Katona József és Kővári Attila

OBJEKTUMOK LÉTREHOZÁSA, MEGSZÜNTETÉSE, MEMÓRIA MENEDZSMENT

GYAKORLATORIENTÁLT SZOFTVERFEJLESZTÉS C++ NYELVEN VISUAL STUDIO COMMUNITY FEJLESZTŐKÖRNYEZETBEN

II. KÖTET

Publio

Objektumok létrehozása, megszüntetése, memória menedzsment

Katona József és Kővári Attila

2016
Felelős kiadó: Publio Kiadó
www.publio.hu

Lektorálta: Kratzmajer Zoltán
ISBN: 9789634245322
DOI: 10.18395/objektum.2016

Előszó

A C++ programozási nyelv egy általános célú, magas szintű nyelv. A nyelv elterjedtségét és létjogosultságát támassza alá, hogy szinte minden operációs rendszer alá létezik C++ fordító. A C++ nyelv elsajátításához szükséges a C nyelv ismerete is, mivel a C++ általánosságban a C nyelv szintaxisára és koncepcióira épül, annak kiterjesztése.

E könyv a C++ nyelven történő objektumorientált szoftverfejlesztést megismertető sorozat második kötete, mely sorozat gyakorlati példákon keresztül segíti a nyelv elsajátítását. Bízunk benne, hogy azok, akik végigolvassák a könyvet, és kidolgozzák a példákat, azok alkalmazás szintjén megismerik a C++ nyelvi elemei által adott lehetőségeket, a nyelv logikáját, az átláthatóbb szoftver készítését elősegítő funkcióját és mélyebben elsajátítják az objektumorientált szemlélet alapjait, mely a hatékonyan szoftverfejlesztéshez elengedhetetlen. A bemutatott ismeretek gyakorlati példákkal illusztráltak, melyek a megértést és az alkalmazás elsajátítását is nagymértékben segítik.

Jelen kötet az objektumorientált programozásban használt osztályt, konstruktort, destruktort, objektumot és ezen fogalmak kapcsolatát ismerteti, továbbá egy projekten keresztül lépésről-lépésre bemutatja ezeknek a nyelvi eszközöknek a használatát és jelentőségét. A mintapélda a könyvsorozat első kötetében bemutatott Visual Studio Community fejlesztőkörnyezetben kerül megvalósításra, melyben a bemutatott mintapélda implementálható és összeállítható, a futtatható kód előállítható.

Köszönettel tartozunk a kézirat lektorának, Kratzmajer Zoltánnak a szakmai tanácsaiért és módszertani javaslataiért.

A Szerzők

Az osztály

Az osztály lényegében egy típus leírás, amely adattagokból és tagfüggvényekből épül fel, amelyeknek más osztályból történő elérését korlátozhatjuk a feladat jellegétől függően. Általánosságban kizárólag az osztály tagfüggvényei hozzáférhetőek.

C++-ban leginkább a következő három hozzáférhetőségi szintet (jelleget) különböztetik meg:

Public (nyilvános): Ennél a hozzáférhetőségi szintnél a deklarációk és definíciók bármely további osztály(ok)ból szabadon elérhetőek. Elsősorban azokat a tagfüggvényeket szokás nyilvánosan hozzáférhetővé tenni, melyek más osztály(ok)ból történő elérése feltétlenül szükséges.

Protected (védett): Az ilyen típusú hozzáférhetőségi szintet az öröklődési hierarchia megalkotása során használjuk. A deklarációk és definíciók csak az őt tartalmazó osztályban (ős, szülő vagy bázis), illetve a belőlük leszármaztatott osztályokban érhetőek el.

Private (magán): A magán hozzáférhetőségi szint esetén a további deklarációk és definíciók csak az őt tartalmazó osztályban – illetve az osztály úgynevezett „barátaiból” (**friend**) – láthatóak és érhetőek el.

Az osztályokkal kapcsolatban általánosan elfogadott kód stilisztikai szabályok:

Az osztály azonosítóját – a leírására jellemző – főnévvel látjuk el, amelyet ajánlott nagybetűvel kezdeni.

Az osztály adattagjainak neve szófajukat tekintve szintén főnevek, amelyek utalnak az általuk leírt tulajdonságokra. Az adattagokat ajánlott kis kezdőbetűvel ellátni és a határolók mentén a szóközt vagy alulvonást mellőzve, nagybetűvel elválasztani azokat.

Az osztály tagfüggvényeinek azonosítójául érdemes igéket alkalmazni, amelyek egyértelműen jelölik annak a függvénynek a feladatát és felelősségét. A tagfüggvényeket ajánlott nagybetűvel kezdeni és a határolók mentén az adattagokhoz hasonlóan nagybetűvel elválasztani.

Az osztály létrehozásának szándékát a **class** kulcsszóval jelölhetjük.

Ne keverjük az angol és magyar kifejezéseket.

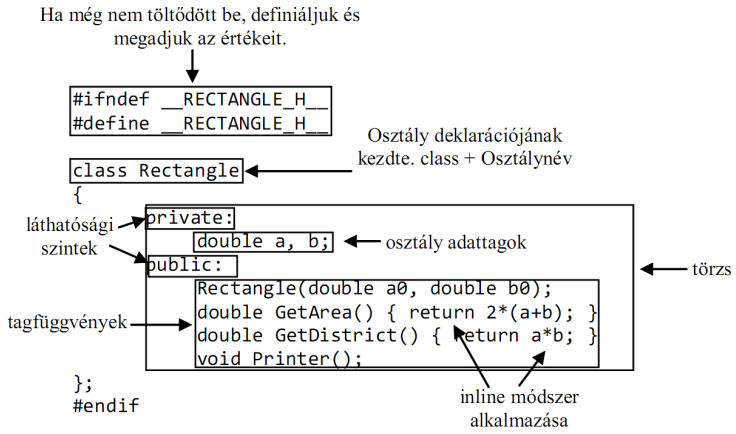
A formai szabályok betartásával, sokkal áttekinthetőbb és karbantarthatóbb kódegységet kaphatunk, amelyek kevésbé igényelnek kommentezést.

Az osztályok kialakításának lehetőségei

C++-ban egy osztály kialakítása több módon véghezvihető. A valós világban ezek a kialakítási módszerek vegyesen fordulnak elő és maga a feladat jellege dönti el, hogy melyiket a legcélszerűbb alkalmazni.

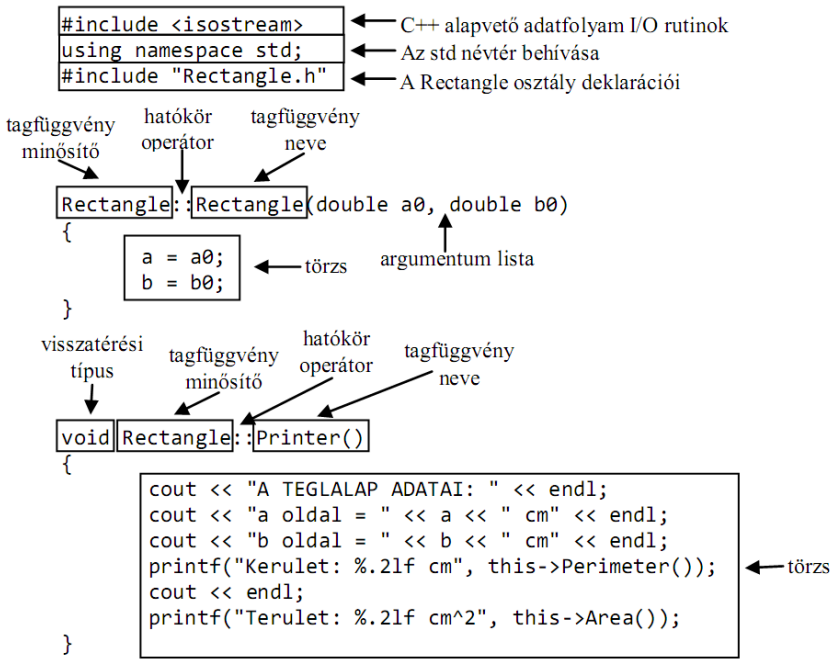
Az első kialakítási lehetőség az úgynevezett implicit **inline** tagfüggvények alkalmazása, ahol a tagfüggvények feladatát az osztály deklarációján belül fejtjük ki. Ennek előnye, hogy a teljes típus leírás egyetlen header fájlban megtalálható, mely könnyebben átlátható, továbbá egyfajta makrónak is tekinthető, mivel a függvényhívásokkal járó adminisztrációra sincs szükség. Az implicit **inline** módszert kisterjedelmű függvényeknél érdemes használni, így egy-két soros függvények esetében ez sokkal hatékonyabb, sokkal olvashatóbb megoldást jelent, azonban a bonyolultabb feladatot megvalósító függvényeket már érdemes fejállományon kívül implementálni, mivel az éppen az átláthatóságot, karbantarthatóságot rontja.

A második kialakítási lehetőség az osztály deklarációjának és implementációjának külön C++-os modulokban történő tárolása. Ezt a módszert elsősorban a nagyobb méretű osztályok és tagfüggvények menedzselésénél ajánlott alkalmazni. Ebben az esetben a deklarációban az adattagok mellett kizárólag a tagfüggvények prototípusa található. Az osztály deklarációt egy úgynevezett header fájlban (.H kiterjesztésű) helyezzük el, amíg az osztály implementációját, ahol többek között a tagfüggvény definícióit tároljuk, egy külön modulban (.CPP kiterjesztésű). Az 1. ábra egy osztály fejlécállománynak lehetséges felépítését mutatja.



1. ábra: **Egy osztály header fájl anatómiája C++ nyelv specifikusan**

A 2. ábra egy osztály implementációjának lehetséges felépítését mutatja.



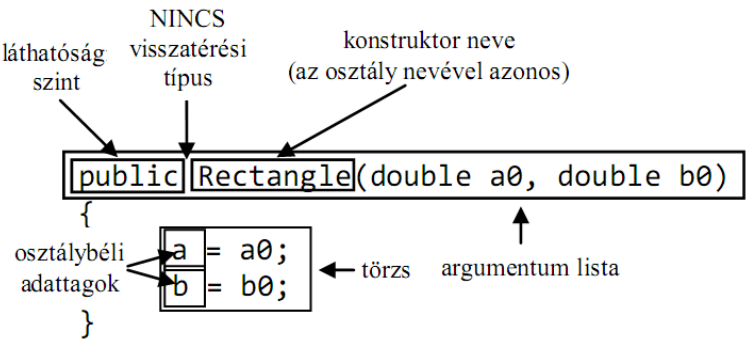
2. ábra: **Egy osztály implementációs moduljának anatómiája C++ nyelv specifikusan**

A konstruktor

Az objektumorientált paradigma egyik legfontosabb jellemzője, hogy a különböző függvényeket, mezőket és jellemzőket egy egységbe zárja, amely egységben a konstruktor, mint egy speciális tagfüggvény foglal helyet. A konstruktor általános feladata az egységbe zárt adattagok kezdőértékeinek beállítása, ahol az egység maga az osztály. A konstruktor legfontosabb tulajdonságai a következő módon foglalhatóak össze:

- Amennyiben nincs általunk implementált konstruktor, a C++ fordító készít egy úgynevezett alapértelmezett jellegűt.
- Az azonosítója minden esetben a konstruktort magában foglaló osztály azonosítójával megegyező.
- A konstruktor visszatérési típussal nem rendelkezik, ez a C++ esetében azt jelenti, hogy még **void** típus használata is tiltott.
- A létrehozandó osztályhierarchia során figyelembe kell vennünk, hogy egy konstruktor nem öröklődhet.
- Az objektumorientált világban a legtöbb függvény túlterhelhető vagy felülírható, nem kivétel ez alól a konstruktor sem, ennek következtében, akár több, eltérő paraméterezésű konstruktor is implementálható egy osztályon belül.
- A konstruktor legtöbb esetben nyilvános (**public**) jellegű.
- Módosító kizárólag a hozzáférési szintet meghatározó kulcsszó és a **static** lehet.

A konstruktor általános, kezdeti értékadási feladata mellett egyéb feladatokat is elláthat. Előfordulhat például, hogy egy olyan osztályt akarunk létrehozni, amelynek a memóriaigénye nagyobb egy átlagos osztályéhoz képest, így használhatjuk azt, mint egyfajta memóriaterület lefoglalásért felelős speciális tagfüggvényt.



3. ábra: **A konstruktor anatómiája**

A konstruktor végrehajtási sorrendje:

- A tartalmazó osztályban a deklaráció(k) sorrendjében meghívásra kerül(nek). Abban az esetben, ha egy osztály több konstruktort is tartalmaz, az argumentumlista alapján dől el, melyik kerül hívásra.
- Amennyiben a tartalmazó osztály egy (bázis) osztály kiterjesztése a bázis osztály konstruktorában szereplő utasítások hajtódnak végre először.
- A konstruktorok hívását követően hajtódnak végre a függvénytörzsben szereplő utasítások.

Implicit és explicit konstruktorok

Egy osztálynak minden esetben van legalább egy konstruktora. Abban az esetben, ha a programozó nem írja meg, a fordító gondoskodik arról, hogy az osztályhoz hozzárendelje az implicit (default) konstruktort. Egy implicit módon hívott konstruktor kizárólag **public** jelleggel bír, az argumentumlistája üres és a törzse sem tartalmaz végrehajtandó utasításokat. Ennek az esetnek a következménye a paraméter nélküli osztály példányosítás. Amennyiben a programozó legalább egy konstruktor létrehoz, úgy az így létrehozott konstruktor explicit módon kerül hívásra. Ilyenkor az implicit konstruktor semmilyen formában nem rendelődik az osztályhoz. Abban az esetben, ha a programozó paraméter nélküli példányosítást szeretne megvalósítani explicit konstruktor hívással, azt kizárólag a programozó által definiált paraméter nélküli konstruktor segítségével teheti meg. Minden olyan esetben, ahol a programozó maga szeretné meghatározni a konstruktor láthatósági szintjét,

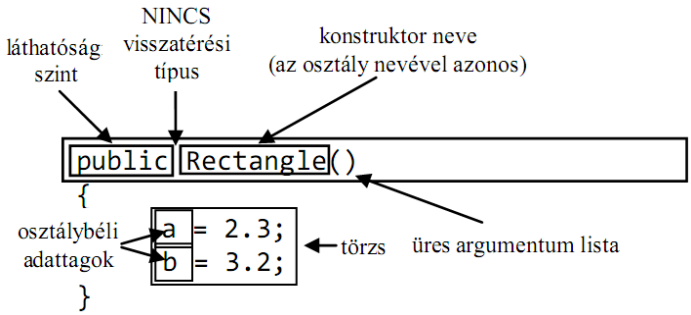
paraméterét és azon utasításokat, amelyeket a törzsben szeretne végrehajtatni kizárólag explicit módon teheti meg.

Konstruktor típusok

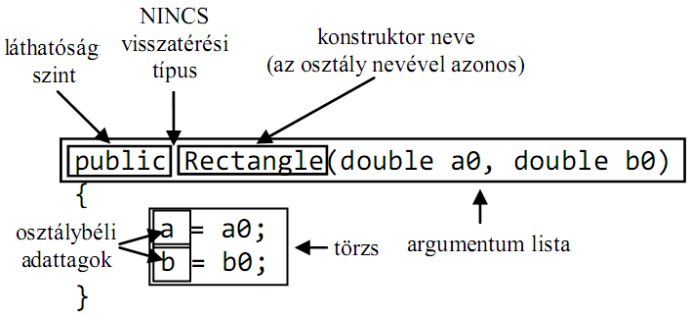
A C++-ban kétféle speciális típusú konstruktort különböztetnek meg, az alapértelmezettet (**default**) és a másoló (**copy**) konstruktort. Az alapértelmezett konstruktorok két további altípusra bonthatóak: a paraméter nélküli és a paraméteres.

A paraméter nélküli konstruktor esetében az argumentumlista üres. Az ilyen jellegű konstruktorok esetében az elsődleges cél lehet az osztály adattagjainak konstans értékkel történő ellátása. A paraméteres konstruktorok esetében az argumentumlistában szereplő értékeket használhatjuk arra, hogy az osztály adattagjait inicializáljuk azokkal.

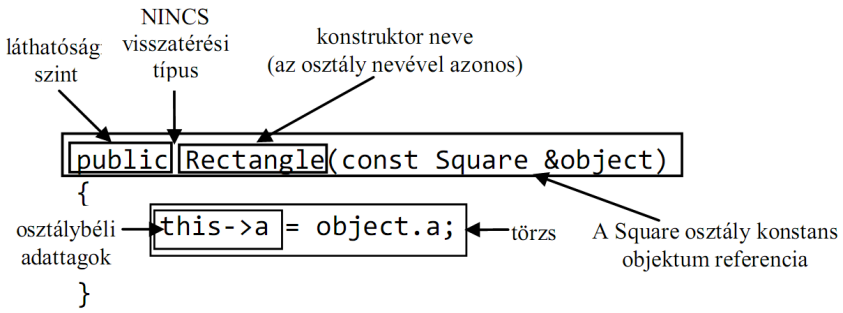
A másoló konstruktor esetében egy objektumpéldány kezdőértékként kaphat egy már létező és inicializált másik objektumpéldány értéket. A másoló konstruktor esetében a paraméterek jellege kizárólag referencia lehet, így biztosítható a végtelen ciklus kivédése, ugyanis a paraméterátadáshoz is másoló konstruktort használható.



4. ábra: **A paraméternélküli, alapértelmezett (default) konstruktor anatómiája**



5. ábra: **A paraméteres, alapértelmezett (default) konstruktor anatómiája**



6. ábra: **A másoló (copy) konstruktor anatómiája**

Objektumok létrehozása, inicializálása

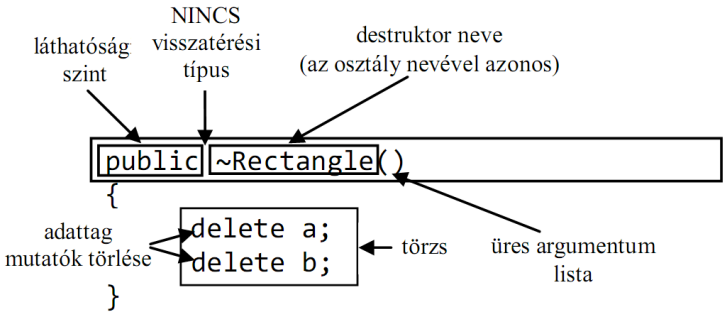
Az előző fejezetben röviden ismertetésre került egy speciális függvény, a konstruktor, ahol megismertük tulajdonságait, általános feladatait és típusait. Jelen fejezetben bemutatásra kerül ezen speciális függvény és az objektum kapcsolata. Egy objektumnál megkülönböztetünk tulajdonságot (belső állapotot), illetve képességet, az előbbit a példányosítani kívánt osztály adattagjai írnak le, amíg az utóbbit a tagfüggvényei. Egy objektum belső állapota bármikor megváltozhat, ahol az osztályban foglalt tagfüggvények segítségével a tulajdonságokat leíró adattagokat manipulálhatjuk. Fontos szabály, hogy egy objektumot egyértelműen és könnyen betudjunk azonosítani. Minden egyes objektum a születésétől kezdve egy adott osztályhoz tartozik.

Az objektumok életük során először memóriát allokálnak (foglalnak) az adattagok számára, ez C++ specifikusan a **new** operátor segítségével történik, amely során az objektumok egy köztes állapotba kerülhetnek. Ebben a köztes állapotban kerülhet sor az ősosztály(ok) adattagjainak allokációjára, amennyiben az(ok) léteznek. Az allokációt követően – megfelelő értékek szerint – az adattagok inicializálása hajtódik végre. Az objektumok megfelelő inicializálásáért a konstruktor felel. Lényegében egy objektum nem létezhet konstruktor nélkül, amennyiben a programozó nem gondoskodik annak megfelelő hívásáról úgy az implicit módon is meg tud hívódni.

A destruktor

Az előző fejezetekben megismerkedtünk egy speciális függvénnyel, amely nélkül egy objektum nem jöhetne létre. Az úgynevezett destruktor a konstruktor feladatának ellentétjét valósítja meg, az objektum megszüntetéséért felelős. Léteznek olyan esetek, amikor a memóriát célszerűbb dinamikusan kezelni, tehát magára a programozóra bízni egy objektumpéldány életciklusát. Abban az esetben, ha az objektum megszüntetésének felelősségét nem az operációs rendszerre szeretnénk bízni, explicit destruktor hívást kell alkalmaznunk. Tehát a destruktor, az objektum megszüntetésért, a lefoglalt erőforrások, mint például a memória, felszabadításáért felelős speciális függvény. A destruktor legfontosabb tulajdonságai a következő módon foglalhatóak össze:

- Ha az osztály rendelkezik destruktorral, akkor az explicit módon kerül hívásra, ha azonban ilyen nem létezik, akkor az adott osztályhoz a fordító a saját alapértelmezése szerinti változatát használja.
- A destruktor nevét tilde karakterrel (~) kell kezdeni, ahol a név megegyezik a magában foglaló osztály nevével (tilde + osztály név).
- Egy destruktornak kizárólag paraméternélküli formája létezik.
- A konstruktorhoz hasonlóan itt sincs vissza térési érték, ez a C++ esetében azt jelenti, hogy még **void** típus használata is tiltott.
- A destruktor aktivizálása a **delete** operátorral történik.
- A létrehozandó osztályhierarchia során figyelembe kell vennünk, hogy a leszármazott osztályok destruktorai is megfelelően meghívódjanak. Erről úgynevezett virtuális destruktorok definiálásával gondoskodhatunk.
- Egy destruktor nem definiálható felül, minden egyes osztálynak kizárólag egy destruktora lehet.
- A konstruktorhoz hasonlóan a destruktor sem öröklődhet.
- Egy destruktor lehet virtuális is, azonban egy konstruktor nem.
- A konstruktorral ellentétben a destruktornak nem léteznek különböző típusai.

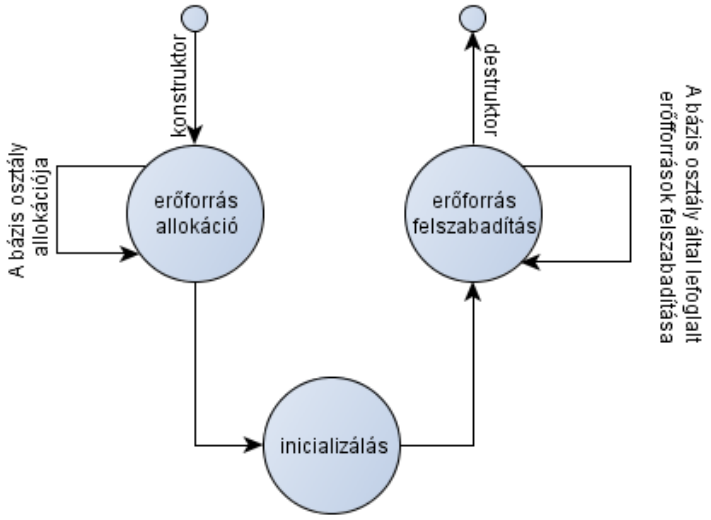


7. ábra: **A destruktor anatómiája**

Objektumok megszüntetése

Az előző fejezetben röviden ismertetett destruktor az, amely az objektum megszüntetésére használható. Minden egyes objektum a születésétől kezdve egy adott osztályhoz tartozik és létrehozása során erőforrásokat (memória, állomány stb.) foglal le, azonban ha egy objektum végrehajtotta feladatát, gondoskodnunk kell annak megszüntetéséről és az általa lefoglalt erőforrások felszabadításáról. Amennyiben nem így tennénk az erőforrások elvesznének, és könnyedén kialakulhatna például a memória szivárgás jelensége. A C++ nyelv a programozó számára a fentebb megismert speciális függvényt, a destruktort biztosítja az erőforrás felszabadítás maradéktalan elvégzésére.

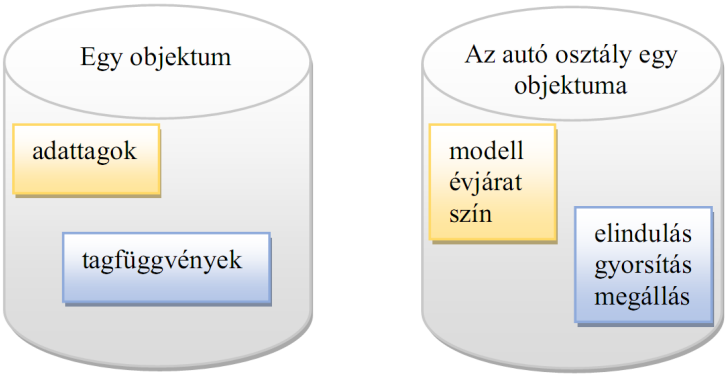
Összegzésként elmondható, hogy egy objektum életciklusa három főszakaszból áll: születés, feladatok végrehajtása és megszűnés. A 6. ábra illusztrálja egy objektum életciklusát.



8. ábra Egy objektum életciklusa

Az osztályok és objektumok kapcsolata

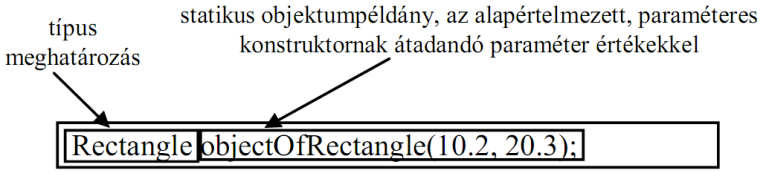
A fentebb tárgyalt fogalmak, mint az osztályok és objektumok együttesen fordulnak elő a programozás során. Az osztálynév, mint típusnév használható, amelyben nem kizárólag csak adatszerkezeteket tárolhatunk, hanem függvényeket is. Az objektum az osztály egy előfordulását jelenti. A 9. ábra baloldala szemlélteti egy objektum általános felépítését, amíg a jobb oldalon egy adott osztály (autó) egy előfordulását. Az autó osztály ezen objektuma rendelkezik (hozzáfér) a következő tulajdonságokkal(hoz): a gépjármű modell nevével, évjáratával és színével, valamint az elindulás, a gyorsítás és a megállás képességekkel.



8. ábra: **Az ábra baloldalán egy általános ábrázolása egy objektumnak, amíg a jobb oldala egy autó osztály adott objektuma tulajdonságokkal és képességekkel.**

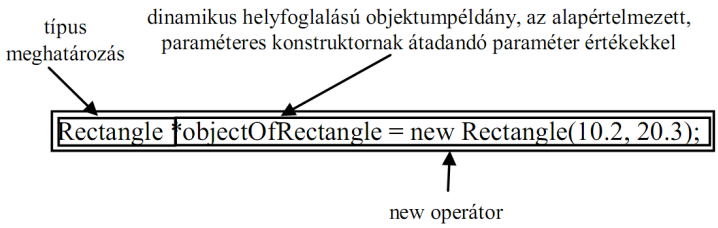
Primitív szinten vizsgálva ez azt jelentené, hogy az osztály lenne a típus az objektum pedig a változó. Az osztály előfordulását példányosításnak is szokták nevezni, ahol az objektum az adott osztály egy példánya. C++-ban a példányosítás két módon történhet. Az első az úgynevezett statikus példányosítás, amikor az objektum megszüntetését magára az operációs rendszerre bízzuk. A második az úgynevezett dinamikus objektumpéldány létrehozása, ebben az esetben a programozóra van bízva, hogy mikor szüntet meg egy objektumot, szabadítja fel az általa lefoglalt erőforrásokat.

A statikus objektumpéldány létrehozása esetén az osztály neve után az objektum azonosító kerül megfogalmazásra. 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. A 10. ábra szemlélteti egy statikus objektumpéldány felépítését.



9. ábra: **A statikus objektumpéldány anatómiája C++ nyelv specifikusan**

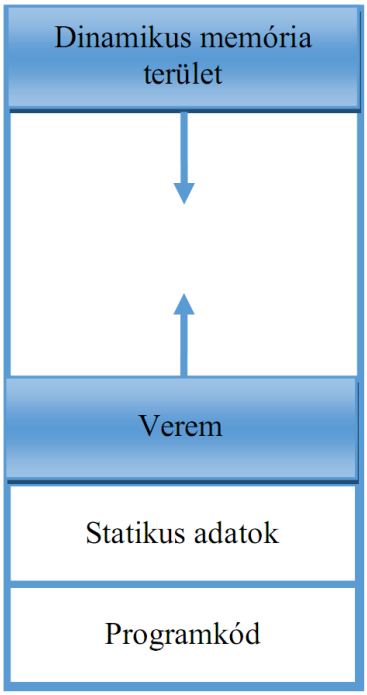
C++–ban operátorok felelnek a dinamikus memóriakezelésért. Dinamikus objektumpéldány létrehozásánál (11. ábra) szükség van a **new** operátorra, amely segítségével dinamikusan tudjuk allokálni a memóriát. A **new** operátor amellett, hogy lefoglalja a szükséges memóriaterületet, a lefoglalt típusra mutató 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.



10. ábra: **A dinamikus objektumpéldány anatómiája C++ nyelv specifikusan**

C++ és a memória menedzsment

Egy C++ alapú program az operációs rendszertől négy memória területet kap. A **CODE** (*Program kód*) helyen a végrehajtandó program helyezkedik el. A **GLOBAL** (*Statikus adatok*) területen a globális és statikus változók helyezkednek el, amelyek a teljes program futása során elérhetőek. Memória menedzselés szempontjából a statikusan létrehozott objektumpéldány a memória végrehajtási *verem* területén (**STACK**) helyezkedik el, amíg a dinamikus objektumpéldány a memória *dinamikus memória területén* (**HEAP**). A dinamikus memória területet szokás még *halomnak* is nevezni. A 12. ábrán látható egy kevert memóriakezelést (a veremben és a dinamikus memória területen is történt helyfoglalás) használó program tipikus memóriastruktúrája.



11. ábra:**A kevert memóriakezelés memóriaábrája**

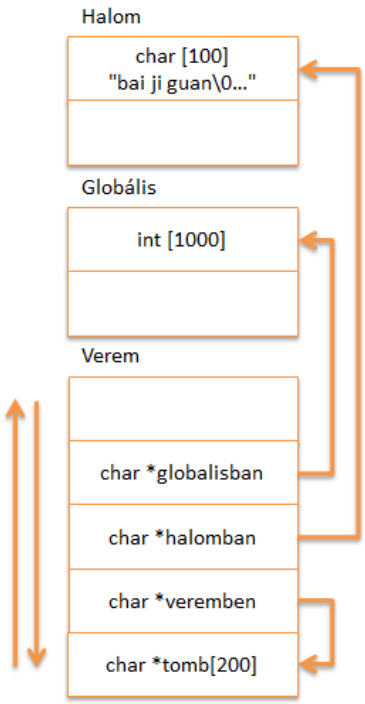
Az ábra legalsó részén található maga a programkód, amely felett a statikus adatok helyezkednek el, amely adatok a program indulásakor töltődnek be. A kevert memóriakezelés eredményeképpen mind a veremterületen, mind a dinamikus memória területen történt helyfoglalás.

A végrehajtási verem egy olyan, statikus lefoglalt memóriaterület, amelybe folytonosan lehet adatokat tölteni, továbbá mindig az utolsóként behelyezett elemet lehet kivenni először, az elsőként behelyezett elem pedig a verem legalsó szegmensében helyezkedik el. Ha függvény hívás történik, akkor a veremben az aktivációs rekord legfelülre kerül, amely tárolja az adott függvény paramétereit és az automatikus helyfoglalású változóit. Amennyiben a függvény futása befejeződik, az aktivációs rekord törlődik annak minden változójával együtt. C++ esetében az első aktivációs rekord a **main()** függvény. A verem akkor tekinthető üresnek, ha a **main()** függvény lefutott.

A dinamikus memória terület gazdálkodásáért a programozó a felelős. A programozó foglalja le a kívánt területet és gondoskodik a felszabadításáról is, amennyiben a felszabadításról elfeledkezik, memória szivárgás következik be. C++-ban a dinamikus helyfoglalás a **new** operátorral történik, amíg a lefoglalt terület felszabadítása a **delete** operátor segítségével hajtható végre. Ha a dinamikus területen tárolunk adatokat, a végrehajtási veremben már csak egy mutató jön létre. A végrehajtási verem legnagyobb problémája, hogy nem végtelen és hamar keletkezhet **stackoverflow** (verem túlcsordulás) kivétel, amely a program fagyásához vezethet, ellenben a dinamikus memória terület szinte korlátlan (a memóriánk méretétől függ). Az alább látható, egyszerű, C++ alapú programrészlet[1], amelynek a memóriabeli struktúrája a 13. ábrán látható.

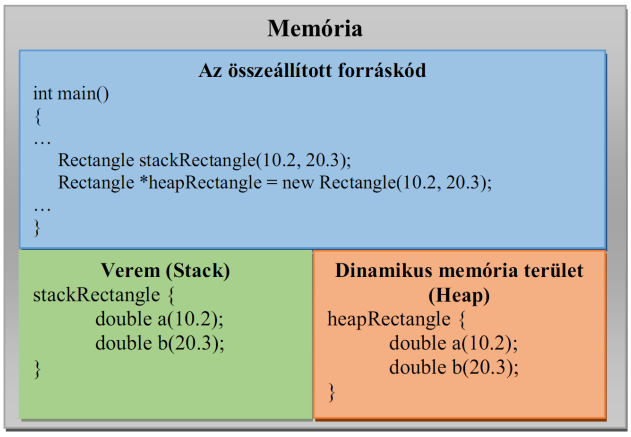
[1]Forrás: http://moodle.autolab.uni-pannon.hu

```
char global[1000];
int main()
{
        char tomb[200];
        char *veremben;
        char *heapen;
        char *globalisban;
        halomban = new(100);
        globalisban = global;
        veremben = tomb;
        delete(heapen);
        return 0;
}
```



12. ábra: **C++ alapú programrészlet memóriaábrája**

A 14. ábra szemlélteti egy statikus és egy dinamikus módon létrehozott objektumpéldány memóriabéli területét.



13. ábra: **Memória területek felosztása a program futása során**

Mintapélda Visual Studio Community

fejlesztőkörnyezetben

A fentebb megismert fogalmak gyakorlati elsajátításához a következőekben leírt minta feladat tervezését és implementálását végezzük el.

A feladat megfogalmazása

Írjunk egy repülőgép-hordozót osztályt (*AircraftCarrier*), amely egy repülőgép-hordozó anyahajót modellez. Az osztály tagváltozói a következőek:

- Életerő (*hitPoints*), mely egy egész szám 0 és 500 között.
- Vízkiszoritas (*draft*), mely egy lebegőpontos szám 1,0 és 100,0 között.
- Hatótávolság (*range*), mely egy szöveges típusú adat.
- Repülőgépek száma (*aircraftCarried*), mely egy egész szám 0 és 150 között.

Ügyeljünk az egységbezárás elvére, tehát a tagváltozók private láthatóságú adattagok legyenek!

Valósítsuk meg az osztály metódusait:

- Írjuk meg az osztály konstruktorát, melyben kötelező megadni a repülőgép-hordozó vízkiszorítását, hatótávolságát és a szállított repülőgépek számát! Az életerőt is lehessen megadni, de ne legyen kötelező (használjunl default paramétereket). Ha a programozó nem ad meg életerőt a példányosítás során, a tagváltozó kezdeti értéke 500 legyen)! A konstruktor megvalósításnál alapértelmezett paraméteres konstruktort használjunk.
- Írjunk egy *Repair()* függvényt, mely 70 ponttal növeli a repülőgép-hordozó életerejét! Ügyeljünk rá, hogy 500 fölé soha ne kerülhessen az érték.
- Írjunk egy *Attack(otherAttackRating, attackAircraftCarried)* függvényt, amely támadás alá helyezi a hajónkat! A függvény tartalma csakis akkor fusson le, ha a repülőgép-hordozó életereje nagyobb mint 0! A támadás hatására a repülőgép-hordozó életereje csökkenjen (*hitPoints - otherAttackRating*) és a repülőgépek száma csökkenjen ((*aeroplaneNumbers – attackAircraftCarried) * 0.5*)

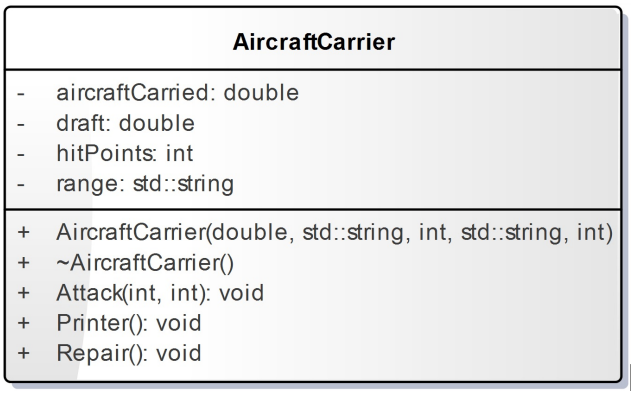
értékkel, ha az ellenség repülőgépeinek a száma fele annyi vagy kevesebb, mint a mi hajónk repülőgépeinek száma, egyébként ((*aeroplaneNumbers – attackAircraftCarried * 0.8)*) csökkentsük ezt az értéket! Ügyeljünk rá, hogy 0 alá nem csökkenhet a repülőgéphordozó életereje.
- Írjunk egy egyszerű *Printer()* függvényt, amely minden fontos adatot kiír az anyahajóról.
- Ne feledkezzünk meg az objektumpéldány által lefoglalt erőforrás felszabadításáról, a destruktor megírásáról sem.

Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén [heap]) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükséges az objektum példányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. Mielőtt a feladatnak nekilátnánk, tervezzük meg azt statikus, használati eset és dinamikus modellek segítségével, hozzájárulva ezzel a kódolás hatékonyságához.

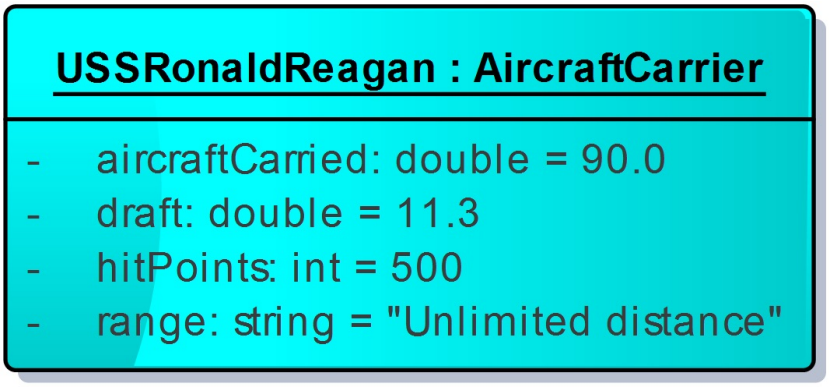
Statikus, használati eset és dinamikus tervezés

Az objektummodellben az adat áll a középpontban és annak a szemszögéből írja le a rendszer statikus tulajdonságait és struktúráit, mely során meghatározásra kerül, hogy a rendszer milyen egységekből, elemekből épül fel. A modelltervezés megkönnyítésére olyan UML (Unified Modelling Language – Egységes Modellező Nyelv) diagramokat fejlesztett ki, amelyek segítségével vizuálisan szemléltethető egy program statikus (időtől független) strukturális felépítése, illetve egy adott időpontban meglévő objektumainak kapcsolatrendszerét, mely maga a statikus modell.

Az objektummodell leírja a rendszerbeli objektumok struktúráit, attribútumait és metódusait, valamint az objektumok közötti kapcsolatokat, relációikat. Általában az objektummodell az alapja a dinamikus és funkcionális modell megalkotásának, hisz a változásokat és a transzformációkat kifejező dinamikus és funkcionális modellek esetén valami alapján meg kell tudni adni, hogy mik változnak, transzformálódnak.



14. ábra:**Osztály diagram**

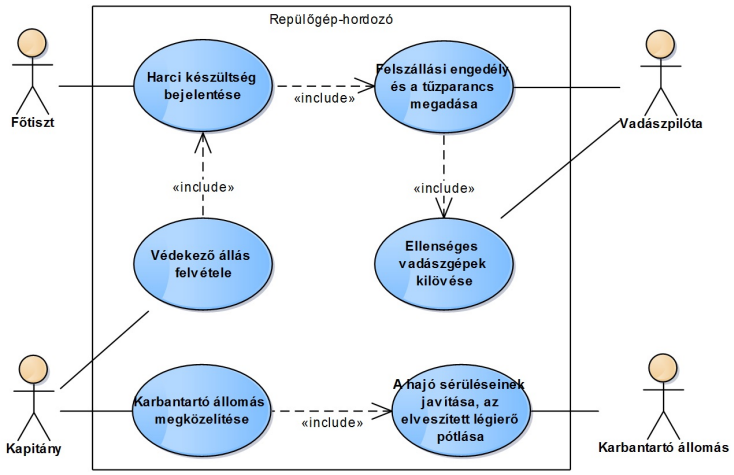


15. ábra:**Objektum diagram**

Funkcionális modell

Általánosságban az objektumokból álló rendszer a külvilágból érkező információkat fogadja, azokat feldolgozza, és arra válaszol. A rendszerek tehát együttműködnek a külvilágban létező emberi vagy automatikus szereplőkkel, az aktorokkal. Az aktorok a rendszer használatától azt várják, hogy az specifikáció alapján kiszámítható, meghatározható módon viselkedjen, adjon választ.

A használati eset (**use case**) diagram definiálja a rendszer, vagy a rendszer valamely jól meghatározható részének a viselkedését, leírva az aktorok és a rendszer közötti együttműködést, mint akciók és reakciók (válaszok) sorozatát. A repülés példáján alapuló használati eset diagram látható a következő ábrán.



16. ábra:**Használati eset (use case) diagram**

Dinamikus modell

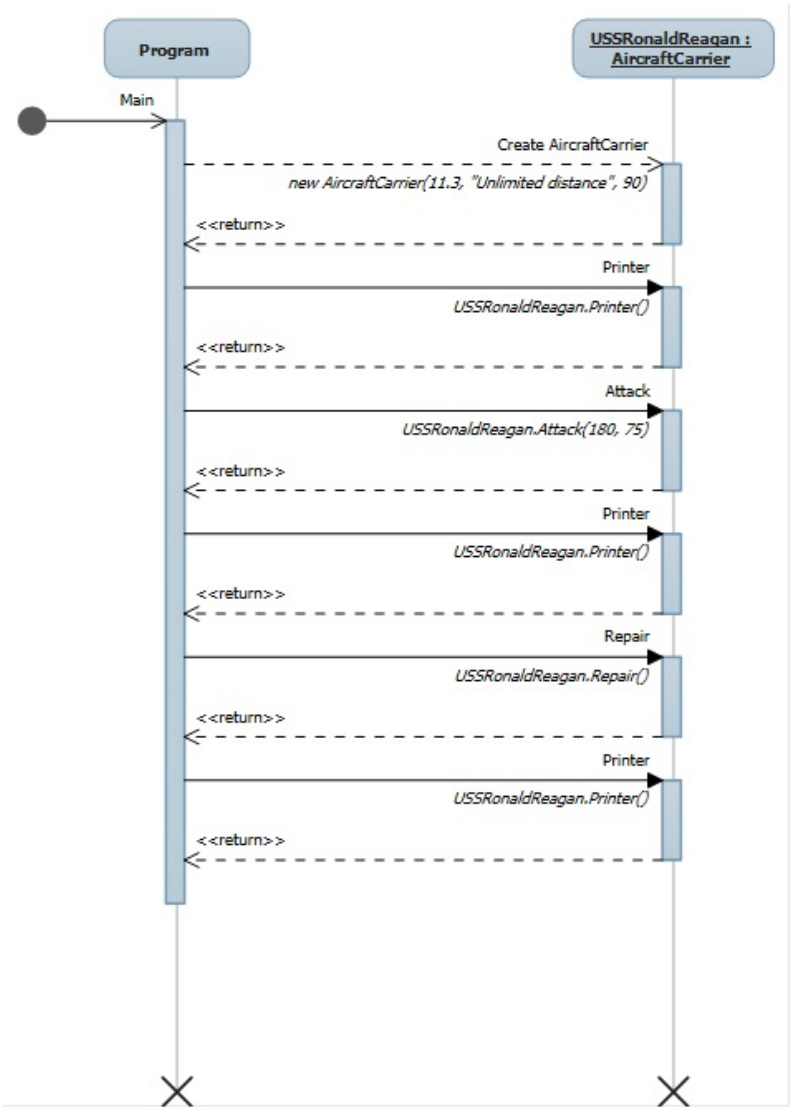
A dinamikus modell a rendszer időbeli viselkedését, sorrendiségét írja le, mely az objektumokat érő hatások, események és ezek sorrendje, a műveletek, a metódusok végrehajtásának ütemezése, az állapotok és azok változásainak rendje.

Azoknak az objektumoknak a viselkedését és együttműködését kell leírni, amelyek az objektummodellben szerepelnek. A viselkedés leírására elsősorban folyamatábrát, az állapotdiagramot és kommunikációs diagramot használják. Az állapotdiagram egy objektumpéldány külső események hatására történő állapotváltozásait és a válaszul adott reakcióinak időbeli sorrendjét adja meg.

A rendszer időbeli viselkedésének leírására, egy művelet végrehajtása során az üzenetek sorrendjét, a funkciót leírható kommunikáció-sorozatot rögzítő kommunikációs diagram is. Az objektumok egymás közötti üzenetváltásainak egy időtengely mentén történő ábrázolására szekvencia diagram segítségével történhet, melyben az objektumok életvonala egy felülről

lefelé mutató időtengelyt képvisel. Az üzenetek nyilakkal ábrázoltak, amelyik nyíl lejjebb található, az követi a felette magadott üzenetet.

A 18. ábrán látható, hogy először az objektumpéldány jön létre, majd az objektumon keresztül kiírjuk a képernyőre a repülőgép-hordozó kezdő adatait. A kiíratást követően támadás alá helyezzük a repülőgép-hordozót. A támadást követően kiíratjuk az értékeket, majd egy javítást végzünk az anyahajón. A javítást követően ismét kiírjuk annak állapotát.



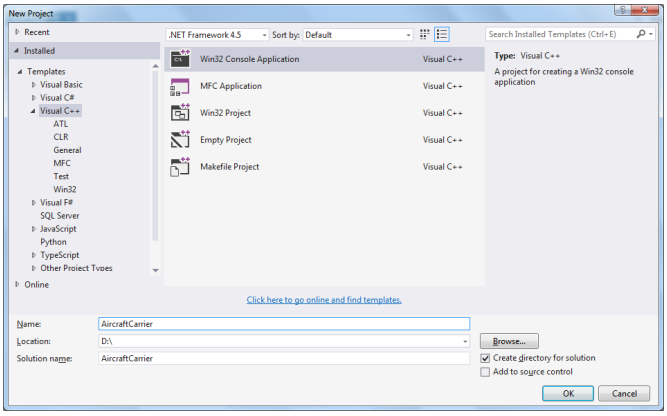
17. ábra:**Szekvencia diagram**

A projekt létrehozása

A projekt megvalósítása során konzolalkalmazást készítünk, ahol a Visual Studio, szabványos C++-nak megfelelő szintaktikát alkalmaz. A projekt három

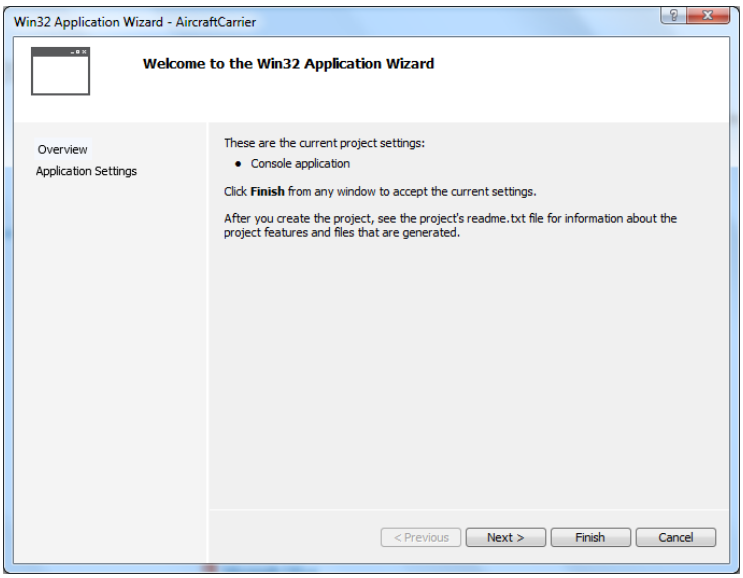
fájlban kerül implementálásra. Az alkalmazás belépési pontja a main függvény, amely a három fájl közül az egyikben fog helyet foglalni. Az előzőekben megfogalmazott minta példa kerül implementálásra és összeállításra, továbbá Windows operációs rendszeren futtatható kóddá történő lefordításra Visual Studio Community fejlesztőkörnyezet felhasználásával. A lefordított 32 bites alkalmazás konzolban kerül futtatásra.

Új projekt a **File** menü **New** menüpontjában hozható létre, a Project almenüt kiválasztva, vagy akár a **Ctrl+Shift+N** gyorsindító gombkombinációval. Új projekt kiválasztása után megjelenő ablakban a **Visual C++ Win32** sablont (**Template**) kell kiválasztani és a **Win32 Console Application**-t, mely az alábbi ábrán látható.



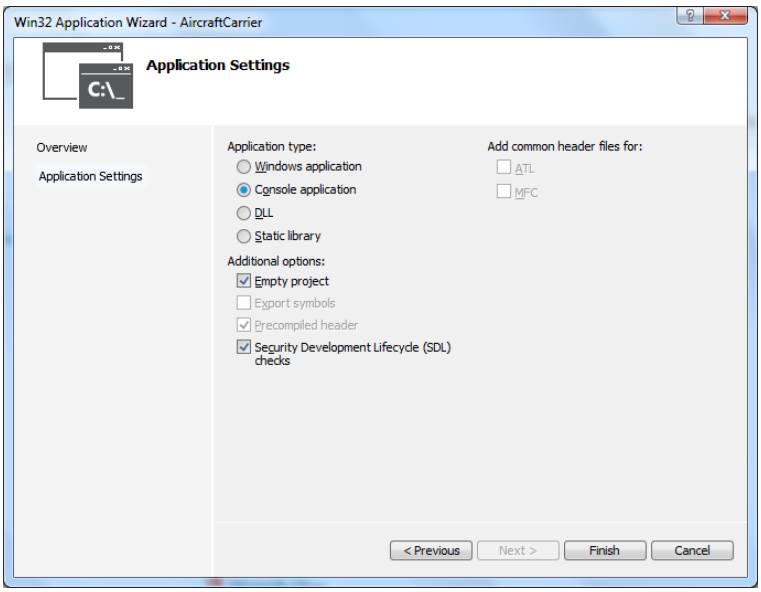
18. ábra: **Win32 Console Application projekt létrehozása**

Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően az **AircraftCarrier** név került megadásra. Ebben a részben lehet módosítani a Projekthez kapcsolódó fájlok mentési helyét is (**Location**). Egy összetett program általában több projektből tevődik össze, ezeket egységes rendszerré a megoldás (**Solution**) fogja össze. Az OK gombra kattintást követően elindul az alkalmazás varázsló, mely a következő ábrán látható.



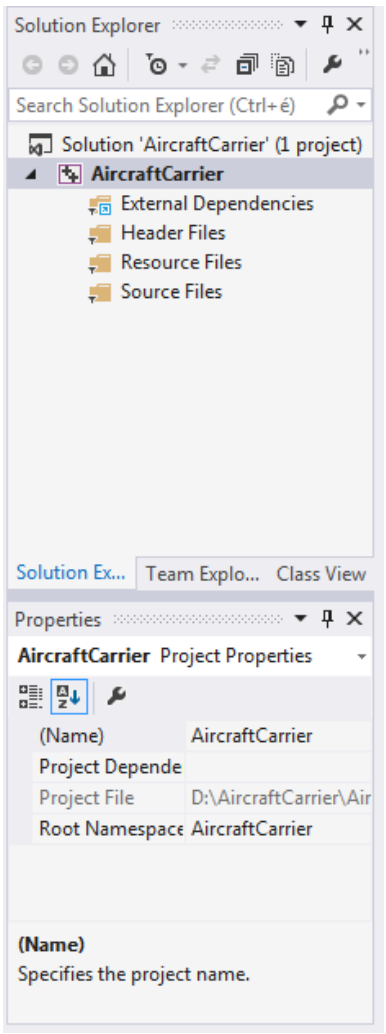
19. ábra: **Win32 konzol alkalmazás varázsló**

A **Next** gombra kattintva a beállítások ablak jelenik, meg, itt megadhatjuk az alkalmazás típusát (**Console Application**), és néhány kiegészítő opciót. Mivel egyszerű alkalmazást készítünk, ezért célszerű az üres projekt (**Empty project**) opció kiválasztása, ahogy az alábbi képen látszódik.



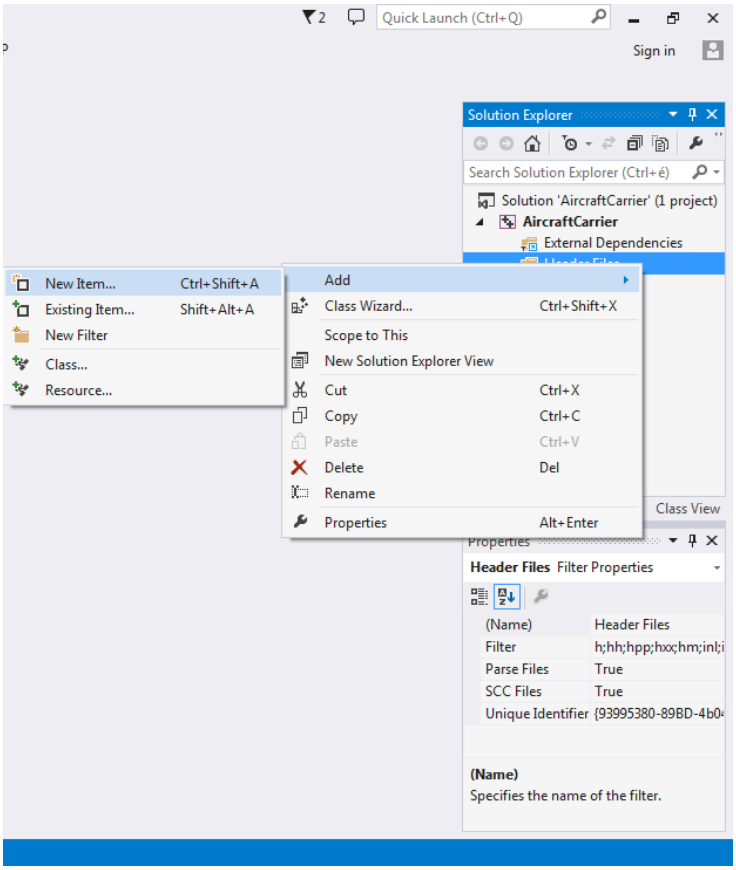
20. ábra: **Alkalmazás beállításai**

A **Finish** megnyomását követően a képernyő jobb szélén, a megoldás ablaka (**Solution Explorer**), alatta a beállítások ablak jelenik meg, ahogy a következő ábrán látható.



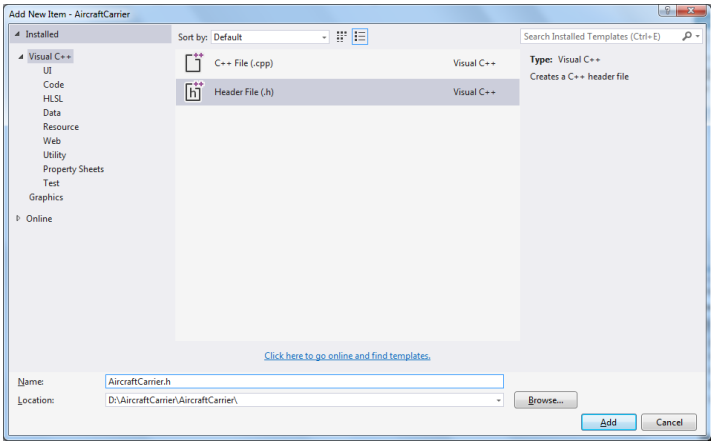
21. ábra:**Megoldás és beállítások ablak**

A program kódjának forrás és header fájljait (**Source Files, Header Files**) a megoldás ablakban a kiválasztott fájl típusra az egér jobb gombjával kattintva adható hozzá az alábbi ábrának megfelelően vagy a **Shift+A** billentyűkombináció megnyomásával.



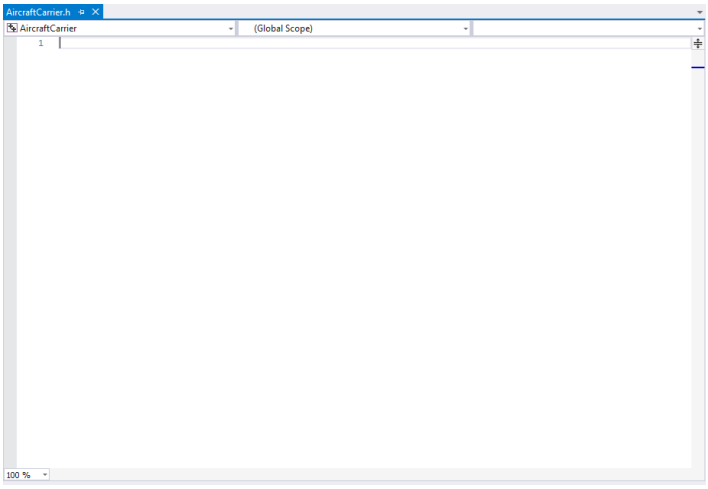
22. ábra: **Új header fájl hozzáadása a projekthez**

Új header fájl (**Header File**) választásával és a név (**Name**) az **AircraftCarrier.h** megadásával hozható létre az **AircraftCarrier.h** fájl, az alábbi ábrának megfelelően.



23. ábra: **Új header fájl hozzáadása a projekthez, név megadása**

Az új fájl hozzáadásával a képernyő középső részén megjelenik a forráskód szerkesztő az alábbi ábrának megfelelően.

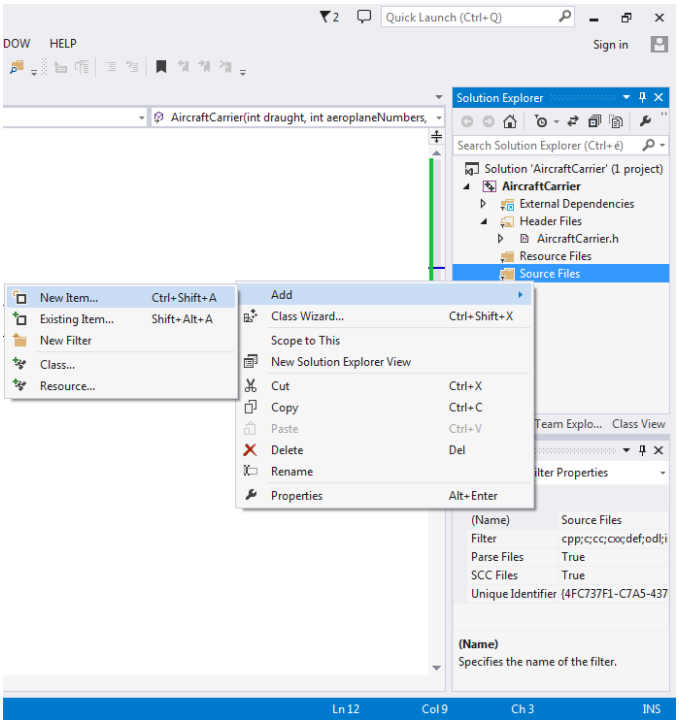


24. ábra: **Forráskód szerkesztő**

AircraftCarrier.h

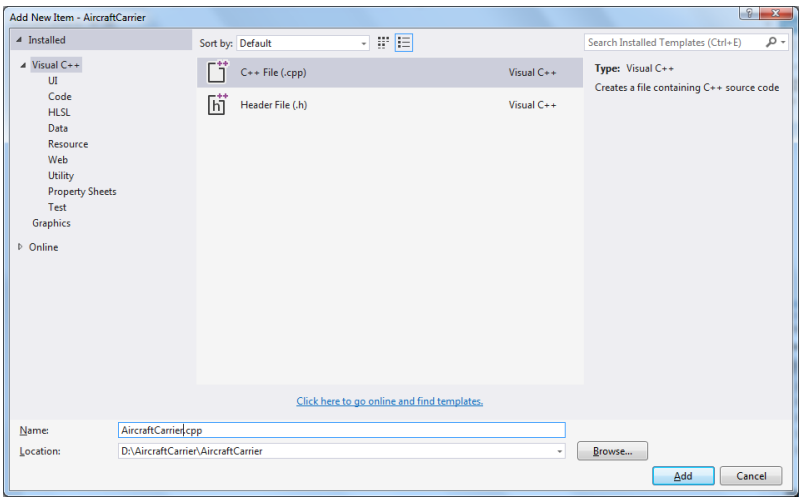
```
#ifndef AIRCRAFTCARRIER_H
#define AIRCRAFTCARRIER_H

class AircraftCarrier
{
private:
        int hitPoints;
        double draft;
        std::string range;
        double aircraftCarried;

public:
        AircraftCarrier(double draft, std::string range, double
        aircraftCarried, int hitPoints = 500);
        void Repair();
        void Attack(int otherAttackRating, double
        attackAeroPlaneNumbers);
        void Printer();
        ~AircraftCarrier(void);
};
#endif
```

A deklarált osztályok megvalósításait az AircraftCarrier.cpp forrásfájlok tartalmazzák, melyeket a **header** fájlokhoz hasonlóan, de a forrásfájl (**Source File**) hozzáadásával hozhatunk létre.



25. ábra:**Új forrásfájl hozzáadása a projekthez**

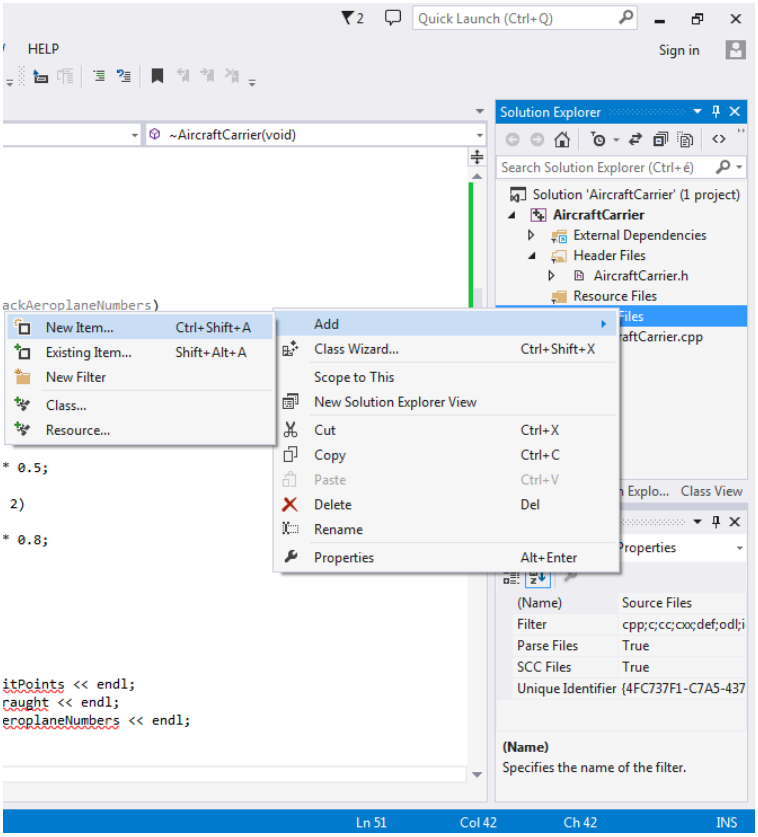
Új C++ fájl (**C++ File**) választásával és a név (**Name**) az AircraftCarrier.cpp megadásával hozható létre.



26. ábra: **Új forrásfájl hozzáadása a projekthez, név megadása**

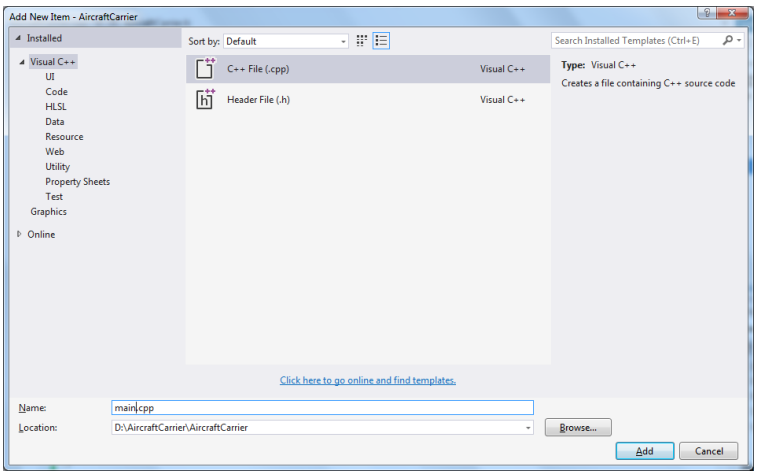
AircraftCarrier.cpp

```
#include <iostream>
#include <string>
#include "AircraftCarrier.h"

using namespace std;

AircraftCarrier::AircraftCarrier(double draft, string range, double
aircraftCarried, int hitPoints)
{
     this->draft = draft;
     this->range = range;
     this->aircraftCarried = aircraftCarried;
     this->hitPoints = hitPoints;
}

void AircraftCarrier::Repair()
{
     if (hitPoints + 70 < 500)
     {
        hitPoints += 70;
     }
     else
     {
        hitPoints = 500;
     }
}

void AircraftCarrier::Attack(int otherAttackRating, double
attackAircraftCarried)
{
     if (hitPoints>0)
     {
        if (hitPoints - otherAttackRating > 0)
        {
             hitPoints -= otherAttackRating;
             if (aircraftCarried / 2 >= attackAircraftCarried)
             {
                aircraftCarried -= attackAircraftCarried * 0.5;
             }
             else if (attackAircraftCarried >= aircraftCarried / 2)
             {
                aircraftCarried -= attackAircraftCarried * 0.8;
             }
        }
     }
}

void AircraftCarrier::Printer()
{
   cout << "A repulogephordozo eletereje          : " << hitPoints << endl;
   cout << "A repulogephordozo hatotavolsaga      : " << range << endl;
   cout << "A repulogephordozo merulesi melysege  : " << draft << endl;
   cout << "A repulogephordozon levo repulok szama: " << aircraftCarried <<
   endl;
}

AircraftCarrier::~AircraftCarrier(void){}
```



27. ábra: **Új forrásfájl hozzáadása a projekthez**

C++ fájl (**C++ File**) választásával és a név (**Name**) a main.cpp megadásával hozható létre.

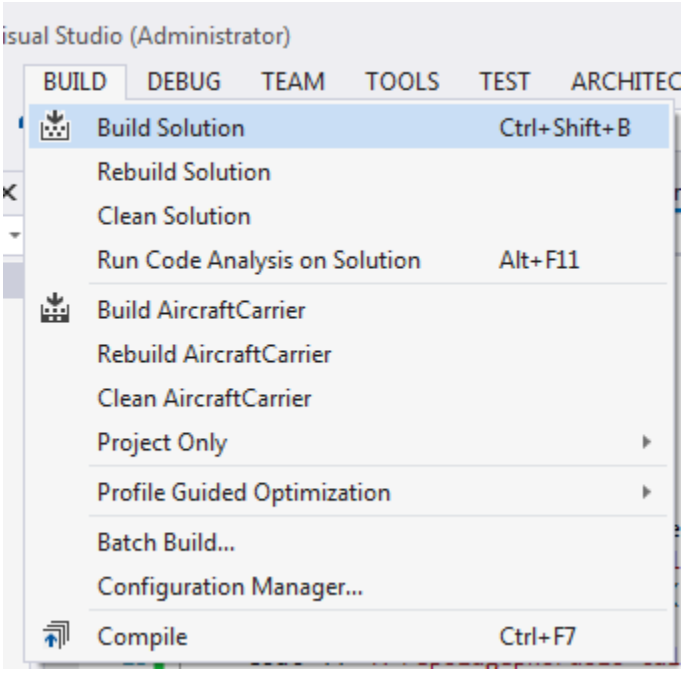


28. ábra: **Új forrásfájl hozzáadása a projekthez, név megadása**

main.cpp

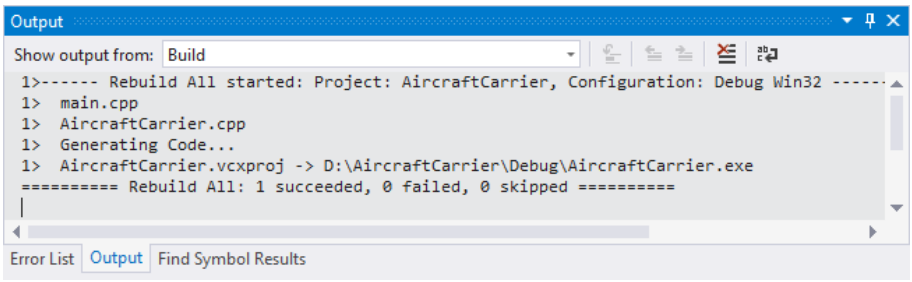
```
#include <iostream>
#include <conio.h>
#include "AircraftCarrier.h"

using namespace std;

int main()
{
    AircraftCarrier *USSRonaldReagan = new AircraftCarrier(11.3,
    "Unlimited distance", 90.0);
    cout << "A repulogephordozo tulajdonsagai! " << endl;
    USSRonaldReagan->Printer();
    cout << endl;
    cout << "A repolugephordozo tulajdonsagai a tamadas utan!" <<
    endl;
    USSRonaldReagan->Attack(180, 75);
    USSRonaldReagan->Printer();
    cout << endl;
    USSRonaldReagan->Repair();
    cout << "A repulogephordozo tulajdonsagai a javitas utan!" <<
    endl;
    USSRonaldReagan->Printer();
    delete USSRonaldReagan;

    _getch();
    return 0;
}
```

A forrásfájlok hozzáadása után a program forráskódja lefordítható, futtatható kód előállítását a **BUILD** menüpont **Build Solution** almenüpontjának kiválasztásával, vagy a **Ctrl+Shift+B** gyorsbillentyű kombináció megnyomásával tehető meg az alábbi ábrának megfelelően.



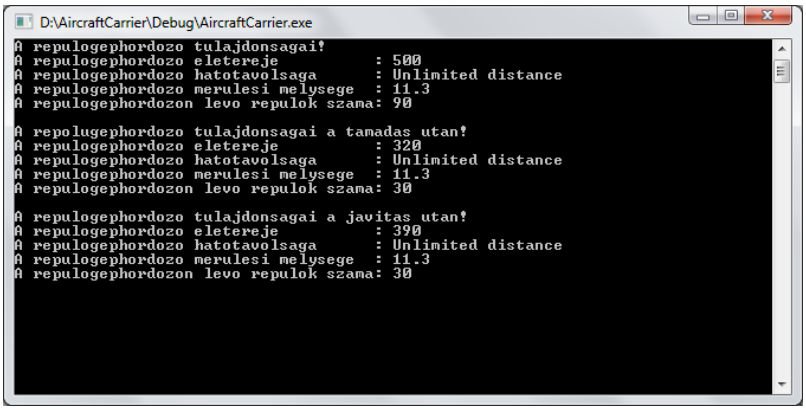
29. ábra: **Program fordítása**

Sikeres fordítás a kimenet (**Output**) ablakban látható.



30. ábra: **Kimenet ablak**

A program futásának eredménye konzolban jelenik meg, amely a következő ábrán látható.



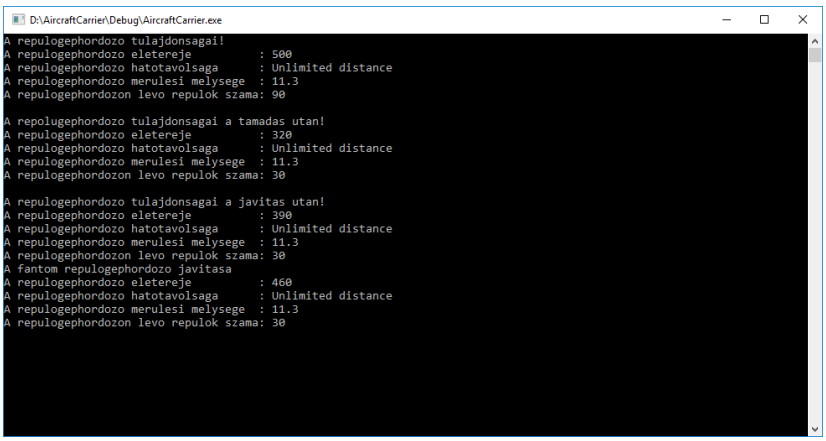
Továbbfejlesztés

Fejlesszük tovább a forráskódunkat oly módon, hogy a fentebb megismert másoló (**copy**) konstruktor segítségével hozzunk létre egy újabb objektumpéldányt, amely kezdőértékként kapja meg a fentebb megvalósított objektumpéldány már létező és inicializált értékeit. A programunkban készítsünk egy olyan klón repülőgép-hordozót, amely képes az eredeti repülőgép-hordozó képességeire és ugyanazon tulajdonságokkal rendelkezik. A klón feladata a bázis állomás meglátogatása, ahol úgy javítják ki a sérüléseket, hogy ezzel az eredeti repülőgép-hordozó életereje is megnő, miközben ő már az óceánon egy ellencsapásra készül. A feladat elvégzéséhez a **main.cpp** állományt kell az alábbi módon átdolgoznunk.

main.cpp

```
#include <iostream>
#include <conio.h>
#include "AircraftCarrier.h"

using namespace std;

int main()
{
	AircraftCarrier *USSRonaldReagan = new AircraftCarrier(11.3,
	"Unlimited distance", 90.0);
	cout << "A repulogephordozo tulajdonsagai! " << endl;
	USSRonaldReagan->Printer();
	cout << endl;
	cout << "A repolugephordozo tulajdonsagai a tamadas utan!" <<
	endl;
	USSRonaldReagan->Attack(180, 75);
	USSRonaldReagan->Printer();
	cout << endl;
	USSRonaldReagan->Repair();
	cout << "A repulogephordozo tulajdonsagai a javitas utan!" <<
	endl;
	USSRonaldReagan->Printer();

	AircraftCarrier *USSRonaldReaganPhantom(USSRonaldReagan);
	cout << "A fantom repulogephordozo javitasa" << endl;
	USSRonaldReaganPhantom->Repair();
	USSRonaldReaganPhantom->Printer();
	delete USSRonaldReagan;

	_getch();
	return 0;
}
```

A program futásának eredménye konzolban jelenik meg, amely a következő ábrán látható.


D:\AircraftCarrier\Debug\AircraftCarrier.exe
A repulogephordozo tulajdonsagai!
A repulogephordozo eletereje : 500
A repulogephordozo hatotavolsaga : Unlimited distance
A repulogephordozo merulesi melysege : 11.3
A repulogephordozon levo repulok szama: 90
A repolugephordozo tulajdonsagai a tamadas utan!
A repulogephordozo eletereje : 320
A repulogephordozo hatotavolsaga : Unlimited distance
A repulogephordozo merulesi melysege : 11.3
A repulogephordozon levo repulok szama: 30
A repulogephordozo tulajdonsagai a javitas utan!
A repulogephordozo eletereje : 390
A repulogephordozo hatotavolsaga : Unlimited distance
A repulogephordozo merulesi melysege : 11.3
A repulogephordozon levo repulok szama: 30
A fantom repulogephordozo javitasa
A repulogephordozo eletereje : 460
A repulogephordozo hatotavolsaga : Unlimited distance
A repulogephordozo merulesi melysege : 11.3
A repulogephordozon levo repulok szama: 30

Gyakorlópélda

Írjunk egy Cirkáló hajót reprezentáló osztályt (*Cruiser*), mely képes lokalizálni az ellenség közeledését és figyelmeztetni a flottát, illetve a lokalizáció mellett legyen képes a támadásra is.

Az osztály tagváltozói a következőek:

- Életerő (*hitPoints*), mely egy egész szám 0 és 500 között.
- Támadóerő (*attackPoint*), mely egy egész szám 500 és 1000 között.
- Radar hatósugara (*scannerReach*), mely egy lebegőpontos szám 0.0 és 500.0 (km) között.

Ügyeljünk az egységbezárás elvére, tehát az adattagok private láthatóságú szinttel rendelkezzenek!

Valósítsuk meg az osztály metódusait:

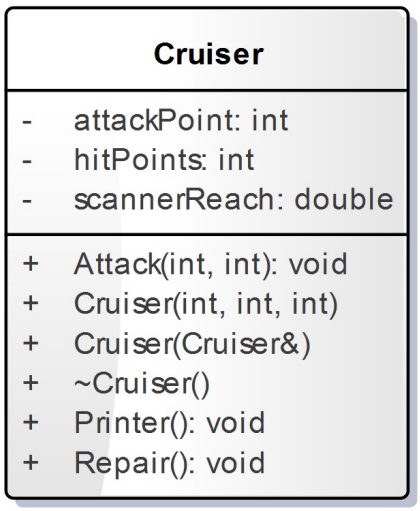
- Írjuk meg az osztály konstruktorát, melyben kötelező megadni a cirkáló életerejét és a támadóerejét! Az radar hatósugarának az értékét is lehessen megadni, de ne legyen kötelező (használjunl default paramétereket). Ha a programozó nem ad meg életerőt a példányosítás során, a tagváltozó kezdeti értéke 500.0 (km) legyen)! A konstruktor megvalósításnál alapértelmezett paraméteres konstruktort használjunk.
- Írjunk egy *Repair()* függvényt, mely 60 ponttal növeli a cirkáló életerejét! Ügyeljünk rá, hogy 500 fölé soha ne kerülhessen az érték.
- Írjunk egy *Attack(otherAttackRating, distance)* függvényt, amely támadás alá helyezi a hajónkat! A függvény tartalma csakis akkor fusson le, ha a cirkáló életereje nagyobb mint 0! A támadás hatására a cirkáló életereje csökkenjen (*hitPoints* - *otherAttackRating*). Ha a radar hatótávolsága nagyobb, mint a *distance* érték, akkor képes figyelmeztetni az ellenségre a flotta többi tagját, ha kisebb, akkor nem. Ügyeljünk rá, hogy 0 alá nem csökkenhet a cirkáló életereje.
- Írjunk egy egyszerű *Printer()* függvényt, amely minden fontos adatot kiír az anyahajóról.
- Ne feledkezzünk meg az objektumpéldány által lefoglalt erőforrás

felszabadításáról, a destruktor megírásáról sem.

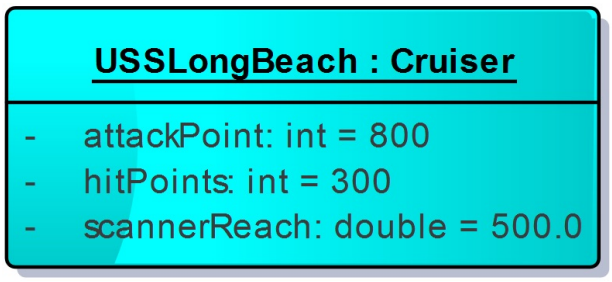
- Írjunk egy egyszerű *main* függvényt, amely létrehoz egy cirkáló hajót, amelyet támadás alá helyezünk. Minden esemény után írjuk ki a cirkáló állapotát!

Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén [heap]) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektum példányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. Amennyiben sikerült megvalósítanunk a feladatot, fejlesszük tovább a cirkálónk képességét. Legyen képes a klónozásra, tehát rendelkezzen ugyanazon képességekkel és tulajdonságokkal, mint az eredeti cirkáló. A klón feladata a bázis állomás meglátogatása, ahol úgy javítják ki a sérüléseket, hogy ezzel az eredeti cirkáló életereje is megnő, miközben ő már az óceánon igyekszik a flotta többi tagjának segítséget nyújtani és egy ellencsapásra készül. A feladat elvégzéséhez a **main.cpp** állományt kell az alábbi módon átdolgoznunk.

Osztály diagram



Objektum diagram



Megoldás

Cruiser.h

```
#ifndef CRUISER_H
#define CRUISER_H

class Cruiser
{
        private:
                int hitPoints;
                int attackPoint;
                double scannerReach;

        public:
          Cruiser(int hitPonts, int attackPoint, int
          scannerReach=500.0);
          Cruiser(const Cruiser &USSLongBeachPhantom);
                void Repair();
                void Attack(int otherAttackRating, int distance);
                void Printer();
                ~Cruiser(void);
};
#endif
```

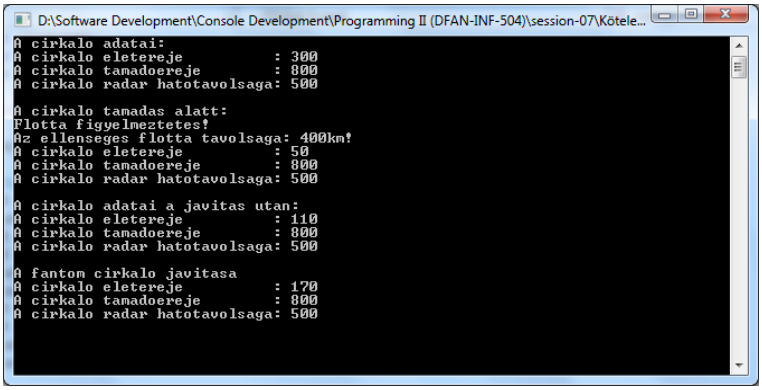
Cruiser.cpp

```
#include <iostream>
#include "Cruiser.h"

using namespace std;

Cruiser::Cruiser(int hitPoints, int attackPoint, int scannerReach)
{
        this->hitPoints=hitPoints;
        this->attackPoint=attackPoint;
        this->scannerReach=scannerReach;
}

Cruiser::Cruiser(const Cruiser &USSLongBeachPhantom)
{
        this->hitPoints = USSLongBeachPhantom.hitPoints;
        this->attackPoint = USSLongBeachPhantom.attackPoint;
        this->scannerReach = USSLongBeachPhantom.scannerReach;
}


void Cruiser::Repair()
{
        if (60 + hitPoints < 500)
        {
                hitPoints += 60;
        }
        else
        {
                hitPoints = 500;
        }
}

void Cruiser::Attack(int otherAttackRating, int distance)
{
        if (hitPoints>0)
        {
                if (hitPoints - otherAttackRating > 0)
                {
                        hitPoints -= otherAttackRating;
                }
        }

        if (scannerReach > distance)
        {
          cout << "Flotta figyelmeztetes! \nAz ellenseges flotta
          tavolsaga: " << distance << "km!\n";
        }
        else
        {
          cout << "Nincs ellenseges flotta a radar hatotavolsagan
          belul!\n";
        }
}
void Cruiser::Printer()
{
     cout << "A cirkalo eletereje          : " << hitPoints << endl;
     cout << "A cirkalo tamadoereje        : " << attackPoint <<
     endl;
     cout << "A cirkalo radar hatotavolsaga: " << scannerReach <<
     endl;
}

Cruiser::~Cruiser(void){}
```

main.cpp

Program futásának eredménye:

```cpp
#include <iostream>
#include <conio.h>
#include "Cruiser.h"

using namespace std;

int main()
{
	Cruiser *USSLongBeach = new Cruiser(300, 800);
	cout << "A cirkalo adatai:" << endl;
	USSLongBeach->Printer();
	cout << endl;
	cout << "A cirkalo tamadas alatt:" << endl;
	USSLongBeach->Attack(250, 400);
	USSLongBeach->Printer();
	cout << endl;
	cout << "A cirkalo adatai a javitas utan: " << endl;
	USSLongBeach->Repair();
	USSLongBeach->Printer();

	Cruiser *USSLongBeachPhantom(USSLongBeach);
	cout << endl;
	cout << "A fantom cirkalo javitasa" << endl;
	USSLongBeachPhantom->Repair();
	USSLongBeachPhantom->Printer();

	delete USSLongBeach;

	_getch();
	return 0;
}
```



Felhasznált irodalom

Andrei Alexandrescu, Herb Sutter: *C++ kódolási szabályok.* Kiskapu Kft. 2005.

B. Stroustrup: *A C++ programozási nyelv I II. kötet.* Kiskapu Kiadó, 2001.

Benedek Zoltán - Levendovszky Tihamér: *Szoftverfejlesztés C++ nyelven.* SZAK kiadó, 2007.

Benkő Tiborné - Poppe András: *Objektumorientált C++ (Együtt könnyebb a programozás*, ComputerBooks, 2010.

Craig Larman: *Applying UML and patterns*, Second Edition, Prentice-Hall, Inc., USA, 2002.

Fóthi Ákos: *Bevezetés a programozáshoz.* ELTE Eötvös Kiadó, 2005.

Hans-Erik Eriksson, Magnus Penker: *Business Modeling with UML*, John Wiley & Sons, Inc., New York, 2000.

Ian Sommerville: *Software Engineering.* Seventh Edition, Pearson Education, Inc., USA, 2004.

Kent Beck: *Implementációs minták.* Panem kiadó, 2008.

Kondorosi Károly - László Zoltán - Szirmay-Kalos László: *Objektum-orientált szoftverfejlesztés*. CompuerBooks, 2004.

Mark Priestley: *Practical Object-Oriented Design with UML.* McGraw-Hill Publishing Company, Great Britain, 2000.

Roger S. Pressman: *Software Engineering.* Fifth Edition, McGraw-Hill Book Company, USA, 2001.

Sike Sándor – Varga László: *Szoftvertechnológia és UML*. ELTE, 2007

Stephen R. Schach: *Object-Oriented and Classical Software Engineering*. Eighth Edition McGraw-Hill, New York, 2011.