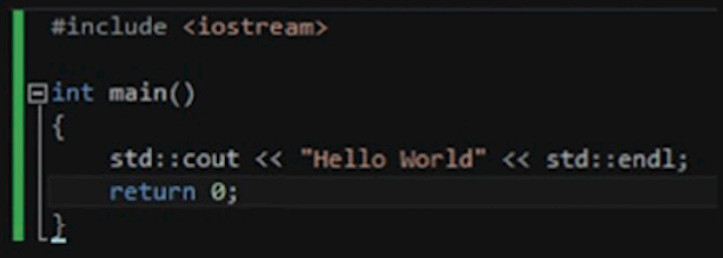
HelloWorld

I. KÖTET

Katona József, Kővári Attila

OBJEKTUMORIENTÁLT SZOFTVERFEJLESZTÉS ALAPJAI

Gyakorlatorientált szoftverfejlesztés C++ nyelven Visual Studio Community fejlesztőkörnyezetben



```
#include <iostream>

int main()
{
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

Objektumorientált szoftverfejlesztés alapjai

Gyakorlatorientált szoftverfejlesztés C++ nyelven

Visual Studio Community fejlesztőkörnyezetben

@Katona József – Kővári Attila
Lektorálta: Dr. Fauszt Tibor

DOI: 10.18395/katona.kovari.objektumorientalt.gyakorlatorientalt.2015
ISBN 978-963-397-708-8

Publio Kiadó
Felelős kiadó: Publio Kiadó Kft.
2015

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. A C++ programozási nyelv szabványának aktuális, 2003-as változatának kódjelzése ISO/IEC 14882:2003. A nyelv törekszik a könnyebben megírható, karbantartható és újrahasznosítható kód előállítására, mely helyes alkalmazása középszintű programozási ismereteket kíván.

E könyv a C++ nyelven történő objektumorientált szoftverfejlesztést megismertető sorozat első kötete, mely sorozat gyakorlati példákon keresztül segíti a nyelv elsajátítását. A könyv segítséget kíván nyújtani a programozók számára a nyelv megismerésében, kiemelve azokat a jellegzetességeket, amikre figyelniük kell a kódolás során, valamint nyelv pontos ismeretével egyben azt is, hogy miként lehet csökkenteni a hibalehetőségeket, megelőzni a helytelen működést a nyelv megfelelő alkalmazásával. A könyvben található példák továbbá segítséget nyújtanak az átlátható kód készítésében is.

A sorozat a C++ nyelven történő programozás elsajátítását nem a legelejétől kívánja ismertetni, hanem támaszkodik egyrészt a C nyelv, az alapvető adatstruktúrák (dinamikus tömbök, láncolt listák, bináris fák) és algoritmusok (keresés, rendezés) ismeretére, valamint alapszintű programozói tudásra (például a vezérlési szerkezetek).

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.

Úgy gondoljuk, hogy ez a könyv nagymértékben hozzájárulhat ahhoz, hogy különböző szakemberek vagy akár középiskolai diákok is bővíteni tudják meglévő programozói ismereteiket. A C++ nyelv megismerése, a programozói tudás bővítése a munkaerőpiacon is igen nagy előnyt jelent.

Jelen kötet az objektumorientált szoftverfejlesztés alapjait mutatja be, ezen belül az objektumorientált programozás módszertanát, legfontosabb alapfogalmait, osztály, öröklés, példányosítás, objektum és más fontos fogalmakat, továbbá röviden bemutatja a szoftver modelleket és egy projekten keresztül lépésről-lépésre a Visual Studio Community fejlesztőkörnyezet használatát, melyben a bemutatott mintapélda implementálható és összeállítható, a futtatható kód előállítható.

Köszönettel tartozunk a kézirat lektorának, Dr. Fauszt Tibornak szakmai tanácsaiért és módszertani javaslataiért.

A Szerzők

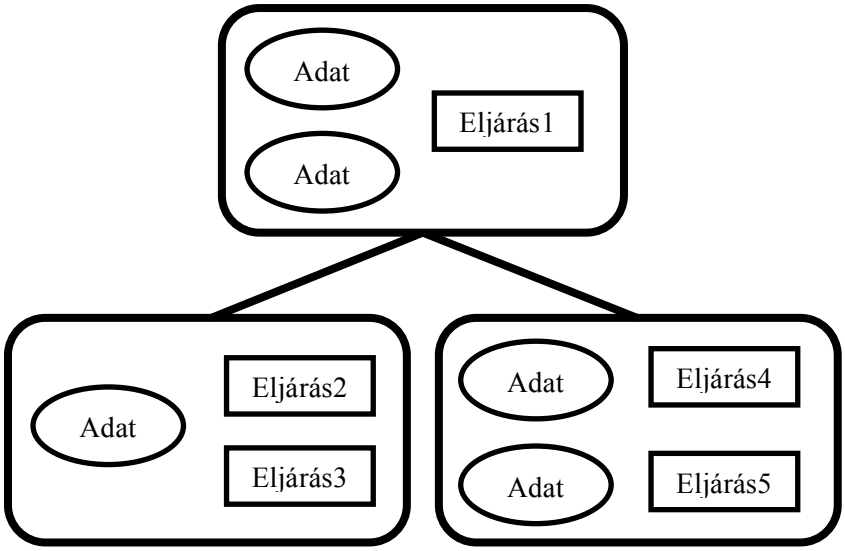
Objektumorientált szoftverfejlesztés

A számítógépes program a körülöttünk lévő világ elemeihez illeszkedik, valamilyen gyakorlati feladatot old meg. Programozás alatt egy adott a feladat megoldási algoritmusának egy bizonyos programozási nyelven történő megvalósítását értjük. Azonban a szoftverfejlesztés alatt ennél sokkal többet értünk, az magában foglalja mindazon módszereket és eszközöket, amelyek célja a hatékony és megbízható program kód készítése. Magát a programozást, mint implementációt, vagyis egy algoritmus megvalósulását, a szoftverfejlesztés egyik lépéseként kezelik. A szoftverfejlesztés lényegében a követelményspecifikációt, programtervezést (programszerkezet, használandó eszközök), forrásprogram elkészítését, vagyis a kódolást/implementációt, tesztelést és a dokumentáció elkészítését foglalja magában.

Objektumorientált programozási módszertana

A szisztematikus programtervezés alkalmazása során egy információfeldolgozási probléma két oldalról közelíthető meg: az algoritmusok, illetve az adatok oldaláról. A feladatok jellegétől függően egyik vagy másik oldal nagyobb hangsúlyt kaphat. A programozás során egy adott megoldandó feladat leképzéséhez célszerű olyan modellt alkalmazni, amely leírásában illeszkedik a környező világ építőelemeihez, leírja azok tulajdonságait és viselkedését. Ezt az elvet valósítja meg az objektumorientált (object-oriented programming, röviden OOP) programozási módszertan. Ellentétben más, korábbi programozási módszertanokkal, nem a műveletek megalkotása áll a középpontban, hanem az egymással kapcsolatban álló programegységek hierarchiájának megtervezése. Az objektum orientált módszertan alkalmazásával a kifejlesztendő rendszert együttműködő objektumokkal modellezzük. Ebben a modellben egyedi objektumok halmazával és azok kapcsolataival, viselkedésével, együttműködésével, adatstruktúrájával és aktuális állapotával, adatainak aktuális értékével írjuk le magát a folyamatot (1. ábra).

Az objektumorientált programozási módszer alkalmazásával a programok bonyolultsága csökkenthető, a megbízhatóság és a hatékonyság növelhető. Az objektumorientált nyelvekben a feladat megfogalmazásában és a megoldásában részt vevő objektumok osztályokba sorolhatók. Az egy osztályba tartozó objektumoknak a viselkedése hasonló, de az adattartalmuk eltér.



1. ábra: Objektumorientált programozás modellje

Az objektumorientált programozást három fontos dolog jellemez:

- Egységbezárás: az adatstruktúrákat és az azokat kezelő függvényeket (metódusokat) egy egységként kezeljük, és elzárjuk őket a külvilág elől. Az így kapott egységet az objektum, mely állapota csak a műveletein keresztül módosítható.
- Öröklés: a meglévő objektumokból leszármaztatott újabb objektumok öröklik a definiálásukhoz használt ős objektumok egyes adatstruktúráit és függvényeit, ugyanakkor újabb tulajdonságokat is definiálhatnak, vagy az ős objektumét újraértelmezhetik.
- Polimorfizmus (többalakúság): egy metódus azonosítója közös lehet egy adott objektum hierarchiában, de a hierarchia minden egyes objektumában az implementációja az adott objektumra nézve specifikus lehet.

Az objektumorientált modellezés alapján egy repülő felfogható a *Jármű* osztály egy tagjaként, annak egy objektumaként. Minden jármű objektum rendelkezik a járművekre jellemző általános tulajdonságokkal (pl.: mozgás módja, szállítható dolgok, szállítható mennyiség, mozgás sebessége stb.), illetve feladat végrehajtási képességgel, funkcióval, amely járművek esetén a szállítás. Az objektum orientált programozásban fontos az öröklődés, amely az osztályok egymásból való származtatása, például a *Repülő* osztály származhat a *Jármű* osztályból, így megörökli a *Jármű* osztály tulajdonságait és képességeit, de akár ki is bővítheti vagy felülbírálhatja azokat az adott repülőgép tulajdonságaival, képességeivel.

Objektumorientált programozással kapcsolatos alapfogalmak

Az 1. Táblázat összefoglalóan tartalmazza az objektumorientált programozással kapcsolatos fontosabb alapfogalmak rövid leírását, melyek magyarázata későbbiekben kerül részletezésre.

1. Táblázat Objektumorientált programozással kapcsolatos fontosabb alapfogalmak

Fogalom	Leírás
Osztály (class)	Metódusok, mezők és jellemzők egy egységbe zárt csoportja, illetve ennek (típus)deklarációja.
Osztály definíció, implementáció	Az osztály deklarációjának egy megvalósítása.
Példányosítás	Egy osztályt létrehozása után példányosítani kell, hogy használhatóvá váljanak a benne lévő rutinok. Példányosításkor hívódik meg az osztály konstruktora.
Objektum	Az osztály egy példánya.
Ősosztály	Az az osztály, amelytől egy másik osztály örököl.
Leszármaztatott osztály	Az az osztály, amelyik egy másik osztálytól örököl.
Osztály- vagy objektumhierarchia	Az osztályok között az öröklési viszonyok meghatározásának következtében kialakuló (család)faszerkeszet.
Konstruktor	Az osztály példányosításakor meghívott, az adott objektumpéldányt alapállapotba állító, kezdőértékeit megadó (inicializáló) speciális metódus.
Destruktor	Az objektumpéldány megszüntetésekor, megsemmisítésekor meghívásra kerülő speciális metódus, mely felszabadítja az objektum által lefoglalt erőforrásokat.
Metódus	Egy adott osztályhoz tartozó, az osztályon vagy az adott típusú objektumpéldányon műveletet végző függvény vagy eljárás.
Mező	Egy adott osztály részét képező, objektumok között nem megosztott (minden példány külön-külön mezőkészlettel rendelkezik) független adatmező.
Osztálymetódus	Olyan metódus, amely nem egy konkrét objektumpéldányon, hanem magán az osztályon végez műveleteket. Az osztálymetódusok az osztály példányosítása nélkül is meghívhatók. Természetesen nem érhetik el és nem hívhatják meg az osztályban deklarált nem osztály mezőket/metódusokat (azok csak egy-egy objektumpéldányon belül léteznek).
Osztálymező	Az osztálymezők az osztály egyetlen mezői, amelyeket az osztálymetódusok is elérnek. Olyan speciális mező, amely az adott osztály – és a belőle leszármazott osztályok – minden

	objektumpéldánya között megosztásra kerül, a globális változókhoz hasonló szerepet tölt be (azaz ha egyikük változtat rajta, az összes többi példány osztálymezője is tükrözni fogja a változtatást).
Jellemző	Olyan speciális osztályelem, amely mező ill. változó módjára viselkedik, de olvasása és írása speciális olvasó (getter) és/vagy író (setter) metódusokkal történik, mellyel ellenőrzés is végezhető.
Hozzáférhetőség (láthatóság)	Public: Az objektumot használó bármely kód számára közvetlenül hozzáférhető. Protected: Közvetlenül nem hozzáférhető, de a származtatott osztályok használhatják. Private: Csak abban az osztályban érhetők el, amelyikben meghatározták őket (ill. friend osz).

Osztály létrehozása a *class* kulcsszóval történik, az osztály neveit nagybetűvel szokás kezdeni. Az osztályok adattagokból és tagfüggvényekből épülnek fel. A hatókör (::) operátor lehetővé teszi, hogy a statikus, a konstans adattagokat illetve tagfüggvényeket elérjük. Ezekhez a tagokhoz való hozzáférést korlátozhatjuk, általánosan elfogadott szabály, hogy az osztály adattagjai *private*, amíg a különböző feladatok végrehajtásáért felelős tagfüggvények *public* hozzáférésűek.

A *this* objektumreferencia 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.

Az osztály példányosítása történhet statikus, illetve dinamikus módon. C++–ban operátorok felelnek a dinamikus memóriakezelésért. Több statikus objektumpéldány definiálása során a memóriában csak az adattagok többszöröződnek, így minden egyes objektumpéldány saját memóriaterülettel rendelkezik. Azonban a metódusok számára a memória csak egyetlen példányban kerül lefoglalására, minden példány közösen használja azokat. Ha egy statikus objektumpéldányon keresztül szeretnénk elérni az osztály nyilvános adattagjait és tagfüggvényeit, akkor azt az objektum neve után megadott pont (.) operátorral tehetjük meg. Dinamikus objektumpéldány a *new* operátor segítségével hozható létre, mely a lefoglalt típusra mutató pointerrel tér vissza, ez objektumpéldány esetében a példányra mutató pointert jelenti. A dinamikusan létrehozott objektum esetén az osztály adattagjai és tagfüggvényei a nyíl (->) operátorral érhetőek el. A lefoglalt területet a delete (~) operátorral szabadíthatjuk fel.

Az alábbi példák a fenti fogalmak C++ nyelven történő implementációját mutatják be, melyek a későbbiekben részletesen ismertetve lesznek:

Osztály deklaráció:

```
//osztálydeklaráció
class Jarmu
{
   //adattagok
   private:
      float fogyasztas;
      float tankMeret;
   protected:
      int maxUtas;

   //tagfüggvények
   public:
      Jarmu(int maxUtas, float fogyasztas, float tankMeret);
   protected:
      float MaximalisSzallitasiTavolsag();
};
```

Osztály definíció / implementáció:

```
Jarmu::Jarmu(int maxUtas, float fogyasztas, float tankMeret)
{
   this->maxUtas = maxUtas;
   this->fogyasztas = fogyasztas;
   this->tankMeret = tankMeret;
}
float Jarmu::MaximalisSzallitasiTavolsag()
{
   return tankMeret/fogyasztas * 100;
}
```

Az egyes változók inicializálása egyszerűen taginicializálással is megadhatóak az alábbi forma szerint:

```
Jarmu::Jarmu(int maxUtas, float fogyasztas, float tankMeret) :
maxUtas(maxUtas), fogyasztas(fogyasztas), tankMeret(tankMeret) {}

float Jarmu::MaximalisSzallitasiTavolsag()
{
   return tankMeret/fogyasztas * 100;
}
```

Példányosítás, objektum:

```
//objektumpéldány (dinamikus)
Jarmu* jarmuObjektum = new Jarmu(7, 6.8, 50.5);
```

Öröklés, ősosztály, leszármaztatott osztály:

```
//leszármaztatás
class Repulo : public Jarmu
{
   private:
      float maxFelszalloTomeg;
      float repuloSzarazTomeg;
      float tuzeloanyagTomeg;

   public:
      Repulo(int maxUtas, float fogyasztas, float tankMeret, float
      maxFelszalloTomeg, float repuloSzarazTomeg, float
      tuzeloanyagTomeg);

   protected:
      float SzallithatoRakomanyMaximalisTomege()
      {
         //pl.: 100 kg egy utas és egy poggyász
         return maxFelszalloTomeg - repuloSzarazTomeg –
         tuzeloanyagTomeg - maxUtas * 100
      }
};
```

```
//objektumpéldány (dinamikus)
Repulo* Boeing737 = new Repulo(141, 378.1, 23817.0, 56740.0, 32881.0,
12618.0);
//oszályon belüli függvényhívás
maximalisRakomany = Plane->SzallithatoRakomanyMaximalisTomege();
//ősosztályból örökölt függvényhívás
hatoTavolsag = Plane->MaximalisSzallitasiTavolsag();
```

Konstruktor, destruktor:

```
//konstruktor
Jarmu::Jarmu(int maxUtas, float fogyasztas, float tankMeret)
{
   this->maxUtas = maxUtas;
   this->fogyasztas = fogyasztas;
   this->tankMeret = tankMeret;
}
//destruktor
Jarmu::~Jarmu(void){}
```

Mező, metódus:

```
//adatmező
int maxUtas;
float fogyasztas;
float tankMeret;
//metódus
float Jarmu::MaximalisSzallitasiTavolsag()
{
   return tankMeret/fogyasztas * 100;
}
```

Jellemző:

```
//getter metódus létrehozása private láthatóságú adattag számára
float Jarmu::GetMaxUtasLetszam() const
{
   return maxUtas;
}
//setter metódus létrehozása private láthatóságú adattag számára
void Jarmu::SetMaxUtasLetszam(int letszam)
{
   maxUtas=letszam;
}
```

Hozzáférhetőség:

```
//public (nyílvános) láthatóság
public:
   int maxUtas;
//protected (védett) láthatóság
protected:
   int fogyasztas;
//private (zárt) láthatóság
private:
   float tankMeret;
```

Szoftver modellek

Az előző fejezetekben leírt tulajdonságok alapján egy objektumorientált szoftver esetében a rendszert elsősorban három szempont szerint vizsgáljuk:

- adatok;
- műveletek (funkciók, adattranszformációk);
- vezérlés (viselkedés).

E három szempont az alapja az objektumorientált módszertanokban használt modelleknek is.

Objektummodell, statikus modelltervezés

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.

Az osztálydiagram objektumokat és az objektumok közötti kapcsolatokat jeleníti meg, míg az objektumdiagram az egyedi objektumpéldányok kapcsolatait tünteti fel.

Az osztálydiagramban az osztályok, illetve azok objektumai közötti kapcsolatot kifejező relációk lehetnek:

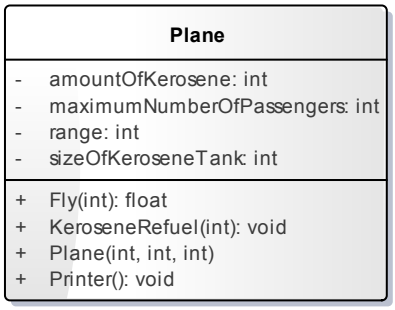
- öröklődés;
- asszociáció;
- aggregáció;
- kompozíció.

Az öröklődés osztályok közötti kapcsolatot adja meg, míg a másik három, a résztvevő osztályok objektumait kapcsolja össze. Az osztályok azonosítóját nagybetűvel szokás kezdeni. Az osztály neve alatt helyezkednek el az adattagok, amelyek az objektum tulajdonságaiért felelnek, illetve az adattagok alatt – az objektumok képességeit megvalósító – tagfüggvények találhatóak. A szabványos osztálydiagramban a public „+”, a private „-”, a protected „#” karakterrel kerül megjelölésre.

A szabványos osztálydiagramban az osztály jelölése a következő:

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

Példa:



2. ábra: Osztály megadása

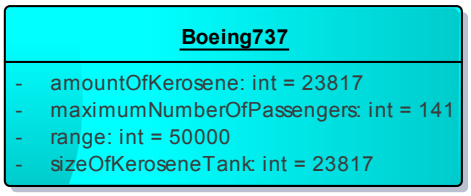
A rendszerhez egy osztálydiagram tartozik, ugyanakkor egy osztálydiagramhoz több objektumdiagram tartozhat. Az objektumdiagramban az osztályok helyébe azok példányai, az objektumok kerülnek. Az objektumok a rendszer működése során dinamikusan jönnek létre, változnak és szűnnek meg, tehát az idő függvényében változó, különböző állapotokat vehetnek fel.

Szabványos megadásában az objektum neve, illetve annak az osztálynak a neve, amelyből az objektum létrejött szerepel a felső részben. Alatta az objektum adott pillanatban meglévő állapotát lehet jelölni, az attribútumai aktuális értékének feltüntetésével.

A szabványos objektumdiagramban az objektum jelölése a következő:

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

Példa:

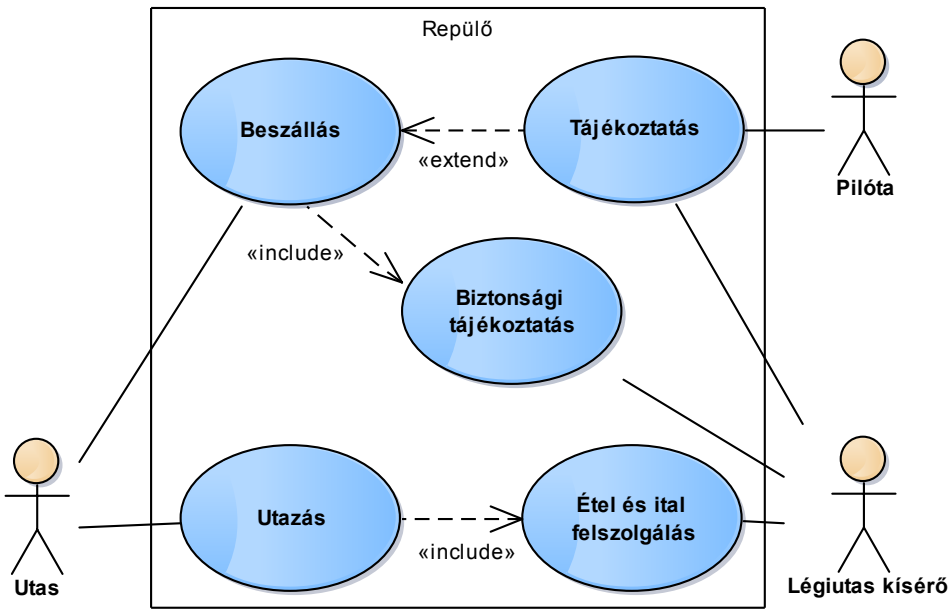


3. ábra: Objektum megadása

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.



4. ábra: Használati eset (use case) diagram

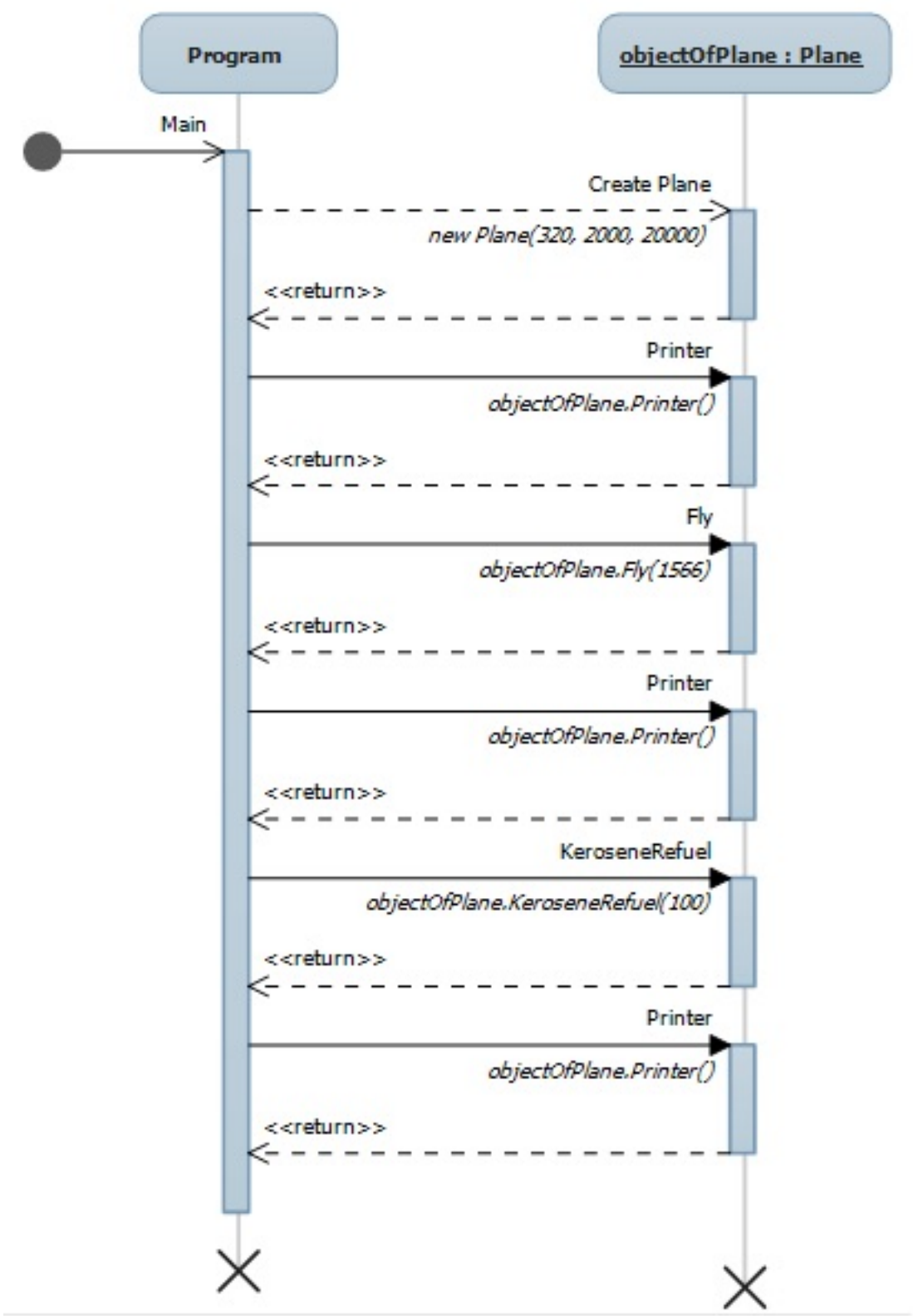
Dinamikus modell

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.

Az alábbi ábrán látható egy repülőgép adatait a repülés előtt, repülés után (Fly) majd üzemanyag feltöltés után (KeroseneRefuel) kiíró program szekvencia diagrammja.



5. ábra: Szekvencia diagram

Objektumorientált programkészítés

Az objektumorientált programkészítés folyamata az alábbi főbb lépésekre bontható:

- a követelmények feltárása után, a megfogalmazott probléma leírása alapján meg kell határozni azokat az objektumokat, adatokat, metódusokat, amelyek segítségével a feladat leképezhető;
- az egyes hasonló objektumok közös tulajdonságait osztály segítségével írjuk le;
- meg kell keresni az egyes osztályok közötti kapcsolatokat;
- implementálni kell az előzőkben meghatározott osztályokat és kapcsolatokat;
- programban megfelelő helyen létre kell hozni az objektumok megfelelő példányait és meg kell valósítani a kommunikációs kapcsolatokat.

Az előzőekben bemutatott mintapéldák alapján C++ nyelven implementált, egy repülőgép hatótávolságát és a maximálisan szállítható rakomány mennyiségét számító egyszerű objektumorientált program forráskódja. A Jarmu.h header fájl tartalmazza a Jarmu osztály, és a Repulo.h a *Repulo* osztály deklarációját. A Jarmu.cpp a *Jarmu* osztály, a Repulo.cpp a Repulo osztály egy megvalósítását tartalmazza. A main.cpp tartalmazza a *Repulo* osztály példányosítását és a Boeing 737-300 *Repulo* osztály és a *Jarmu* osztály függvényhívásait. A függvényhívások eredményeként a program kiírja a rakomány maximális tömegét és a gép hatótávolságát konzolban. A getch() függvény meghívásával a konzol csak gombnyomásra záródik be.

Az egyes forráskódok a könnyű átláthatóság, későbbi karbantarthatóság miatt kerültek külön fájlokba.

Jarmu.h

```
#ifndef JARMU_H
#define JARMU_H

class Jarmu
{
        private:
                float fogyasztas;
                float tankMeret;
        protected:
                int maxUtasLetszam;

        public:
                Jarmu(int maxUtasLetszam, float fogyasztas, float
                tankMeret);
                float MaximalisSzallitasiTavolsag();
};
#endif
```

Repulo.h

```
#ifndef REPULO_H
#define REPULO_H
#include "Jarmu.h"

class Repulo : public Jarmu
{
        private:
                float maxFelszalloTomeg;
                float repuloSzarazTomeg;
                float tuzeloanyagTomeg;

        public:
                Repulo(int maxUtasLetszam, float fogyasztas, float
                tankMeret, float maxFelszalloTomeg, float
                repuloSzarazTomeg, float tuzeloanyagTomeg);

        public:
                float SzallithatoRakomanyMaximalisTomege()
                {
                        return repuloSzarazTomeg + tuzeloanyagTomeg
                        + maxUtasLetszam * 100;
                }
};
#endif
```

Jarmu.cpp

```
#include "Jarmu.h"

Jarmu::Jarmu(int maxUtasLetszam, float fogyasztas, float tankMeret)
{
        this->maxUtasLetszam = maxUtasLetszam;
        this->fogyasztas = fogyasztas;
        this->tankMeret = tankMeret;
}

float Jarmu::MaximalisSzallitasiTavolsag()
{
        return tankMeret / fogyasztas * 100;
}
```

Repulo.cpp

```
#include "Repulo.h"

Repulo::Repulo(int maxUtasLetszam, float fogyasztas, float tankMeret,
float maxFelszalloTomeg, float repuloSzarazTomeg, float
tuzeloanyagTomeg) : Jarmu(maxUtasLetszam, fogyasztas, tankMeret)

{
        this->maxUtasLetszam = maxUtasLetszam;
        this->maxFelszalloTomeg = maxFelszalloTomeg;
        this->repuloSzarazTomeg = repuloSzarazTomeg;
        this->tuzeloanyagTomeg = tuzeloanyagTomeg;
}
```

main.cpp

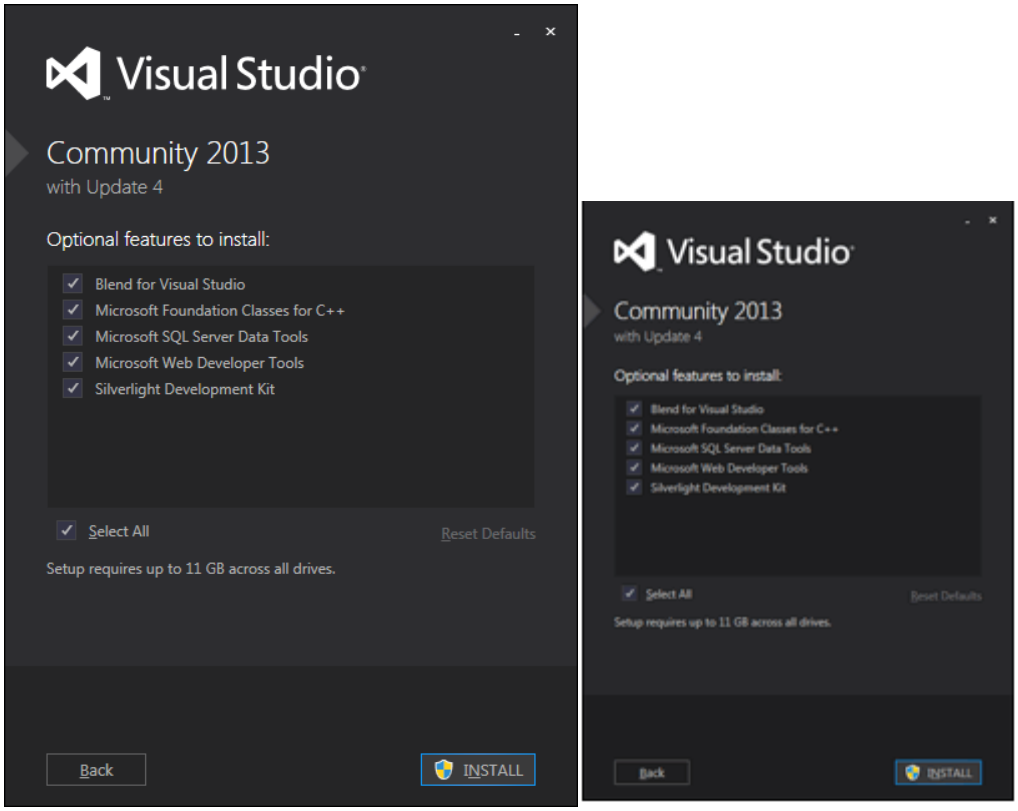
```cpp
#include "Repulo.h"
#include <conio.h>
#include <iostream>

using namespace std;

int main()
{
	Repulo* Boeing737 = new Repulo(141, 378.1, 23817.0, 56740.0,
	32881.0, 12618.0);

	float maximalisRakomanyTomege = Boeing737
	->SzallithatoRakomanyMaximalisTomege();

	float hatoTavolsag = Boeing737
	->MaximalisSzallitasiTavolsag();

	cout << "A Boeing 737-es repulogep rakomany maximalis tomege
	" << maximalisRakomanyTomege << " kg" << " es hatotavolsaga "
	<< hatoTavolsag << " km";

	_getch(); //várakozás billentyűnyomásra

	return 0;
}
```

Visual Studio Community fejlesztőkörnyezet

Az előző fejezetben ismertetett program implementálásához, a kód szerkesztéséhez, adott operációs rendszeren futtatható program előállításához, teszteléséhez, hibakereséséhez, összességében fejlesztéséhez integrált fejlesztőkörnyezeteket alkalmaznak.

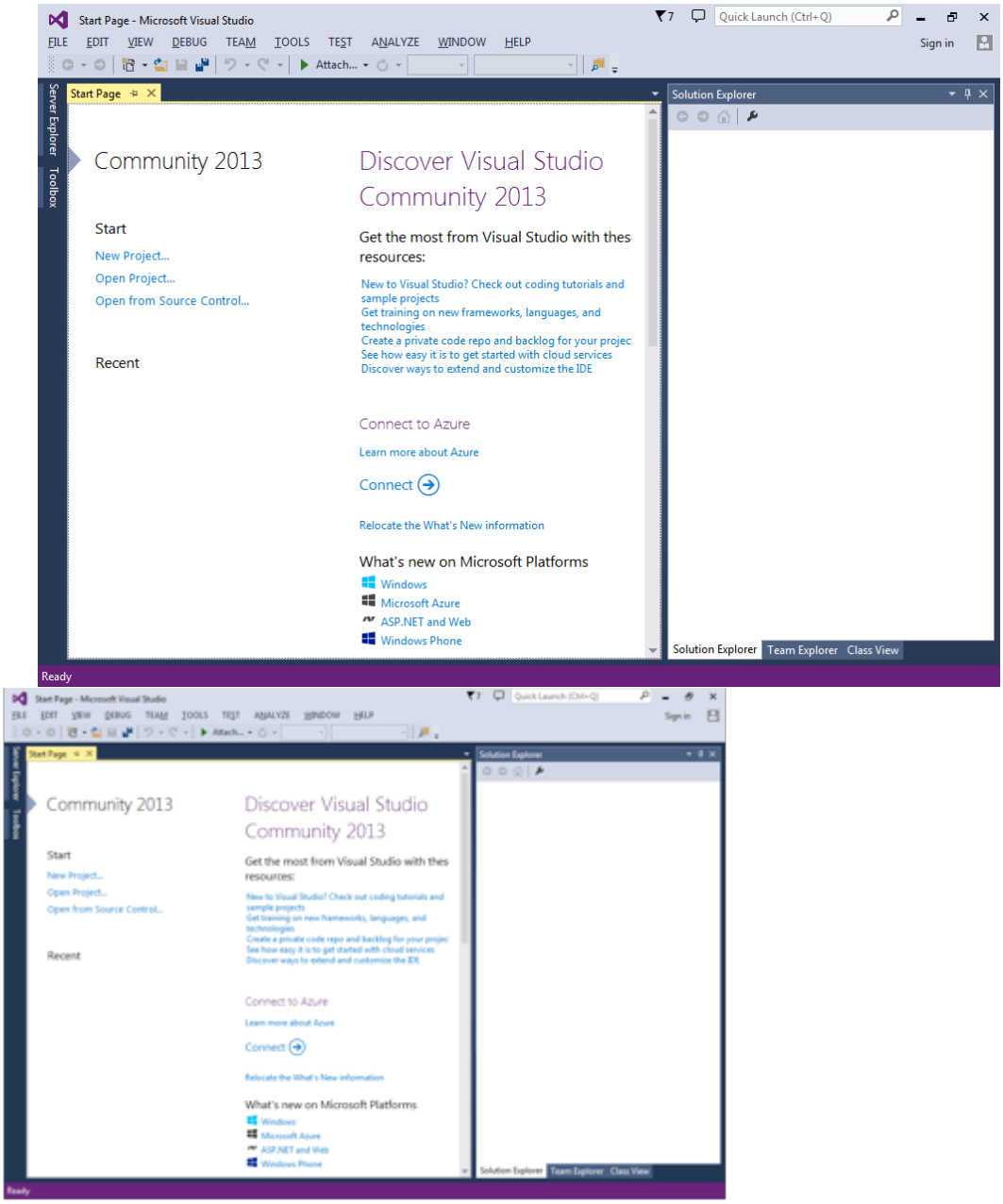
A Visual Studio Community a Microsoft ingyenes, több programozási nyelvet támogató fejlesztőkörnyezete, mely természetesen a felhasználói felület (GUI – Graphical User Interface) elkészítésére is támogatást nyújt. A Community 2013 kiadás jellemzője, hogy meghatározott feltételek mellett ingyenes, a Visual Studio Professional 2013 minden lényeges funkcionalitását tartalmazó termékről van szó, amely a Visual Studio kiterjesztések kezelésére is képes. Az eddig kínált ingyenes Express verzió a jövőben megszűnik, a Community annak helyébe lép.

A Visual Studio Community fejlesztőkörnyezet a www.visualstudio.com oldalról tölthető le web és DVD9 ISO telepítő változatban (http://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx). A telepítéshez több mint 10 GB helyre van szükség, és Windows 7 SP1 vagy újabb operációs rendszerre. A telepítés néhány lépésből áll, Windows 7 SP1 alatti telepítés esetén az alábbi képen látható opcionális funkciók telepítése is kiválasztható, melyek közül a későbbi fejlesztési igények végett célszerű mindegyiket kiválasztva hagyni. A telepítés után az első indításkor a program bejelentkezést kér egy Microsoft fiók adatainak megadásával. Az ingyenes használat ingyenes Microsoft fiók regisztrációhoz kötött, de fejlesztőkörnyezet próbaváltozata 30 napig regisztráció nélkül is használható. A regisztrációs adatok megadása, vagy annak kihagyása „Not now, maybe later.” után a fejlesztői környezet felületének színsémáit lehet kiválasztani 3 lehetőség közül: kék (Blue), sötét (Dark) és világos (Light).



6. ábra: Visual Studio Community opcionális funkciók

Kék színsémát választva a fejlesztőkörnyezet kezdőablaka az első indítás után:

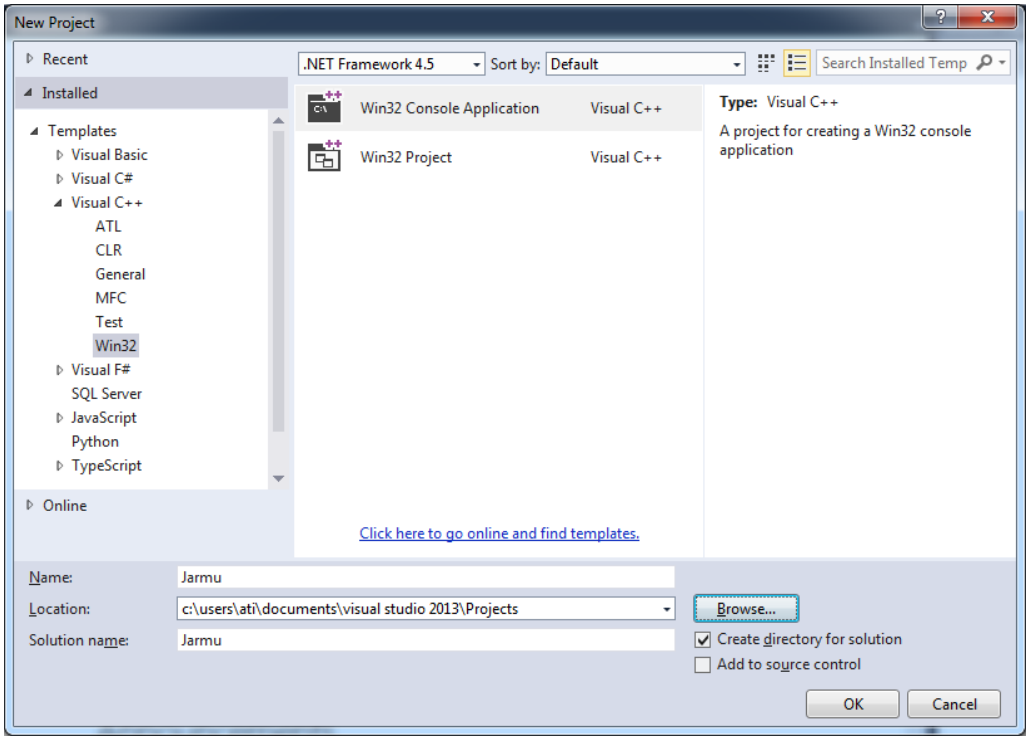


7. ábra: Visual Studio Community fejlesztőkörnyezet kezdőképernyője

Konzolalkalmazás fejlesztés

Amennyiben úgynevezett konzolalkalmazást készítünk, a szabványos C++-nak megfelelő szintaktikát alkalmaz a Visual Studio. A C++ alkalmazások forrásfájljainak kiterjesztése .cpp, a header fájlt .h, mint C nyelv esetén. Az alkalmazás belépési pontja a main függvény, ahol a program futása kezdődik. Az alábbiakban az előzőkben ismertetett Jármű példa kerül implementálásra, összeállítása és Windows operációs rendszeren futtatható kóddá történő lefordításra Visual Studio Community fejlesztőkörnyezet felhasználásával. A lefordított 32 bites alkalmazás konzolban kerül futtatásra.

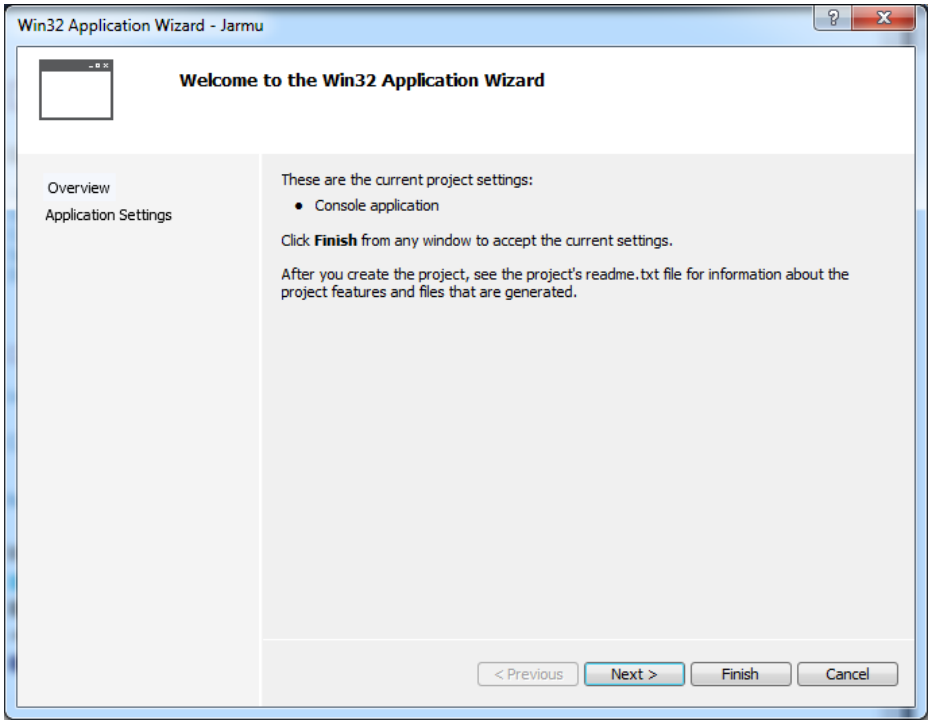
A legtöbb fejlesztőrendszerben a program elkészítéséhez felhasznált forráskódokat, erőforrásfájlok stb. egy ún. Projekt összeállítás fogja össze. Új projekt a File menü New menüpontjában hozható létre, a Project almenüt kiválasztva, vagy akár a Ctrl+Shift+N gyorsindító gombkombinációval. Új projekt kiválasztása után megjelenő ablakban a közül a Visual C++ Win32 sablont (Template) kell kiválasztani és a Win32 Console Application-t, mely az alábbi ábrán látható.



8. ábra: Win32 Console Application projekt létrehozása

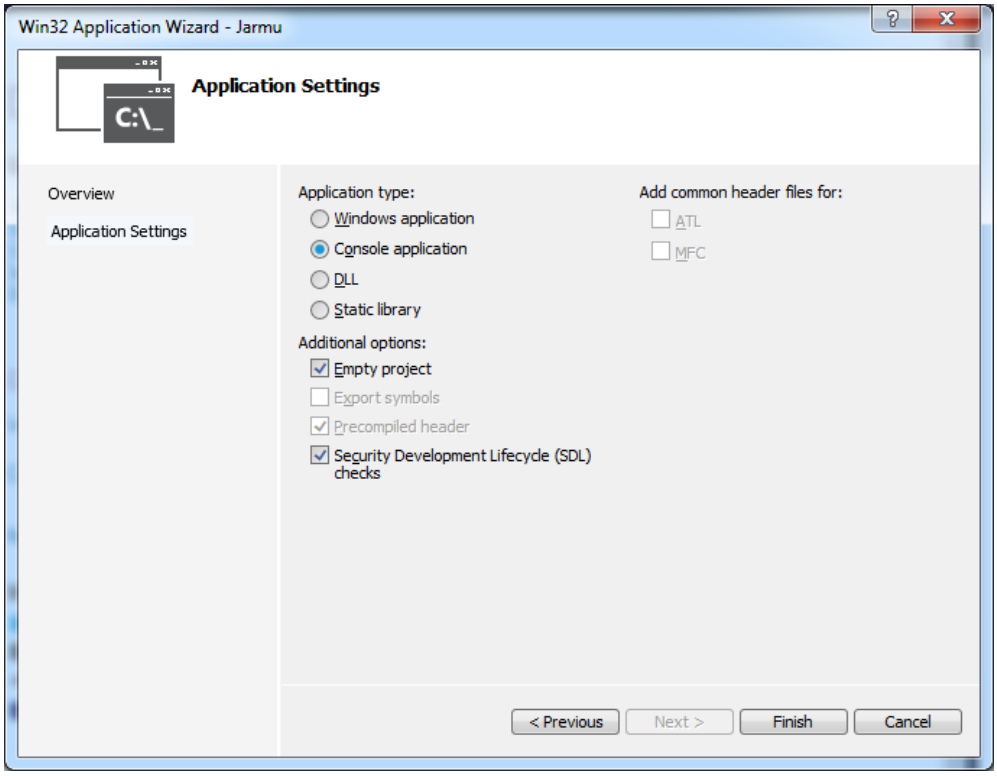
Az ablak alsó részén a projekt nevének (Name) a példának megfelelően a Jarmu név került megadásra. Ebben a részben lehet módosítani a Projekthez kapcsolódó fájlok mentési helyét is (Location). Egy összetett program általában több projektből tevődik össze, ezeket egységes rendszerré a megoldás (Solution) fogja össze.

Az OK gombra kattintást követően elindul az alkalmazás varázsló, mely a következő ábrán látható.



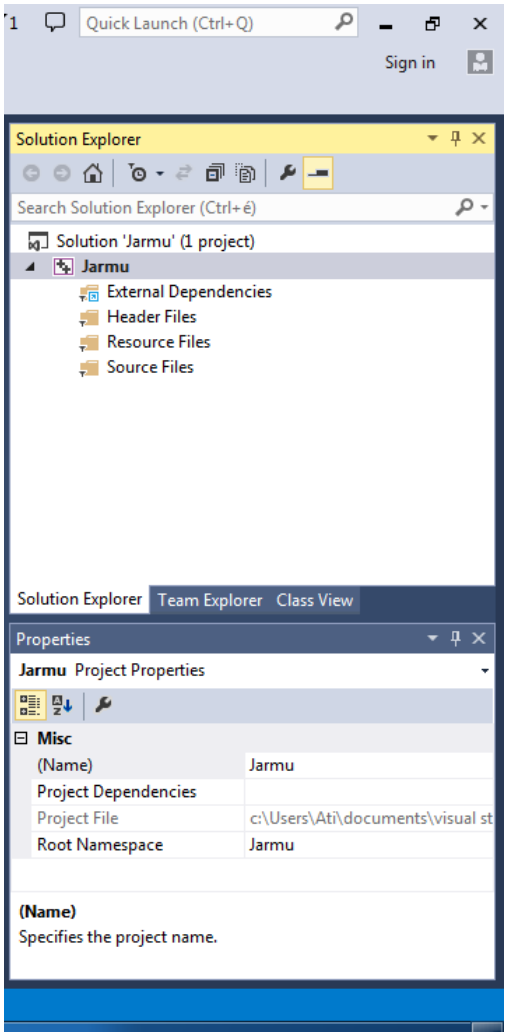
9. ábra: Win32 konzol alkalmazás varázsló

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.



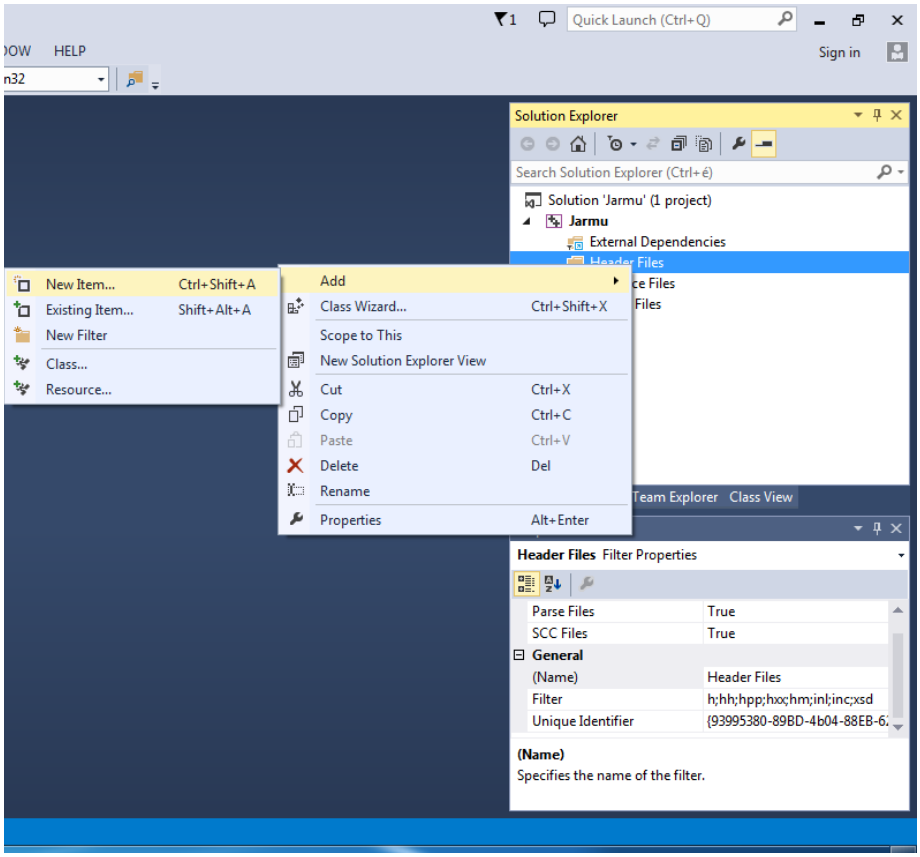
10. ábra: Alkalmazás beállításai

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



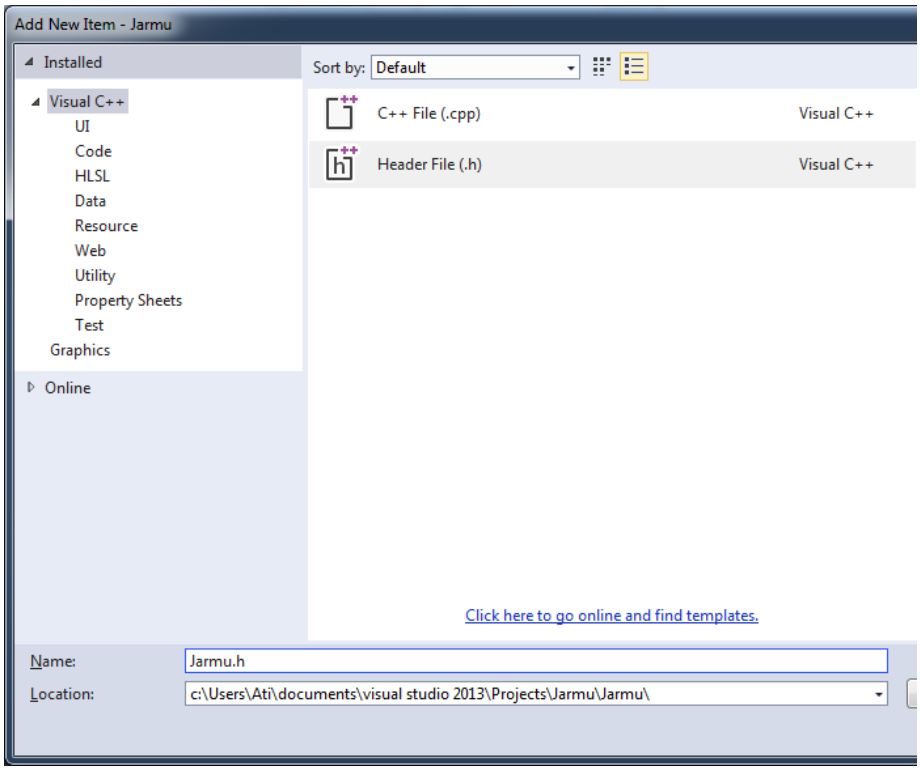
11. ábra: Megoldás és beállítások ablak

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.



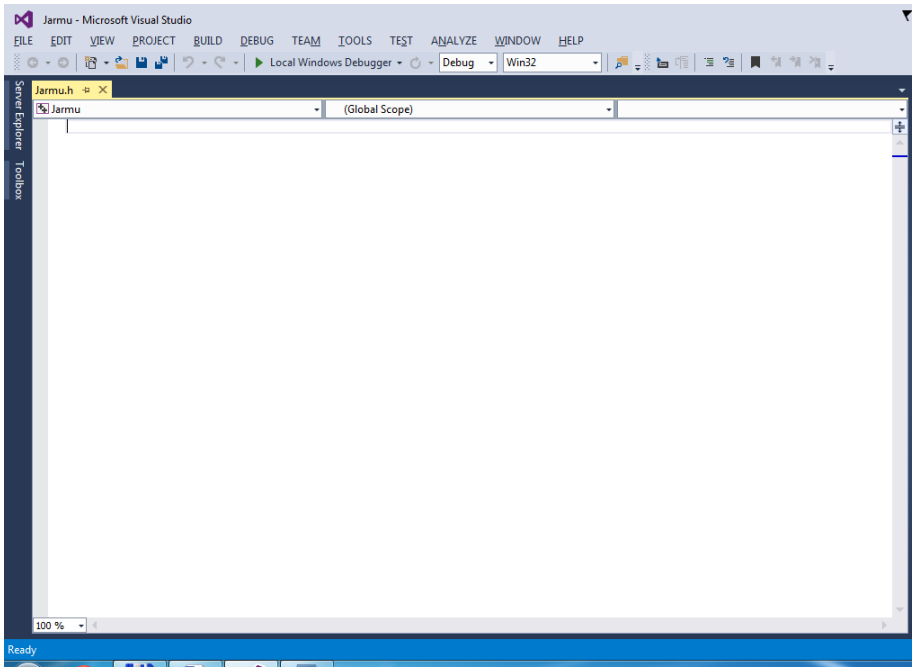
12. ábra: Új header fájl hozzáadása a projekthez

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



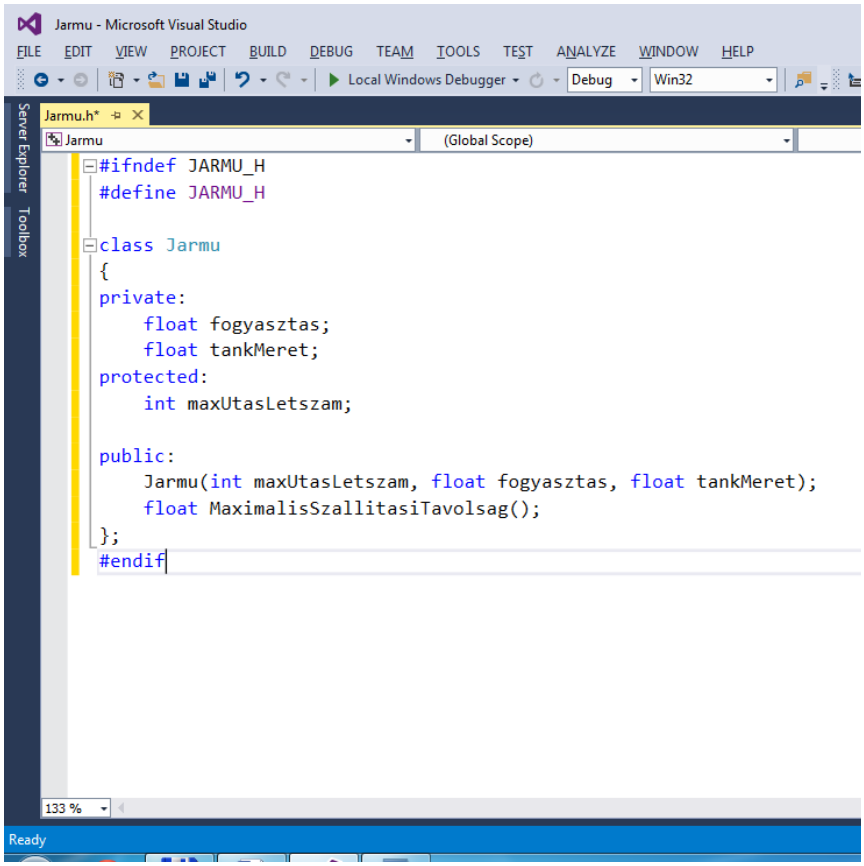
13. ábra: Új header fájl hozzáadása a projekthez, név megadása

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

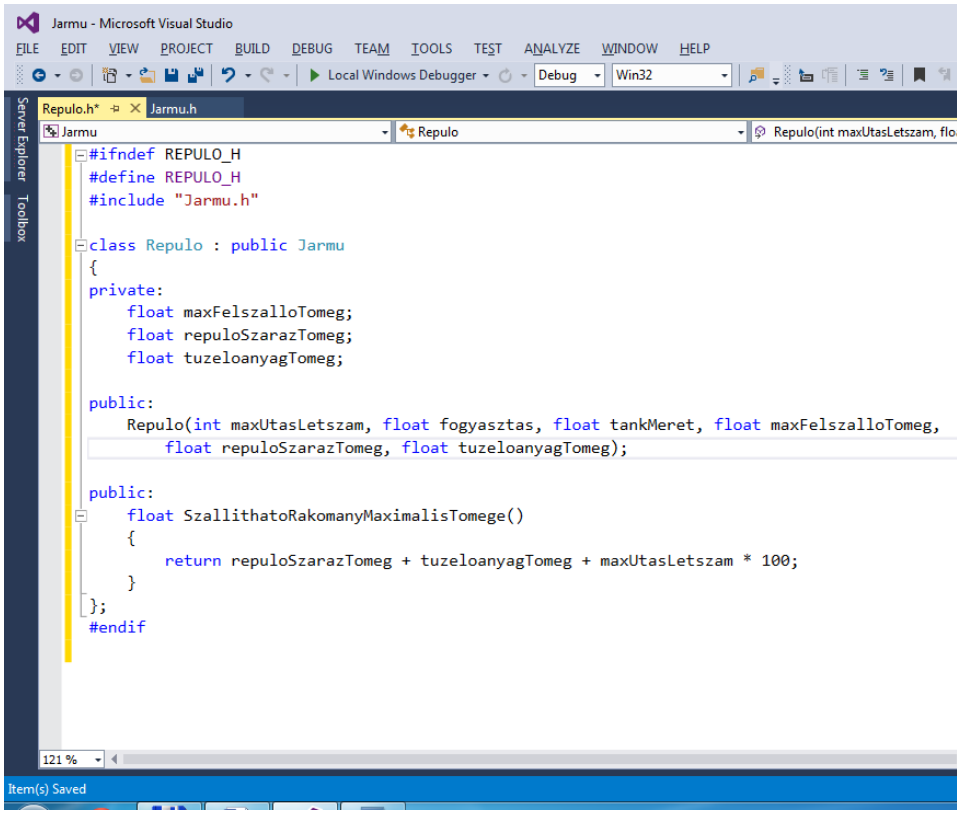


14. ábra: Forráskód szerkesztő

A *Jarmu* és *Repulo* osztály korábban már bemutatott deklarációjának forráskódja a szerkesztőben megadható az alábbi ábrák szerint.

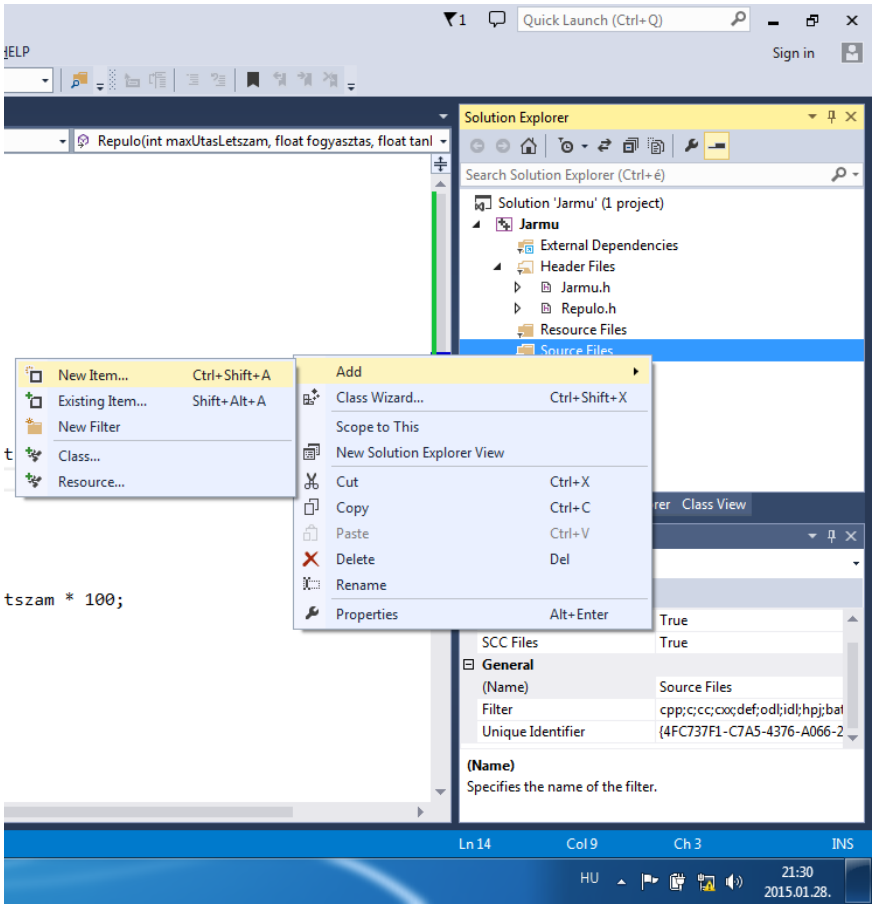


15. ábra: Jarmu osztály deklarációja a forráskód szerkesztőben



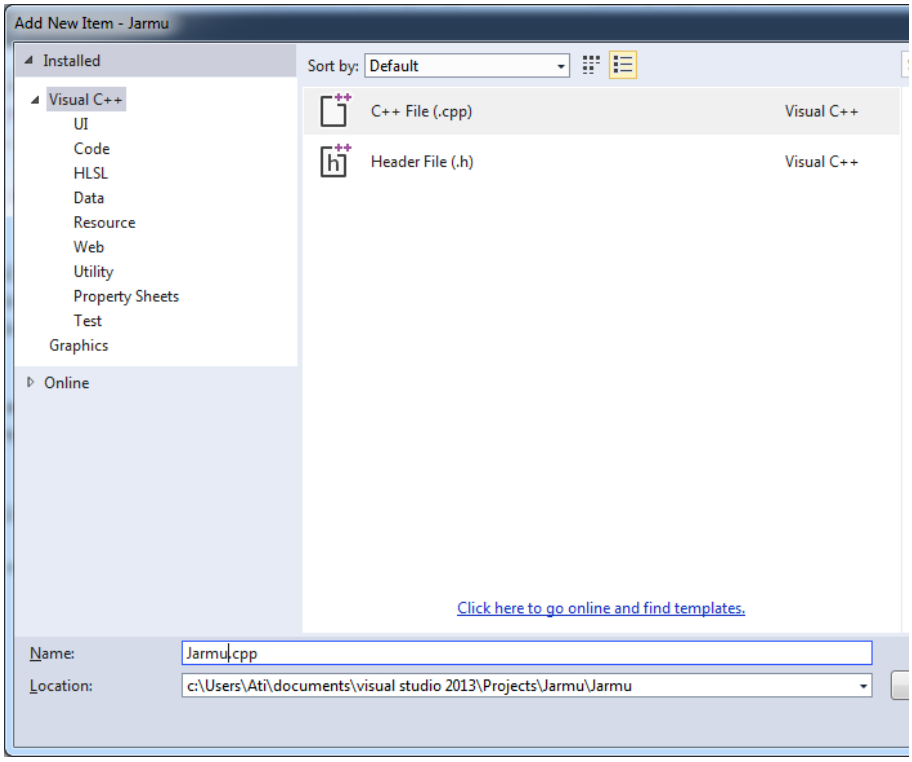
16. ábra: Repulo osztály deklarációja a forráskód szerkesztőben

A deklarált osztályok megvalósításait a Jarmu.cpp és a Repulo.cpp forrásfájlok tartalmazzák, melyeket a header fájlokhoz hasonlóan, de a forrásfájl (Source File) hozzáadásával hozhatunk létre és adhatjuk meg azután a kódot a szerkesztőben a következő ábráknak megfelelően.



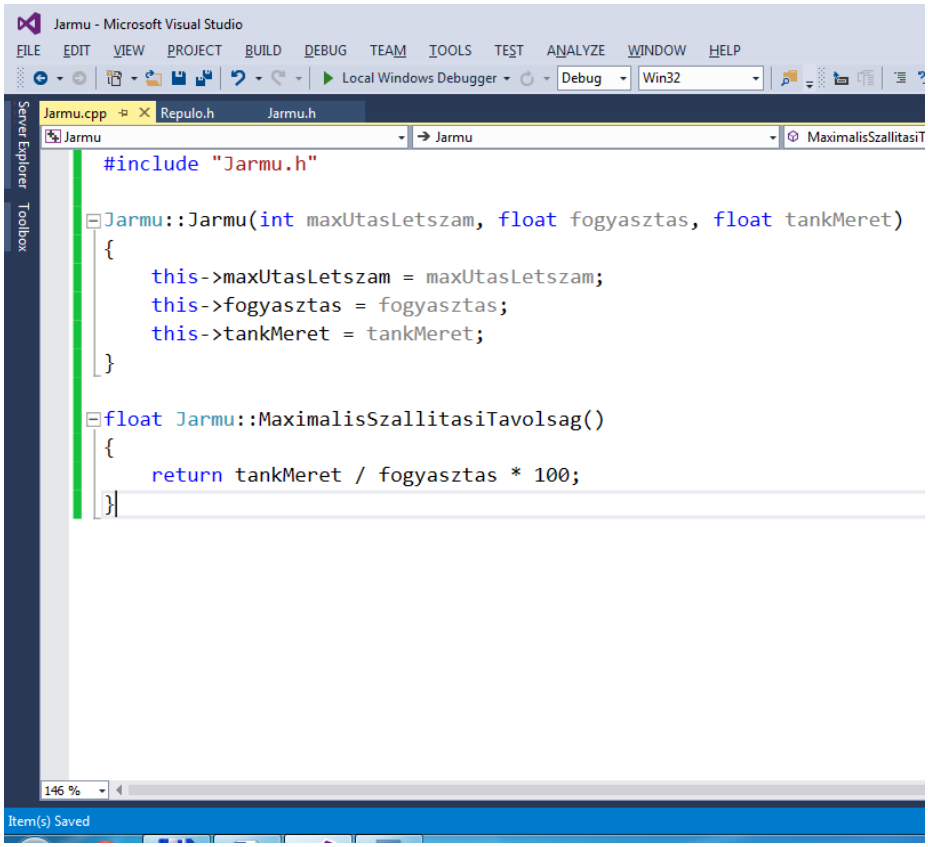
17. ábra: Új forrásfájl hozzáadása a projekthez

Új C++ fájl (C++ File) választásával és a név (Name) a Jarmu.cpp megadásával hozható létre a Jarmu.cpp fájl, az alábbi ábrának megfelelően.

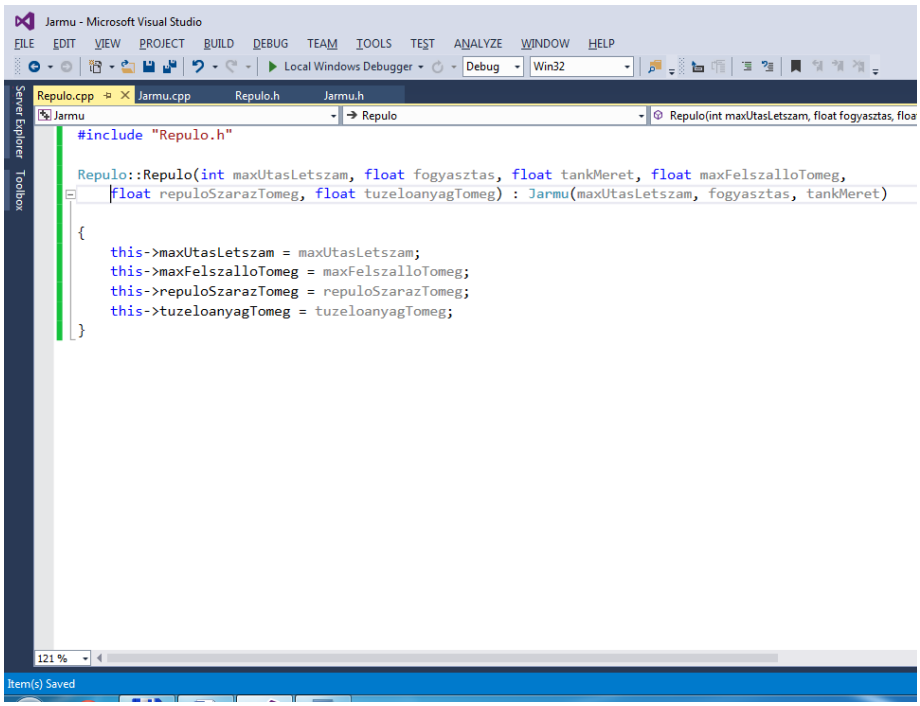


18. ábra: Új forrásfájl hozzáadása a projekthez, név megadása

A *Jarmu* és *Repulo* osztályok megvalósításának a korábban már bemutatott forráskódja a szerkesztőben megadható az alábbi ábrák szerint.

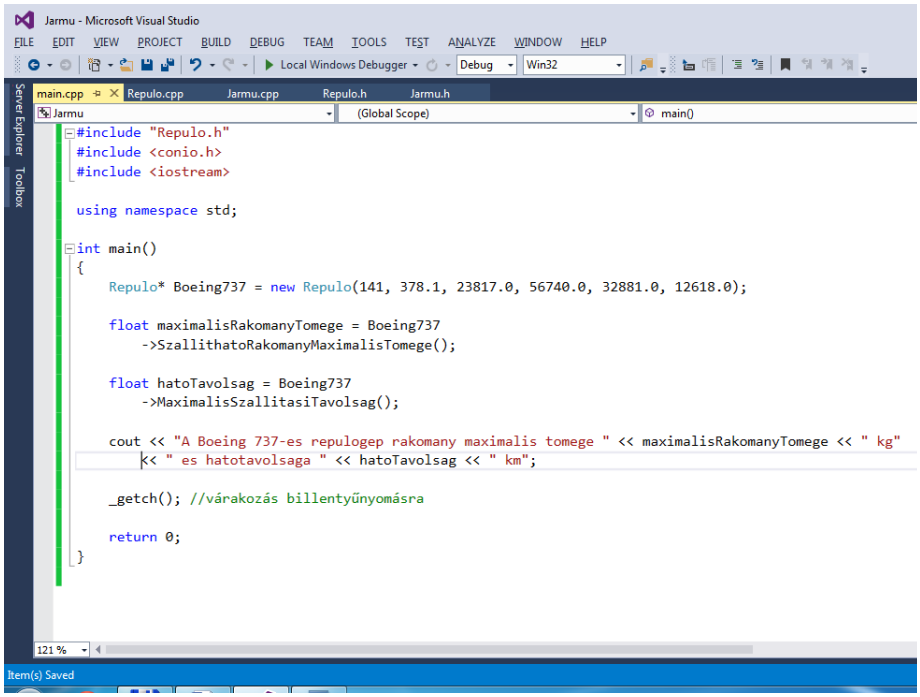


19. ábra: Jarmu osztály megvalósítása a forráskód szerkesztőben



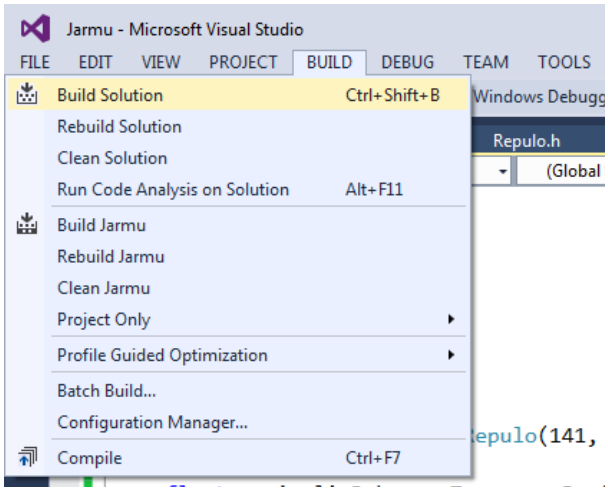
20. ábra: Repulo osztály megvalósítása a forráskód szerkesztőben

A main.cpp mely a *Repulo* osztály példányosítását és függvényhívásait tartalmazza, az alábbi ábrán látható.



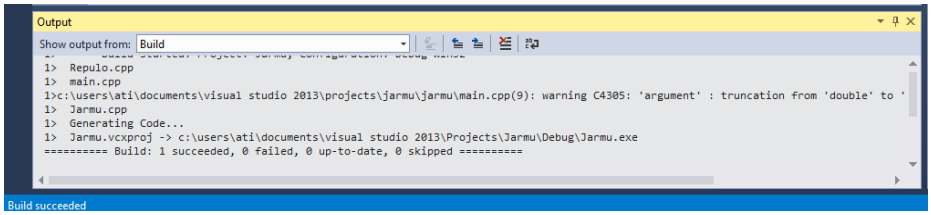
21. ábra: main függvény a forráskód szerkesztőben

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.



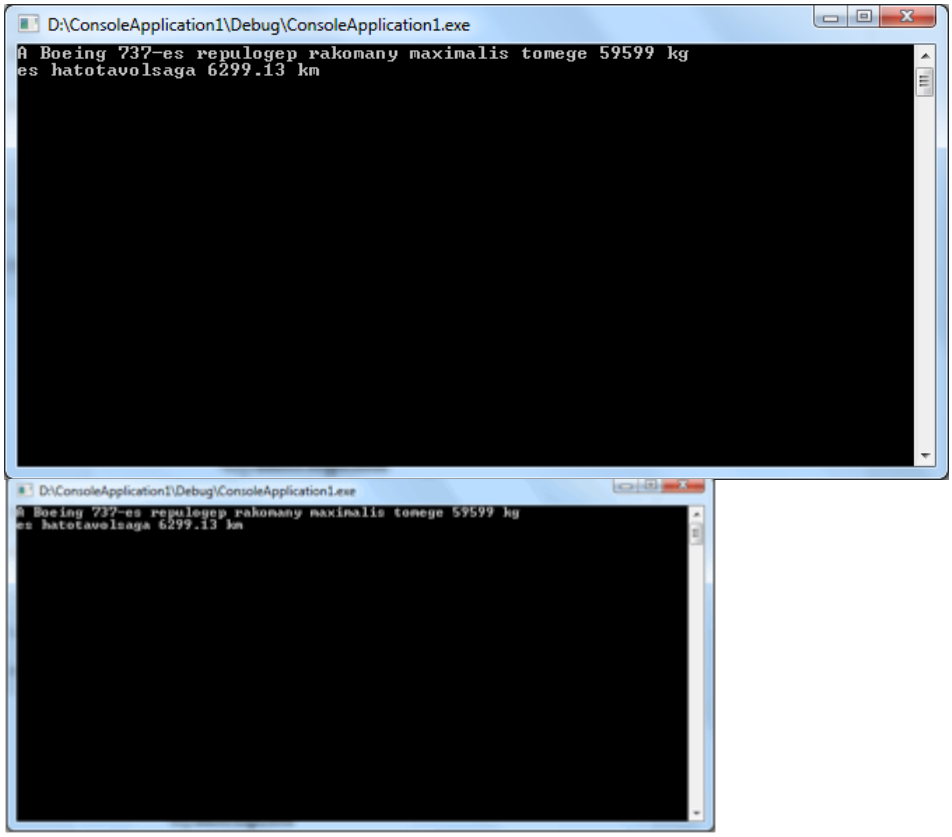
22. ábra: Program fordítása

Sikeres fordítás a kimenet (Output) ablakban látható.



23. ábra: Kimenet ablak

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


D:\ConsoleApplication1\Debug\ConsoleApplication1.exe
A Boeing 737-es repulogep rakomany maximalis tomege 59599 kg
es hatotavolsaga 6299.13 km

Gyakorlópélda

Írjunk egy tengeralattjárót modellező osztályt (*Submarine*), mely képes más tengeralattjáróval harcolni!

Az osztály tagváltozói a következőek:

- Életerő (hitPoints), mely egy egész szám 0 és 20 között;
- Támadóerő (attackRating), mely egy egész szám 1 és 200 között;
- Védekezőerő (defenseRating), mely egy egész szám 0 és 200 között;
- Maximális merülési mélység (maxSubmersionDepth), mely egy lebegőpontos szám 0.0 és 5000.0 között.

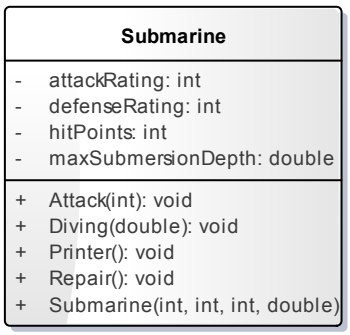
Ügyeljünk az egységbezárás elvére, tehát a tagváltozók private láthatóságú adattagok legyenek!

Valósítsuk meg az osztály metódusait:

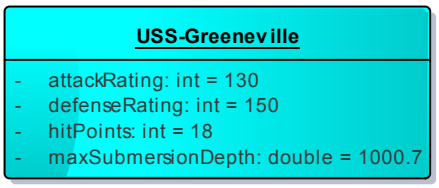
- Írjuk meg az osztály konstruktorát, melyben kötelező megadni a tengeralattjáró támadóerejét, védekezőerejét és életerejét! A maximális merülési mélységet is lehessen megadni, de ne legyen kötelező (használjon default paramétereket: ha a programozó nem ad meg életerőt, akkor a tagváltozó kezdeti értéke 1000.0 lesz);
- Írjunk egy Repair() függvényt, mely 2 ponttal növeli a tengeralattjáró életerejét! Ügyeljünk rá, hogy 20 fölé soha ne kerüljön az érték.
- Írjunk egy Attack(otherAttackRating) függvényt, mely támadás alá helyezi a tengeralattjárónkat! A függvény tartalma csakis akkor fusson le, ha a tengeralattjárónk életereje nagyobb mint 0! A támadás hatására a tengeralattjárónk életereje csökkenjen ((defenseRating - otherAttackRating)/10) értékkel! Ügyeljünk rá, hogy 1 alá nem csökkenhet a tengeralattjárónk életereje. Ha a támadás sorozat alatt eléri az életerő 1 egység alá csökkene, akkor a tengeralattjárót menekítsük ki úgy, hogy lemerítjük egy tetszőleges mélységre a Diving(depth) függvény segítségével.
- Diving(depth): A tengeralattjáró semmisüljön meg, ha a paraméterben átadott érték nagyobb, mint a tengeralattjáró maximális merülési mélysége, egyébként írja ki, hogy sikeres a merülés;
- Írjon egy Printer() függvényt, ahol minden adatot megjelenít a tengeralattjáróról.

Írjunk egy egyszerű *main()* függvényt, amely létrehoz egy tengeralattjárót, amelyet támadás alá helyezünk. Minden esemény után írjuk ki a tengeralattjáró állapotát!

Osztály diagram



Objektum diagram



Megoldás

Submarine.h

```
#ifndef SUBMARINE_H
#define SUBMARINE_H

class Submarine
{
        private:
                int hitPoints;
                int attackRating;
                int defenseRating;
                double maxSubmersionDepth;

        public:
                Submarine(int hitPoints, int attackRating, int
                defenseRating, double maxSubmersionDepth);
                void Repair();
                void Attack(int attackRating);
                void Diving(double depth);
                void Printer();
};
#endif
```

Submarine.cpp

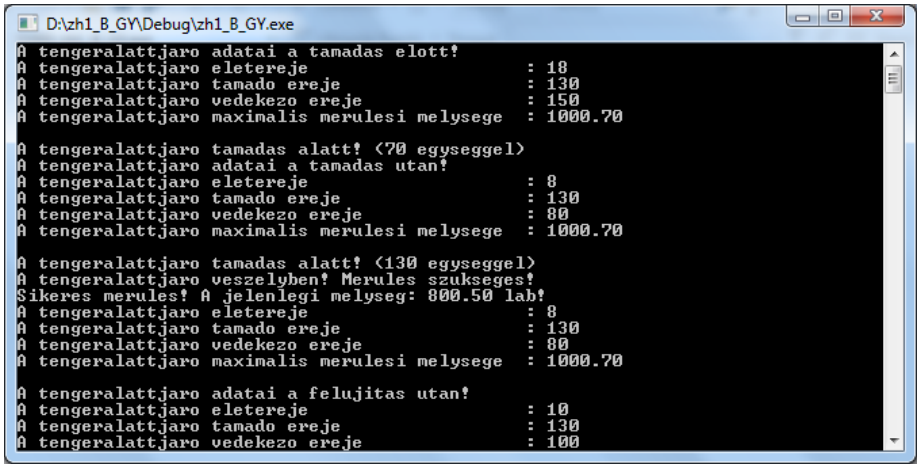
```
#include <iostream>
#include "Submarine.h"

using namespace std;

Submarine::Submarine(int hitPoints, int attackRating, int
defenseRating, double maxSubmersionDepth)
{
        this->hitPoints = hitPoints;
        this->attackRating = attackRating;
        this->defenseRating = defenseRating;
        this->maxSubmersionDepth = maxSubmersionDepth;
}

void Submarine::Repair()
{
        if (this->hitPoints+2<20)
        {
                this->hitPoints+=2;
                this->defenseRating+=20;
        }
        else
        {
                this->hitPoints=20;
                this->defenseRating=200;
        }
}

void Submarine::Attack(int attackRating)
{
        if(this->hitPoints>0)
        {
                if (((this->defenseRating - attackRating)/10) < 1)
                {
                        cout << "A tengeralattjaro veszelyben!
                        Merules szukseges!\n";
                        this->Diving(800.5);
                }
                else
```

```
{
                        this->hitPoints = (this->defenseRating -
                        attackRating)/10;
                        this->defenseRating = this->defenseRating -
                        attackRating;
                }
        }
}

void Submarine::Diving(double depth)
{
        if(depth > this->maxSubmersionDepth)
        {
                cout << "A merules sikertelen! A tengeralattjaro
                megsemmisult!\n";
                cout << endl;
        }
        else
        {
                printf("Sikeres merules! A jelenlegi melyseg: %.2lf",
                depth); cout << " lab!";
                cout << endl;
        }
}

void Submarine::Printer()
{
        cout << "A tengeralattjaro eletereje                    : "
        << this->hitPoints << endl;
        cout << "A tengeralattjaro tamado ereje                 : "
        << this->attackRating << endl;
        cout << "A tengeralattjaro vedekezo ereje               : "
        << this->defenseRating << endl;
        printf("A tengeralattjaro maximalis merulesi melysege  :
        %.2lf ", this->maxSubmersionDepth);
}
```

main.cpp

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

using namespace std;

int main()
{
        Submarine *s = new Submarine(18, 130, 150, 1000.7);
        cout << "A tengeralattjaro adatai a tamadas elott!" << endl;
        s->Printer();
        cout << endl;
        cout << endl;
        cout << "A tengeralattjaro tamadas alatt! (70 egyseggel)" <<
        endl;
        s->Attack(70);
        cout << "A tengeralattjaro adatai a tamadas utan!" << endl;
        s->Printer();
        cout << endl;
        cout << endl;
        cout << "A tengeralattjaro tamadas alatt! (130 egyseggel)" <<
        endl;
        s->Attack(130);
        s->Printer();
        cout << endl;
        cout << endl;
        cout << "A tengeralattjaro adatai a felujitas utan!" << endl;
        s->Repair();
        s->Printer();

        _getch();
        return 0;
}
```

Program futásának eredménye:



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.