A programozás alapjai 2.

Hivatalos segédlet a 06. laborhoz.

Tartalomjegyzék

Konstans tagváltozók, tagfüggvények

Osztályok **konstans tagváltozóit** kezdőérték-adás után nem lehet megváltoztatni. Továbbá olyan osztályra, mely rendelkezik konstans tagváltozóval, nem alkalmazható értékadás operátor, viszont copy konstruktor igen (létrehozáskor lehet kezdőértéket adni neki, és soha többé).

A **konstans taggfüggvények** garantálják, hogy nem fogják megváltoztatni, illetve nem hívnak meg olyan függvényt, mely megváltoztathatja az **objektum állapotát** (tehát nem változtatja az objektum tagváltozóinak értékét). Pl. getter függvények konstansok lehetnek (és legyenek is mindig), mivel csak visszaadják egy tagváltozó értékét, nem módosítják azt. Ahhoz, hogy egy tagfüggvény konstans legyen a "const" kulcsszót kell a paraméterlista és függvénytörzs közé helyezni. Ezt mind a függvény deklarációjánál, mind a definíciójánál jelölni kell (a header és .cpp fájlban is).

Ha egy konstans függvényben olyan utasítás szerepel, mely módosítja az objektum állapotát, vagy nem konstans függvényt hív, **fordítás idejű hiba** keletkezik.

Konstruktor soha nem lehet konstans, mivel inicializálja a tagváltozókat, tehát változtatja őket, és ezt nem tilthatjuk meg neki.

Konstans objektumon csak konstans tagfüggvény hívható.

Statikus tagváltozók, tagfüggvények

A statikus tagváltozó olyan tagváltozó, mely nem egy objektum sajátja, hanem az egész osztályra közösen vonatkozik. Tehát nem az objektum tulajdonsága, hanem az osztályé. Értéke minden objektum számára ugyanaz.

Egy osztály statikus tagváltozója akkor is létezik és elérhető, ha az adott osztályból még egyetlen objektum sem lett példányosítva.

Statikus taggfüggvények az osztályhoz tartoznak, nem kapnak 0. paraméterként this pointert. A statikus tagváltozókon dolgoznak, nem érik el egy objektum saját, nem statikus tagváltozóit és nem hívhatnak nem statikus függvényeket.

Elérhetőek egy példányon keresztül vagy az osztály nevén keresztül.

Mire lehet ezt használni például?

Egyedi objektum azonosító

Gyakori feladat, hogy az objektumokat **egyedi azonosítóval** szeretnénk ellátni. Ezt meg lehetne úgy oldani, hogy az objektum konstruktorában átadjuk neki paraméterül. Ekkor viszont semmi sem garantálja, hogy az objektumok azonosítói egyediek lesznek, a létrehozót nem tudjuk kötelezni arra, hogy létrehozás előtt ellenőrizze az egyediséget.

Random szám generálás sem tökéletes megoldás, mivel előfordulhat ismétlődés. Ráadásul nem tudjuk előre, hogy hány objektum lesz példányosítva, így nehéz megbecsülni, hogy mekkora intervallumon kéne random számot generálni.

A legjobb (és egyetlen igazán elfogadható megoldás), ha az osztály rendelkezik egy 0 kezdőértékű statikus változóval, és egy ugyanolyan típusú egyedi azonosítót tároló tagváltozóval (jó, ha konstans). Amikor egy új objektum példányosodik, a konstruktorában az egyedi azonosítójának a statikus változó értékét adjuk értékül, és a statikus változó értékét növeljük eggyel.

Figyelem! Másoló konstruktorban az egyedi azonosítót nem másoljuk, hanem a statikus változóból vesszük az értéket, majd növeljük, mint más konstruktornál. Hiába másoló konstruktor, az egyedi azonosítót nem másolhatjuk.

Meyers' singleton pattern (tervezési minta)

Tegyük fel, hogy egy adott objektumot kell használnunk az alkalmazás egész életciklusa alatt. C++-ban lehetőségünk *lenne* egy globális példányt létrehozni, amit a programban bárhol használhatnánk. Azonban egy jó objektum orientált design esetén nem engedhető meg globális változó vagy függvény, mivel ellentmondásban lenne a beágyazás (encapsulation) és az adatrejtés (data hiding) elvekkel. A másik probléma, hogyha át szeretnénk állni egy másik programozási nyelvre, például C#-ra, ahol már minden objektumnak számít (még egy egyszerű int is), ott már nincs is lehetőség globális változók létrehozására. Ezért találták ki a Singleton tervezési mintát (pattern), amit minden OOP-t támogató nyelvben meg lehet valósítani (bár ennek is több változata létezik). Egyébként az ilyen univerziális praktikákból nagyon sok van (akit érdekelne bővebben: http://www.oodesign.com/).

Példák alkalmazásra:

- Logger singleton osztály
  - különböző logolási szinteket szeretnénk megvalósítani
  - grafikusan és konzolosan is szeretnénk logolni
    - nem elégszünk meg a printf-fel, cout-tal és társaival
  - ha nem lenne, nagyon sok overheaddel járna
    - mindig copy-pastelni kellene X sor kódot csak azért, hogy tudjunk logolni egy adott helyen
  - több Logger osztálynak nem lenne értelme
- PrintSpooler singleton osztály
  - nyomtatási sor kezelő
  - erőforrás hozzáférési konkurrencia kezelés
    - két processz/szál szeretne egyszerre nyomtatni
    - versenyhelyzet alakul ki köztük
    - arbitrációs folyamat: annak eldöntése, hogy ki nyomtathasson először

Fontos: ne Singleton osztály köré szervezzük az egész programunkat, különben Clean Code elvet sértünk ("god class"). Magyarán **irányító szerepet ne töltsön be**. Megsértése esetén a program egyes részei a függőség miatt nem lesznek újrafelhasználhatók. Ez azt jelenti, hogya nekikezdetek egy másik szoftver projektnek, az alapoktól kell felépíteni az egészet (korábban megírt modulok nélkül), ami rengeteg időt elvisz feleslegesen. Javallott, hogy **Singletont csak a fentebb említett példákhoz hasonló szolgáltatásokhoz használjatok.**

Felmerülhet a kérdés, hogy miért nem olyan osztályt készítünk inkább, amelyikben minden függvény és változó statikus. Ekkor ugyan jobb teljesítményt érténk el (hiszen fordítási idő végére már használhatók lesznek), azonban számos olyan feature-t elveszíthetünk, amit a Singleton biztosít: függvénynév túlterhelés, könnyebb tesztelni ("dummy"/"mock"/teszt adattal feltöltött Singleton), könnyebb belső állapotnyilvántartás (objektumoknak van állapota). (Van még pár fontos előnye (polimorfizmus és öröklés), de ezekről majd később fogtok hallani).

singleton.h

```
#pragma once
#include <iostream>

class Singleton
{
	// - priváttá tesszük a konstruktorokat
	// - így csak osztályon belül lehet példányosítani
	Singleton() {
		someValue = 10;
	}
	Singleton(const Singleton&){}
	int someValue;
public:
	// privát konstruktor miatt
	// az osztályon beüli scope-ot csak statikus metóduson keresztül érjük el
	// ezen belül kell elérni, hogy csak egy példány létezhessen
	static Singleton& getInstance();

	// funkcionalitást megtestesítő függvény
	void doStuff();
};
```

singleton.cpp

```
#include "singleton.h"

Singleton& Singleton::getInstance() {
	// csak egyszer jön létre, bárhányszor hívjuk meg a getInstance()-t
	static Singleton s;
	return s;
}

void Singleton::doStuff() {
	std::cout << "Method of a singleton class. Value of someValue: " << this->someValue << std::endl;
}
```

main.cpp

```
#include "singleton.h"

int main() {
	// kívülről akár teljesen különböző példányoknak is lehetne nézni őket
	Singleton& mySingleton1 = Singleton::getInstance();
	Singleton& mySingleton2 = Singleton::getInstance();

	mySingleton1.doStuff(); // kimenet: Method of a singleton class. Value of someValue: 10
	mySingleton2.doStuff(); // kimenet: Method of a singleton class. Value of someValue: 10

	return 0;
}
```

Osztály tagváltozóinak inicializálása

Inicializálásnak a változók/tagváltozók létrehozásakor történő kezdőérték-adást nevezzük. Egy egyszerű értékadás nem számít inicializálásnak.

pl. `int x; x = 0;  // definíció + értékadás`

pl. `int x = 0; // definíció + inicializáció`

Egy osztály tagváltozóinak inicializációját a **konstruktorok** teszik lehetővé. Az értékadásokat elvégezhetjük a konstruktor **törzsében** vagy **inicializációs listán**. Az összes tagváltozónak a memóriafoglalás és az inicializációs lista műveletei még a konstruktor törzsének lefutása előtt végrehajtódnak. Ha egy tagváltozó nem szerepel az inicializációs listában, alapértelmezett értékkel foglalódik le számára a memóriaterület. Ha szerepel, akkor a megadott értékkel.

Tegyük fel, hogy van egy “Computer” osztályunk, aminek két int típusú tagváltozója van, a “cpu” és “ram”.

Inicializáció a konstruktor törzsében: `Computer(int c, int r){ cpu = c; ram = r; }`

Inicializációs listával: `Computer(int c, int r): cpu(c), ram(r) {}`

Beépített, **egyszerű** típusú tagváltozók esetén mindegy, hogy azokat milyen módon inicializáljuk (ld. fenti mindkét példa jó).

Ezzel ellentétben, ha egy **objektum** tagváltozóról van szó (valamilyen osztály típusú), és az inicializációs listában nem szerepel, akkor default konstruktorral fog létrejönni, mielőtt a konstruktor törzse lefutna. Ha valamilyen paraméterekkel szeretnénk létrehozni, akkor ezt mindenképp az inicializációs listában tegyük meg, különben feleslegesen jön létre belőle egy default példány. Ha pedig nincs is default konstruktora a tagváltozó osztályának, akkor muszáj inicializációs listában valamilyen paraméterrel inicializálni.

Referencia tagváltozó inicializációja csak és kizárólag inicializációs listában hajtható végre, és itt kötelező is, mivel referenciának muszáj értéket adni.

Konstans tagváltozónak mindig inicializációs listában adunk kezdőértéket.

Statikus tagváltozókat az osztályon kívül kell definiálni és inicializálni. Ezt nem lehet header fájlban megtenni, mivel ekkor minden fájlnak, mely include-olja ezt a header fájlt, lenne egy definíciója a statikus változóra. Összességében több definíciója lenne ugyanannak a változóban, ami tiltott (pontosan egy lehet neki), és a linker hibát fog dobni. Tehát *.cpp fájlban, mindenen kívül (global scope) kell definiálni.

pl. `int MyClass::staticVar = 10;` → osztályhoz tartozik, így hivatkozunk rá (mivel több osztálynak is lehetne ugyanilyen nevű statikus változója, meg kell adni, melyikre gondoltunk)

Általános szabályok az előbbi, egyedi eseteken kívül:

- Ha egy tagváltozónak **nem alapértelmezett értéket** szeretnénk beállítani (pl. egy objektumnak a default konstruktora által biztosított értékeket), akkor használjunk **inicializációs listát**. E mögött az a motiváció, hogy ha a konstruktor törzsében végezzük az értékadásokat, akkor addigra már alapértelmezett értékkel lefoglalódott számára a memória, és még azt felülírjuk (2 művelet), ahelyett hogy egyetlen, jól paraméterezett kezdőérték-adással oldottuk volna meg (1 művelet) és időt takarítanánk meg.
- Minden olyan típusú tagváltozónak **inicializációs listában** kell kezdőértéket adni, melynek **nincs értékadás operátora** vagy **default konstruktora**.

Friendship

Egy osztályban deklarálhatunk olyan függvényeket, melyek nem tagfüggvények, mégis engedélyt kapnak arra, hogy a private (és protected - ld. később) láthatóságú tagváltozókat és tagfüggvényeket elérjen. Ezt úgy tehetjük meg, ha a külső függvény deklarációját felvesszük az osztályba és elé tesszük a “friend” kulcsszót.

Egy osztály is lehet friend, mely a friend függvényekhez hasonlóan hozzáfér egy másik osztály privát (és protected) tagváltozóihoz és tagfüggvényeihez. Ezt az engedélyt mindig explicit ki kell írni, tehát egy osztály “barátjának a barátja nem a barátunk”. Friend osztályt deklarálni kell a “friend” kulcsszóval abban az osztályban, aminek a privát (és protected) tagjaihoz hozzáférhet.

pl.

```
class Person; // Person osztály deklarációja
class Address {
        friend class Person; // a Person osztály hozzáfér a privát tagokhoz
        // ...
}
class Person {
        Address live;   // Address típusú tagváltozó
        int age;
public:
        friend void aging(Person);    // külső függvény, ami hozzáfér a ptivát tagokhoz
        // ...
}
void aging(Person p){
        p.age++;
}
```

Névterek

Előfordulhat az az eset (bármikor), hogy amikor egy vagy több fájlt/libraryt include-olunk, ugyanazzal az azonosítóval létezik több függvény vagy osztály. Például írhatunk egy olyan függvényt, aminek *printf*() a neve, és ha include-oljuk az *stdio.h* könyvtárat, abban szintén létezik egy *printf*() metódus. Ilyen névütközés esetén a fordítás folyamán hibát kapunk, mert nem tudja kitalálni, hogy mi melyikre gondoltunk, amikor meghívtuk azt a függvényt, amiből kettő is van (pl. a mi saját *printf*() vagy az *stdio.h*-ban lévő *printf*() metódust hívja meg), vagy melyik osztályt szeretnénk példányosítani, ha több lehetőség is van. Ezeknek a **névütközéseknek** a **feloldására** használhatóak a névterek.

Egy névtér egy kódrészt fog egybe, melyen belül garantálni kell az azonosítók (a függvény, változó és osztálynevek) **egyediségét**. Egy névtér elemei egymást elérik. Névteren belül található függvényekhez/osztályokhoz csak úgy férhetünk hozzá, ha megadjuk a a névteret is, amelyen belül definiálva lettek. Ehhez a **hatókör-feloldó (scope resolution)** operátort (**::**) használjuk. Ennek a használatával tudjuk megmondani a fordítónak, hogy egy bizonyos névtérben található függvényt/osztályt használjon.

Tegyük fel, hogy az előbb említett *printf*() függvényünket egy *MagicNamespace* nevű névtérbe helyeztük át. Most, ha meghívjuk a *printf*() függvényt, akkor az *stdio.h*-ban lévő függvény fog lefutni. Ha pedig a sajátunkat szeretnénk meghívni, ahhoz a *MagicNamespace::printf()* függvényhívás szükséges.

Névterekkel nem csak a névütközéseket oldhatjuk meg, hanem **logikai egységet** is létrehozhatunk összetartozó osztályoknak, függvényeknek stb. Pl. egy névtérbe helyezhetjük a geometriai alakzatok osztályait.

Egy névtér **több fájlba** is felbontható, és névterek **egymásba ágyazhatók**.

using

Amikor egy bizonyos névtérben lévő függvényeket, osztályokat stb. sűrűn használjuk, ezért nagyon sokszor ki kell írnunk a névtér nevét is, a C++ lehetőséget ad ennek egyszerűsítésére.

Meg tudjuk mondani a fordítónak, hogy ha egy bizonyos azonosítót használunk, akkor azt melyik névtérben keresse.

- szintaxis: **using** nevter::azonosito;
- beágyazott névterek esetén: **using** nevter1::nevter2::azonosito;

Abban az esetben viszont, amikor nagyon sok függvényt stb. szeretnénk használni egy névtérből, nem hatékony ennek a használata. Ezt a problémát úgy tudjuk megoldani, hogy a fordítónak azt az utasítást adjuk, hogy egy adott névtérből használjon minden azonosítót.

- szintaxis: **using namespace** nevter;

Figyelem, *using namespace* használatával legyünk óvatosak, mert az eredetileg megoldott névütközés probléma visszatérhet, ezért kerüljük! Pl. két olyan névtérre is alkalmazzuk, melyekben szerepel ugyanaz az azonosító. Ekkor az elsőként bemutatott módszerrel lehet a compilernek megmondani, melyik névtérből szeretnénk használni az azonosítót vagy explicit kiírjuk a névteret.

Using-ot használhatunk globálisan és függvényeken belül is.