Katona József–Király Zoltán

OBJEKTUMORIENTÁLT SZOFTVERFEJLESZTÉS C++ NYELVEN

FEJLESZTŐI KÖRNYEZET, OSZTÁLYOK ÉS OBJEKTUMOK

Statikus modelltervezés
A fejlesztő környezet rövid ismertetése
Osztályok deklarálása, implementálása és példányosítása
általánosságban és nyelv specifikusan

Dunakavics
D·U·F PRESS

INFORMATIKAI KÖNYVEK

1.

Katona József–Király Zoltán

OBJEKTUMORIENTÁLT SZOFTVERFEJLESZTÉS C++ NYELVEN

FEJLESZTŐI KÖRNYEZET, OSZTÁLYOK ÉS OBJEKTUMOK

© Katona József–Király Zoltán, 2015

Lektorálta: Dr. Kusper Gábor, tanszékvezető főiskolai docens

D▪U▪F PRESS
DUNAÚJVÁROSI FŐISKOLA
www.duf.hu

Kiadóvezető: Németh István
Felelős szerkesztő: Nemeskéry Artúr
Layout és tördelés: Duma Attila

Kiadja: DUF Press, a Dunaújvárosi Főiskola kiadója
Felelős kiadó: Dr. András István

Készült a HTSART nyomdában.
Felelős vezető: Halász Iván

ISBN 978-963-9915-52-7
ISSN 2415-9115

Katona József–Király Zoltán

OBJEKTUMORIENTÁLT SZOFTVERFEJLESZTÉS C++ NYELVEN

FEJLESZTŐI KÖRNYEZET, OSZTÁLYOK ÉS OBJEKTUMOK

Statikus modelltervezés
A fejlesztő környezet rövid ismertetése
Osztályok deklarálása, implementálása és példányosítása
általánosságban és nyelv specifikusan

Dunakavics
DUF PRESS
Dunaújváros, 2015

INFORMATIKAI KÖNYVEK 1.

Tartalom

1. Fejezet

Általánosságban az objektumorientált programozásról 9

Statikus modelltervezés 10

Osztálydiagramok (Class diagram) 11

Objektumdiagramok (Object diagram) 12

A fejlesztő környezet rövid ismertetése 13

Új Managed C++ Project létrehozása Visual Studio környezetben 13

Managed C++ osztály (állomány és header fájl) létrehozása Visual Studio környezetben 17

Osztályok deklarálása, implementálása és példányosítása általánosságban és nyelvspecifikusan 19

Mintapélda 20

Mintapélda 25

Gyakorló példák 27

2. Forráskódok

Calculator osztály deklarációja (Calculator.h) 32

Calculator osztály implementálása (Calculator.cpp) 33

Calculator osztály példányosítása (main.cpp) 34

Text osztály deklarációja (Text.h) 35

Text osztály implementálása (Text.cpp) 36

Text osztály példányosítása (main.cpp) 38

3. Felhasznált irodalom 39

Előszó

Úgy gondoljuk, hogy ez a könyvsorozat hozzájárulhat ahhoz, hogy különböző szakemberek kamatoztathassák tudásukat a tématerületen.

A műsorozat elsődleges célja az objektumorientált világ megismertetése. Az ismeretek elsajátításhoz igyekeztünk egy olyan magas szintű programozási nyelvet választani, amely az előismeretekre támaszkodva hatékonyan képes szemléltetni az OOP[1] világában használt szabályokat, fogalmakat, elveket és modelleket. A választott C++ nyelv mindamellett, hogy a fentebb leírt előírásoknak megfelel, a munkaerőpiacon is az egyik legsikeresebbnek számító magas szintű nyelv.

A szoftverfejlesztési és annak tanítási tapasztalatai alapján arra az elhatározásra jutottunk, hogy a választott C++ nyelvet nem a legelejétől kívánjuk ismertetni. Nem szeretnénk például elismételni azokat a vezérlési szerkezeteket, amelyeket már egy középszinten lévő programozó számtalanszor hallott és jó eséllyel használt. Természetesen ez nem azt jelenti, hogy ezeket az eszközöket el is felejthetjük, ugyanis, ahhoz, hogy jó programozó váljon belőlünk nem elég ismernünk ezeket az utasításkészleteket, hanem fel is kell ismernünk, hogy hol és mikor a legcélszerűbb azokat használni.

A könyv összeállításakor az ipari és oktatói tapasztalatainkra és számos, neves szakirodalomra támaszkodtunk. Reméljük, hogy ez a segédlet hozzásegíti az olvasót a tananyag megértéséhez és könnyed elsajátításához. Ehhez kellő kitartást és lelkesedést kívánunk.

Köszönettel tartozunk a kézirat lektorának, Dr. Kusper Gábornak gondos munkájáért, szakmai tanácsaiért és módszertani javaslataiért, melyek jelentősen hozzájárultak a könyvsorozat minél magasabb szakmai szintre emeléséhez.

A Szerzők

OOP: angolul object-oriented programming egy programozási módszertan

KATONA JÓZSEF *–KIRÁLY ZOLTÁN **

Objektumorientált szoftverfejlesztés C++ nyelven

(Fejlesztői környezet, osztályok és objektumok)

* *Dunaújvárosi Főiskola, Informatika Intézet*
E-mail: katonaj@mail.duf.hu

** *Dunaújvárosi Főiskola, Informatika Intézet*
E-mail: kiru@mail.duf.hu

1. Fejezet

Általánosságban az objektumorientált programozásról

Az objektumorientált programozás (angolul **object-oriented programming** röviden **OOP**) egy programozási módszertan. Azok a programozási nyelvek, amelyek objektumorientáltak sokkal strukturáltabbak, modulárisabbak és absztraktabbak, mint azok, amelyek csak a hagyományos strukturált programozást teszik lehetővé. Az objektumorientált szemléletmód megértéséhez új gondolkodásmódra van szükség. Amíg a módszer nem született meg, addig a műveletek tervezése és megalkotása állt a középpontban, ahol a programokat különböző feladatok végrehajtásáért felelős függvényekből építették fel, illetve a függvényeken belül különböző vezérlési szerkezetek (mint például az elágazások) kaptak szerepet. Később a felhasználói igények és az egyre nagyobb programméretek következtében a strukturált programozás már nem tette lehetővé azt, hogy könnyen megtervezzük és átlássuk a forráskódokat, sőt az egyre hosszabb élettartamú programok miatt, a karbantartás és **support**olás is fontossá vált. Új szemléletmód megalkotására volt szükség. Fontossá vált a kapcsolatban álló programegységek hierarchiájának tervezése és megalkotása. Ez a gondolkodásmód elsősorban a valós világ modellezésén alapul. Képzeljünk el egy teljesen hétköznapi eszközt – egy autót – amely felfogható egy osztály tagjaként, annak egyik objektumaként. Minden autó-objektum rendelkezik az autókra jellemző tulajdonságokkal (szín, kerekek száma, súly stb.), illetve feladat végrehajtási képességgel (szállítás). Az objektumorientált programozásban fontos szerep jut az úgynevezett öröklődésnek, ami az osztályok egymásból való származtatását teszi lehetővé.

Ismét egy példával élve: a kutyák osztály származhat az állatok osztályból, így megörökli az állatok tulajdonságait és képességeit, valamint kibővítheti vagy felülbírálhatja azokat a kutyák tulajdonságaival, képességeivel. Az alábbiakban megismerkedünk olyan fogalmakkal, amelyek nélkülözhetetlenek az objektum-orientált programozás világában, továbbá megnézzük ezek gyakorlati felhasználási lehetőségeit. Az objektum-orientált megoldások jelentősége az alábbiakban rejlik:
– Az egységbezárás elvét használva lehetővé válik az osztály metódusainak és adattagjainak elrejtése, mely utóbbiak így már csak az adatokat kezelő tagfüggvények segítségével manipulálhatóak (**encapsulation**).
– A nagyméretű programok esetében szükségessé válhat az adatrejtés (**data hiding**).
– Lehetőség nyílik az operátorok túlterhelésére (**operator overloading**).
– Az öröklődés által nyújtott osztályhierarchia felépítése is lehetséges (**inheritance**).
– Többszörös öröklődés (**multiple inheritance**).
– A származtatás során a kiterjesztett osztályok öröklik az alaposztály (ős vagy szülő) adattagjait és tagfüggvényeit, de lehetőség van ezek megváltoztatására is (lehetőség van új tagok hozzáadására, a tagfüggvények túlterhelésére).
– Korai kötés, amikor is a fordítás során megvalósított az összerendelés a metódushívások között (**early binding**).
– Amikor a program futása során dől el, hogy melyik tagfüggvényt kell aktivizálni késői kötésnek nevezzük (**late binding**).
– C++ egyik jellegzetességének nevezzük az absztrakt (elvont) osztályokat (**abstract class**), amelyeket használhatunk egyfajta felületként (**interface**), illetve más osztályok bázisosztályaként is. Az ilyen jellegű típusleírások célja „absztrakt", azaz üres virtuális függvények – más néven pure virtuális függvények – egységbe gyűjtése. Ezeket az üres virtuális metódusokat később az implementálandó osztályban kötelező jelleggel meg kell valósítanunk. Az ilyen típusú osztályból objektumpéldány nem készíthető, azonban objektum-referencia igen.
– Egy objektumot több osztály példányaként is használhatók. (Polymorphism).

Statikus modelltervezés

Ahhoz, hogy elkezdjük az osztályok implementálását, esetleges integrálását, szükséges őket megtervezni. Az ember vizuális lény, így olyan UML (**Unified Modelling Language** – Egységes Modellező Nyelv) diagramokat fejlesztett ki, amelyek segítségével szemléltethetjük egy program (időtől független) strukturális felépítését, illetve egy adott időpontban meglévő objektumainak kapcsolatrendszerét.

Ezeket a modelleket statikus (avagy strukturális) diagramoknak nevezzük. A programok megalkotása során ezeket a diagramokat fogjuk használni. Természetesen egy programot nem csak egy adott időpillanatban lehet vizsgálni, hanem egy úgynevezett időtérben is, de ennek a szemináriumnak nem az a célja, hogy különböző állapotokat és együttműködéseket mutasson be, hanem az idő egy adott pillanatára koncentrálva megalkotni a valós világ modelljét. A két statikus diagram, amellyel foglalkozni fogunk az osztály- és objektumdiagram, amelyek a következőekben röviden bemutatásra kerülnek.

Osztálydiagramok (Class diagram)

Az osztálydiagram a problématérben a megoldás szerkezetét leíró, összefüggő gráf, amelynek
– csomópontjaihoz az osztályokat;
– éleihez pedig az osztályok közötti relációkat rendeljük.
(A gráf összefüggő, hiszen az osztályoknak kapcsolódniuk kell egymáshoz, különben független rendszert alkotnának. Két osztályt ugyanakkor több él is összeköthet, mert több reláció is lehet közöttük, illetve egy relációhoz több él is tartozhat.)

A relációk osztályok, illetve azok objektumai közötti kapcsolatot fejeznek ki. Az osztályok között a következő relációk állhatnak fenn:
– öröklődés;
– asszociáció;
– aggregáció;
– kompozíció.

Az öröklődés osztályok közötti kapcsolat, a másik három a résztvevő osztályok objektumait kapcsolja össze.

A szabványos osztálydiagramban az osztály jelölése a következő:

Osztály neve
<láthatóság> <adattag neve> : <típus>
<láthatóság> <tagfüggvény neve> (<esetleges paraméterlista>) : visszatérési érték

Az osztályoknak létezik azonosítójuk, amelyeket – a programozás világában – nagybetűvel szokás kezdeni. Az osztály neve alatt helyezkednek el az adattagok (tagadatok), amelyek az objektum tulajdonságaiért felelnek, illetve az adattagok alatt – az objektumok képességeit megvalósító – tagfüggvények találhatóak.

Objektumdiagramok (Object diagram)

Az objektumdiagram egyszeresen összefüggő gráf, amelynek
– csomópontjaihoz az objektumokat;
– éleihez pedig az objektumok közötti összekapcsolásokat rendeljük.

A rendszerhez egy osztálydiagram tartozik, ugyanakkor egy osztálydiagramhoz több objektumdiagram tartozhat. Mindegyik objektumdiagramnak összhangban kell lennie az osztálydiagrammal. Az objektumok a rendszer működése során dinamikusan jönnek létre, változnak és szűnnek meg, ezért az idő függvényében változó, különböző állapotokat vehetnek fel. Az osztálydiagram a rendszer egész idejére jellemző. Az objektumdiagramban az osztályok helyébe azok példányai, az objektumok kerülnek. Az összekapcsolások az osztálydiagramban szereplő relációk példányai. Az objektumdiagram összekapcsolásai multiplicitás nélküliek, mert az osztálydiagramban szereplő relációk multiplicitásának megfelelő számú objektum jelenik meg az adott helyen. Az öröklődési reláció nem jelenik meg az objektumdiagramban, hiszen ebben a diagramban konkrét objektumok szerepelnek.

A szabványos objektumdiagramban az objektum jelölése a következő:

<objektumazonosító> : <osztály neve>
<adattag> = <aktuális állapot>

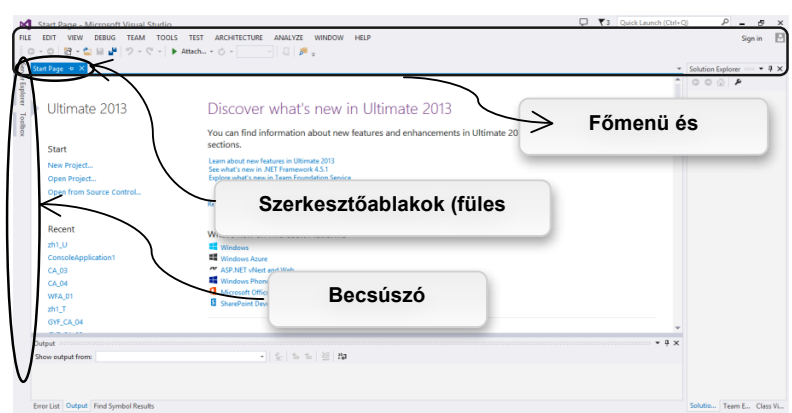
Az ábra felső részében helyezkedik el az objektum neve (azonosítója), illetve annak az osztálynak a neve, amelyből az objektum létrejött. Az ábra alsó részében az objektum adott pillanatban meglévő állapotát jelölhetjük attribútumai aktuális értékének feltüntetésével.

A fejlesztő környezet rövid ismertetése

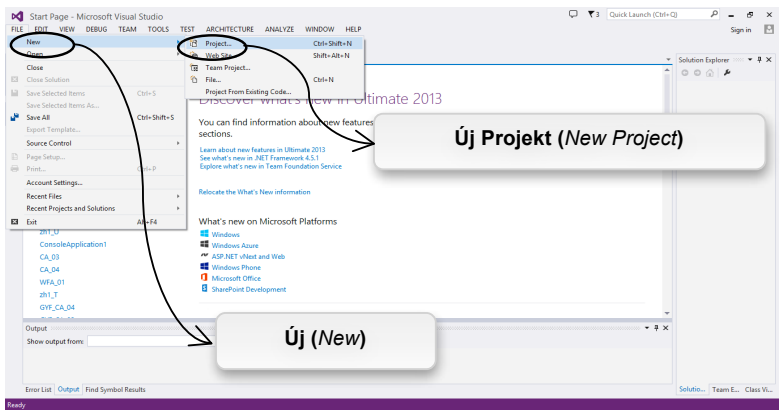
A szemináriumi órák keretein belül az úgynevezett **Microsoft Visual Studio** fejlesztőkörnyezetet használjuk. Az elmúlt évek során számos bővítményt és frissítést kapott, sőt a fejlesztői csomag napjainkig támogatott és több mint valószínű, hogy még sokáig ez a fejlesztői környezet lesz a legprofibb termékcsomag Windows operációs rendszerre. Ezen tárgy kereteiben azokkal a legfontosabb fejlesztőikörnyezet beállításokkal ismerkedünk meg, amelyek nélkülözhetetlenek. A tárgy programozási nyelve az objektum orientáltságot támogató C++ lesz. Először az úgynevezett **Managed C++**–t vesszük górcső alá, majd kiterjesztjük ismereteinket a **C++/CLI** nyelvre. A szabványos nyelvi elemek megismerését úgynevezett konzolos alkalmazások segítségével végezzük. Amennyiben Windows alatt úgynevezett konzolalkalmazást készítünk, a szabványos C++-nak megfelelő szintaktikát alkalmaz a Visual Studio. Így fordíthatóak le a Unix, Mac vagy más rendszerre készített programok, programrészletek. A konzolos alkalmazásaink során több C++ kiterjesztésű (.cpp) fájlt és **header** fájlt (.h) fogunk készíteni. Természetesen a logikailag összetartozó részeket külön is elhelyezhetjük, például csapatmunka során. A fejlesztés során számos elő feldolgozót (preprocesszor) fogunk felhasználni, mint például a **#define**ok feloldásra vagy **#include**ok beszúrásra. Az alkalmazásunknak mindig lesz egy belépési pontja az úgynevezett (main) függvény, ahol a program futása fog kezdődni.

Új Managed C++ Project létrehozása Visual Studio környezetben

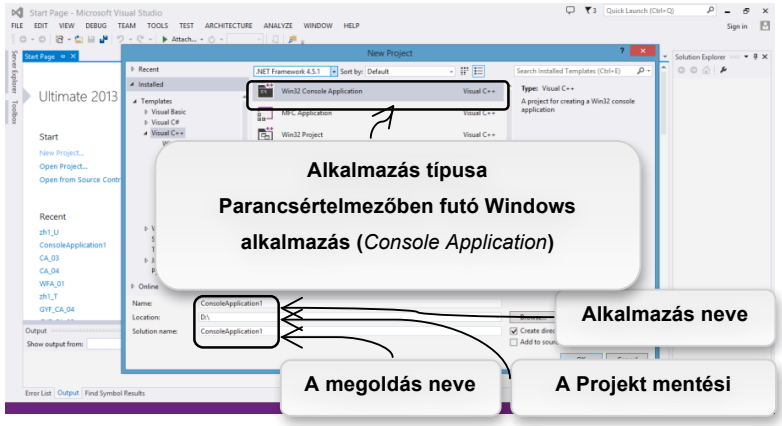
A fejlesztő környezetünk kezdő képernyője az alábbi ábrán látható.



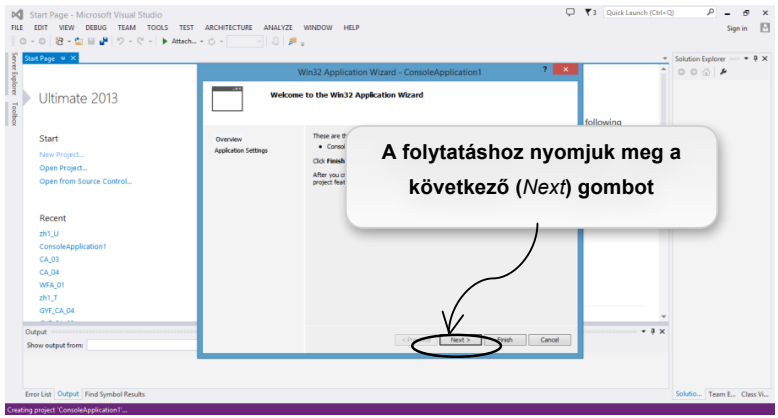
A legtöbb fejlesztőrendszerben a programkészítés alapját egy ún. *Projekt összeállítás*a adja. Ha új Project-et szeretnénk létrehozni, akkor válasszuk ki a fájl menüben az New menüpontot, majd kattintsunk **Project** almenüre, de ezt egyszerűen, akár a Ctrl+Shift+N gyorsindító gombkombinációval is megtehetjük.



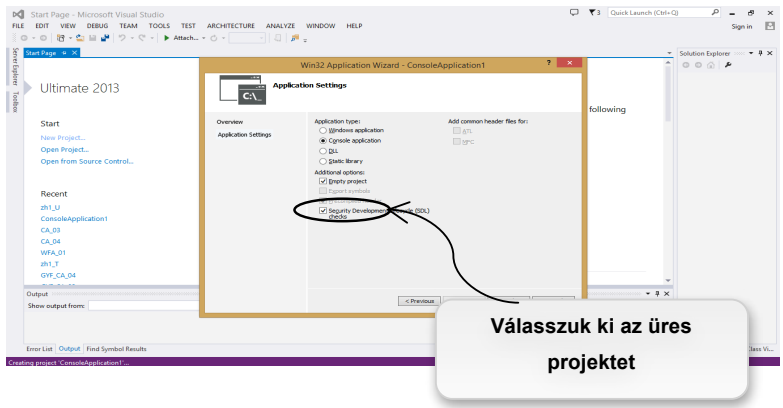
A Visual C++ rendszer számos lehetősége közül a Win32 konzolalkalmazás az egyszerű, szöveges felületű C++ alkalmazások típusa. Az új projekt kiválasztása után válasszuk ki a Visual C++ nyelvet, azon belül a Win32-es alkalmazást. A Win32-es alkalmazáson belül válasszuk ki **Win32 Console Application** applikációt. A Visual Studio-ban gyakran előfordul, hogy egy-egy részfeladaton több fejlesztő is dolgozik, amelynek a produktuma lehet maga a programtermék. A programtermék sok részből tevődik össze, ezeket a részeket egységes rendszerré kell összefogni. Ezeket az összefoglalt egységeket **Solution**nek (megoldásnak) nevezzük. A megoldás lényegében magában foglalja a készülő programtermék alkotó elemeit. Tehát a **Solution** beviteli mezőben kell megadnunk a programtermék nevét. A megoldás összetevőit, a **Name** beviteli mezőben adhatjuk meg. Ezeket az összetevőket **Project**nek nevezünk. Természetesen a megoldás egy vagy akár több projektet is tartalmazhat, amíg a projekt az éppen adott alkalmazáshoz szükséges fájlokat tartalmazza. A **Location** résznél beállíthatjuk a projekt mentési helyét.



Az **OK** gombra kattintást követően elindul a konzolalkalmazás-varázsló, ahol egy üres projectet készítünk.



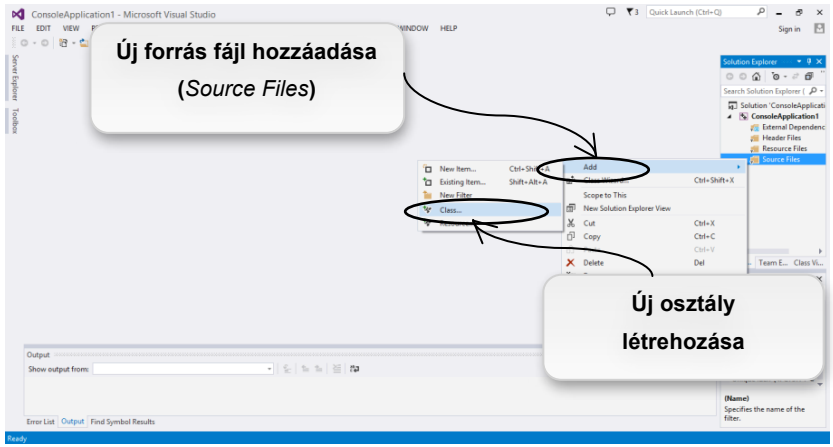
A **Next** gombra kattintva az applikáció varázsló jelenik meg, ahol beállíthatjuk az alkalmazás típusát – esetünkben ez a **Console Application** – illetve a további beállítási lehetőségek közül válasszuk ki az **Empty project**et.



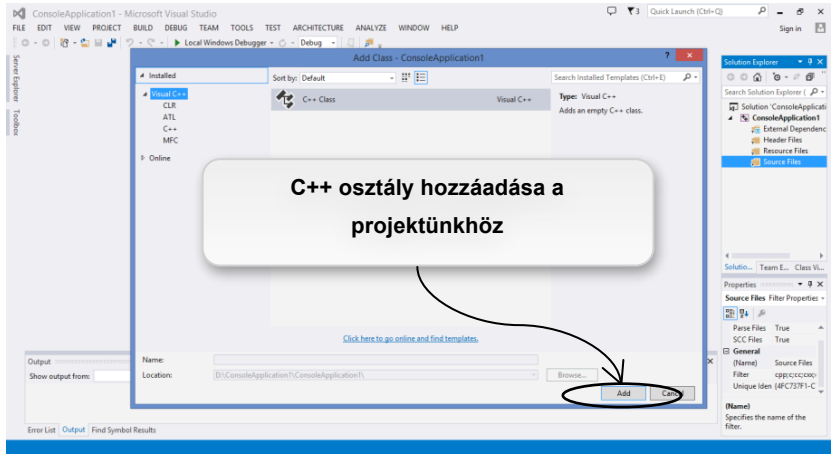
A **Finish** gomb megnyomását követően megjelenik a megoldás ablaka (**Solution Explorer**), ahol a **Source Files** elemre kattintva az egér jobb gombjával, új forrásállományt adhatunk a projekthez (**Add/New Item…**). Ezt a tevékenységet gyorsabban a Ctrl+Shift+A billentyűkombináció hármassal is elérhetjük, valamint lehetséges új forrásállomány hozzáadása a **Project** menüpont alatt, az **Add New Item** lehetőségre kattintva.

Managed C++ osztály (állomány és header fájl) létrehozása Visual Studio környezetben

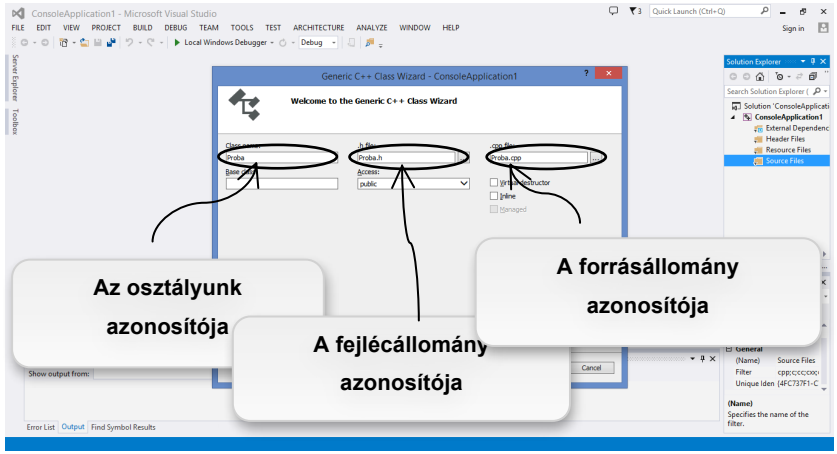
A mi esetünkben ne az új elemet, hanem az osztályt válasszuk.



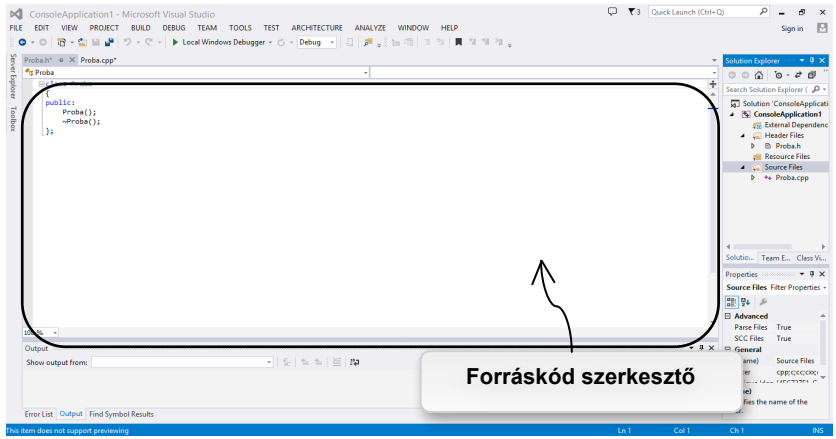
A menüpontra kattintva az alábbi varázslóval találjuk szembe magunkat.



Az **Add** gombra kattintva az osztályt generáló varázsló jelenik meg, ahol megadhatjuk az osztályunk, **header** file-unk és C++ állományunk nevét, illetve további beállításokra is lehetőség van, de számunkra az alapértelmezett beállítások megfelelőek.



A **Finish** gombra kattintva a forráskód szerkesztőben találjuk magunkat, ahol elkezdhetjük a munkánkat.



Osztályok deklarálása, implementálása és példányosítása általánosságban és nyelvspecifikusan

Maga az osztály egy típus leírása, amelynek alapján példányok (objektumok) hozhatók létre. Az osztályok alapvetően adattagokból és tagfüggvényekből épülnek fel. Ezekhez a tagokhoz való hozzáférést korlátozhatjuk, vagy épp engedélyezhetjük, attól függően mire van szükségünk, de általánosan elfogadott szabály az egységbezárás, amely szerint az osztály adattagjait **private** láthatósággal, amíg a különböző feladatok végrehajtásáért felelős tagfüggvényeket **public** jelleggel látjuk el. A következő három láthatóságot (jelleget) különböztetjük meg:

– **Public** (*nyilvános*): Ennek a kulcsszónak a jelentősége az, hogy a deklarációk és definíciók bármely további osztályban szabadon elérhetőek. Ahhoz, hogy az adatelrejtés elve érvényesüljön ajánlott, hogy **public** elérhetőséggel csak az osztály tagfüggvényeit deklaráljuk. A szabványos osztálydiagramban a jelölésük „+" karakterrel történik.

– **Private** (*magán*): Ennek a láthatóságnak a jelentősége, hogy a további deklarációk és definíciók csak az őt tartalmazó osztályban – illetve az osztály „barátaiból" (**friend**) – láthatóak és érhetőek el. A szabványos osztálydiagramban a jelölésük „-" karakterrel történik.

– **Protected** (*védett*): A kulcsszó jelentősége az öröklődés folyamán mutatkozik meg. A deklarációk és definíciók csak az őt tartalmazó osztályban (ős vagy szülő), illetve a belőlük leszármaztatott osztályokban érhetőek el. A szabványos osztálydiagramban a jelölésük „#" karakterrel történik.

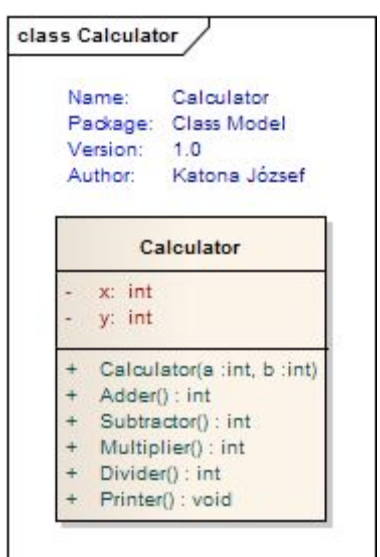
Osztály létrehozása a **class** kulcsszóval történik, további általános szabály – a programozás világában – az, hogy az osztály neveit nagybetűvel kezdjük. Az osztályokat azonban valamilyen módon életre is kell keltenünk, amelyekhez kiváló szolgálatot nyújt az úgynevezett objektum. Az objektum orientált programozás szépsége nem csak osztályokban és objektumokban mutatkozik meg, hanem az osztályokon végigmenő folyamaton is, mint az öröklődés, amely egy szép, egységes hierarchiát alkothat. Ezeken az osztályokon lépésről, lépésre haladva vagyunk képesek eljutni ahhoz az osztályhoz, amely számunkra a megfelelő képességű. Az osztály példányosítása történhet statikus, illetve dinamikus módon. A statikus objektumpéldány létrehozásánál nincs más dolgunk, mint az osztály neve után megadni az objektum azonosítót. Több statikus objektumpéldány definiálása során a memóriában csak az adattagok többszöröződnek, így minden egyes objektumpéldány saját memóriaterülettel rendelkezik. Azonban az objektumok képességeit megvalósító metódusok csak egyetlen példányban kerülnek a memória területre és minden példány közösen használja azokat. Ha egy statikus objektumpéldányon keresztül szeretnénk elérni az osztály nyilvános adattagjait, akkor azt az objektum neve után megadott pont (.) operátorral tehetjük meg. C++–ban operátorok felelnek a dinamikus memóriakezelésért. Dinamikus objektumpéldány létrehozásánál szükség van a **new** operátorra, amely segítségével dinamikusan tudjuk allokálni a memóriát.

A **new** operátor amellett, hogy lefoglalja a szükséges memóriaterületet, a lefoglalt típusra mutató pointerrel tér vissza, ez objektumpéldány esetében a példányra mutató pointert jelenti. Amíg a statikus objektumpéldánynál a nyilvános jellegű adattagokat és tagfüggvényeket a példány után írt pont (.) operátorral érjük el, addig a dinamikusan létrehozott pointer segítségével elérhető objektum esetén ez a nyíl (->) operátorral lehetséges. (A nyíl (név->) operátor tulajdonképpen a (*név) operátor kombinációjával analóg, azt helyettesítő, egyszerűsítő operátor.) A lefoglalt területet a **delete** operátorral szabadíthatjuk fel. Több példány létrehozása esetén használhatjuk a **this** objektumreferenciát, amely mindig az aktuális objektumpéldányra mutat. A **this** kulcsszó jelentősége akkor mutatkozik meg, ha a konstruktor paraméterlistájának a neve megegyezik az osztály adattagjainak az azonosítóival, ekkor a fordító a **this** referenciamutató segítségével dönti el, hogy melyik az osztály adattagja, ami az aktuális objektumhoz tartozik, illetve melyik az argumentumlistabeli változó.

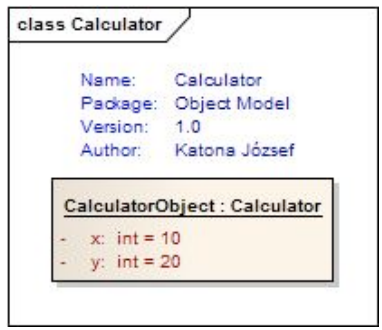
Mintapélda

Alkossunk egy konzolos alkalmazást. Hozzunk létre egy egyszerű osztályt, amely képes a négy alapvető aritmetikai művelet végrehajtására. (CA_01)

Osztálydiagram (Class diagram)



Objektumdiagram (Object diagram)



Az osztály deklarációját a (*Calculator.h*) állományban adjuk meg. Az osztály deklarálása előtt szerepel egy úgynevezett #pragma once előfordító direktíva. A direktívára azért van szükség, hogy több fájlban tudjuk használni ugyanazt a **header**t. Fontos megjegyezni, hogy ez a direktíva nem szabványos, tehát nem minden fordító tudja lefordítani (figyelmen kívül hagyja), de a Visual C++ esetén működik. A **class** kulcsszavat felhasználva létrehozzuk az osztályt. Az osztályunk neve **Calculator**. Az osztály törzsében deklarálunk két magánjellegű változót, amelyek az osztály adattagjai lesznek. Ha nem írjuk ki a láthatóságot, akkor az alapértelmezetten **private** jellegű. Később megfogalmazzuk a metódusokat. Pontosabban a függvény fejeit hozzuk létre (tagfüggvény deklarációi), majd az osztályon kívül kifejtjük őket, ennek az ellenkezője az **inline** módszer, amely azt jelenti, hogy még az osztály törzsében kifejtjük a függvények feladatait. A metódusok **public** láthatósággal rendelkeznek, amely lehetővé teszi azok elérhetőségét az osztályon kívülről. Az osztály törzsét (;) karakterrel zárjuk.

A **Calculator** osztály implementációs (*Calculator.cpp*) állománya a következő részeket tartalmazza: Az első két sorban behívjuk a szükséges fájlokat. Az #include <iostream> deklarálja a C++ alapvető adatfolyam I/O rutinokat. Az #include "Calculator.h" tartalmazza a szükséges osztály deklarációkat. De mi is a különbség a két különböző jelölés között?

– **<file_nev>**: a fordítóhoz tartozó szabványos **include** könyvtárban keres először a preprocesszor.
A könyvtári ún. **header** fileokhoz (deklarációs állományokhoz) használjuk.

–**„file_nev"**: a saját, pl. aktuális könyvtárban keres először. Saját **header** fileokhoz szoktuk használni.

Tovább haladva a kódon az első függvényünk az úgynevezett *konstruktor*, amely a példában paramétereseként szerepel. Itt álljunk is meg egy szóra.

– Mi is az a konstruktor?
– Mire szolgál?
– Milyen tulajdonságokkal rendelkezik?

A konstruktort tekinthetjük egy olyan speciális függvénynek, melynek a törzsében lévő kód akkor fut le, amikor az adott osztályból létrehozunk egy példányt. A konstruktor általános feladata az osztály adattagjainak inicializálása. A legfontosabb tulajdonságai a következőek:
– A neve egyezzen az Osztály nevével.
– Nincs visszatérési értéke, még void sem.
– Mint minden más metódus a konstruktor is túlterhelhető, így eltérő paraméterezésű konstruktor is létrejöhet.
– Ha nincs általunk létrehozott konstruktor, akkor a fordító elkészíti az alapértelmezett konstruktort.
– Nem öröklődhet.

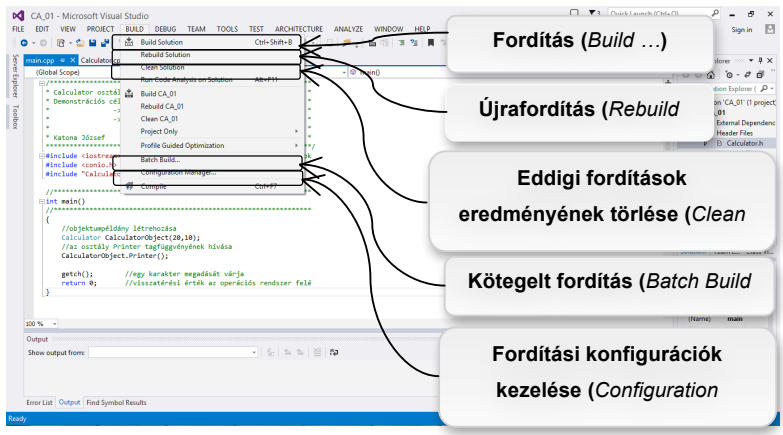
A továbbiakban szerepel négy függvény, amelyeknek feladatai a négy alapvető aritmetikai művelet végrehajtása. A *Printer()*-függvény segítségével íratjuk ki a konzolra. Az **std**-névtér segítségével vagyunk képesek elérni a cout-függvényt, amely a szövegünk kiírását segíti. Az **std**-névtérben elérhető még az **endl**-utasítás, amely sortörést tesz lehetővé.

A **Calculator** osztály példányosítását végrehajtó (main.cpp) állomány tartalmazza az osztály példányait és a szükséges függvényhívásokat. Az #include <conio.h> a DOS konzol I/O rutinok hívásában használt különféle függvényeket deklarál, például tartalmazza a **_getch()** függvényt, amelyet arra használunk, hogy meggátoljuk a konzolunk azonnali bezáródását a programunk lefutása után (azáltal, hogy a függvény egy karakter megadására vár.) A függvények kifejtését befejezvén létrehozásra kerül a **main()**-függvény, amelyben készítünk egy objektumot. Az objektum neve tetszőleges, de ajánlott valami beszédes névvel ellátni. A példában használt objektumpéldány statikus helyfoglalású, ami a már fentebb is leírt tulajdonságokat írja elő:
– A program adatterületén hozza létre a fordító;
– Ha több példányunk van, akkor az adattagokat többszörözi, így minden példány saját adatterülettel rendelkezik;
– A tagfüggvények egyetlen példányban léteznek és minden példány azokat használja.

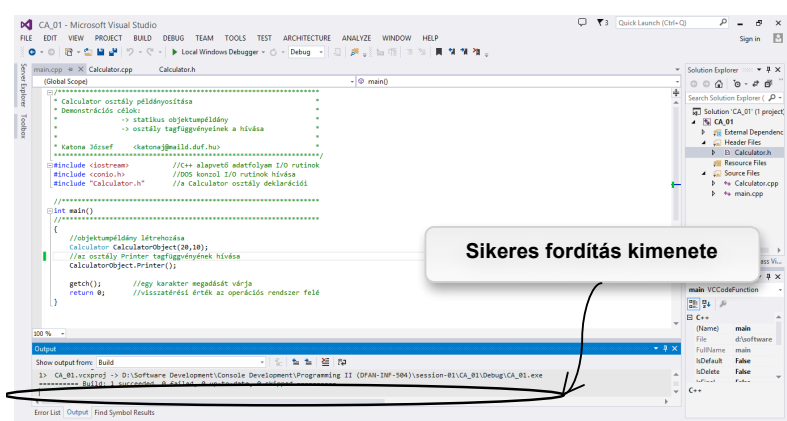
A példányosítás során látjuk el bemeneti paraméterértékekkel a konstruktort és ebben a speciális esetben adunk át kettő darab egész típusú számot. Az utolsó lépések között szerepel a kiíró függvény meghívása, amelyet az objektum segítségével érünk el. A kód végén szereplő **return 0** az operációs rendszer felé adja vissza ezt az értéket, amely jelentése lehet például, hogy sikeresen lefutott a programunk.

A program szövegének beírása után a fordításhoz a **Build/Build Solution** vagy a **Build/Rebuild Solution** menüpontokat használhatjuk. A **Build**elést egyszerűen az F6 billentyű lenyomásával is végrehajthatjuk. A **Build** menüpont alatt további számos és hasznos almenüpont található:
– **Batch Build** (*kötegelt fordítás*): Az almenüre kattintva egy dialógus-ablakkal találjuk szembe magunkat, ahol egyszerre több projektet konfigurálhatunk. Eldönthetjük például, hogy a fordításparancs kiadása után mely projektek legyenek azok, amelyek egy időben fordulnak;

– **Rebuild** (*újrafordítás*): A projekt ismételt – újra – fordítása.
– **Configuration Manager** (*fordítási konfigurációk*): Az itt megjelenő dialógus-ablakban megoldás és projektbeállítási lehetőségeket érhetünk el. Az egyik ilyen beállítási lehetőség azon platform kiválasztása, amely platformra szeretnénk lefordítani a megoldásunkat vagy projektünket.



Sikeres fordítás esetén a kimeneti ablakban a következő feliratot kell kapnunk **Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped.** Érdemes a **View** menüpont alatt az úgynevezett **Error list**–et is bekapcsolni, mert az esetleges fordítási hibák itt is megjelennek, és több információval szolgálhatnak, segítve ezzel a hiba feltárását.

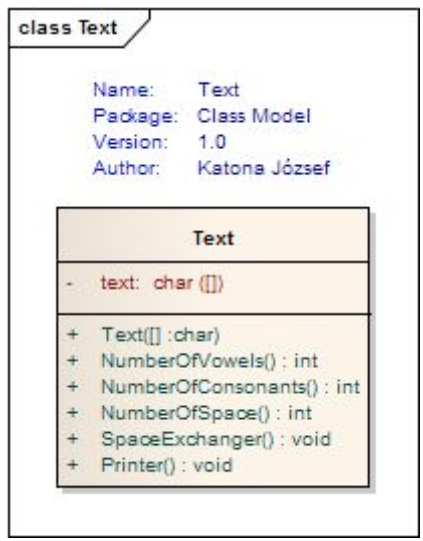


A **Debug/Start Debugging (F5)** vagy a **Debug/Start Without Debugging (Ctrl+F5)** menüválasztással indíthatjuk a programot. A **Build/Configuration Manager** ... menüválasztás hatására megjelenő párbeszédablakban megadhatjuk, hogy nyomon követhető (**Debug**) vagy pedig végleges (**Release**) változatot kívánunk fordítani. (Ez a választás meghatározza a keletkező futtatható állomány tartalmát, illetve a helyét a lemezen.) Bármelyik **Build** (*felépítés*) választásakor a fordítás valójában több lépésben megy végbe. Az elő feldolgozó értelmezi a kettős kereszttel (#) kezdődő sorokat, melynek eredményeként keletkezik a C++ nyelvű forráskód. Ezt a kódot a C++ fordító egy olyan tárgykóddá fordítja, amelyből hiányzik a könyvtári elemeket megvalósító gépi kód. Utolsó lépésben a szerkesztő pótolja ezt a hiányt, és futtaható alkalmazássá alakítja a már teljes gépi (natív) kódot. Több forrásfájlt (modult) tartalmazó projekt esetén a fordítást modulonként végzi az elő feldolgozó és a C++ fordító, majd az így keletkező tárgymodulokat a szerkesztő építi egyetlen futtatható állománnyá. A futtatást követően még el kell mentenünk az elkészült programot, hogy a későbbiekben ismét tudjunk vele dolgozni. A sokféle megoldás közül a következő bevált lépéssor lehet segítségünkre: először minden fájlt lemezre mentünk (**File/Save All**), majd pedig lezárjuk a projektet a megoldással együtt. (**File/Close Solution**). Végezetül nézzük meg a projekt fordításakor keletkező könyvtárstruktúrát a merevlemezen! A magasabban elhelyezkedő **Debug**-illetve **Release**-könyvtárak tartalmazzák a kész futtatható alkalmazást, míg a mélyebben fekvő, azonos nevű könyvtárakban munkafájlokat találunk. Ez a négy mappa törölhető, hisz fordításkor ismét létrejönnek. Ugyancsak ajánlott eltávolítani a fejlesztői környezet **intellisense** szolgáltatásait segítő CA_01.ncb fájlt, melynek mérete igen nagyra nőhet. A megoldás (projekt) ismételt megnyitását a CA_01.sln állománnyal kezdeményezhetjük (**File/Open/Project/Solution**).

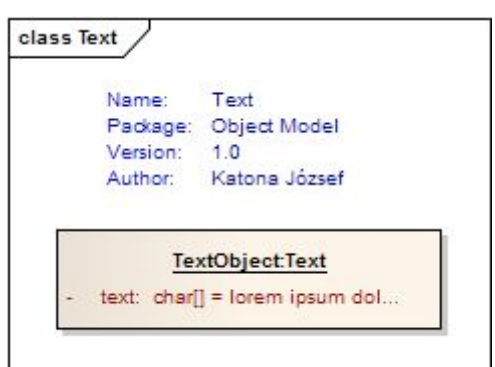
Írjunk egy olyan osztályt, amelynek van egy paraméteres konstruktora. A konstruktoron keresztül adjunk át egy rövidebb szöveget. A konstruktor feladata legyen az osztály adattagjának inicializálása. Az osztály legyen képes a következő adatokat szolgáltatni: magánhangzók, mássalhangzók és szóközök száma. Az osztály egy függvénye cserélje le a szóközöket #-re. Az egyes műveletek végrehajtásához külön függvényeket használjon, amelyeknek visszatérési értéke a feladattól függ. Az osztály függvényeinek meghívását objektumon keresztül végezze. Az eredményt egy konzolos kiíratással szemléltesse. (CA_02)

Mintapélda

Osztálydiagram (Class diagram)



Objektumdiagram (Object diagram)



Az osztály deklarációját *(Text.h)* állományban adjuk meg. Az előbbi példában beszéltünk az előfordító direktíváról, azonban az nem volt szabványos. A szabványos direktíva használata a következő: makró (**if-not-defined** = ha nincs deklarálva) megnézi, hogy a TEXT_H definiálva van–e. Mivel ez az első eset és ez a név egyedi így a TEXT_H nincs deklarálva és így „belépünk" az **include** file-ba. A második sorban deklaráljuk a TEXT_H–t. Ha ez az **include** file másodszor is beillesztésre kerül, akkor az **#ifndef** makró azt fogja találni, hogy a TEXT_H már deklarálva van és így az egész **include** file-t „átugorja", a definíciókat nem hajtja végre még egyszer, ezáltal nem fogunk hibaüzenetet kapni az újradefiniálásra vonatkozóan. Az **#ifndef** makró párja az **#endif** amely lezárja azt. Az „átugrás" azt jelenti, hogy az **#endif** makró utáni sorra ugrik a fordító. Az **#endif** makrót mindenképpen ki kell adni, mert az zárja le az **#ifndef** makrót. A **class** kulcsszavat felhasználva megalkotjuk az osztályt. Az osztályunk neve **Text**. Az osztály törzsében deklarálunk egy magánjellegű változót, amely az osztály adattagja. Később megfogalmazzuk a metódusok prototípusait.
Pontosabban a függvények fejeit hozzuk létre (tagfüggvény prototípus), majd az osztályon kívül kifejtjük őket. A metódusok **public** láthatósággal rendelkeznek. Az osztály törzsét (;) karakterrel zárjuk.

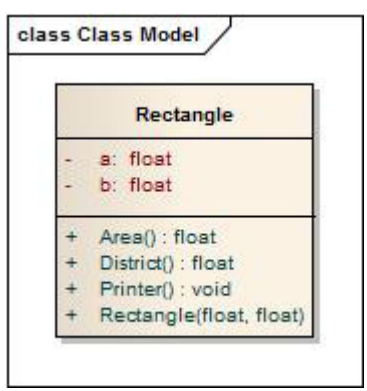
A (*Text.cpp*) állomány a következő sorokat tartalmazza: A #include <conio.h> a DOS konzol I/O rutinok hívásában használt különféle függvényeket deklarál. A #include "Text.h" tartalmazza a szükséges osztály deklarációkat. Tovább haladva az első függvényünk a konstruktor, amely a példában ismét paraméteresként szerepel. A továbbiakban szerepel öt függvény, amelyeknek feladatai rendre a magánhangzók, mássalhangzók és a szóközök összeszámolása, továbbá a negyedik metódus az összes szóközt lecseréli „#" karakterre. A metódusoknál használt strlen()-függvény a szövegünk hosszát adja vissza. A Printer()-függvény segítségével írunk ki a konzolra.

A (main.cpp) állomány tartalmazza az osztály példányait és a hívásokat. Létrehozásra kerül a **main()**-függvény, amelyben készítünk egy objektumot. A példában használt objektumpéldány statikus helyfoglalású. A példányosítás során látjuk el bemeneti értékkel a konstruktort és ebben a speciális esetben átadunk neki egy rövidebb szöveget. Az utolsó lépések között szerepel a kiíró függvény meghívása, amelyet az objektum segítségével érünk el. A kód végén szereplő **return 0** az operációs rendszer felé jelzi a program sikeres futását.

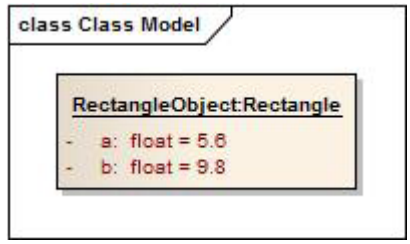
Gyakorló példák

Írjunk egy olyan osztályt, amelynek van egy paraméteres konstruktora. A konstruktoron keresztül adjunk át az osztálynak két darab lebegőpontos számot (ezek lesznek egy téglalap oldalai). A konstruktor feladata legyen az osztály adattagjainak inicializálása. Az osztály legyen képes a két szám értékből kiszámolni egy téglalap területét és kerületét. Az egyes műveletek végrehajtásához külön függvényeket használjon, amelyeknek visszatérési értéke valós típusú (float). Az osztály függvényeinek meghívását objektumon keresztül végezze. Az eredményt egy konzolos kiíratással szemléltesse. (GYF_CA_01)

Osztálydiagram (Class diagram)

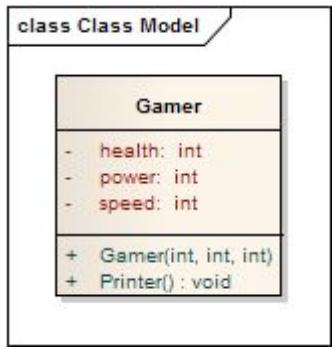


Objektumdiagram (Object diagram)

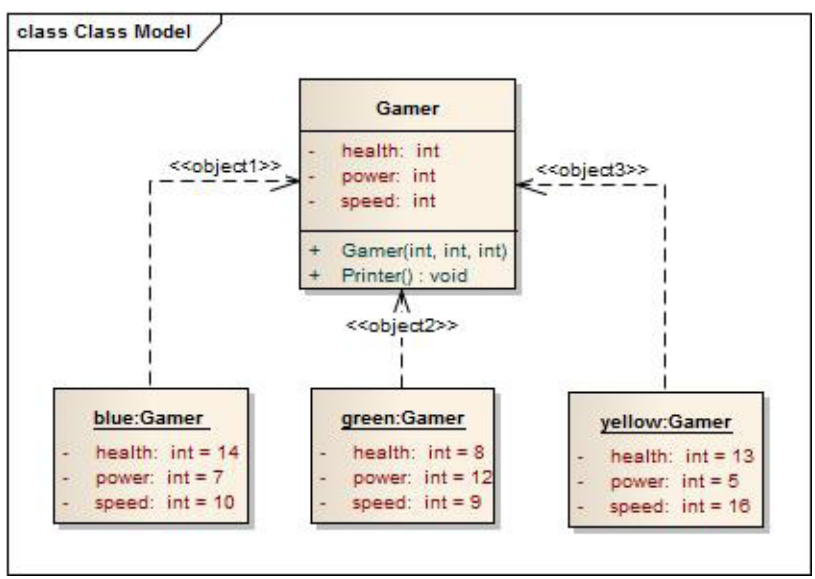


Hozzunk létre egy olyan osztályt, amely egy a valósvilágból vett játékost modellez. A játékosnak legyen egészsége, ereje és gyorsasága. Az egységbezárás elvét figyelembe véve az adattagokat private **láthatósággal lássuk el. Az osztály adattagjait paraméteres konstruktor segítségével inicializáljuk. Ne feledkezzen el a this-objektum-referenciáról. Az osztálynak legyen egy olyan tagfüggvénye, amely ezeket az értékeket kiírja. Három játékost (objektumot) hozzon létre és írja ki a konzolra az egészségének, erejének és gyorsaságának az értékeit. (GYF_CA_02)**

Osztálydiagram (Class diagram)

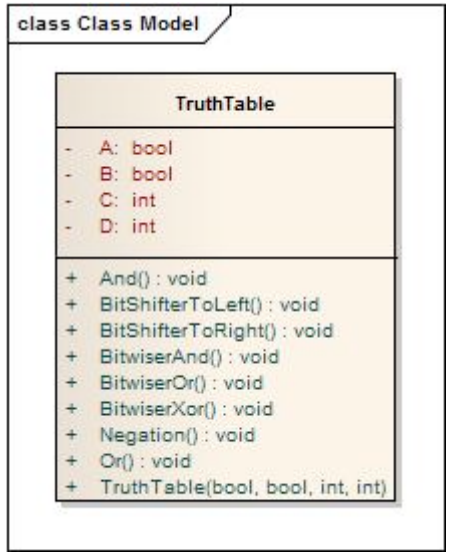


Osztály – Objektumdiagram (Object diagram)

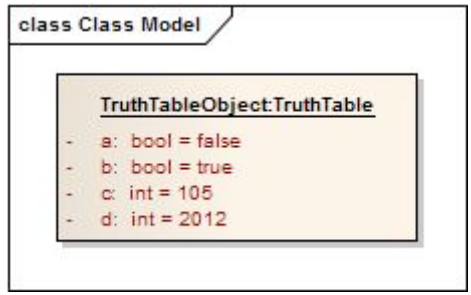


Hozzon létre egy TruthTable-osztályt. Az osztály legyen képes elvégezni az és, vagy és a negációs logikai műveleteket két operanduson. További metódusok hajtsák végre a bitenkénti és, a bitenkénti vagy és a bitenkénti kizáró vagy műveleteket két tízes számrendszerbeli számon, amelyet az osztály szintén a konstruktortól kap meg. Az osztály legyen még képes biteltolást végezni a két tízes számrendszerbeli számon, mind jobbra, mint balra egy-egy bittel. Az eredményeket bináris és decimális formában is írassa ki. (GYF_CA_03)

Osztálydiagram (Class diagram)

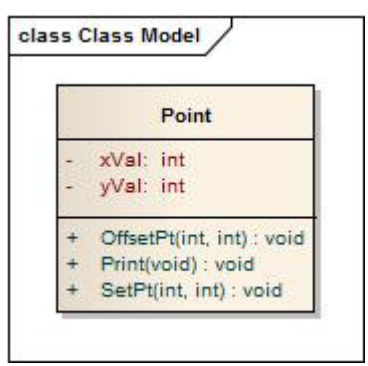


Objektumdiagram (Object diagram)

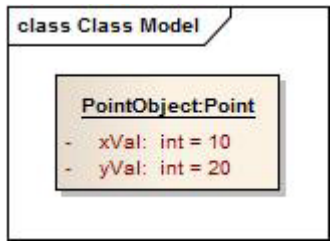


Hozzon létre egy Pont osztályt, ahol két pont értékeit tárolja, illetve megnöveli egy függvény segítségével. Az osztály adattagjait konstruktor nélkül inicializálja. Legyen egy olyan függvény is, amely a konzolos kiíratásért felelős. (GYF_CA_04)

Osztálydiagram (Class diagram)



Objektumdiagram (Object diagram)



Forráskódok

Calculator-osztály deklarációja (Calculator.h)

```
/*******************************************************************
* Calculator osztály deklarációja                                  *
* Demonstrációs célok:                                             *
*                -> egységbezárás: class                           *
*                -> adattagok                                      *
*                -> konstruktor                                    *
*                -> tagfüggvények                                  *
*                -> láthatóságok                                   *
*                                                                  *
* Katona József     <katonaj@mail.duf.hu>                          *
*******************************************************************/
#pragma once               //előfordító direktíva (nem szabványos!)
//==================================================================
class Calculator
{
//==================================================================
    //kívülről nem elérhető adattagok (ha nem írjuk ki a láthatóságot
    //alapértelmezetten private, a további példákban már nem írom ki)
    private:
        int x, y;                 //adattagok

    public:                      //kívülről is elérhető tagfüggvények
        Calculator(int a, int b);//paraméteres konstruktor
        int  Adder();            //Adder függvény deklaráció
        int  Subtractor();       //Substractor függvény deklaráció
        int  Multiplier();       //Multiplier függvény deklaráció
        int  Divider();          //Divider függvény deklaráció
        void Printer();          //Printer függvény deklaráció
};
```

Calculator-osztály implementálása (Calculator.cpp)

```
/*******************************************************************
* Calculator osztály definiálása/implementálása                    *
* Demonstrácios célok:                                             *
*                -> this referenciamutató                          *
*                -> adattagok  inicializálása                      *
*                -> :: hatókör operátor                            *
*                -> tagfüggvények feladatainak kifejtése           *
*                -> std névtér                                     *
*                                                                  *
* Katona József     <katonaj@mail.duf.hu>                          *
*******************************************************************/
#include <iostream>           //C++ alapvető adatfolyam I/O rutinok
#include "Calculator.h"       //a Calc osztály deklarációi

//-----------------------------------------------------------------
Calculator :: Calculator (int a, int b)
//-----------------------------------------------------------------
{
        x = a;
        y = b;
}

//-----------------------------------------------------------------
int Calculator :: Adder()
//-----------------------------------------------------------------
{
        return this->x + this->y;
}

//-----------------------------------------------------------------
int Calculator :: Subtractor()
//-----------------------------------------------------------------
{
        return this->x - this->y;
}

//-----------------------------------------------------------------
int Calculator :: Multiplier()
//-----------------------------------------------------------------
{
        return this->x * this->y;
}

//-----------------------------------------------------------------
int Calculator :: Divider()
//-----------------------------------------------------------------
{
```

```
        return this->x / this->y;
}

//-----------------------------------------------------------------
void Calculator :: Printer()
//-----------------------------------------------------------------
{
        std::cout << this->x << " + " << this->y << " = " <<
                this->Adder() << std::endl;
        std::cout << this->x << " - " << this->y << " = " <<
                this->Subtractor()  << std::endl;
        std::cout << this->x << " * " << this->y << " = " <<
                this->Multiplier() << std::endl;
        std::cout << this->x << " / " << this->y << " = " <<
                this->Divider() << std::endl;
}
```

Calculator-osztály példányosítása (main.cpp)

```
/*****************************************************************
* Calculator osztály példányosítása                              *
* Demonstrációs célok:                                           *
*                -> statikus objektumpéldány                     *
*                -> osztály tagfüggvényeinek a hívása            *
*                                                                *
* Katona József     <katonaj@mail.duf.hu>                        *
*****************************************************************/
#include <iostream>           //C++ alapvető adatfolyam I/O rutinok
#include <conio.h>            //DOS konzol I/O rutinok hívása
#include "Calculator.h"       //a Calculator osztály deklarációi

//*****************************************************************
int main()
//*****************************************************************
{
        //objektumpéldány létrehozása
        Calculator CalculatorObject(20,10);
        //az osztály Printer tagfüggvényének hívása
        CalculatorObject.Printer();
        std::cout << "\nPress any key to exit...";

        _getch();     //egy karakter megadását várja
        return 0;     //visszatérési érték az operációs rendszer felé
```

Text-osztály deklarációja (Text.h)

```
/*****************************************************************
* Text osztálydeklarációja                                       *
* Demonstrációs célok:                                           *
*                -> egységbe zárás: class                        *
*                -> adattagok                                    *
*                -> konstruktor                                  *
*                -> tagfüggvények                                *
*                -> láthatóságok                                 *
*                -> char tömb                                    *
*                                                                *
* Katona József     <katonaj@maild.duf.hu>                       *
*****************************************************************/
#ifndef TEXT_H   //if not define egy szimbólum nemlétét ellenőrzi
#define TEXT_H   //egy szimbólum definiálása és értékének megadása
//===============================================================
class Text
{
//===============================================================
        //osztály adattagja
        char text[54];

    public:
        //paraméteres konstruktor
        Text(char[]);
        //a magánhangzók számával tér vissza
        int  NumberOfVowels();
        //a mássalhangzók számával tér vissza
        int  NumberOfConsonants();
        //a szóközök számával tér vissza
        int  NumberOfSpace();
        //a szóközöket lecseréli a # karakterre
        void SpaceExchanger();
        //az eredmény kiírásáért felel
        void Printer();
};
#endif //feltétel lezárása
```

Text-osztály implementálása (Text.cpp)

```
/*******************************************************************
* Text osztály definiálása/implementálása                          *
* Demonstrációs célok:                                             *
*                -> this referenciamutató                          *
*                -> adattagok inicializálása                       *
*                -> :: hatókör operátor                            *
*                -> tagfüggvények feladatainak kifejtése           *
*                -> std névtér                                     *
*                -> strlen függvény bemutatása                     *
*                                                                  *
* Katona József     <katonaj@mail.duf.hu>                          *
*                                                                  *
*******************************************************************/
#include <iostream>           //C++ alapvető adatfolyam I/O rutinok
#include "Text.h"             //a Szoveg osztály deklarációi

//-------------------------------------------------------------------
Text :: Text(char x[])
//-------------------------------------------------------------------
{
    for (int i = 0; i < strlen(x); ++i) //strlen->return szoveg hossz
        text[i] = x[i];                 //adattag inicializálás
//-------------------------------------------------------------------
int Text::NumberOfVowels()
//-------------------------------------------------------------------
{
        int numberOfVowles = 0;
        char vowels[5] = {'a','e','i','o','u'};
        for (int i = 0; i < strlen(text); ++i)
        {
                for (int j = 0; j < strlen(vowels); ++j)
                        if(text[i] == vowels[j])
                                ++numberOfVowles;
        }
        return numberOfVowles;
}

//-------------------------------------------------------------------
int Text::NumberOfConsonants()
//-------------------------------------------------------------------
{
        int numberOfConsonants = 0;
        char consonants[21] =
        {'b','c','d','f','g','h','j','k','l','m','n',
         'p','r','q','s','t','v','w','x','y','z'};
        for (int i = 0; i < strlen(this->text); i++)
        {
```

```
                for (int j = 0; j < strlen(consonants); j++)
                        if(text[i] == consonants[j])
                                ++numberOfConsonants;
        }
        return numberOfConsonants;
}

//-----------------------------------------------------------------
int Text::NumberOfSpace()
//-----------------------------------------------------------------
{
        int numberOfSpace = 0;
        for (int i = 0; i < strlen(this->text); i++)
                if(text[i] == ' ')
                        ++numberOfSpace;

        return numberOfSpace;
}

//-----------------------------------------------------------------
void Text::SpaceExchanger()
//-----------------------------------------------------------------
{
        for (int i = 0; i < strlen(this->text); i++)

                if(text[i] == ' ')
                        text[i] = '#';
}

//-----------------------------------------------------------------
void Text::Printer()
//-----------------------------------------------------------------
{
        std::cout << "AZ EREDETI SZOVEG" << std::endl;
        std::cout << "=================" << std::endl;
        for (int i = 0; i < 54; ++i)
                std::cout << text[i];
        std::cout << std::endl << std::endl;
        std::cout << "ADATOK"            << std::endl;
        std::cout << "=================" << std::endl;
        std::cout << "maganhangzok szama  : " <<
        this->NumberOfVowels() << "db" << std::endl;
        std::cout << "massalhangzok szama : " <<
        this->NumberOfConsonants() << "db" << std::endl;
        std::cout << "szokozok szama      : " <<
        this->NumberOfSpace() << "db" << std::endl;
        std::cout << std::endl;
        std::cout << "SZOKOZOK CSEREJE"  << std::endl;
```

```
        std::cout << "=================" << std::endl;
        this->SpaceExchanger();
        for (int i = 0; i < 54; ++i)
                std::cout << text[i];
        std::cout << std::endl << std::endl;
}
```

Text-osztály példányosítása (main.cpp)

```
/*********************************************************************
* Text osztály példányosítása                                        *
* Demonstrácios célok:                                               *
*                -> statikus objektumpéldány                         *
*                -> osztály tagfüggvényeinek a hívása                *
*                                                                    *
* Katona József     <katonaj@mail.duf.hu>                            *
*********************************************************************/
#include <conio.h>                  //DOS konzol I/O rutinok hívása
#include "Text.h"                   //a Text osztály deklarációi

//*********************************************************************
int main()
//*********************************************************************
{
        //objektumpéldány létrehozása

        Text TextObject("lorem ipsum dolor sit amet, consectetur
        adipisicing...");
        //az osztály Kiir() tagfüggvényének hívása
        TextObject.Printer();

        _getch();    //egy karakter megadását várja
        return 0;   //visszatérési érték az operációs rendszer felé
}
```

Felhasznált irodalom

Andrei Alexandrescu–Herb Sutter(2005): *C++ kódolási szabályok.* Kiskapu kiadó.
Robert A. Maksimchuck–Eric J. Naiburg (2006): *UML földi halandóknak.* Kiskapu kiadó.
Alex Allain (2013): *Jumping Into C++*. Cprogramming.com kiadó.
Mike McGrath(2011): *C++ Programming In Easy Steps 4th Edition*. In Easy Steps Limited kiadó.
Bjarne Stroustrup (2013): *The C++ Programming Language*. Addison Wesley kiadó
Nicolai M. Josuttis (2012): *The C++ Standard Library: A Tutorial and Reference.* Addison Wesley kiadó.
Ivor Horton (2014): *Ivor Horton's Beginning Visual C++ 2013* (Wrox Beginning Guides). John Wiley & Sons kiadó.
David Vandevoorde (2014): *C++ Templates: The Complete Guide.* Addison Wesley kiadó.
Marc Gregoire (2014): *Professional C++*. John Wiley & Sons kiadó.
Martin Fowler(2003): *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (Object Technology Series). Addison Wesley kiadó.
Craig Larman(2004): *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall kiadó.
Simon Bennet–Steve Mcrobb–Ray Farmer (2006): *Object-Oriented Systems Analysis and Design Using UML.* McGraw-Hill Higher Education kiadó.
David P. Tegarden–Alan Dennis–Barbara Haley Wixom (2012): *Systems Analysis and Design with UML.* John Wiley & Sons kiadó.

Segédanyagok
Továbbá számos olyan nagyobb magyar és külföldi egyetemektől származó publikáció, könyv és segédlet került felhasználásra, amelyek külön nem kaptak helyet a felsorolásban.