I. – konstruktorok túlterhelése

- a függvényekhez hasonlóan a konstruktorok is túlterhelhetők különböző paraméterekkel: különböző számú és/vagy típusú paraméterek
- alapértelmezett konstruktor: nincs paramétere, akkor hívódik meg, amikor az objektumot deklaráljuk, de nem inicializáljuk argumentumokkal

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

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

Rectangle::Rectangle () {
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Osztályok I. – uniform inicializáció

- eddigi példákban a konstruktorokat úgy hívtuk, hogy a paramétereket zárójelek között adtuk meg (functional form)
- egy paraméterrel rendelkező konstruktorokat hívhatunk a variable initialization szintaktikával:
  - `class_name object_name = initialization_value;`
- uniform inicializáció:
  - `class_name object_name { value, value, value, ... }`

Osztályok I. – uniform inicializáció

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

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

Osztályok I. – tagok inicializációja konstruktorban

- a konstruktor definiálása után közvetlenül inicializálhatjuk az adattagokat
- objektum adattagok esetén ha nem inicializáljuk őket a kettőspont után, akkor alapértelmezett konstruktorukkal jönnek létre

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

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }

Rectangle::Rectangle (int x, int y) : width(x) { height=y; }

Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

Osztályok I. - tagok inicializációja konstruktorban

```
// member initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
  public:
    Cylinder(double r, double h) : base (r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo (10,20);

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

Osztályok I. – mutatók osztályokra

- az osztályokra is definiálhatunk mutatókat, a mutató az objektumra mutat ilyenkor
- az objektum adattagjai a mutatón keresztül a -> operátorral érhetők el

expression	can be read as
`*x`	pointed to by `x`
`&x`	address of `x`
`x.y`	member `y` of object `x`
`x->y`	member `y` of object pointed to by `x`
`(*x).y`	member `y` of object pointed to by `x` (equivalent to the previous one)
`x[0]`	first object pointed to by `x`
`x[1]`	second object pointed to by `x`
`x[n]`	(`n`+1)th object pointed to by `x`

Osztályok I. – mutatók osztályokra

```cpp
// pointer to classes example
#include <iostream>
using namespace std;

class Rectangle {
  int width, height;
public:
  Rectangle(int x, int y) : width(x), height(y) {}
  int area(void) { return width * height; }
};


int main() {
  Rectangle obj (3, 4);
  Rectangle * foo, * bar, * baz;
  foo = &obj;
  bar = new Rectangle (5, 6);
  baz = new Rectangle[2] { {2,5}, {3,6} };
  cout << "obj's area: " << obj.area() << '\n';
  cout << "*foo's area: " << foo->area() << '\n';
  cout << "*bar's area: " << bar->area() << '\n';
  cout << "baz[0]'s area:" << baz[0].area() << '\n';
  cout << "baz[1]'s area:" << baz[1].area() << '\n';
  delete bar;
  delete[] baz;
  return 0;
}
```

Osztályok II. – operátorok túlterhelése

- az osztályok segítségével gyakorlatilag új típusokat hozunk létre
- ezekkel az új típusokkal is végezhetünk különböző műveleteket operátorok segítségével
- az osztályok esetén nem egyértelmű, hogy az egyes operátoroknak hogyan kell viselkedniük, ezért szükséges az operátorok túlterhelése (operator overloading)

```
struct myclass {
  string product;
  float price;
} a, b, c;
a = b + c;
```

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

Osztályok II. – operátorok túlterhelése

- szintaktika: `type operator sign (parameters) { /*... body ...*/ }`

Expression	Operator	Member function	Non-member function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@(B)	-
a(b,c...)	()	A::operator()(B,C...)	-
a->b	->	A::operator->()	-
(TYPE) a	TYPE	A::operator TYPE()	-

```cpp
// overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

Osztályok II. – operátorok túlterhelése

- operátorok túlterhelése történhet tagfüggvényként (ahogy az eddigi példákban szerepelt), illetve nem tagfüggvényként is: ilyenkor a túlterhelő függvény első paramétere a megfelelő osztály egy objektuma

```
// overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

Osztályok II. – `this` kulcsszó

- a this kulcsszó egy mutatót reprezentál arra az objektumra, amelynek a tagfüggvénye éppen végrehajtódik
  - így pl. ellenőrizhetjük, hogy egy függvénynek paraméterként átadott ugyanolyan típusú objektum megegyezik-e azzal az objektummal, amelyiken meghívtuk
  - az = operátor felülírása esetén gyakran használjuk a referenciaként való visszatérést

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

```
// example on this
#include <iostream>
using namespace std;

class Dummy {
  public:
    bool isitme (Dummy& param);
};

bool Dummy::isitme (Dummy& param)
{
  if (&param == this) return true;
  else return false;
}

int main () {
  Dummy a;
  Dummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b\n";
  return 0;
}
```

Osztályok II. – statikus tagok

- egy osztály tartalmazhat statikus adattagokat és függvényeket is
- egy statikus adattag (class variable) egy közös adattagként funkcionál minden objektum között, amik az adott osztályból jöttek létre, minden objektumban ugyanaz az értéke
  - pl. használható egy statikus számláló arra, hogy számoljuk hány példányt hoztunk létre egy adott osztályból

```
// static members in classes
#include <iostream>
using namespace std;

class Dummy {
  public:
    static int n;
    Dummy () { n++; };
};

int Dummy::n=0;

int main () {
  Dummy a;
  Dummy b[5];
  cout << a.n << '\n';
  Dummy * c = new Dummy;
  cout << Dummy::n << '\n';
  delete c;
  return 0;
}
```

Osztályok II. – statikus tagok

- statikus tagok osztály hatókörben (class scope) vannak, az osztályon belül nem lehet inicializálni őket, osztályon kívül kell:

```
int Dummy::n=0;
```

- mivel a statikus tag közös minden objektum között ezért hivatkozhatunk rá bármelyik objektumon keresztül vagy közvetlenül az osztályon keresztül:

```
cout << a.n << '\n';
cout << Dummy::n << '\n';
```

- statikus függvények: nem érhetnek nem statikus tagokat és nem használhatják a `this` kulcsszót

Osztályok II. – konstans tagfüggvények

- amikor egy objektumot a const kulcsszóval adunk meg, akkor minden adattagja csak olvasható az osztályon kívülről, a konstruktor viszont a megszokott módon hívódik meg és inicializálja az osztály adattagjait

```
// constructor on const object
#include <iostream>
using namespace std;

class MyClass {
  public:
    int x;
    MyClass(int val) : x(val) {}
    int get() {return x;}
};

int main() {
  const MyClass foo(10);
// foo.x = 20;              // not valid: x cannot be modified
  cout << foo.x << '\n';  // ok: data member x can be read
  return 0;
}
```

Osztályok II. – konstans tagfüggvények

- egy konstans objektum tagfüggvényei csak akkor hívhatók ha maguk is konstansok:

```
int get() const {return x;}
```

- a const kulcsszó több helyen is megjelenhet más-más jelentéssel:

```
int get() const {return x;}          // const member function
const int& get() {return x;}         // member function returning a const&
const int& get() const {return x;} // const member function returning a const&
```

- konstans tagfüggvények nem módosíthatják az osztály nem statikus adattagjait és nem hívhatják a nem konstans tagfüggvényeket sem
- konstans objektumok csak a konstans tagfüggvényeket érhetik el, nem konstans objektumok viszont elérhetik mind a konstans és nem konstans tagfüggvényeket is

Osztályok II. – konstans tagfüggvények

- konstans objektumok használata gyakori a fejlesztők körében
- a legtöbb függvény, ami egy objektumot vár paraméterként konstans referenciaként veszi azt át így a függvény csak a konstans tagokat éri el

```
// const objects
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
};

void print (const MyClass& arg) {
  cout << arg.get() << '\n';
}

int main() {
  MyClass foo (10);
  print(foo);

  return 0;
}
```

Osztályok II. – konstans tagfüggvények

- tagfüggvények konstans szempontból is túlterhelhetők: megadhatunk két függvényt úgy, hogy csak a const kulcsszóban különböznek
- ilyenkor ha a függvényt egy konstans objektumon keresztül hívjuk, akkor a konstans tagfüggvény hívódik meg, nem konstans objektumon keresztül pedig a nem konstans tagfüggvény

```
// overloading members on constness
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
    int& get() {return x;}
};

int main() {
  MyClass foo (10);
  const MyClass bar (20);
  foo.get() = 15;         // ok: get() returns int&
// bar.get() = 25;         // not valid: get() returns const int&
  cout << foo.get() << '\n';
  cout << bar.get() << '\n';

  return 0;
}
```

Osztályok II. – osztály sablonok (class templates)

- osztályok esetében is használhatunk sablon paramétereket, amiket az osztályban típusokként használhatunk:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

- az osztály példányosításakor megadhatjuk a konkrét típust, amit az objektum használ:

```
mypair<int> myobject (115, 36);
mypair<double> myfloats (3.0, 2.18);
```

Osztályok II. – osztály sablonok

- a sablon paraméterrel rendelkező osztályok tagfüggvényeit az osztály definícióban adhatjuk meg
- amennyiben az osztályon kívül szeretnénk kifejteni a függvényeket, akkor szükséges ott is megadni a sablon deklarációt:

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

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

Osztályok II. – sablon specializáció (template specialization)

- sablon osztályokat megadhatunk úgyis, hogy bizonyos konkrét típusok esetén máshogy viselkedjenek a sablon tagfüggvények
- pl.: a megadott osztály megnöveli eggyel az element értékét az increase tagfüggvényben, de szeretnénk ha a konkrét char típus esetén más legyen a függvénye működése, ilyenkor adjuk meg az osztály sablon specializációját

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```