12. gyakorlat – felsorolók

Az objektumorientált programozás és a C++

A programozási nyelvek igen sokféle megközelítés (*paradigma*) mentén működhetnek. Az hogy egy nyelv melyik kategóriába tartozik, alapjaiban meghatározza a jelölésrendszerét (*szintaktika*) és a használható utasítások körét, valamint azoknak végrehajtási módját (*szemantika*) is.

E mentén látványos ellentétpárt alkot az *imperatív* és a *deklaratív* paradigmájú nyelvek köre.

Az előbbi csoportba olyan nyelvek tartoznak, amelyek segítségével a processzor által direktben végrehajtandó utasításokat tudunk megfogalmazni. Ha még ráadásul a nyelv eszközeivel le tudjuk írni az ebben a félévben is tanult szekvenciális programozást támogató vezérlési szerkezeteket, azaz a struktogramokat is, akkor azt a nyelvet a *strukturált* programozási nyelvek családjába soroljuk. Ezek általában ismerik a „változó” fogalmát, hozzáférhetünk velük a memória bizonyos részeihez, az egyre bonyolultabb utasítássorozatokat pedig függvények (*procedurális* nyelvek) beiktatásával tudjuk magasabb absztrakciós szintre emelni, ezzel a kódunkat logikai egységekre bontani, összességében átláthatóbbá tenni. A C++ is egy ilyen nyelv.

A *deklaratív* nyelvek ezzel szemben olyanok, amelyeknél nem echte utasításokban gondolkozunk, hanem „kifejezésekben”. A fordítóprogram dolga ezen kifejezések ekvivalens átalakításával azokat a létező legegyszerűbb alakra (*normálforma*) hozni. Ez a fordító számára egy jóval nehezebb feladat, ezért ezek a nyelvek sosem a hatékonyságukról voltak híresek. De a fordítók és a hardverek fejlődése miatt az utóbbi időben egyre nagyobb teret nyernek ezek a nyelvek is. Az ELTE-n oktatott nyelvek közül a *funkcionális* nyelvek (Haskell, Clean) tartoznak ide, valamint bár ezek nem is igazán programozási nyelvek, de „deklaratív”-nak számít pl. a HTML és a CSS is, hiszen ezekben is csak leírjuk, hogy mit szeretnénk, de arról nem szólunk, hogy hogyan.

Természetesen egy nyelv több paradigmához is tartozhat, a paradigmák között nincsen éles határ, mondhatjuk azt, hogy egy nyelv „kicsit” ide tartozik, „kicsit” oda, és „nagyon” amoda. A paradigmákon belül lehetnek al-kategóriák is. A kettővel ezelőtti bekezdésben a C++ nyelvet például rögtön 3 (imperatív, procedurális, strukturált) paradigmába is besoroltam.

Az sincs kőbe vésve, hogy tulajdonképpen hány paradigma is van, ráadásul ezek nyilván az új nyelvek bevezetésével, a régiek fejlesztésével vagy esetleg kihalásával állandóan változnak. Mindenesetre álljon itt egy lista a fontosabbakról. Ezek BSc-MSc alatt mind részletezésre is kerülnek:

- imperatív
- procedurális
- strukturált
- objektumorientált
- deklaratív
- funkcionális
- logikai
- komponensalapú
- generikus

A C++ nyelv (ellentétben elődjével, a C-vel) *objektumorientált* is.

A félév második felében ezzel a nézőponttal ismerkedünk meg, nem is annyira a tervezés szintjén (azt majd a *programozási technológia* című tárgyból), hanem a gyakorlat oldaláról.

Az objektumorientáltság abból indul ki, hogy mikor elolvassuk, feldolgozzuk a megoldandó feladatot, nem egyből azon kezdünk el gondolkodni, hogy milyen utasításokat próbáljunk majd alkalmazni (mert ez egy bizonyos bonyolultság felett már átláthatatlan volna). Inkább bizonyos adatok illetve „felelősségi körök" mentén megpróbáljuk az egész feladatot szétbontani több, egyszerűbb egységre (ezzel megvalósítva az ún. *adatabsztrakció*t), majd ezen egységekhez definiálunk rajtuk értelmezett műveleteket, felépítünk közöttük valamilyen „kommunikációt", létrehozunk konkrét objektumokat, és ha mindent jól csináltunk, egy emberi szemlélettel nézve logikus, koherens, átlátható rendszert kapunk, amit később könnyen átalakíthatunk, bővíthetünk, ha épp arra van szükség.

Legyen a feladat például az, hogy egy vállalat dolgozóinak bizonyos szempontok szerint fizetésemelést kell adnunk. Ekkor nyilván maga a vállalat lesz az egyik „objektum", a vállalathoz tartozik egy költségvetés, ez is egy entitás a modellünkben, a vállalathoz tartozik a dolgozók listája, a dolgozók pedig olyan objektumok, amelyeknek vannak bizonyos adataik, lekérdezhetjük a fizetésüket, és módosíthatjuk is azt, ráadásul úgy, hogy az adott dolgozó adataitól (pl. milyen rég óta van a cégnél…) is függhet az emelés mértéke.

Miután ezzel megvagyunk, a feladat megoldása annyi, hogy létrehozunk egy darab „cég" objektumot, annak meghatározzuk az adatait, regisztráljuk a dolgozókat, stb., majd a cégen magán meghívunk egy olyan műveletet, ami a cég kasszája és a dolgozók listája alapján az utóbbin végigiterál és minden dolgozón, mint objektumon meghívja azt a függvényt, ami módosítja annak fizetését. Az, hogy mi alapján módosul a fizetés, az függhet magától a dolgozótól (pl. annak attribútumaitól; hogy milyen *altípus*ba tartozik), valamint minden a „fizetésnövelő" metódusnak átadott paramétertől. Nyilván ha a fenti feladatot egyértelműbben fogalmazom meg, akkor egy konkrétabb modellt tudok összeállítani.

Mindenesetre a résztvevő objektumok és az azokon értelmezett műveletek tisztán meghatározzák a „felelősségeket" és „jogosultságokat". Az a kód, ami egy adott dolgozó fizetését az adott dolgozó adataitól függően megnöveli, a dolgozó osztály kódjával lesz egy helyen. A „cég" osztálynál csak meg kell hívni ezt a függvényt minden dolgozóra, anélkül, hogy a cég osztály tudna róla, hogy az egyes beosztásban levő dolgozóknál pontosan hogy is működik a fizetésemelés. Egy jól végiggondolt modellel biztosan nem fordulhat elő, hogy valaki kimarad, vagy esetleg többször részesül a fizetésemelésből. Akkor jó a modell, ha az osztályhierarchia mentén minden „réteg" csak egy kicsit tesz hozzá a teljes feladat megoldásához. Ekkor a program könnyebben tesztelhető, könnyebben bővíthető lesz. Általában kerülendő a „mindenható osztály" (*god object*), azaz amikor minden lényegi működés egy osztályba kerül. Ekkor nem valósul meg a dekompozíció, így az osztályok bevezetésének egyik fontos előnyét nem tudjuk kihasználni. Az osztályokat – az objektumorientáltságot – lehet jól és lehet rosszul használni. Erről bővebb információt a *design pattern* és az *antipattern* szavakra kereséssel lehet találni. A god object az egyik leghíresebb antipattern.

Az órai példában egy mátrix osztályt valósítottunk meg, a későbbi alkalmakkor pedig a felsoroló lesz a téma, amit a lehetőségekhez képest objektumorientáltan fogunk megvalósítani.

Álljon most itt néhány fontos alapfogalom, azok rövid magyarázatával, angol nevével és C++-os megvalósításával:

Osztály (class)

Gyakorlatilag egy „sablon”, egy kitöltetlen keret, amit majd a konkrét objektumok konkrét adatokkal töltenek meg.

Tulajdonképpen azt soroljuk fel itt, hogy mik alkotják az osztályt, és hogy „miket tud” az osztály.

```
class Osztálynév
{
  mezők…
};
```

Objektum (object)

Konkrét adatokkal feltöltött változó, melynek a *típusa* valamelyik osztály. Ezen elvégezhetők mindazok a *művelet*ek, amiket az osztályon definiáltunk.

```
Osztálynév változónév; //deklaráció
változónév.metódusnév(); //metódushívás
v.push_back(4); //egy konkrét példa, a vector osztály egyik műveletére
```

Példány, példányosítás (instance, instantiation)

Egy objektum, illetve létrehozásának pillanata. Lefoglalódik a megfelelő memóriarész, majd lefut a *konstruktor*.

Konstruktor (constructor)

Példányosítás során lefutó metódus. Az osztályon belül visszatérési érték nélküli az osztály nevét viselő metódusként kell megadni.

Több konstruktor is lehet, ezt hívják konstruktor *túlterhelés*nek. Természetesen a konstruktoroknak paraméterezésükben el kell térniük, különben nem tudná a fordító, melyiket hívja meg!

```
Osztálynév() { … }
Osztálynév(double d) { … }
```

Viszont olyan konstruktort létre lehet hozni, amelyiknek paramétere *altípus*a egy másik konstruktor paraméterének…

Például ha a fenti osztályt így példányosítom, akkor az egyparaméteres `double`-t váró konstruktor fut le:

```
Osztálynév o1(1.5);
```

De akkor is, ha így, mivel az `int` *implicit kaszt*olódik `double`-ra:

```
Osztálynév o2(1);
```

De mi van akkor, ha nekem nemcsak a `double`-ös, hanem egy ilyen konstruktorom is van:

```
Osztálynév(int i) { … }
```

A fenti, `o1` és `o2` objektum létrehozását jelentő kód ekkor is helyesnek bizonyul, fordul és fut, de akkor `o1` a `double`-ös, `o2` pedig az `int`es paraméterezésű konstruktorral fog létrejönni!

Tehát mindig az hívódik meg, amelyiknek a *formális* paraméterei „közelebb" vannak az *aktuális*hoz.

Ez nem csak a konstruktorra igaz, hanem tetszőleges metódusra, sőt függvényre. *Túlterhelés*nek, *overloading*nak hívjuk. Ilyenkor a függvény különféle paraméterezésű változatai ténylegesen különböző függvényeknek fognak számítani a fordítóprogram szemszögéből. A programozónak viszont esetenként szebb, olvashatóbb kódot jelent ez a megoldás.

Default konstruktor[1]

Paraméter nélküli konstruktor.

```
vector<int> v; //itt meghívódik a param. nélküli konstruktor és amúgy azt
csinálja, hogy a vektort 0 méretűre inicializálja
vector<int> v(1); //itt az egy darab intet váró konstruktor hívódik meg
```

Implicit konstruktor

Amennyiben nem írunk egy osztályhoz semmilyen konstruktort (se paraméterest, se paraméter nélkülit), akkor alapból létrejön egy paraméter nélküli konstruktor, ami nem csinál mást, mint az *attribútum*ok default konstruktorait sorra meghívja.

Destruktor (destructor)

Az a függvény, ami a példány *élettartam*ának végén, a megszűnésekor fut le. Ha nem írunk sajátot, akkor ebből is van alapértelmezett, ami az adattagok destruktorait hívja végig.

A destruktor hívása tehát *statikus* memóriafoglalású változók esetében a szülő blokk végén, *dinamikus* változók esetén (következő félév) a `delete` utasítás kiadásakor fut le.

Destruktort mi nem igazán fogunk írni, mert a dinamikus adattagot nem tartalmazó osztályok esetében nincs mit eltakarítanunk magunk után.

Paramétere nem lehet.

```
~Osztálynév() { … }
```

Másoló konstruktor (copy constructor)

Ebben a félévben nem tanuljuk. Olyan konstruktor, aminek paramétere egy másik példány ugyanebből az osztályból. Ha az osztályunk dinamikus változókat is tartalmaz, akkor van igazi jelentősége, ekkor ugyanis nem feltétlenül helyes a *sekély másolat* (*shallow copy*), ehelyett az úgynevezett *mély másolás*t (*deep copy*) kell megvalósítani.

```
Osztálynév(const Osztálynév& o) { … }
```

---

[1] Megadhatunk default értéket is a konstruktor paraméterének: `Osztálynév(int n=0) {…}`, ekkor az `Osztálynév o;` deklaráció érvényes és úgy viselkedik, mintha `Osztálynév o(0)`-t írtunk volna.
Ilyen default értékeket mindenféle függvény paraméterének lehet adni, sőt osztály adattagnak is.
Ha a fenti konstruktor mellett van ténylegesen üres paraméterezésű konstruktorunk is, az fordítási hibát okoz, hiszen a fordító nem tudhatja, hogy a paraméter nélküli példányosításoknál az egy paraméteres default értékes vagy az eleve 0 paraméterest akartuk-e meghívni

hívása:

```
Osztálynév o;
Osztálynév o2(o);
```

vagy:

```
Osztálynév o2 = o;
```

Ez ilyenkor (tehát amikor a deklarációval van egybekötve egy értékadás) *nem* egyszerű értékadás, hanem kezdeti értékadás (változóinicializálás), azaz a másoló konstruktor meghívása.

Értékadó operátor (assignment operator)

Ugyanaz mint a *copy constructor*, csak egy már eddig is létező példánynak adunk új értéket.

Úgy szokták megvalósítani, hogy előbb lefuttatják a *destruktor* majd a *másoló konstruktor* kódját.

```
Osztálynév& Osztálynév operator=(const Osztálynév& o) { … }
```

hívása:

```
Osztálynév o,o2;
o2 = o;
```

Alapértelmezett értékek, adattag-inicializálás

Az attribútumoknak nem csak a konstruktorban adhatunk default értéket, hanem a deklarálásukkor is

```
class Osztálynév
{
  int i = 2;     //ha a konstruktorban nem mondok mást, i 2 lesz alapból
};
```

Speciális helyzetben vannak a *konstans* attribútumok. Ezek olyanok, akiknek a létrejöttük pillanatában *muszáj* valamilyen értéket kapniuk, és utána ezek a változók már „csak olvashatók” lesznek, a kezdeti értéküket megváltoztatni nem lehet. Két módon lehet nekik értéket adni: az egyik a most mutatott deklarációkori inicializálás, a másik pedig *nem* a konstruktor törzsében való értékadás – hiszen a konstruktor már a változók számára történő memóriafoglalás és valamilyen kezdő értékre való inicializálás *után* fut le – hanem az úgynevezett *inicializálólistá*ban való megadás.

Ez szintaktikailag így néz ki:

```
class Osztálynév
{
  Osztálynév(…) : változó1(érték1), változó2 (érték2), …
  {
    …
  }
  …
};
```

Tehát a konstruktor paraméterlistája és törzse között kell megadni vesszőkkel elválasztva azon attribútum–érték-párokat, amelyeket eleve egy adott értékre szeretnénk inicializálni. Ezek a változók tehát nem már létrejöttük után kapnak értéket, hanem eleve a paraméterként megadott értékre inicializálódnak, így az inicializálólista használata hatékonyabb, mint a konstruktortörzsbeli értékadások sorozata. Konstans attribútumokra pedig utóbbi nem is működne. Inicializálólistába kerülhet függvényhívás is, az „érték”-ek pedig kikerülhetnek a konstruktor paraméterei közül is, tehát nem csak konstansokat adhatunk meg.

Egy konstans attribútumnak adhatunk kezdeti értéket deklarációkor *is* és inicalizálólistában *is*, ekkor az utóbbi lesz az „erősebb”. Ennek az az értelme, hogy ha több féle konstruktor is van definiálva egy típushoz, és ezeknek eltérnek az inicializálólistái, akkor ha adunk default értéket az adott konstans változónak, akkor biztosak lehetünk abban, hogy akármelyik konstruktor is fut le, mindenképp lesz neki valamilyen értéke.

Egy apróságra érdemes[2] odafigyelni. Tekintsük az alábbi esetet:

```
class C
{
  const int a;
  int b;
  C(int n) : b(n), a(b+1) { … }
};
```

(az, hogy `a` konstans, `b` nem, az ebből a szempontból nem számít, csak szerettem volna illusztrálni, hogyan lehet megadni a konstans adattagokat)

Ha pl. `n=4` paraméterrel létrehozunk egy `C` objektumot, azt várnánk, hogy `b` értéke 4, `a` értéke pedig 5 legyen. `b` értéke valóban 4 lesz, `a` értéke pedig nem definiált (*memóriaszemét*). Az oka ennek az, hogy az inicializálólistában megadott kezdeti értékadások egy bizonyos jól meghatározott sorrendben futnak le, ami nem az, amilyen sorrendben az inicializálólista elemeit megadtuk, hanem az, amilyen sorrendben az érintett adattagokat deklaráltuk az osztályon belül! Azaz, ahogy a pirossal kiemelt részből látszik, mivel `a` előbb volt deklarálva, mint `b`, előbb `a` fog `b+1`-re inicializálódni, de ekkor még `b` értéke nem tisztázott, tehát `a` értéke se az lesz, amit vártunk. Ha a deklaráció sorrendjét megváltoztatjuk, vagy `a(n+1)`-et írunk a listába, akkor `a` is „rendes” értéket fog kapni.

Mező (field)

Más néven az osztály *tag*jai (*member*). *Attribútum* vagy *metódus*.

Szelektor (selector)

Egy olyan függvény, amivel egy adott osztály egy példányához hozzárendeljük annak egy mezőjét. Magyarán, egy objektumra alkalmazva ezzel tudjuk elérni az adattagokat és metódusokat. Például, ha `T` osztályban egy *publikus* `a` adattag és `f` függvény található, akkor azokat `T` egy `t` nevű példányán így érjük el:

```
t.a;
t.f();
```

[2] Ez tipikus vizsgakérdés a C++ tárgyból és tipikus állásinterjús felvételi kérdés is ;)

Attribútum (attribute)

Vagy *adattag*. Tulajdonképpen egy változó, mely hozzá van rendelve az adott objektumpéldányhoz, *élettartam*a vele azonos[3].

```
class T
{
  int a;
  string s;
};
```

Metódus (method)

Vagy *tagfüggvény*. Egy olyan függvény, ami az osztályhoz lett deklarálva, ezt az osztály „példányain lehet elvégezni". Ez abban nyilvánul meg, hogy mindig van egy *implicit* paramétere, ami egy az arra az objektumra mutató *pointer*[4], akin elvégeztük. Ezt úgy hívjuk, „`this`", és rajta keresztül lehet elérni az osztálypéldány attribútumait vagy egyéb metódusait. A `this` kiírása csak akkor kötelező, ha a metódus egy paramétere *elfed*né a hivatkozni kívánt mezőt.

```
class T
{
  int a = 5;
  void f() { cout<<a<<endl; }
  int g(int a) { cout<<(this->a)<<a<<endl; }
};

T t;
t.f();     //kiírja, hogy 5
t.g(3);    //kiírja, hogy 53
```

Operátorok (operator)

Azok a függvények, amelyeket valamilyen szimbólummal *infix*, vagy más egyéb módon (is) meg lehet hívni. Új operátort nem lehet behozni, de már meglévő operátorokat *túl* lehet *terhel*ni.

Ezek gyakorlatilag a műveleti jelek (`+`,`*`,…), az indexelések (`[]`), a függvényhívás (`()`), a kiírás (`<<`), stb.

A függvényhívásos jelölés előnye az indexeléssel szemben az, hogy több *operandus*a (azaz paramétere) is lehet.

Használatukkal igen olvashatóvá, elegánssá lehet tenni a kódot.

```
T operator+(const T& a, const T& b) { … }
T a,b;
T c = a + b; //ekvivalens vele: T c = a.operator+(b);
```

[3] Kivétel: ún. *lusta példányosítás*, vagy ha elrontjuk a *destruktor*t ☺ de ez nem e féléves anyag…

[4] A *pointer* (vagy *mutató*) se idei anyag, most annyit kell csak megjegyezni, hogy ha pointeren (pl. `this`) keresztül érjük el a mezőket, akkor szintaktikusan egy nyíllal (`->`), nem pedig a . szelektorral tudjuk ezt jelölni

Elérhetőség (accessibility/access modifiers/access specifiers)

Szokták *láthatóság*nak is nevezni, bár az eredetileg mást jelent. (*lásd 1. gyakorlat pdf-je*).

Arról van itt szó, hogy az osztályok megvalósításakor nem minden belső dolgot szeretnénk a használóik orrára kötni (*enkapszuláció* elve). Például, ha egy mátrix osztályt akarok megvalósítani, azt nem akarom kifelé mutatni, hogy azt pl. egymásba ágyazott vektorokkal implementáltam, én csak szeretnék egy megfelelő elem-lekérő és elem-módosító függvényt írni, de meg szeretném tiltani hogy másképp hozzáférjenek az adatokhoz.

Ennek értelme kettős: egyrészt ha egy *szabványos interfész*en keresztül tud csak az osztály és a külvilág kommunikálni, akkor az osztály adattárolási megvalósításai könnyedén cserélhetők, nem kell mást átírni csak az osztály kódját. De a használó kódrészleteket nem. Másrészt, ha bármilyen *invariáns*[5] tulajdonságot definiáltam az osztályra, akkor annak megtartását nehézkes és elég bizonytalan dolog lenne a külső hívóra rábízni. Egyszerűbb és biztonságosabb azt mondani, hogy közvetlenül inkább nem engedem elérni az adattagot, hanem definiálok hozzá egy módosító műveletet, ami esetlegesen visszadobja az illegális módosítási kérelmeket. Emellett egyébként összességében kevesebb kódismétléssel is jár ez a megoldás, mintha mindig minden alkalommal a használatot venném körbe egy feltételvizsgálattal.

Tehát mind a műveletek, mind az adattagok lehetnek rejtettek. Az adattagokat a fentebb részletezett okok miatt általában nem szoktuk nyilvánosnak megtartani, a műveletek közül pedig azokat, amiket a kívülről használható műveletek „segédműveleteinek" gondoljuk, azokat elrejtjük, a többit nyilvánosnak hagyjuk. Elég kézenfekvő, hogy a konstruktor és destruktor nyilvános, kívülről hívható legyen. Most ebben a félévben az is lesz, aztán később fogunk tanulni olyan eseteket, amikor a konstruktort érdemes priváttá tenni. Aki nem bír magával és utána szeretne olvasni az a „*factory osztály*"-ra, illetve a „*`virtual` függvény/absztrakt osztály*" témakörökre keressen rá.

Az elérhetőség 3 fajta lehet:

- `public` – mindazok a műveletek és adattagok, amiket nyilvánossá akarunk tenni
- `private` – amiket nem (osztályon belül persze elérhetők)
- `protected` – ezek az osztályon belül illetve az osztály leszármazottjaiban[6] érhetőek el, ebben a félévben ezt a módot nemigen fogjuk használni

Az elérhetőség *osztályszintű* fogalom! Azaz ha egy `T` osztályból van egy `t1` és `t2` példányom, `t1` függvényei elérik `t2` privát adattagjait (ha paraméterként átadom neki `t2`-t)!

Az egyes mezők elérhetőségét úgy adjuk meg, hogy kiírjuk a megfelelő kulcsszót majd egy kettőspontot, és az ezt követő összes mező a következő elérhetőséges kulcsszóig ennek megfelelő elérhetőségű lesz… Egy osztályban nem kötelező az összes publikus „dolgot" egymás után írni, lehet többször is kitenni a `public` jelölést.

Ha nem írunk semmit, akkor az a mező alapértelmezésben `private`.

---

[5] Olyan logikai állítás, aminek mindvégig fenn kell maradnia. Mondjuk egy „totó tipp" típus tartalmazzon egy karaktert, de amikor módosítjuk ezt a karaktert, akkor minden olyan próbálkozást, ami nem arra irányult, hogy `'1'`, `'2'`, vagy `'X'` legyen ez a karakter, el kell utasítani…

[6] Újabb olyan fogalom, ami nem idei anyag: *származtatás* (*inheritance*)

```
class T
{
  int a; //private
  private:
  int b; //private
  int c; //private
  public:
  int d; //public
  private:
  int e; //private
  int f; //private
}
```

Barát függvények és osztályok (`friend`)

Ha egy *függvény* a `T` típus barátjaként lett definiálva, akkor az hozzáfér a `T` *privát* mezőihez.

A barát függvény lehet egy másik osztály *metódus*a, vagy egy *globális függvény*.

Egy egész *osztály* is lehet barát.

A barátságot a `T` osztály definíciójában kell jelölni, a globális függvényeket akár itt definiálhatjuk is, de nem szokás, hiszen nem a `T`-hez tartoznak, csak haverok vele ☺

Egy jellemző használati esete: amikor bevezetek egy új `T` típust, és ehhez olyan függvényt (sőt, mondjuk *operátor*t) szeretnék készíteni, aminek jobb oldali operandusa `T` típusú, a bal oldali viszont vagy egy olyan osztályé amit nincs jogom szerkeszteni, vagy esetleg primitív típusú (`int`, `bool`, stb). Ekkor ezt a függvényt jobb híján globálisként kell elkészítenem, de azért az megengedhető, hogy belelássak egyik vagy másik paraméterének (pl. `T`) a megvalósításába is. Így mégiscsak úgy viselkedik, mintha ez `T` tagfüggvénye lenne, csak épp nem `T`-n meghívva. *Kommutatív* operátoroknál nagyon jól jön (konkrétumokat lásd a mellékelt `cpp` fájlban).

Fejléc- (header) és forrásfájlok (source)

A mezőket általában az osztály *definíció*jában csak *deklarál*juk, a *definíció*jukat egy külön fájlba szoktuk írni.

Inline definíciónak hívjuk azt, amikor a deklarálás helyén definiáljuk is a metódust.

Egyébként egy osztályt külön szoktunk bontani két fájlra, egyrészt egy a deklarációkat tartalmazó `.h` kiterjesztésű fejlécfájlra, másrészt a függvények kifejtéseit tartalmazó `.cpp` állományra.

Mind a forrásfájlba, mind azokba a forrásfájlokba, ahol az így definiált osztályt használni szeretnénk, be kell *include*-olni a *header* fájlt.

A forrásfájlban a függvényeket az osztálynévvel *minősít*ve érjük el – hiszen azok az osztály tagjai. Az osztályban esetlegesen barátként deklarált függvényeket minősítés nélkül érjük el, hiszen azok nem az osztály részei.

Header fájlba nem illik `using namespace`-eket írni, mivel ezeket a fájlokat elvben bárhova (oda is ahol nem szeretnénk `namespece`-eket használni) `include`-olhatjuk. Forrásfájlokat nem `include`-olunk, ezért oda szabad ilyesmit írni.

Egy példa:

`tipus.h` tartalma:

```
class Tipus
{
  int Metod1(int a);
  void Metod2();
  int Metod3() { return 3; } //inline definíció
  friend int Method4(const Tipus&); //barát függvény
};
```

`tipus.cpp` tartalma:

```
#include"tipus.h"
int Tipus::Metod1(int a) { return 1; }
void Tipus::Metod2() { … }
int Method4(const Tipus& t) { … } //nem a Tipus része
```

Konstans műveletek

A konstansként megjelölt műveletekben a `this` mutató egy *„konstans* `T`*-re mutató”* típusú pointer. Ez azt jelenti, hogy egy ilyen műveleten keresztül azt a példányt, amin a metódust meghívtuk, nem lehet módosítani

```
void f() const { this->n = 1; } //ez tilos
```

Statikus adattagok

Olyan adattagok, amik nem az osztály egy példányához, hanem *magához* az osztályhoz vannak rendelve. Például egy olyan szám típusú változó, ami minden konstruktorhíváskor nő és minden destruktorhíváskor csökken, el tudja tárolni, mennyi példány van most az adott osztály objektumaiból.

Statikus műveletek

Olyan műveletek, amik nem az osztály egy példányához, hanem magához az osztályhoz vannak rendelve. Például a fenti számláló lekérdezése. Ez meghívható egy példányon, de a *scope resolution* operátor segítségével akár létező példány nélkül a „típuson” is.

```
class T
{
  static int darab() { … }
}

T t;

t.darab();
T::darab();
```

Most, hogy megtudtuk mi az osztály, tisztában vagyunk mindenféle szintaktikai lehetőségekkel, adja magát a kérdés:

Mi a különbség az *osztály* és a *rekord* között?

Egyrészt az a szokás, hogy `struct`tal a *POD* osztályokat szoktuk jelölni. A rövidítés azt takarja, hogy *Plain Old Data*, ami annyit tesz, hogy egyszerű, mindenféle komolyabb metódusoktól mentes rekord típus. Például egy *pár*, egy *hármas*, egy *négyes*…

Osztállyal olyan dolgokat szoktunk megadni, ami túlmutat az adattagjain. Ami több mint a részei összessége. Ahol igazán van jelentősége az elérhetőségnek, a metódusoknak, az öröklésnek, és kb. mindennek ami az objektumorientáltságnál szóba jöhet.

Másrészt annyi, hogy a default elérhetőség a `struct`nál (összhangban az előző ponttal) a `public`, a `class`nál a `private`.

Van még különbség az *öröklés*i módok és a *template* osztályokban való alkalmazhatóság terén, de ez ebben a félévben lényegtelen.