

**A C++ szabványos sablonkönyvtár
(STL) szekvenciális tárolói**

Gyakorlatorientált szoftverfejlesztés C++ nyelven

Visual Studio Community fejlesztőkörnyezetben

@Katona József – Kővári Attila
Lektorálta: Kratzmayer Zoltán

Publio Kiadó
Felelős kiadó: Publio Kiadó Kft.
2016
Minden jog fenntartva!
DOI: ???

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 negyedik kötete, mely sorozat gyakorlati példákon keresztül segíti a nyelv elsajátítását. Bízunk benne, hogy azok, akik végig olvassák a könyvet, és kidolgozzák a példákat, alkalmazás szintjén megismerik a C++ nyelvi elemei által adott lehetőségeket, a nyelv logikáját, az átláthatóbb szoftverkészítést elősegítő funkcióját és mélyebben elsajátítják az objektumorientált szemlélet alapjait, mely a hatékony szoftverfejlesztéshez elengedhetetlen. A bemutatott ismeretek gyakorlati példákkal illusztráltak, melyek a megértést és az alkalmazás elsajátítását is nagymértékben segítik.

Jelen kötet a C++ szabványos sablonkönyvtár szekvenciális tárolói ismerteti. A könyv olvasása során megismerjük a szekvenciális tárolók különböző tulajdonságait és eltérő működési módjait, amelyek biztosíthatják számunkra, hogy egy programozási feladatra hatékony, gyors és elegáns megoldást tudjunk nyújtani. Továbbá a mű néhány 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.

Figyelem! A könyv sorozat elsősorban az objektum-orientált szemléletmód hatékony elsajátítását próbálja segíteni gyakorlati példákon keresztül, még azokban az esetekben is, ahol nem kifejezetten a legelegánsabb megoldást képes nyújtani, így a rutinosabb, az STL tárolókat kiválóan alkalmazó, valamint az objektum-orientált világban biztonságosan mozgó olvasónak bizonyos megoldások kényszerítettnek tűnhetnek.

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

A Szerzők

STL (Standard Template Library) – A C++ nyelv szabványos sablonkönyvtára

A könyvsorozat előző, harmadik kötetének témája a generikus programozás, amelynek elsajátítása nélkülözhetetlen ahhoz, hogy megértsük az STL (Standard Template Library), magyarul Szabványos Sablonkönyvtár elemeinek paraméterezhető osztályait és függvényeit. A sablonkönyvtár számos gyűjteményt, illetve komponens biztosít számunkra, annak érdekében, hogy minél hatékonyabb, gyorsabb és elegánsabb megoldást tudjunk nyújtani egy programozási feladatra.

Konténerek (tárolók)

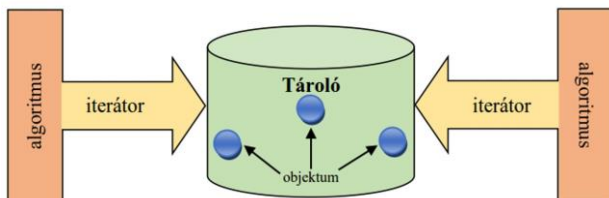
Olyan adatstruktúrák – például objektumok – amelyek más, azonos típusú objektumokat (elemeket) tárolnak, az adatok memóriában történő tárolását teszik lehetővé és tárolási lehetőségeik mellett karbantartó műveleteket is biztosítanak. (array – statikus tömb, vector – dinamikus tömb, deque – kétvégű sor, list – kétszeresen láncolt lista, set – halmaz, map – asszociatív tömb stb.) *Terjedelmi korlátok miatt a jelenlegi kötet kizárólag a szekvenciális tárolókat ismerteti részletekbe menően, az asszociatív tárolók mélyebb bemutatására a könyvsorozat 6. kötetében kerül sor.*

Algoritmusok

Olyan művelethalmaz, amely úgy van implementálva, hogy a konténerekben tárolt adatokkal maximálisan együtt tudjanak dolgozni, így ezek az algoritmusok minden egyes tárolóra vagy legalább annak részhalmazára alkalmazhatók. (sort, copy, search, min, max stb.) *Az algoritmusok alaposabb bemutatása a könyvsorozat 8. kötetében kerül ismertetésre.*

Iterátorok (bejárók)

Olyan általánosított mutatók, amelyek lehetővé teszik azon adatok eltérő módon történő elérését, amelyek a konténerben helyezkednek el, így kínálja lehetőséget arra, hogy minden konténeren egyformán dolgozhassunk. (iterator – közvetlen elérésű bejáró, const_iterator – egy tetszőleges elérésű bejáró, de rajta keresztül nem módosítható a mutatott elem, reverse_iterator – Egy tetszőleges elérésű bejáró, csak ezzel visszafelé lehet menni stb.) *Az iterátorok részletezése a könyvsorozat 7. kötetében található.*



A sablonkönyvtár komponensek

Konténer adaptációk

A konténer adaptációk a tárolókra épülő magasabb szintű adatszerkezet tárolók, amelyeket használhatjuk veremként, sorként vagy fontossági sorként. (*stack* - verem, *queue* - sor, *priority_queue* - fontossági sor) A **stack** adapter egy vermet implementál, így amit utoljára betettünk azt vesszük ki először (FIFO – first-in, first-out). A **queue** adapter egy sort implementál, így azt az elemet vehetjük ki először, amelyet először elhelyeztünk benne (LIFO – last-in, first-out). A **priority_queue** adapter egy fontossági sorrend adaptert implementál, ahol minden esetben a legmagasabb értékű elem van az elején. A *konténer adaptációk részletezése a könyvsorozat 5. kötetében lelhető fel.*

Függvényobjektumok (funktorok, predikátumok)

A műveleteket funktor osztályok fedik le, más komponensek számára, felülbírálják a () operátort, így egyfajta állapottal rendelkező függvényként használhatóak. Minden egyes operátornak megvannak a megfelelő funktor objektumai. (*plus* - +, *greater_equal* - >=, *logical_and* - && stb.) A gyakorlatban az előre definiált funktorokat gyakran használjuk együtt a C++-ban már implementált algoritmusokkal. Az alábbi példában a **mismatch()** előredefiniált függvényt felhasználva két tömbben lévő értékek egyezőségét vizsgáljuk, ahol az **equal_to** funktort használva döntjük el az azonos indexen lévő két tömb elem értékéről, hogy egyeznek-e. A *függvényobjektumokról részletesebben a könyvsorozat 9. számában olvashat.*

```

                                funktor alkalmazása
                                |
std::mismatch(array0, array0+2, array1, std::equal_to<int>());

```

Funktor	Operátor
plus <i>(összeadás)</i>	+
minus <i>(kivonás)</i>	-
multiplies <i>(szorzás)</i>	*
divides <i>(osztás)</i>	/
modulus <i>(maradékos osztás)</i>	%
negate <i>(negálás)</i>	-
equal_to <i>(egyenlőség vizsgálata)</i>	==
not_equal_to <i>(nem egyenlőség vizsgálata)</i>	!=
greater <i>(nagyobb, mint)</i>	>
less <i>(kisebb, mint)</i>	<
greater_equal <i>(nagyobb vagy egyenlő, mint)</i>	>=
less_equal <i>(kisebb vagy egyenlő, mint)</i>	<=
logical_and <i>(logikai és)</i>	&&
logical_or <i>(logikai vagy)</i>	
logical_not <i>(logikai negáció)</i>	!

Néhány a `<functional>` fejlálmányban előre definiált predikátumok

A sablonkönyvtárban olyan adatszerkezeteket valósítottak meg, mint például a dinamikus vektor, a kétszeresen láncolt lista vagy az asszociatív tömb. Mivel a könyvtárban fellelhető komponensek sablonokkal¹ (**template**) dolgoznak, így bármilyen típussal képesek együtt működni.

A sablonkönyvtár szolgáltatásainak definiálásait az **std** névtérben, valamint azok deklarációit, illetve sablon (**template**) használata esetén, implementációt is, különböző fejlálmányokban helyezték el. A sablonkönyvtár szolgáltatásainak beépítéséről szabványos fejlálmány megfogalmazásával gondoskodhatunk:

```
#include <vector>
#include <list>
#include <map>
```

A fentebbi szabványos fejlálmány biztosítja számunkra a **vector**, a **list** és a **map** tárolók minden deklarációját és implementációját. Ahhoz, hogy használni is tudjuk például a vektor szolgáltatásait, szükséges az **std** névtér „behívása”. Erről három féle módon gondoskodhatunk:

¹A sablonokról, mint például osztály- vagy függvénysablonok, részletesen a C++ sorozat 3. kötetében olvashat, továbbá elcsajátíthatja azok használatát.

- Globális névfeloldással, ebben az esetben a névtér összes eleme elérhető lesz. Előnyei közé sorolható, hogy nem kell közvetlen módon feloldani a kívánt elemet, így olvashatóbb kód állítható elő. Az ilyen módon történő felhasználás hátrányai közé sorolható a névütközés és lassulás, mivel ebben az esetben egész névterek importálása történik.

```
using namespace std;
```

- Explicit névfeloldással, amikor a közvetlenül feloldani kívánt névteret a szükséges helyen megadjuk. Előnye a gyorsabb kódfutás, valamint könnyebb a névütközések elkerülése, hátrányai közé sorolható a redundáns kódrészletek előállása, amely a kód olvashatóságát nehezítheti.

```
std::cout << "Explicit névfeloldási módszer" << std::endl;
```

- Használhatjuk még a kevésbé megszokott úgynevezett **using** deklarációt, ahol csak a kívánt elem nevét oldjuk fel.

```
using std::cout;  
cout << "A cout elem nevének feloldása" << std::endl;
```

Konténerek, tárolók (containers)

A gyakorlatban elmondható, hogy csak ritkán írunk saját tároló osztályokat, mivel azokat már az STL-ben megírták és mindenki számára elérhetővé tették. Minden egyes tároló rendelkezik saját alapértelmezett konstruktorral, másoló konstruktorral és destruktorral, amelyek a tároló osztályok objektumpéldányainak létrehozásában, másolatában és megszüntetésében játszanak nagy szerepet, továbbá minden tároló képes lehet egy másik tároló részartományának eltárolására. A különböző tárolók elemeinek mindig van egy meghatározott sorrendje. A leggyakrabban használt tároló elemeket az alábbi táblázat foglalja össze.

Szabványos tárolók

Fejállomány	Konténer típus	Általános, rövid leírás
<vector>	Szekvenciális	A vektor lényegében egy dinamikus tömb, amelynek elemei folytonos memória területen helyezkednek el.
<list>	Szekvenciális	Kétszeresen láncolt lista, az elemek szétszórtan helyezkednek el a memóriában.
<deque>	Szekvenciális	Kettősvégű sor, ahol az egybefüggő memóriaterületek szétszórtan helyezkednek el.
<array>	Szekvenciális	Statikus tömb, amelyek elemei fix méretű memóriaterületen, folytonosan helyezkednek el.
<forward list>	Szekvenciális	Egyszeresen láncolt lista.
<queue>	Adapter	A sor lehetővé teszi az elemek beszűrésát, továbbá a kivételét az elejéről.
<stack>	Adapter	Verem egy olyan tároló, amelyből mindig az utoljára elhelyezett elemet tudjuk

<code><priority queue></code>	Adapter	Egy fontossági sorrend adaptert implementál, ahol minden esetben a legmagasabb értékű elem van az elején.
<code><map></code>	Asszociatív	Olyan asszociatív tároló, amely egyedi kulcs/érték adatpárok tárolására alkalmas.
<code><set></code>	Asszociatív	Olyan asszociatív tároló, amely a halmazokat valósítja meg, így minden elem legfeljebb egyszer szerepelhet.
<code><multimap></code>	Asszociatív	Olyan asszociatív tároló, amely az asszociatív tömb egy variánsa az úgynevezett rendezett multi-asszociatív tömb, ahol ellentétben a map asszociatív tárolókkal a kulcsismétlődés megengedett.
<code><multiset></code>	Asszociatív	A halmaz egy variánsa az úgynevezett rendezett multi-halmaz, ahol ellentétben a halmazokkal a kulcsismétlődés megengedett.
<code><unordered_map></code>	Asszociatív	Olyan hasított (rendezetlen) asszociatív tömb, amely kulcs / érték párokat tárol.
<code><unordered_set></code>	Asszociatív	Olyan hasított (rendezetlen) halmaz, amely egyedi kulcsokkal rendelkezik.
<code><unordered_multi map></code>	Asszociatív	A rendezetlen asszociatív tömb egy variánsa az úgynevezett hasított (rendezetlen) asszociatív tömb, ahol ellentétben a

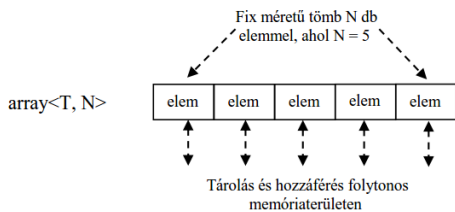
<unordered_multi set>	Asszociatív	rendezetlen asszociatív tömb tárolókkal, a kulcsismétlődés megengedett. Olyan hasított (rendezetlen) multi-halmaz, amelyeknél megengedett a kulcsismétlődés.
<bitset>	Asszociatív	Bithalmaz, logikai értékek tömbje, amely logikai értékek halmazát tárolja.

Szekvenciális tárolók

Az STL kezdetben (C++98) három eltérő szekvenciális tárolót valósított meg: **vector<AdatTípus>** - dinamikus tömb (vektor) folytonos tárolással, **list<AdatTípus>** - kétszeresen láncolt lista, és **deque<AdatTípus>** - kétvégű sor. A C++11 szabvány bevezetését követően további két darab szekvenciális tárolót implementáltak: **array<AdatTípus, Méret>** - statikus tömböt (vektor) folytonos tárolással, és a **forward_list<AdatTípus>** - egyszeresen láncolt listát. A szekvenciális tárolóknál a sorrendet a programozó határozza meg. Ezek a tárolók különböző módon valósítanak meg bizonyos feladatokat, és a programozóra van bízva, hogy az adott feladathoz melyiket választja, pontosabban a művelet milyensége határozza meg azt, hogy melyik tárolót használjuk. A tárolókat a műveletek végrehajtásának hatékonysága szerint is szokás differenciálni.

std::array

A tömb egy olyan szekvenciális tároló, amelynek elemei statikus (fix méretű) memóriaterületen, folytonos módon helyezkednek el. A tömb mérete már a fordítási időben ismert. Összességében tetszőleges elemeket képes tárolni, de típusát tekintve azonosnak (homogénnek) kell lenniük. Az STL-ben megvalósított statikus tömb, a „konvencionális” statikus tömb egy olyan továbbfejlesztett változata, ahol számos implementált tagfüggvény lehet segítségünkre, mint például az ellenőrzött indexelés.



Az Array tároló sematikus ábrája

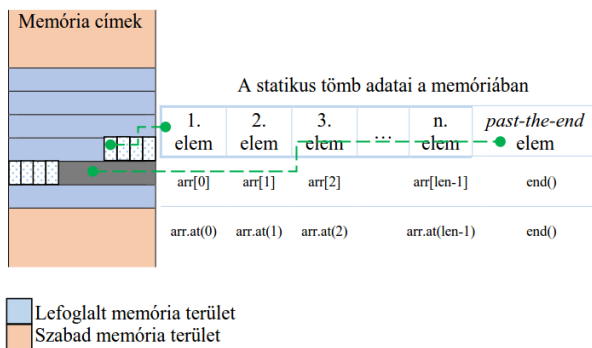
A statikus tömb deklarálásának általános szintaktikája, ahol az **ObjectType** a tömbben tárolt elemek típusát, amíg az **ArraySize** a tömb méretét jelöli:

```
array< ObjectType, ArraySize > ArrayName;
```

Egy statikus, egész szám típusú, 5 elemből álló tömböt, például a következő módon is létre hozhatunk:

```
#include <array>
int main()
{
    std::array<int, 5>;
}
```

Csakúgy, mint a „konvencionális” tömböknél, az elemeket közvetlen módon, indexelve ([] operátort használva) is elérhetjük vagy használhatjuk az ellenőrzött indexelést biztosító **at()** tagfüggvényt:



Egy statikus tömb elhelyezkedése a memóriában és az elemek közvetlen elérésének módja

A tömb első elemére mutató referencia értékét (referencia az első elemre) kérhetjük le a **front()** függvény segítségével (ekvivalens a ***(ArrayName.begin()-el)**, amíg az utolsó elemére mutató referencia értékét (referencia az utolsó elemre) a **back()** függvény alkalmazásával (ekvivalens a ***(ArrayName.end()-1-el)** tudhatjuk meg. A függvények alkalmazása előtt érdemes az **empty()** függvény hívásával meggyőződni arról, hogy a tömb üres-e, ellenkező esetben egy üres tömbre történő hívás nem értelmezett, mivel a referencia érvénytelen lesz. A fentebbi két tagfüggvény segítségével nem csak a statikus tömb két végén szereplő referencia értékét tudhatjuk meg, hanem egyfajta értékadó utasításként is használhatjuk. A következő példa jól szemlélteti a fentebb tárgyalt függvények használatát:

```

#include <array>
#include <iostream>

int main()
{
    std::array<char> Letters{ 'q', 'w', 'e', 'r', 't', 'z' };

    if (!Letters.empty())
    {
        std::cout << "A statikus tömb első betűje: " << Letters.front() << '\n';
        std::cout << "A statikus tömb utolsó betűje: " << Letters.back() << '\n';
        Letters.front() = 'a';
        Letters.back() = 'b';
    }
}

```

A fentebbi forráskód részlet eredményként a statikus tömb első- és utolsó karakterét írja ki, valamint beállításra (félülírásra) kerül az első (q karakter helyett a) és az utolsó elem (z karakter helyett b).

A teljes tömböt (0 ... N-1, ahol N a tömb maximális mérete) a **fill(const T& érték)** módosító függvény segítségével tölthetjük fel a paraméterben megadott specifikus értékkel.

ArrayName.fill(500) →

500	500	500	500	500
-----	-----	-----	-----	-----

A statikus tömb tároló feltöltés műveletének ábrája

A **swap(array& CserelendoTomb)** módosító tagfüggvény segítségével két azonos típusú és méretű tömb tartalma cserélhető meg. Az alábbi rövid kódrészlet szemlélteti a **swap()** függvény működését.

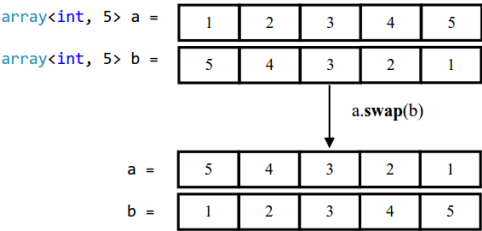
```

#include <array>

int main()
{
    array<int, 5> a = { 1,2,3,4,5 };
    array<int, 5> b = { 5,4,3,2,1 };

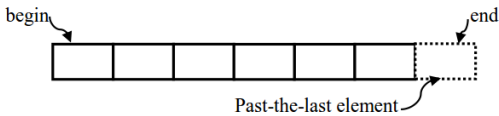
    a.swap(b);
}

```



A statikus tömb tároló csere műveletének ábrája

A statikus tömbben tárolt elemek számát a **size()** tagfüggvény segítségével, míg a tároló maximális méretét a **max_size()** tagfüggvény révén kérhetjük le. Azonban mivel minden egyes statikus tömb fix méretű, folytonos memória területen helyezkedik el, így a **max_size()** függvény által visszaadott érték minden esetben egyenlő lesz a tömb méretével (amelyet jelöljünk **N** konstans értékkel). A **begin()** tagfüggvény a tároló legelső pozícióját reprezentálja, amely egy iterátor objektum, amíg az **end()** tagfüggvény a tároló végét reprezentálja szintén egy iterátor objektum segítségével.



Elem hozzáférés	Általános, rövid leírás
at	Az elemek indexelt elérésének azon változata, ahol kivétel (out_of_range exception) keletkezik abban az esetben, ha az indexelés érvénytelen.
[] operátor	Az elemek indexelt elérésének azon változata, ahol nincs indexhatár ellenőrzés, akárcsak egy hagyományos tömb esetében.

front	Az első elem referencia értékét adja vissza vagy állítja be.
back	Az utolsó elem referencia értékét adja vissza vagy állítja be.
data	Az első elemre mutató referenciával tér vissza.

Kapacitás	Általános, rövid leírás
size	A tárolt elemek számát adja vissza.
max_size	A maximális méretet adja vissza.
empty	Üresség vizsgálata. Logikai igaz értékkel tér vissza, ha a tömb tároló üres, ellenkező esetben logikai hamis lesz az érték.

Módosítók	Általános, rövid leírás
fill	Felöltés adott értékkel.
swap	Két egyező típusú és méretű statikus tömb tároló értékeinek megcserélése.

Bejárók	Általános, rövid leírás
begin	A tömb elejét (első elemét) reprezentálja egy tetszőleges elérésű bejáróval.
end	A tömb utolsó utáni helyét (utolsó utáni elem – <i>past-the-end</i>) reprezentálja egy tetszőleges elérésű bejáróval.
rbegin	A tömb végét (utolsó elem) reprezentálja tetszőleges elérésű, fordított bejáróval.
rend	A tömb első elem előtti helyét (első előtti elem) reprezentálja egy tetszőleges elérésű, fordított bejáróval.
cbegin	A tömb elejét (első elem) reprezentálja egy konstans elérésű bejáróval.
cend	A tömb utolsó utáni helyét (utolsó utáni elem – <i>past-the-end</i>) reprezentálja egy konstans elérésű bejáróval.
crbegin	A tömb végét (utolsó elem) reprezentálja egy konstans elérésű, fordított bejáróval.

crend

A tömb első elem előtti helyét (első előtti elem)
reprezentálja egy konstans elérésű, fordított bejáróval.

Statikus tömb objektum tagfüggvényei

Minta példa a statikus tömb (array) tároló használatára Visual Studio Community fejlesztőkörnyezetben

Írjunk egy `ArrayDemo` (*ArrayDemo*) osztályt, amely különböző, statikus tömb specifikus műveleteket ír le, majd egy objektumpéldány segítségével ezeket a műveleteket hajtassuk végre. Minden egyes tagfüggvény feladat megfogalmazását, zárójelben követi az adott tárolóhoz tartozó tagfüggvény, amelyet fel kell használni a feladat helyes megoldásához, például a konstruktor esetében ilyen a **fill()**. Az osztály adatait a következők:

- `array`, egy statikus tömb jellegű szekvenciális tároló, amely 10 db egész szám tárolására képes.

Ügyeljünk az egységbezáras elvére, így a fentebb megfogalmazott adattag(ok) **private**, az alábbi tagfüggvények **public** láthatósági szintekkel bírnak!

Valósítsuk meg az osztály metódusait:

- Írjuk meg az osztály konstruktorát, amely segítségével inicializáljuk a statikus tömb tárolót az 5-ös értékkel. (**fill()**)
- Írjunk egy `getArrayOfArray()` függvényt, amely a tömb aktuális méretét adja vissza. (**size()**)
- Írjunk egy `getMaxSizeOfArray()` függvényt, amely a tömb maximális méretének értékével tér vissza. (**max_size()**)
- Írjunk egy `isArrayEmpty()` függvényt, amely megvizsgálja, hogy a tömb tartalmaz-e elemeket, majd annak függvényében vissza tér egy logikai értékkel. Igazsal, ha található elem a tömbben, hamissal, ha üres. (**empty()**)
- Írjunk egy `assignFibonacciValueBySquareBracketsOperator()` függvényt, amely a tömb elemeit új értékekkel látja el, amely értékek a Fibonacci sorozat első 10 értéke. (**[] operator**)
- Írjunk egy `assignPrimValueByAt()` függvényt, amely a tömb elemeit új értékekkel látja, amelyek az 3 és 31 közötti prímszámok. (**at()**)
- Írjunk egy `printArray()` függvényt, amely kiírja egy konstans iterátor felhasználásával a tömb minden egyes elemét. (**cbegin()**, **cend()**)

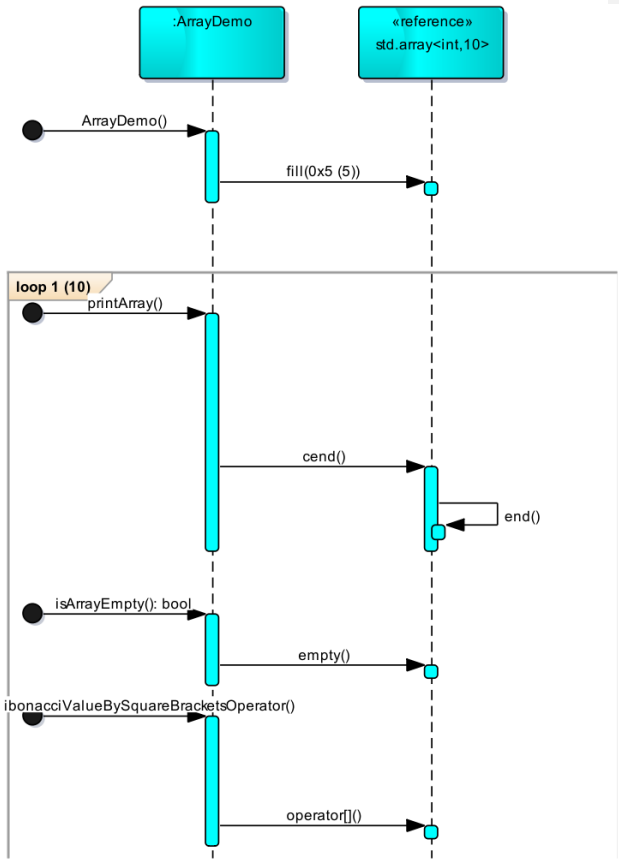
Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén **[heap]**) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektumpéldányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. Figyelem az **auto** kulcsszóval a fordítóra is bízhatjuk az iterátorok típusának felismerését! Mielőtt a feladatnak nekilátnánk, tervezzünk meg az osztály- és szekvencia diagramot. **Figyelem! A könyvben ábrázolt szekvenciadiagramok terjedelmi korlátok miatt a teljesség igénye nélkül készültek!**

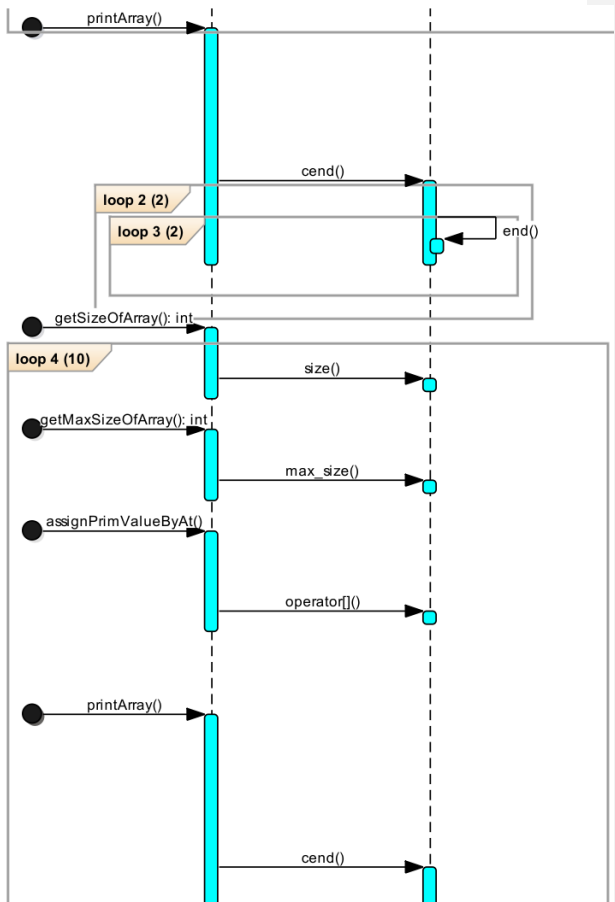
Statikus tervezés

ArrayDemo
- array: std::array<int, 10>
+ ArrayDemo() + assignFibonacciValueBySquareBracketsOperator(): void + assignPrimValueByAt(): void + getMaxSizeOfArray(): int + getSizeOfArray(): int + isArrayEmpty(): bool + printArray(): void

Osztálydiagram

Dinamikus diagram





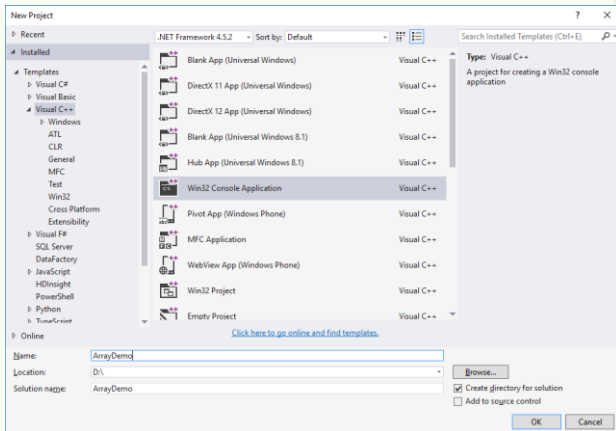


Szekvenciadiagram

A projekt létrehozása

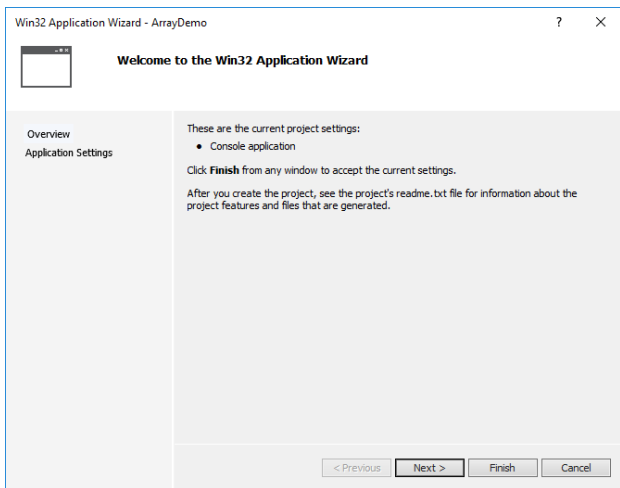
A projekt megvalósítása során konzolalkalmazást készítünk, ahol a Visual Studio, szabványos C++-nak megfelelő szintaktikát alkalmaz. A projekt három fájlban kerül implementálásra. Az alkalmazás belépési pontja a **main** függvény, amely a három fájl közül az egyikben fog helyet foglalni. Az előzőekben 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 közül a **Visual C++ Win32** sablont (**Template**) kell kiválasztani és a **Win32 Console Application**-t, mely az alábbi ábrán látható.



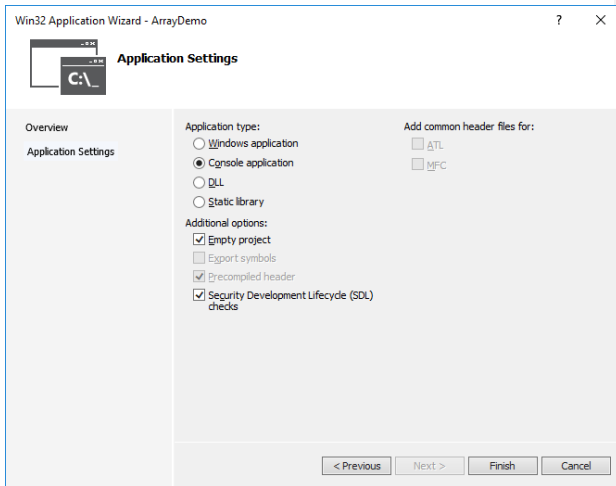
Win32 Console Application projekt létrehozása

Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően az **ArrayDemo** név került megadásra. Az OK gombra kattintást követően elindul az alkalmazás varázsló, mely a következő ábrán látható.



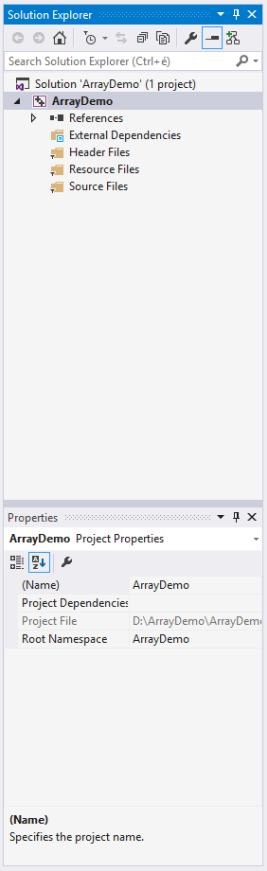
Win32 konzol alkalmazás varázsló

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.



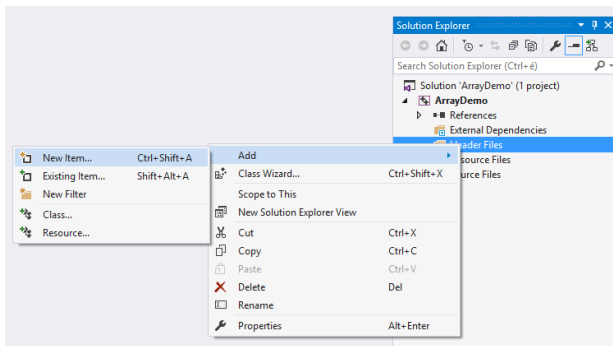
Alkalmazás beállításai

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



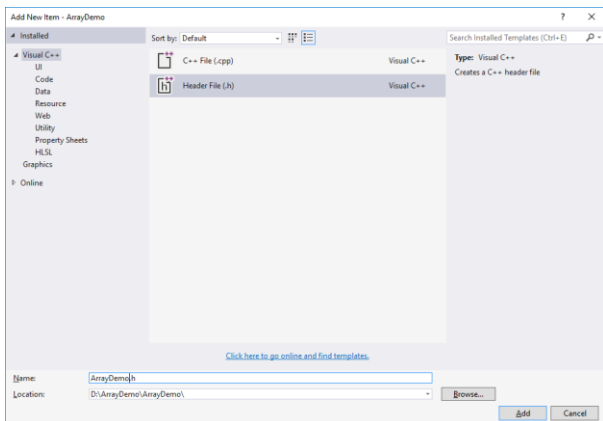
Megoldás és beállítások ablak

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.



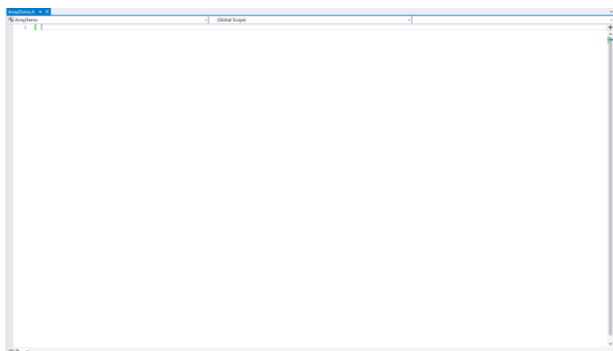
Új header fájl hozzáadása a projekthez

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



Új header fájl hozzáadása a projekthez, név megadása

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



Forráskód szerkesztő

ArrayDemo.h

```
#ifndef ARRAYDEMO_H
#define ARRAYDEMO_H

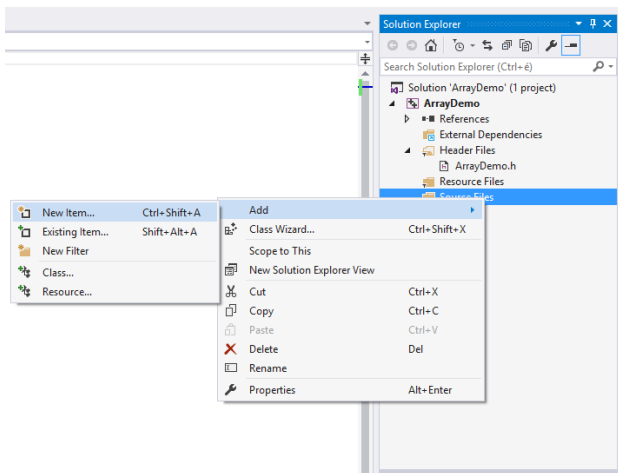
#include <array>

class ArrayDemo
{
private:
    std::array<int, 10> array;

public:
    ArrayDemo();
    int getArrayOfArray();
    int getMaxSizeOfArray();
    bool isArrayEmpty();
    void assignFibonacciValueBySquareBracketsOperator();
    void assignPrimValueByAt();
    void printArray();
};

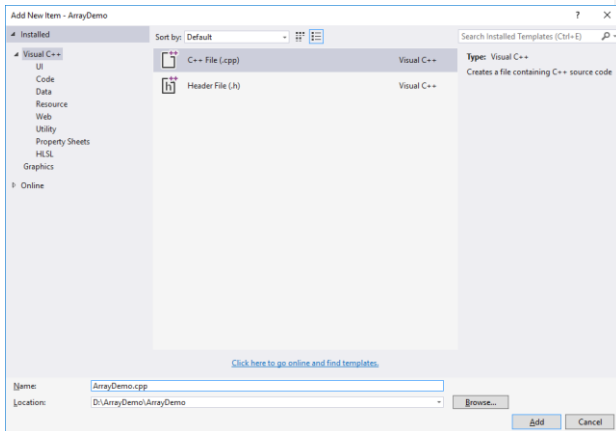
#endif
```

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



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása
ArrayDemo.cpp

```

#include <iostream>
#include "ArrayDemo.h"

ArrayDemo::ArrayDemo()
{
    array.fill(5);
}

int ArrayDemo::getArrayOfArray()
{
    return array.size();
}

int ArrayDemo::getMaxSizeOfArray()
{
    return array.max_size();
}

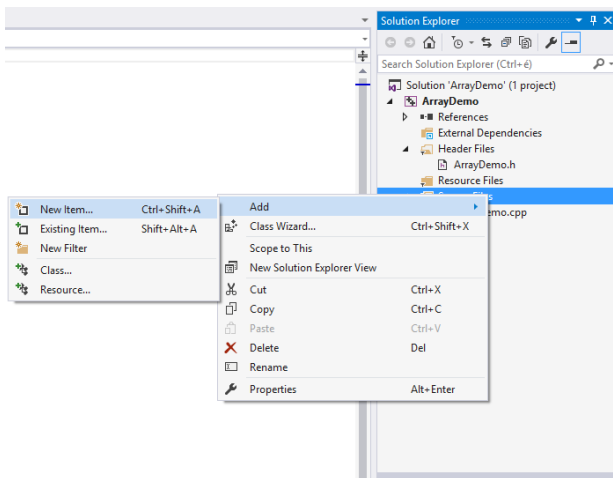
bool ArrayDemo::isArrayEmpty()
{
    return array.empty();
}

void ArrayDemo::assignFibonacciValueBySquareBracketsOperator()
{
    int firstTerm = 1, secondTerm = 1;
    array[0] = firstTerm;
    array[1] = secondTerm;
    for (unsigned int i = 2; i < 10; ++i)
    {
        array[i] = firstTerm + secondTerm;
        firstTerm = secondTerm;
        secondTerm = array[i];
    }
}

void ArrayDemo::assignPrimValueByAt()
{
    for (int i = 3, k = 0; k < 10; ++i)
    {
        int prim=1;
        for (int j = 2; j < i; j++)
        {
            if (i % j == 0)
            {
                prim=0;
            }
        }
        if(prim) array[k++] = i;
    }
}

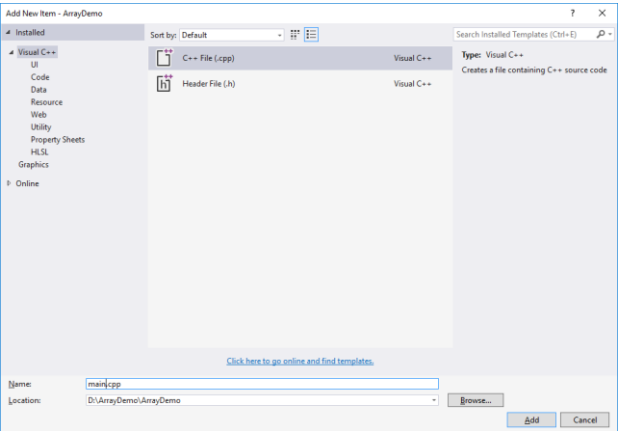
void ArrayDemo::printArray()
{
    for (std::array<int, 10>::const_iterator constIterator = array.cbegin();
        constIterator != array.cend(); ++constIterator)
        std::cout << *constIterator << ' ';
}

```



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása
main.cpp

```

#include <iostream>
#include <conio.h>
#include "ArrayDemo.h"

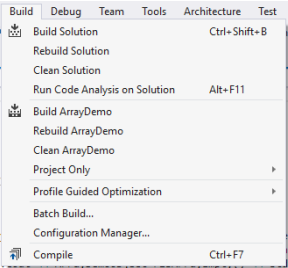
int main()
{
    ArrayDemo *ArrayDemoObject = new ArrayDemo();

    std::cout << "MUVELET => ArrayDemo() #konstruktor hivasa, statikus tomb
feltoltese..." << std::endl;
    std::cout << "ALLAPOT: "; ArrayDemoObject->printArray(); std::cout <<
std::endl;
    std::cout << "MUVELET => empty() #uresseg vizsgalata..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << ArrayDemoObject->isArrayEmpty()
<< std::endl;
    std::cout << "MUVELET => [] operator #(ujra)feltoltes..." << std::endl;
    ArrayDemoObject->assignFibonacciValueBySquareBracketsOperator();
    std::cout << "ALLAPOT: "; ArrayDemoObject->printArray(); std::cout <<
std::endl;
    std::cout << "MUVELET => size() #meret lekerdezese..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << ArrayDemoObject-
>getSizeOfArray() << std::endl;
    std::cout << "MUVELET => max_size() #maximalis meret lekerdezese..." <<
std::endl;
    std::cout << "EREDMENY: "; std::cout << ArrayDemoObject-
>getMaxSizeOfArray() << std::endl;
    std::cout << "MUVELET => at() #(ujra)feltoltes..." << std::endl;
    ArrayDemoObject->assignPrimValueByAtPrim();
    std::cout << "ALLAPOT: "; ArrayDemoObject->printArray(); std::cout <<
std::endl;

    delete ArrayDemoObject;
    _getch();
    return 0;
}

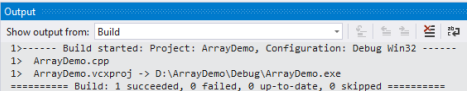
```

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.



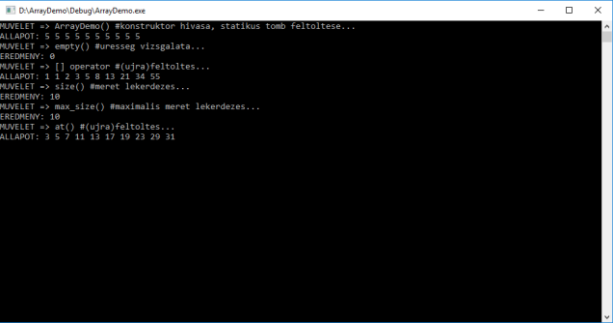
Program fordítása

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



Kimenet ablak

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



A program futásának eredménye

A feladathoz kapcsolódó magyarázatok

Az osztály deklarációját az (**ArrayDemo.h**) állományban adtuk meg. A deklarációban egy magán (**private**) jellegű, 10 elemű, egész szám típusú (**integer**) adatok tárolására képes, statikus tömb került megfogalmazásra, amelyet az **std** névtéren keresztül értünk el:

```
private:
    std::array<int, 10> array;
```

A konstruktor neve tulajdonságaiból fakadóan megegyezik az osztály nevével, láthatósági szintje nyilvános (**public**) visszatérési értéke nincs, még **void** sem, továbbá az adott feladatban paraméter nélküli formában jelent meg:

```
public:
    ArrayDemo();
```

A *getSizeOfArray()* és a *getMaxSizeOfArray()* tagfüggvények a leírásban megfogalmazott tömb aktuális- és maximális méret lekérdezését biztosítják. A láthatósági szintjük nyilvános (**public**), visszatérési típusuk egész szám (**integer**) és argumentumlistájuk üres:

```
int getSizeOfArray();
int getMaxSizeOfArray();
```

Az *isArrayEmpty()* tagfüggvény a tömb ürességének vizsgálatát hajtja végre, láthatósági szintje nyilvános (**public**), visszatérési típusa logikai (**bool**), argumentumlistája üres:

```
bool isArrayEmpty();
```

Az *assignFibonacciValueBySquareBracketsOperator()* tagfüggvény segítségével a [] operátoron keresztül értük el a tömb egyes elemeit. Ne feledjük, hogy az ilyen módon történő tömb elemek indexelt elérésének esetében nincs ellenőrzés. Az *assignPrimValueByAt()* tagfüggvény feladata az ellenőrzött indexelt tömb elemek elérése. Végül megfogalmazásra került egy *printArray()*, tagfüggvény, amely egy konstans iterátor segítségével a képernyőn megjeleníti a tömb aktuális elemeit. A fentebb említett három tagfüggvény visszatérési típusaik **void**, láthatósági szintjeik nyilvánosak (**public**), valamint az argumentumlistájuk üres.

```
void assignFibonacciValueBySquareBracketsOperator();
void assignPrimValueByAt();
void printArray();
```

Az **ArrayDemo.cpp** állomány tartalmazza az osztály teljeskörű implementációját. Az implementáció során az egyes tagfüggvények neve előtt a **::** (hatókör) operátor került alkalmazásra, azon célból, hogy pontosan meghatározzuk melyik osztály tagfüggvényét valósítottuk meg.

A konstruktor feladata a statikus tömb minden elemének 5-el történő feltöltése, amelyet egyszerűen **fill()** tagfüggvény segítségével hívásával valósítottunk meg.

```
ArrayDemo::ArrayDemo()
{
    array.fill(5);
}
```

Az aktuális és maximális méret lekérdezését végrehajtó függvények implementációja során az **array** objektumon keresztül egyszerűen hívásra kerültek a **size()**, illetve a **max_size()** metódusok.

```
int ArrayDemo::getArrayOfArray()
{
    return array.size();
}
int ArrayDemo::getMaxArrayOfArray()
{
    return array.max_size();
}
```

Az üresség vizsgálata során a méret lekérdezéshez hasonlóan az **array** objektumon keresztül hívtuk meg a szükséges tagfüggvényt, amely ebben az esetben az **empty()** metódus.

```
bool ArrayDemo::isArrayEmpty()
{
    return array.empty();
}
```

A tömb elemeinek eléréséhez két módszert, az index helyességét nem vizsgáló **[] operátor**-t és ellenőrzést végző **at()** alkalmaztunk. A **[] operátor** alkalmazása során a Fibonacci sorozat első 10 értékét adtuk értékül a tömbünknek, amely sorozatról tudjuk, hogy egy elem mindig az előző kettő összegeként szerepel.

[User1] megjegyzést írt: módosítani kell a kódot

Az `at()` tagfüggvény esetében 3-tól kezdődően 31-ig a prímszámokat adtuk értékül a tömbünknek. A prímszámokról tudjuk, hogy maradék nélkül, kizárólag egyel és önmagukkal oszthatóak, mely ciklusban kerül ellenőrzésre. Az indexelt ellenőrzés vizsgálathoz próbáljuk túlindexelni a tömböt. Mi történik ekkor? A tömb elemek megjelenítéséhez konstans iterátort alkalmaztunk, mivel az elemeket kizárólag olvasni szeretnénk. A `cbegin()` tagfüggvény a tömb elejét (első elem), addig a `cend()` a tömb végét (utolsó utáni elem) reprezentálja egy konstans elérési bejáróval.

```
void ArrayDemo::assignValueBySquareBracketsOperator()
{
    int firstTerm = 1, secondTerm = 1;
    array[0] = firstTerm;
    array[1] = secondTerm;
    for (unsigned int i = 2; i < 10; ++i)
    {
        array[i] = firstTerm + secondTerm;
        firstTerm = secondTerm;
        secondTerm = array[i];
    }
}

void ArrayDemo::assignPrimValueByAt()
{
    for (int i = 3, k = 0; k < 10; ++i)
    {
        int prim=1;
        for (int j = 2; j < i; j++)
        {
            if (i % j == 0)
            {
                prim=0;
            }
        }
        if(prim) array[k++] = i;
    }
}

void ArrayDemo::printArray()
{
    for (std::array<int, 10>::const_iterator constIterator = array.cbegin();
        constIterator != array.cend(); ++constIterator)
        std::cout << *constIterator << ' ';
}
```

A `(main.cpp)` állomány tartalmazza a `main()` függvényt, a program belépési pontját. A fő függvényben történő dinamikus példányosítása után hívjuk meg a feladat teljesítéséhez szükséges függvényeket. A műveletek végrehajtását

követően, gondoskodjunk a lefoglalt erőforrások felszabadításáról az alapértelmezett explicit destruktorki hívásának segítségével.

std::vector

A vektor egy olyan szekvenciális tároló, amely lényegében egy tömb és elemei folytonos memóriaterületen helyezkednek el, továbbá ezt a területet dinamikusan, akár futás közben is képes változtatni, ellentétben a hagyományos tömbök rögzített memória terület méretével. Így a vektor többek között lehetőséget biztosít arra, hogy elkerüljük a memóriahely pazarlását (túl méretes tömb), vagy éppen a tömb túlszordulását (alul méretes tömb) jelenségét. Mind ezek mellett a **vector** osztály objektumpéldánya és a fordító közösen gondoskodik a memória felszabadításáról, így akár kezeletlen kivétel esetén is felszabadítják a lefoglalt erőforrásokat, elkerülve ezzel például a memóriaszivárgás (**memory leak**) jelenséget. A vektor deklarálásának általános szintaktikája:

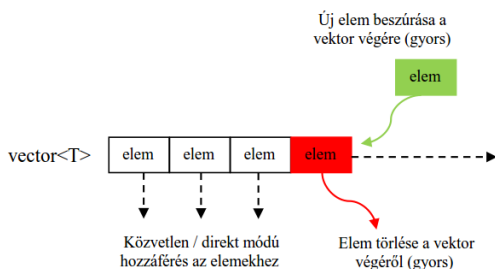
```
vector< ObjectType > VectorName;
```

Egy vektort a következő módon hozhatunk létre:

```
#include <vector>

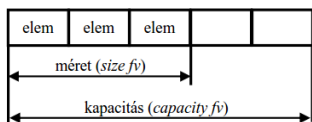
int main()
{
    std::vector<int> MyVector;
}
```

A vektor memóriához allokációjáért felelős algoritmus általában (azért az általánosítás, mivel egyes **STL** verziókban ez külön szabályozható) dupla mennyiségű memóriaterületet foglal, abban az esetben, ha méret növelésre lenne szükség, így minimalizálva a másolásból eredő művelet idő növekedését (**overhead**), azonban a tároló lehetőséget kínál a memóriához allokáció programozó által történő végrehajtására is a **reserve()** tagfüggvény segítségével. A függvényről és annak működéséről, egy külön bekezdésben lehet olvasni. Csakúgy, mint a már jól megszokott egyszerű tömböknél, itt is lehetőség van indexelt hozzáférésre, a mérete 0-tól (**size()**-1) terjed. A vektor a végéhez való hozzáfűzést vagy éppen törlést nagyon gyorsan hajtja végre, abban az esetben, ha az elemeket odébb kell csúsztatni lassúvá válhat, ugyanis ez sok másolási művelettel jár a folytonos memóriaterületen való elhelyezkedés miatt.



A Vector tároló sematikus ábrája elem hozzáférése és törlése esetén

A vektornak két fontos mérőszáma létezik. Az első a kapacitás (**capacity**), amely a vektor számára lefoglalt maximális memória területet jelenti, amelyen képes elemeket tárolni anélkül, hogy újra allokálná (foglalná) a memória területet. A második a méret (**size**), amely az aktuális elemek által a vektorban lefoglalt memória területet mutatja. Célszerű a vektor számára nagyobb helyet foglalni, mint az elemek aktuális száma, mivel ekkor nem kell a vektor végére történő beszúrások esetén átmásolni az adatokat egy nagyobb memóriaterületre. Összességében elmondható, hogy a vektor kapacitásának mindig nagyobbnak kell lennie, mint a méretének. A kapacitás értéket az úgynevezett **capacity()** függvény, amíg a méret értéket a **size()** függvény segítségével kérhetjük le.



A vektor tároló kapacitása és mérete

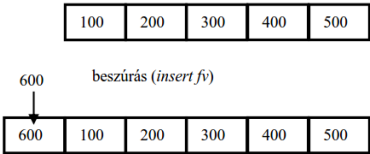
A kapacitást manuálisan a **reserve()** függvény segítségével állíthatjuk be:

```
std::vector<int> v;  
v.reserve(100);
```

A fenti kódrészletben egy **v** nevezetű vektort deklaráltunk, ahol a kapacitás és az elemszám is egységesen 0 nagyságú és 0 darabszámú. A deklarálást követően

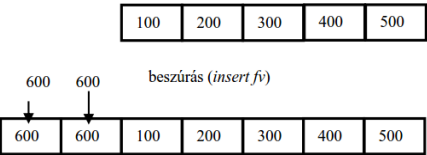
kapacitást „manuálisan” 100 egységre állítottuk. Amennyiben a vektorba szeretnénk beszúrní egy elemet, akkor a vektor **insert()** függvényét kell meghívni, amelynek első paramétere egy iterátort, a második paramétere egy opcionálisan megadandó számot (olyan mennyiség, amely azt mondja meg, hogy az érték hányszor szerepeljen egymás után a vektorban), a harmadik paramétere egy beszúrandó értéket vár. A tagfüggvény mindig a megadott iterátorral jelzett elem elé szúrja be az új elem értékét. Az alábbi ábra szemlélteti a vektor **insert()** függvényének működését. A példában 100, 200, 300, 400 és 500-as értékek kerültek tárolásra, majd az **insert()** függvény megfelelő paraméterezésével beszúrára kerül a 600-as érték a vektor első helyére.

```
std::vector<int> v{ 100 , 200 , 300 , 400 , 500 };
vector<int>::iterator i = v.begin();
v.insert(i, 600);
```



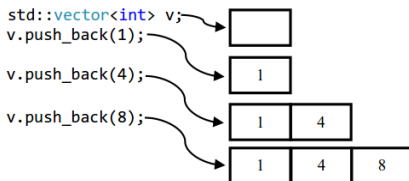
A vektor tároló beszúrás műveletének ábrája

```
std::vector<int> v { 100 , 200 , 300 , 400 , 500 };
vector<int>::iterator i = v.begin();
v.insert(i,2,600);
```



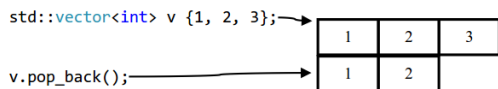
A vektor tároló beszúrás műveletének ábrája

A vektor végéhez történő elem beszúrása a **push_back()** metódus segítségével hajtható végre, ahol paraméterben megadhatjuk a tárolóhoz fűzendő értéket. A következő ábra szemlélteti a **push_back()** függvény működését:



A Vektor tároló hozzáfűzés műveletének ábrája

A vektorból való törlés többféle módon is történhet. Amennyiben a vektor utolsó elemét szeretnénk törölni, akkor a **pop_back()** tagmetódus használható:



A vektor tároló utolsó elemének törlése

Egyébként, ha egy tetszőleges elem kitörlése a cél, akkor az **erase(begin, end)** függvény használatával tehetjük meg, amely a törlendő elemre mutató iterátort várja paraméterül:

```
std::vector<int> v { 100 , 200 , 300 , 400 , 500 };
vector<int>::iterator i = v.end();
v.erase(i);
```

100	200	300	400	500
-----	-----	-----	-----	-----

törlés (*erase fv*)

100	200	300	400
-----	-----	-----	-----

A vektor tároló elemének törlése **erase()** függvény segítségével

A tároló összes elem törléséhez a **clear()** metódus áll rendelkezésre. Az alábbi táblázat röviden összefoglalja a kifejezetten vektor tárolónál használható tagfüggvényeket és azok feladatait.

Elem hozzáférés	Általános, rövid leírás
at	Az elemek indexelt elérésének azon változata, ahol kivétel (out_of_range exception) keletkezik abban az esetben, ha az indexelés érvénytelen.
[] operátor	Az elemek indexelt elérésének azon változata, ahol nincs indexhatár ellenőrzés, akárcsak egy hagyományos tömb esetében.
front	Az első elem referencia értékét adja vissza vagy állítja be. Üres vektor esetén a referencia érvénytelen.
back	Az utolsó elem referencia értékét adja vissza vagy állítja be. Üres vektor esetén a referencia érvénytelen.
data	Az első elemre mutató referenciával tér vissza.
Kapacitás	Általános, rövid leírás
size	A tárolt elemek számát adja vissza.
max_size	A maximális méretet adja vissza.
resize	A vektor átméretezése.
capacity	Azon üres (még felhasználható) elemszámok mennyiségét adja vissza, amelyekbe még úgy helyezhetünk értékeket, hogy az újabb memória foglalást igényelne.
empty	Üresség vizsgálata. Logikai igaz értékkel tér vissza, ha a vektor tároló üres, ellenkező esetben logikai hamis lesz az érték.
reserve	A függvény paramétereként megadott számértékkel növeli a kapacitását.
shrink_to_fit	A kapacitást a mérethez igazítja.
Módosítók	Általános, rövid leírás
assign	Több előfordulása ismert. Az egyik változat során a tagfüggvény az elem n darab másolatát behelyezi a vektorba, amíg egy másik változatnál az iterátorokkal

	megadott tartományt (iterator_start és iterator_end között) helyezi be a vektorba.
push_back	A paraméterben megadott elem másolatát a vektor tároló végéhez fűzi.
pop_back	Az utolsó elem törlése.
insert	Több előfordulása ismert. Az első változat során a függvény paraméterében megadott pozíció helyére beszúrja a szintén paraméterként átadott értéket. Egy második változat esetében a függvény paraméterében megadott pozíció helyére, <i>n</i> alkalommal beszúrja a szintén paraméterként átadott értéket. A harmadik változat során a függvény paramétereként megadott pozícióra beszúrja a tartományként megadott elemeit.
erase	Több előfordulása ismert. Az első változat során a paraméterben megadott pozíció helyen törli az elemet, amíg egy második változat esetében a paraméterben megadott tartományból törli az összes elemet.
swap	Két vektor tartalmának megcserélése.
clear	Az összes elem törlése.
emplace	Specifikus pozícióba, új elemet helyez, így növelve a méretet. Amennyiben a méretnek nagyobb kapacitásra lenne szüksége, úgy a vektor újra allokálódik.
emplace_back	Az utolsó pozícióban, új elemet helyez, így megnövelve a méretet. Amennyiben a méretnek nagyobb kapacitásra lenne szüksége, úgy a vektor újra allokálódik.

Bejárók	Általános, rövid leírás
begin	A vektor elejét (első elemét) reprezentálja egy tetszőleges elérésű bejáróval.
end	A vektor utolsó utáni helyét (utolsó utáni elem – <i>past-the-end</i>) reprezentálja egy tetszőleges elérésű bejáróval.
rbegin	A vektor végét (utolsó elem) reprezentálja egy tetszőleges elérésű, fordított bejáróval.

rend	A vektor első elem előtti helyét (első előtti elem) reprezentálja egy tetszőleges elérésű, fordított bejáróval.
cbegin	A vektor elejét (első elem) reprezentálja egy konstans elérésű bejáróval.
cend	A vektor utolsó utáni helyét (utolsó utáni elem – <i>past-the-end</i>) reprezentálja egy konstans elérésű bejáróval.
crbegin	A vektor végét (utolsó elem) reprezentálja egy konstans elérésű, fordított bejáróval.
crend	A vektor első elem előtti helyét (első előtti elem) reprezentálja egy konstans elérésű, fordított bejáróval.

A vektor objektum tagfüggvényei

Minta példa a dinamikus tömb (vector) tároló használatára Visual Studio Community fejlesztőkörnyezetben

Írjunk egy `VectorDemo` (*VectorDemo*) osztályt, amely különböző, vektor specifikus műveleteket ír le, majd egy objektumpéldány segítségével ezeket a műveleteket hajtassuk végre. Minden egyes tagfüggvény feladatának megfogalmazását, zárójelben követi az adott tárolóhoz tartozó tagfüggvény, amelyeket kötelező jelleggel kell használni a feladat helyes megoldásának elérése érdekében, például a konstruktor esetében ilyen a **push_back()** metódus. Az osztály adatait a következő:

- *vector*, egy dinamikus tömb jellegű szekvenciális tároló, amely 10 db egész szám tárolására képes.

Ügyeljünk az egységbezárás elvére, így a fentebb megfogalmazott adattag(ok) **private**, az alábbi tagfüggvények **public** láthatósági szintekkel bírnak!

Valósítsuk meg az osztály metódusait:

- Írjuk meg az osztály konstruktorát, amely segítségével inicializáljuk a vektor tárolót 100-tól - 109-ig. (**push_back()**)
- Írjunk egy *isVectorEmpty()* függvényt, amely megvizsgálja, hogy a vektor tartalmaz-e elemeket, majd annak függvényében vissza tér egy logikai értékkel. Igazsal, ha található elem a vektorban, hamissal, ha üres. (**empty()**)
- Írjunk egy *getSizeOfVector()* függvényt, amely a vektor aktuális méretét adja vissza. (**size()**)

- Írjunk egy `getCapacityOfVector()` függvényt, amely a vektor kapacitás értékét adja vissza. (**capacity()**)
- Írjunk egy `getMaxSizeOfVector()` függvényt, amely a vektor maximális méretének értékével tér vissza. (**max_size()**)
- Írjunk egy `resizeVector(int size)` függvényt, amely paraméterül vár egy egész szám típusú értéket, majd ezzel az értékkel átméretezi a vektort. (**resize()**)
- Írjunk egy `reserveVector(int value)` függvényt, amely paraméterül vár egy egész szám típusú értéket, majd ezzel az értékkel átméretezi a vektor kapacitását. (**reserve()**)
- Írjunk egy `shrinkToFit()` függvényt, amely a vektor kapacitását újraméretezi a tartalomhoz igazodóan. (**shrink_to_fit()**)
- Írjunk egy `recreateValuesWithAtToVector()` függvényt, amely a vektor elemeit új értékekkel látja el. Ezen értékek a "negafibonacci" sorozat 34, -21, 13, -8, 5, -3, 2, -1, 1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 és 89 elemei. (**at()**)
- Írjunk egy `assignValuesToVector()` függvényt, amely a vektort újra feltölti 7 hosszán 100-as értékekkel. (**assign()**)
- Írjunk egy `removeLastElementOfVector()` függvényt, amely a vektor utolsó elemét eltávolítja egészen addig, amíg az üres nem lesz, illetve a kivett elemeket jelenítsük meg a konzolon. (**pop_back()**)
- Írjunk egy `eraseElements(int length)` függvényt, amely a vektor elejétől, (**vector.begin()**) a függvény paramétereként megkapott indexig, törli az értékeket. (**erase()**)
- Írjunk egy `insertElements()` függvényt, amely a vektor elejétől a 3. indexig beszúrja a 7-es értéket. (**insert()**)
- Írjunk egy `emplaceBack(int value)` függvényt, amely a vektor végéhez hozzáfüzi a paraméterben kapott számértéket. (**emplace_back()**)
- Írjunk egy `printContent()` függvényt, amely kiírja egy konstans iterátor felhasználásával a vektor minden egyes elemét. (**cbegin(), cend()**)

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. Figyelem az **auto** kulcsszóval a fordítóra is bízhatjuk az iterátorok típusának felismerését! Mielőtt a feladatnak neki látnánk, tervezzük meg az osztály- és szekvencia diagramot. **Figyelem! A könyvben ábrázolt**

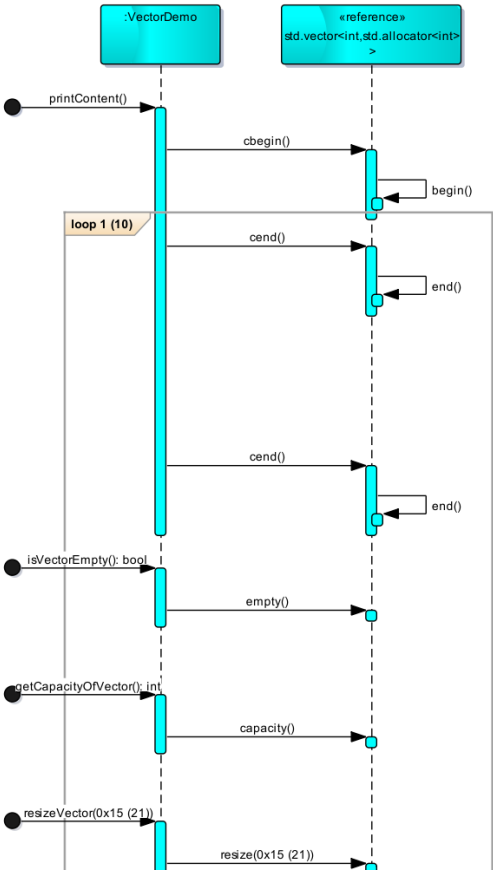
szekvenciadiagramok terjedelmi korlátok miatt a teljesség igénye nélkül készültek!

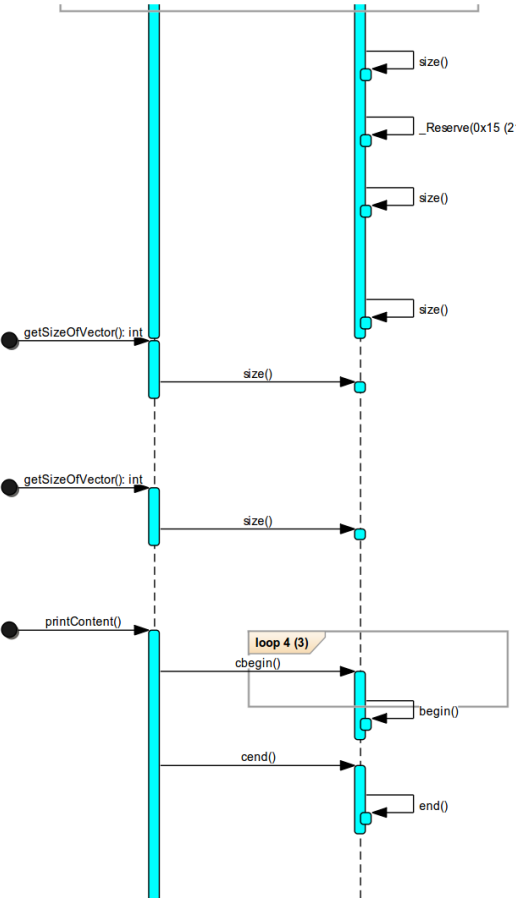
Statikus tervezés

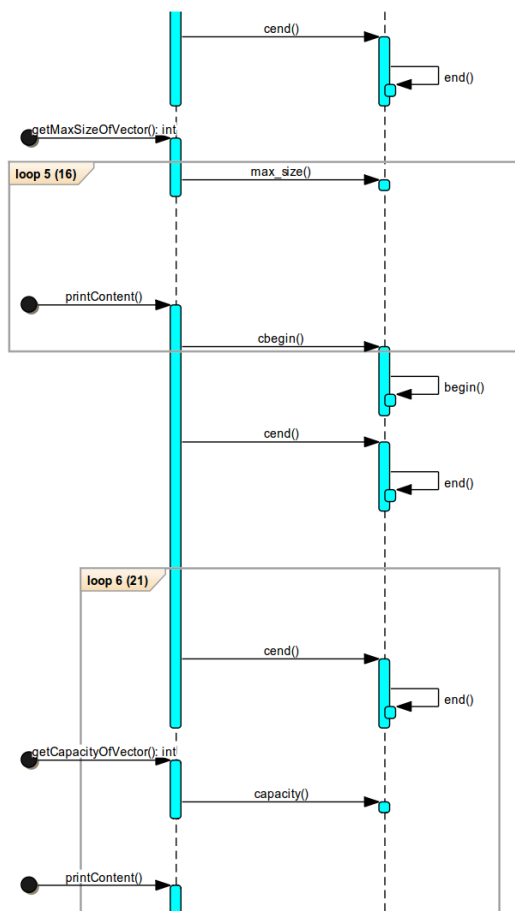
VectorDemo
- vector: std::vector<int>
+ assignValuesToVector(): void
+ emplaceBack(int): void
+ eraseElements(int): void
+ getCapacityOfVector(): int
+ getMaxSizeOfVector(): int
+ getSizeOfVector(): int
+ insertElement(int, int): void
+ isVectorEmpty(): bool
+ printContent(): void
+ recreateValuesWithAtToVector(): void
+ removeLastElementOfVector(): void
+ reserveVector(int): void
+ resizeVector(int): void
+ shrinkToFit(): void
+ VectorDemo()

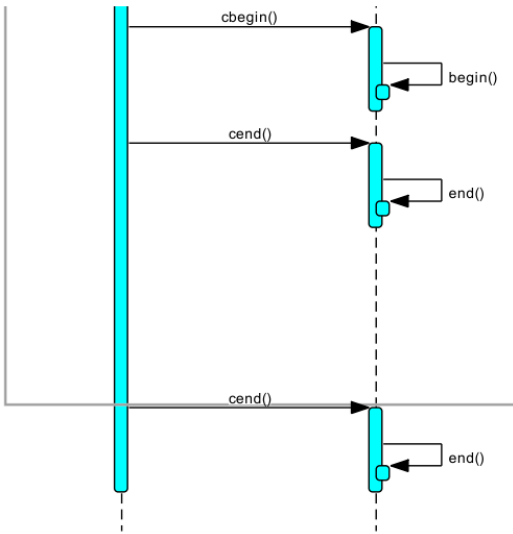
Osztálydiagram

Dinamikus diagram









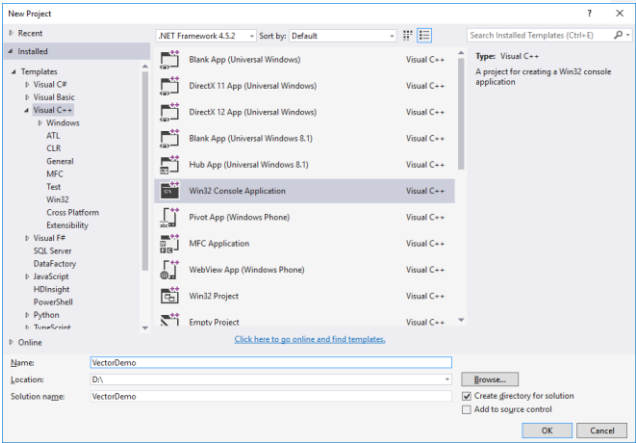
Néhány tagfüggvény szekvenciadiagramja

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.

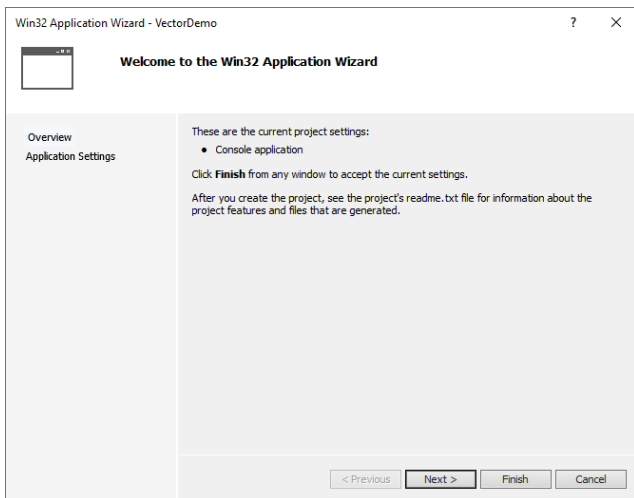
Új projekt a **File** menü **New** menüpontjában hozható létre, a **Project** almenüt kiválasztva, vagy akár a **Ctrl+Shift+N** gyorsindító gombkombinációval. Új projekt kiválasztása után megjelenő ablakban a közül a **Visual C++ Win32**

sablont (**Template**) kell kiválasztani és a **Win32 Console Application**-t, mely az alábbi ábrán látható.



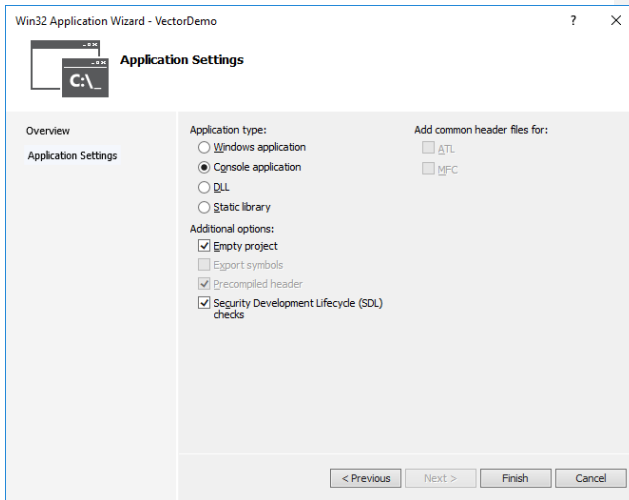
Win32 Console Application projekt létrehozása

Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően az **VectorDemo** név került megadásra. Az OK gombra kattintást követően elindul az alkalmazás varázsló, mely a következő ábrán látható.



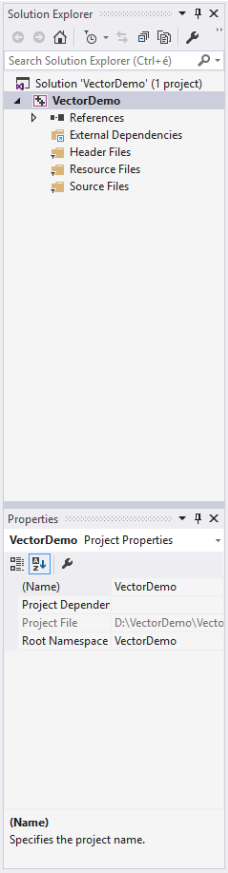
Win32 konzol alkalmazás varázsló

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.



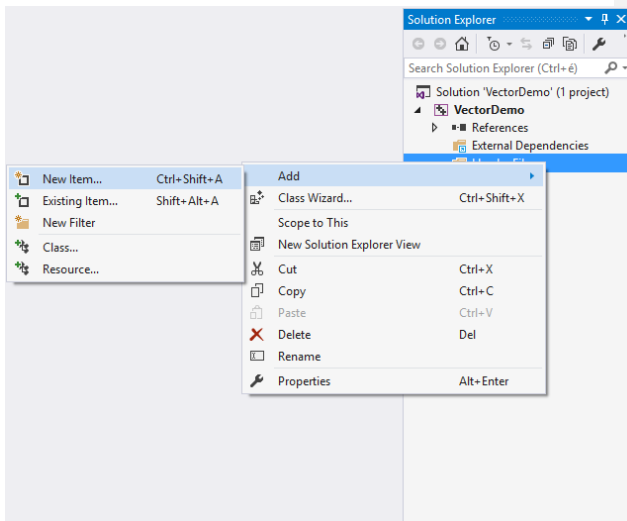
Alkalmazás beállításai

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



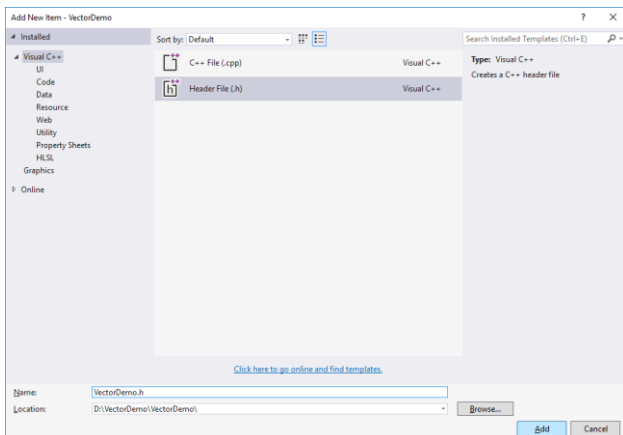
Megoldás és beállítások ablak

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.



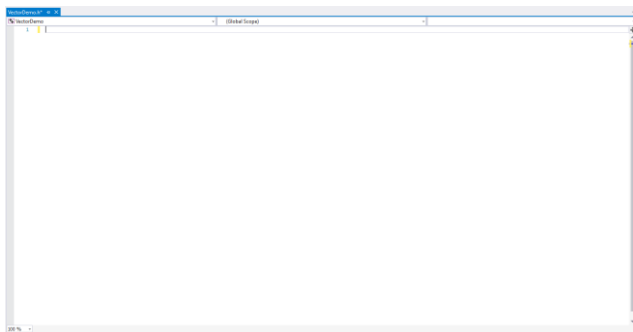
Új header fájl hozzáadása a projekthez

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



Új header fájl hozzáadása a projekthez, név megadása

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



Forráskód szerkesztő

VectorDemo.h

```
#ifndef VECTORDEMO_H
#define VECTORDEMO_H

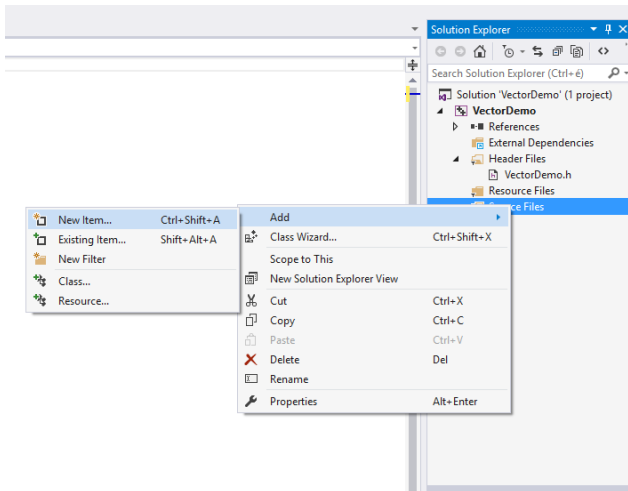
#include <vector>

class VectorDemo
{
private:
    std::vector<int> vector;

public:
    VectorDemo();
    int getSizeOfVector();
    int getCapacityOfVector();
    int getMaxSizeOfVector();
    void resizeVector(int size);
    bool isVectorEmpty();
    void reserveVector(int value);
    void shrinkToFit();
    void recreateValuesWithAtToVector();
    void assignValuesToVector();
    void removeLastElementOfVector();
    void insertElement(int length, int value);
    void eraseElements(int length);
    void emplaceBack(int value);
    void printContent();
};

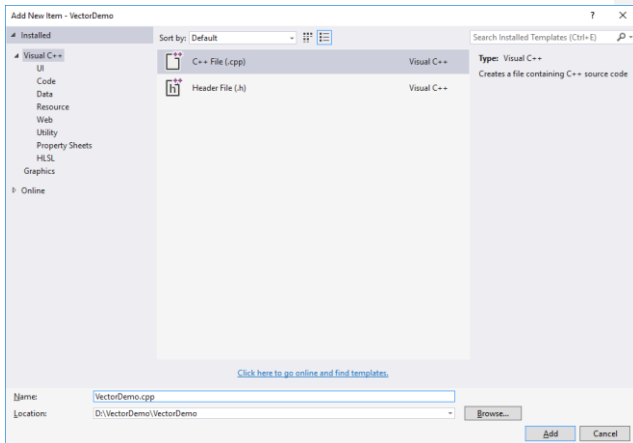
#endif
```

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



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása

VectorDemo.cpp

```
#include <iostream>
#include <math.h>
#include "VectorDemo.h"

VectorDemo::VectorDemo()
{
    for(unsigned int i = 0; i < 10; ++i)
        vector.push_back(100 + i);
}

bool VectorDemo::isVectorEmpty()
{
    return vector.empty();
}

int VectorDemo::getSizeOfVector()
{
    return vector.size();
}

void VectorDemo::resizeVector(int size)
{
    vector.resize(size);
}

int VectorDemo::getCapacityOfVector()
{
    return vector.capacity();
}

int VectorDemo::getMaxSizeOfVector()
{
    return vector.max_size();
}

void VectorDemo::reserveVector(int value)
{
    vector.reserve(value);
}

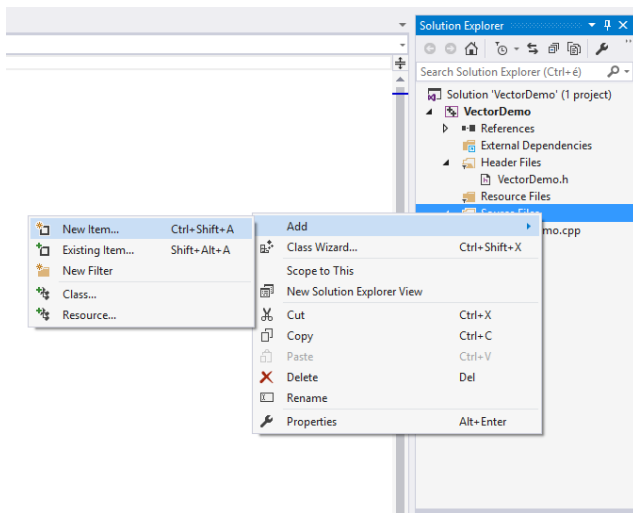
void VectorDemo::shrinkToFit()
{
    vector.shrink_to_fit();
}

void VectorDemo::recreateValuesWithAtToVector()
{
    int firstTermOffFibonacciSequence = 1, secondTermOffFibonacciSequence = 1;
    vector[getSizeOfVector() / 2] = firstTermOffFibonacciSequence;
    vector[getSizeOfVector() / 2 + 1] = secondTermOffFibonacciSequence;
    for (unsigned int i = getSizeOfVector() / 2 + 2; i < getSizeOfVector();
        ++i)
    {
        vector.at(i) = firstTermOffFibonacciSequence + s
econdTermOffFibonacciSequence;
        firstTermOffFibonacciSequence =
            secondTermOffFibonacciSequence;
        secondTermOffFibonacciSequence = vector.at(i);
    }
}
```

```

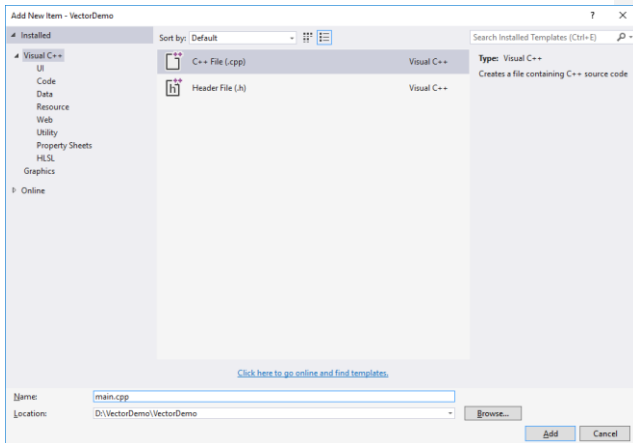
int firstTermOfNegaFibonacciSequence = 0,
secondTermOfNegaFibonacciSequence = 1;
vector[getsizeofvector() / 2 - 1] = firstTermOfNegaFibonacciSequence;
vector[getsizeofvector() / 2 - 2] = secondTermOfNegaFibonacciSequence;
for (int j = sizeofvector() / 2 - 3; j >= 0; --j)
{
    vector.at(j) = firstTermOfNegaFibonacciSequence -
secondTermOfNegaFibonacciSequence;
firstTermOfNegaFibonacciSequence =
secondTermOfNegaFibonacciSequence;
secondTermOfNegaFibonacciSequence = vector.at(j);
}
}
void VectorDemo::assignValuesToVector()
{
    vector.assign(7, 100);
}
void VectorDemo::removeLastElementOfVector()
{
    while (vector.begin() != vector.end())
    {
        std::cout << "v.back(): "; std::cout << "(" << (vector.back()) << " "
"; vector.pop_back();
    }
}
void VectorDemo::insertElement(int length, int value)
{
    std::vector<int>::iterator iterator = vector.begin();
    vector.insert(iterator, length, value);
}
void VectorDemo::eraseElements(int length)
{
    vector.erase(vector.begin(), vector.begin()+length);
}
void VectorDemo::emplaceBack(int value)
{
    vector.emplace_back(value);
}
void VectorDemo::printContent()
{
    for (std::vector<int>::const_iterator constIterator=vector.cbegin();
constIterator!=vector.cend(); ++constIterator)
        std::cout << ' ' << *constIterator;
}

```



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása

main.cpp

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

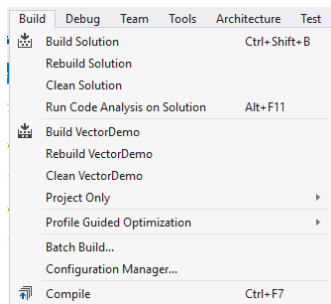
int main()
{
    VectorDemo *VectorDemoObject = new VectorDemo();

    std::cout << "MUVELET => VectorDemo() #konstruktor hivasa, vector tarolo
feltoltese..." << std::endl;
    std::cout << "ALLAPOT: "; VectorDemoObject->printContent(); std::cout <<
std::endl;
    std::cout << "MUVELET => empty() #uresseg vizsgalata..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << VectorDemoObject->isVectorEmpty()
<< std::endl;
    std::cout << "MUVELET => pop_back() #elemek kivetele..." << std::endl;
    VectorDemoObject->removeLastElementOfVector(); std::cout << std::endl;
    std::cout << "MUVELET => reverse() #kapacitas ujramerazetes..." <<
std::endl;
    VectorDemoObject->reserveVector(22);
    std::cout << "MUVELET => capacity() #kapacitas lekerdezese..." <<
std::endl;
    std::cout << "EREDMENY: "; std::cout << VectorDemoObject-
>getCapacityOfVector(); std::cout << std::endl;
    std::cout << "MUVELET => resize() #meret ujramerazetes..." << std::endl;
    VectorDemoObject->resizeVector(21);
    std::cout << "MUVELET => size() #meret lekerdezese..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << VectorDemoObject-
>getSizeOfVector(); std::cout << std::endl;
    std::cout << "MUVELET => at() #(ujra)feltoltes..." << std::endl;
    VectorDemoObject->recreateValuesWithAtToVector();
    std::cout << "ALLAPOT: "; VectorDemoObject->printContent(); std::cout <<
std::endl;
    std::cout << "MUVELET => max_size() #maximalis meret lekerdezese..." <<
std::endl;
    std::cout << "EREDMENY: "; std::cout << VectorDemoObject-
>getMaxSizeOfVector(); std::cout << std::endl;
    std::cout << "MUVELET => insert() #beszuras..." << std::endl;
    VectorDemoObject->insertElement(3, 7);
    std::cout << "ALLAPOT: "; VectorDemoObject->printContent(); std::cout <<
std::endl;
    std::cout << "MUVELET => shrink_to_fit() #kapacitas ujramerazetese a
tartalomhoz igazodva..." << std::endl;
    VectorDemoObject->shrinkToFit();
    std::cout << "MUVELET => capacity() #kapacitas lekerdezese..." <<
std::endl;
    std::cout << "EREDMENY: "; std::cout << VectorDemoObject-
>getCapacityOfVector(); std::cout << std::endl;
    std::cout << "MUVELET => erase() #also harom elem torlese..." <<
std::endl;
    VectorDemoObject->eraseElements(3);
```

```
std::cout << "MUVELET => emplace_back() #89-es elem hozzafuzese..." <<
std::endl;
VectorDemoObject->emplaceBack(89);
std::cout << "ALLAPOT: "; VectorDemoObject->printContent(); std::cout <<
std::endl;
std::cout << "MUVELET => assign() #(ujra)feltoltes..." << std::endl;
VectorDemoObject->assignValuesToVector();
std::cout << "ALLAPOT: "; VectorDemoObject->printContent(); std::cout <<
std::endl;

delete VectorDemoObject;
_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.



Program fordítása

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

```
Output
Show output from: Build
1>----- Rebuild All started: Project: VectorDemo, Configuration: Debug Win32 -----
1> VectorDemo.cpp
1> main.cpp
1> Generating Code...
1> VectorDemo.vcxproj -> D:\VectorDemo\Debug\VectorDemo.exe
***** Rebuild All: 1 succeeded, 0 failed, 0 skipped *****
```

Kimenet ablak

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

```
D:\VectorDemo\Debug\VectorDemo.exe
MUNLET -> VectorDemo() #konstruktor hívása, vector tartaló feltöltése...
ALLAPOT: 100 101 102 103 104 105 106 107 108 109
MUNLET -> empty() #üresesség vizsgálata...
EREDMENY: 0
MUNLET -> pop_back() #elemek kivétele...
v.back(): (100) v.back(): (100) v.back(): (107) v.back(): (106) v.back(): (105) v.back(): (104) v.back(): (103) v.back(): (102) v.back(): (101) v.back(): (100)
MUNLET -> reverse() #kapacitás újraméretezése...
MUNLET -> capacity() #kapacitás lekerdezes...
EREDMENY: 22
MUNLET -> resize() #meret újraméretezes...
MUNLET -> size() #meret lekerdezes...
EREDMENY: 21
MUNLET -> at() #újrafeltöltés...
ALLAPOT: 34 -21 13 -8 5 -3 2 -1 1 0 1 2 3 5 8 13 21 34 55 89
MUNLET -> max_size() #maximalis meret lekerdezes...
EREDMENY: 107343823
MUNLET -> insert() #beszúrás...
ALLAPOT: 7 7 7 34 -21 13 -8 5 -3 2 -1 1 0 1 2 3 5 8 13 21 34 55 89
MUNLET -> shrink_to_fit() #kapacitás újraméretezése a tartalomhoz igazodva...
MUNLET -> capacity() #kapacitás lekerdezes...
EREDMENY: 24
MUNLET -> erase() #első három elem törlése...
MUNLET -> emplace_back() #89-es elem hozzáfuzése...
ALLAPOT: 34 -21 13 -8 5 -3 2 -1 1 0 1 2 3 5 8 13 21 34 55 89 89
MUNLET -> assign() #újrafeltöltés...
ALLAPOT: 100 100 100 100 100 100 100 100 100 100
```

A program eredménye

A feladathoz kapcsolódó magyarázatok

Az osztály deklarációját a (**VectorDemo.h**) állományban adtuk meg. A deklarációban egy magán (**private**) jellegű, egész szám típusú (**integer**) adatok tárolására képes, dinamikus tömb került megfogalmazásra, amelyet az std névtéren keresztül értünk el:

```
private:
std::vector<int> vector;
```

A konstruktor neve tulajdonságaiból fakadóan megegyezik az osztály nevével, láthatósági szintje nyilvános (**public**), visszatérési értéke nincs, még void sem, továbbá az adott feladatban paraméter nélküli formában jelent meg:

```
public:  
    VectorDemo();
```

A `getVector()`, `getCapacityOfVector()` és a `getMaxSizeOfVector()` tagfüggvények a leírásban megfogalmazott vektor aktuális- és maximális méretének, valamint kapacitásának lekérdezését biztosítják. A láthatósági szintjük nyilvános (**public**), visszatérési típusuk egész szám (**integer**) és argumentumlistájuk üres:

```
int getVector();  
int getCapacityOfVector();  
int getMaxSizeOfVector();
```

Az `isEmpty()` tagfüggvény a vektor ürességének vizsgálatát hajtja végre, láthatósági szintje nyilvános (**public**), visszatérési típusa logikai (**bool**), argumentumlistája üres:

```
bool isEmpty();
```

A `resize(int size)` tagfüggvény a vektor aktuális méretének átméretezését végzi, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a vektor jövőben használni kívánt méretének értéke:

```
void resize(int size);
```

A `reserve(int value)` tagfüggvény a vektor aktuális kapacitásának átméretezését végzi, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a vektor jövőben használni kívánt kapacitásának értéke:

```
void reserve(int value);
```

A `shrinkToFit()` tagfüggvény a vektor aktuális kapacitásának átméretezését végzi a tartalomhoz legjobban illeszkedően, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void shrinkToFit();
```

A `recreateValuesWithAtToVector()` tagfüggvény feladata ellenőrzött indexeléssel érje el a vektor egyes elemeit, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void recreateValuesWithAtToVector();
```

Az `assignValuesToVector()` tagfüggvény iterátorokkal megadott tartományba helyez be elemeket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void assignValuesToVector();
```

A `removeLastElementOfVector()` tagfüggvény a vektor utolsó elemeit törli, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void removeLastElementOfVector();
```

Az `insertElement(int index, int value)` tagfüggvény egy megadott pozícióba n alkalommal beszúr egy értéket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint két egész szám típusú (**integer**) értéket vár paraméterül, ahol az első paraméter határozza meg a beszúrandó tartomány hosszát, a második argumentum a beszúrnivaló kívánt értéket:

```
void insertElement(int length, int value);
```

Az `eraseElements(int length)` tagfüggvény egy megadott pozíciótól n hosszan törli az értéket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a vektorból törölni kívánt tartomány hossza:

```
void eraseElements(int length);
```

Az `emplaceBack(int value)` tagfüggvény az utolsó utáni pozícióba új elemet fűz a vektorhoz, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték vektorhoz hozzáfűzni kívánt érték:

```
void emplaceBack(int value);
```

A `printContent()` tagfüggvény konstans iterátor segítségével megjeleníti a konzolon a vektor aktuális elemeit, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void printContent();
```

A **VectorDemo.cpp** állomány tartalmazza az osztály teljeskörű implementációját. Az implementáció során az egyes az egyes tagfüggvények neve előtt a `::` (hatókör) operátor került alkalmazásra, azon célból, hogy pontosan meghatározzuk melyik osztály tagfüggvényét valósítottuk meg.

A konstruktor feladata, hogy 100-tól kezdődően 109-ig feltöltse a vektort, amelyet a **push_back()** módosító függvény hívásával valósítottunk meg.

```
VectorDemo::VectorDemo()
{
    for(unsigned int i = 0; i < 100; ++i)
        vector.push_back(100 + i);
}
```

Az aktuális- és maximális méret, valamint a kapacitás lekérdezését végrehajtó függvények implementációja során a **vector** objektumon keresztül egyszerűen hívásra kerültek a **size()**, **max_size()**, illetve a **capacity()** metódusok.

```
int VectorDemo::getSizeOfVector()
{
    return vector.size();
}
int VectorDemo::getCapacityOfVector()
{
    return vector.capacity();
}
int VectorDemo::getMaxSizeOfVector()
{
    return vector.max_size();
}
```

Az üresség vizsgálata során a méret lekérdezéséhez hasonlóan a **vector** objektumon keresztül hívjuk meg a szükséges tagfüggvényt, amely ebben az esetben az **empty()** metódus.

```
bool VectorDemo::isVectorEmpty()
{
    return vector.empty();
}
```

A méret és kapacitás módosító függvények közül az újra méretezéshez a **resize()**-t használtuk, amely a paraméterben kapott nagyságú értékre állította a vektor aktuális méretét, a kapacitás specifikus értékkel történő módosításához a **reserve()** állt rendelkezésünkre, amíg a kapacitás nagyságának a vektor méretéhez történő igazításához a **shrink_to_fit()** függvényt alkalmaztuk.

```
void VectorDemo::resizeVector(int size)
{
    vector.resize(size);
}
void VectorDemo::reserveVector(int value)
{
    vector.reserve(value);
}
void VectorDemo::shrinkToFit()
{
    vector.shrink_to_fit();
}
```

A vektor elemeinek elérését több módon is végezhetjük, azonban ebben a példában, kizárólag az indexhatár ellenőrzést végző **at()** tagfüggvényt alkalmaztuk. Az **at()** alkalmazása során a „negafibonacci” sorozat 34, -21, 13, -8, 5, -3, 2, -1, 1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 és 89 elemeit adtuk értékül a vektorunknak, amely sorozatról tudjuk, hogy egy elem mindig az előző kettő összegeként szerepel, a negatív tartományban minden páros indexű elem negatív előjelű, például $F_{(-40)} = -21$.

```

void VectorDemo::recreateValuesWithAtToVector()
{
    int firstTermOfFibonacciSequence = 1, secondTermOfFibonacciSequence = 1;
    vector[getsizeofvector() / 2] = firstTermOfFibonacciSequence;
    vector[getsizeofvector() / 2 + 1] = secondTermOfFibonacciSequence;
    for (unsigned int i = getsizeofvector() / 2 + 2; i < getsizeofvector();
        ++i)
    {
        vector.at(i) = firstTermOfFibonacciSequence +
            secondTermOfFibonacciSequence;
        firstTermOfFibonacciSequence = secondTermOfFibonacciSequence;
        secondTermOfFibonacciSequence = vector.at(i);
    }
    int firstTermOfNegaFibonacciSequence = 0, secondTermOfNegaFibonacciSequence
        = 1;
    vector[getsizeofvector() / 2 - 1] = firstTermOfNegaFibonacciSequence;
    vector[getsizeofvector() / 2 - 2] = secondTermOfNegaFibonacciSequence;
    for (int j = getsizeofvector() / 2 - 3; j >= 0; --j)
    {
        vector.at(j) = firstTermOfNegaFibonacciSequence -
            secondTermOfNegaFibonacciSequence;
        firstTermOfNegaFibonacciSequence = secondTermOfNegaFibonacciSequence;
        secondTermOfNegaFibonacciSequence = vector.at(j);
    }
}

```

A vektor módosító függvények közül az **assign()** alkalmazása során a 100-as értéket N-szer helyeztük a vektorba, ahol N=7.

```

void VectorDemo::assignValuesToVector()
{
    vector.assign(7, 100);
}

```

A vektor utolsó elemének eltávolításához a **pop_back()** tagfüggvényt használtuk. Az elemek eltávolítása során az éppen eltávolított elemet a **back()** tagfüggvény segítségével kiírtuk a konzolra. A kivételt addig ismételtük, amíg volt elem a vektorban.

```

void VectorDemo::removeLastElementOfVector()
{
    while (vector.begin() != vector.end())
    {
        std::cout << "v.back(): "; std::cout << "(" << (vector.back()) << ") ";
        vector.pop_back();
    }
}

```

A vektorba történő beszúrást az **insert()** tagfüggvény segítségével végeztük el ahol az első paraméter a vektor elejét jelzi **begin()**, a második azt az index helyet, ameddig a beszúrást kívánjuk elvégezni, amíg a harmadik a beszúrandó értéket.

```
void VectorDemo::insertElement(int index, int value)
{
    std::vector<int>::iterator iterator = vector.begin();
    vector.insert(iterator, index, value);
}
```

Az **erase()** tagfüggvényt felhasználva a paraméterben megadott tartományból töröltük az elemeket, ahol a tartomány a vektor elejétől kezdődik és a függvény paraméterében átadott hosszra tart.

```
void VectorDemo::eraseElements(int length)
{
    vector.erase(vector.begin(), vector.begin()+length);
}
```

Az **emplace_back()** tagfüggvény segítségével az utolsó utáni pozícióba a 89-es elemet fűztük a vektorhoz.

```
void VectorDemo::emplaceBack(int value)
{
    vector.emplace_back(value);
}
```

A vektor elemek megjelenítéséhez konstans iterátort alkalmaztunk, mivel az elemeket kizárólag olvasni szeretnénk. A **cbegin()** tagfüggvény a vektor elejét (első elem), addig a **cend()** a vektor végét (utolsó utáni elem) reprezentálja egy konstans elérési bejáróval.

```
void VectorDemo::printContent()
{
    for (std::vector<int>::const_iterator constIterator=vector.cbegin();
         constIterator!=vector.cend(); ++constIterator)
        std::cout << ' ' << *constIterator;
}
```

A **(main.cpp)** állomány tartalmazza a **main()** függvényt, a program belépési pontját. A fő függvényben történő dinamikus példányosítása után hívjuk meg a szükséges függvényeket. A műveletek végrehajtását követően, gondoskodjunk

a lefoglalt erőforrások felszabadításáról az alapértelmezett explicit destruktorki hívásának segítségével.

std::deque

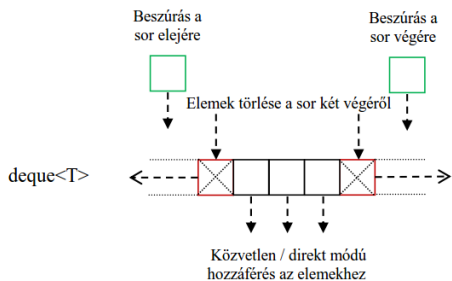
Egy olyan sorozat (**double-ended queue**, kétvégű sor), amely a kétszeresen láncolt lista és a vektor egyfajta hibridjeként tekinthető. A fejlesztők oly módon implementálták a tárolót, ahol a sorozat mindkét végén ugyanolyan hatékonyan végezhető el művelet, mint egy kétszeresen láncolt lista esetében, illetve az elemekhez való hozzáférés (indexelés) ugyanolyan hatékony, mint egy vektor esetében. A vektortól való különbség abból fakad, hogy nem csak a sor végén gyorsak a beszúrási és törlési műveletek, hanem az elején is. Tehát a tároló mindkét irányba tudja változtatni a méretét anélkül, hogy az összes elemet át kellene mozgatni. Azonban a kétvégű sor belsejében az elemek törlése és beszúrása ugyanolyan lassú, mint a vektornál. Ennek következtében a **deque** tárolót akkor használjuk, ha beszúrák és törlések általában csak a végeken fordulnak elő. A sor eleji beszúrás és törlés a **push_front()** és a **pop_front()** tagfüggvények valósítják meg. Az iterálási módszer teljesen megegyezik a vektoréval, tehát véletlenszerű iterálást tesz lehetővé. A vektorral szemben nem folytonos memóriaterületen tárolja az elemeket (nem garantálja, hogy az elemek pontosan egymás mellett vannak), így nem használható C tömbként, ebből fakadóan nem állíthatjuk a kapacitást. A **deque** felszabadítja ugyan a kihasználatlan memóriát, de nem tudjuk, hogy pontosan mikor, ez implementációfüggő. A kétvégű sor deklarálásának általános szintaktikája:

```
deque< ObjectType > DequeName;
```

Egy kétvégű sort a következő módon hozhatunk létre:

```
#include <deque>

int main()
{
    std::deque<int> DequeName{1, 2, 3};
}
```



A kétvégű tároló sematikus ábrája elem hozzáfűzése és törlése esetén

A kétvégű sor elérhető adatait, tagfüggvényei és iterátorai, illetve azok működés elvei csaknem teljesen megegyeznek a vektor tárolóknál olvasottakkal, ezért itt most nem kerülnek külön részletezésre, azok egy összefoglalását az alábbi táblázatban tekinthetjük át.

Elem hozzáférés	Általános, rövid leírás
at	Az elemek indexelt elérésének azon változata, ahol kivétel (out_of_range exception) keletkezik abban az esetben, ha az indexelés érvénytelen.
[] operátor	Az elemek indexelt elérésének azon változata, ahol nincs indexhatár ellenőrzés, akárcsak egy hagyományos tömb esetében.
front	Az első elem referencia értékét adja vissza vagy állítja be.
back	Az utolsó elem referencia értékét adja vissza vagy állítja be.
Kapacitás	Általános, rövid leírás
size	A tárolt elemek számát adja vissza.
max_size	A maximális méretet adja vissza.

resize	A kétvégű sor átméretezése.
empty	Üresség vizsgálata. Logikai igaz értékkel tér vissza, ha a kétvégű sor tároló üres, ellenkező esetben logikai hamis lesz az érték.
shrink_to_fit	A lefoglalt memória mennyiséget a valós méretéhez igazítja.

Módosítók	Általános, rövid leírás
assign	Új tartalom hozzárendelése.
push_front	Elem beszúrása az elején.
push_back	Elem hozzáadása a végéhez.
pop_front	Elem törlése az elejéről.
pop_back	Az utolsó elem törlése, így csökkentve a sor méretét.
erase	Elemek törlése.
swap	Tartalmat cserél, oly módon, hogy a bejárók érvényesek maradnak.
clear	Elemek törlése, minden elemre mutató bejáró érvénytelen lesz.
emplace	Specifikus pozícióba, új elemet helyez, így megnövelve a méretet eggyel.
emplace_front	Az első pozícióba, új elemet helyez, így megnövelve a méretet eggyel.
emplace_back	Az utolsó pozícióban, új elemet helyez, így megnövelve a méretet eggyel.

Bejárók	Általános, rövid leírás
begin	A kétvégű sor elejét (első elemét) reprezentálja egy tetszőleges elérésű bejáróval.
end	A kétvégű sor utolsó utáni helyét (utolsó utáni elem – <i>past-the-end</i>) reprezentálja egy tetszőleges elérésű bejáróval.

rbegin	A kétvégű sor végét (utolsó elem) reprezentálja tetszőleges elérésű, fordított bejáróval.
rend	A kétvégű sor első elem előtti helyét (első előtti elem) reprezentálja tetszőleges elérésű, fordított bejáróval.
cbegin	A kétvégű sor elejét (első elem) reprezentálja egy konstans bejáróval.
cend	A kétvégű sor utolsó utáni helyét (utolsó utáni elem – <i>past-the-end</i>) reprezentálja egy konstans bejáróval.
crbegin	A kétvégű sor végét (utolsó elem) reprezentálja egy konstans, fordított bejáróval.
crend	A kétvégű sor első elem előtti helyét (első előtti elem) reprezentálja egy konstans, fordított bejáróval.

A kétvégű sor objektum tagfüggvényei

Minta példa a kétvégű sor (deque) tároló használatára Visual Studio Community fejlesztőkörnyezetben

Írjunk egy DequeDemo (*DequeDemo*) osztályt, amely különböző, kétvégű sor specifikus műveleteket ír le, majd egy objektumpéldány segítségével ezeket a műveleteket hajtassuk végre. Minden egyes tagfüggvény feladatának megfogalmazását, zárójelben követi az adott tárolóhoz tartozó tagfüggvény, amelyeket kötelező jelleggel kell használni a feladat helyes megoldásának elérése érdekében, például a konstruktor esetében ilyen a **push_back()** módszer. Az osztály adataitja a következő:

- *deque*, egy kétvégű sor jellegű szekvenciális tároló, amely 10 db egész szám tárolására képes.

Ügyeljünk az egységbezárás elvére, így a fentebb megfogalmazott adattag(ok) **private**, az alábbi tagfüggvények **public** láthatósági szintekkel bírnak!

Valósítsuk meg az osztály metódusait:

- Írjuk meg az osztály konstruktorát, amely a ciklusváltozó szorzatával inicializálja a kétvégű sor tárolót. (**push_back()**)
- Írjunk egy *isEmpty()* függvényt, amely megvizsgálja, hogy a kétvégű sor tartalmaz-e elemeket, majd annak függvényében visszatér egy

logikai értékkel. Igazzal, ha található elem a kétvégű sorban, hamissal, ha üres. (**empty()**)

- Írjunk egy *clearDeque()* függvényt, amely törli a kétvégű sor elemeit. (**clear()**)
- Írjunk egy *getSize()* függvényt, amely a kétvégű sor aktuális méretét adja vissza. (**size()**)
- Írjunk egy *getMaxSizeOfDeque()* függvényt, amely a kétvégű sor maximális méretének értékével tér vissza. (**max_size()**)
- Írjunk egy *changeSize(int size)* függvényt, amely paraméterül vár egy egész szám típusú értéket, majd ezzel az értékkel átméretezi a sort. (**resize()**)
- Írjunk egy *shrinkToFit()* függvényt, amely a kétvégű sort újra méretezi a tartalomhoz igazodóan. (**shrink_to_fit()**)
- Írjunk egy *accessElementWithAt()* függvényt, amely a kétvégű sor elemeit új értékekkel látja el. Ezen értékek a tribonacci sorozat 0, 1, 1, 2, 4, 7, 13, 24, 44 és 81 elemei. (**at()**)
- Írjunk egy *getFront()* függvényt, amely a kétvégű sor első elemét adja vissza. (**front()**)
- Írjunk egy *getBack()* függvényt, amely a kétvégű sor utolsó elemét adja vissza. (**back()**)
- Írjunk egy *assignValues()* függvényt, amely a kétvégű sort újra feltölti 3 hosszan 100-as értékekkel (**assign()**)
- Írjunk egy *addElementAtTheEnd(int value)* függvényt, amely a paraméterül kapott értéket a kétvégű sor végéhez fűzi. (**push_back()**)
- Írjunk egy *addElementAtBeginning (int value)* függvényt, amely a paraméterül kapott értéket a kétvégű sor elejéhez fűzi. (**push_front()**)
- Írjunk egy *deleteFirstElement()* függvényt, amely a kétvégű sor elejéről töröli az első elemet. (**pop_front()**)
- Írjunk egy *deleteLastElement()* függvényt, amely a kétvégű sor végétől töröli az utolsó elemet. (**pop_back()**)
- Írjunk egy *insertElements(int index, int value)* függvényt, amely a kétvégű sor elejétől, a függvény paramétereként kapott *index* határig beszúrja a paraméterként kapott *value* értéket. (**insert()**)
- Írjunk egy *eraseElements(int length)* függvényt, amely a kétvégű sor elejétől, (**begin**) a függvény paramétereként megkapott *index* határig, töröli az értékeket. (**erase()**)
- Írjunk egy *emplaceFront(int value)* függvényt, amely a kétvégű sor elejéhez hozzáfűzi a paraméterben kapott számértéket. (**emplace_front()**)

- Írjunk egy `emplaceBack(int value)` függvényt, amely a kétvégű sor végéhez hozzáfűzi a paraméterben kapott számértéket. (**`emplace_back()`**)
- Írjunk egy `printElements()` függvényt, amely kiírja egy konstans iterátor felhasználásával a kétvégű sor minden egyes elemét, a **`cbegin()`** és a **`cend()`** metódusok felhasználásával.
- Ne feledkezzünk meg az objektumpéldány által lefoglalt erőforrás felszabadításáról, a destruktor megírásáról sem.

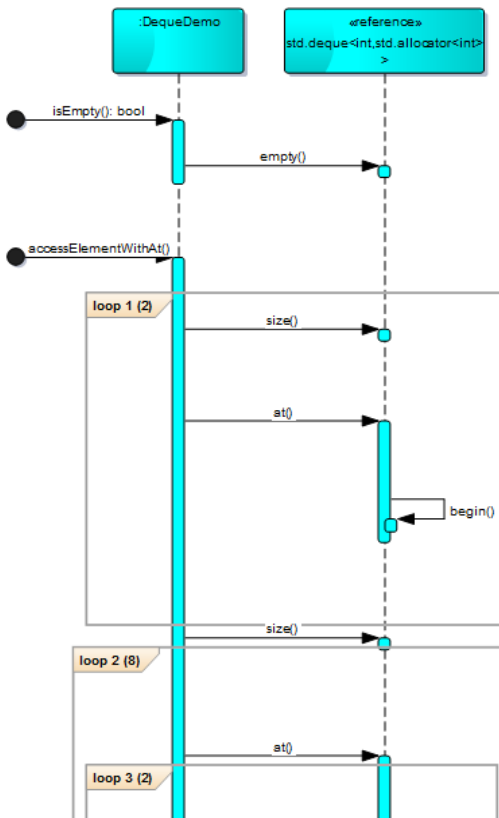
Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén **[heap]**) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektum példányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. Figyelem az **`auto`** kulcsszóval a fordítóra is bízhatjuk az iterátorok típusának felismerését! Mielőtt a feladatnak neki látnánk, tervezzük meg az osztály- és szekvencia diagramot. **Figyelem! A könyvben ábrázolt szekvenciadiagramok terjedelmi korlátok miatt a teljesség igénye nélkül készültek!**

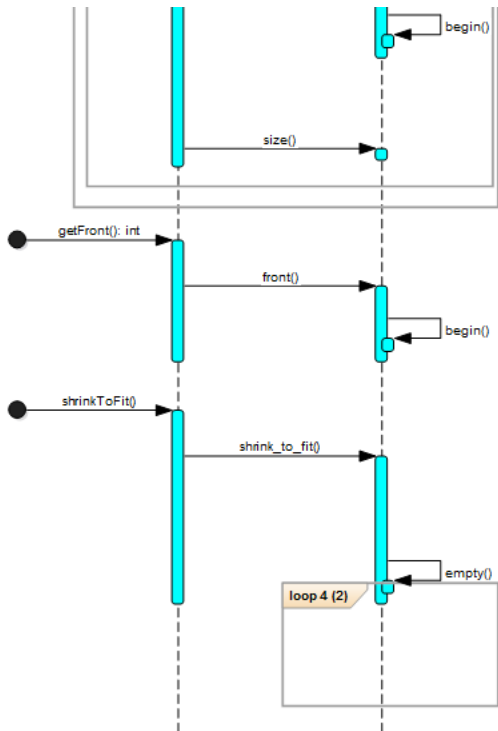
Statikus tervezés

DequeDemo
- deque: std::deque<int>
+ accessElementWithAt(): void
+ addElementAtBeginning(int): void
+ addElementAtTheEnd(int): void
+ assignValues(): void
+ changeSize(int): void
+ clearDeque(): void
+ deleteFirstElement(): void
+ deleteLastElement(): void
+ DequeDemo()
+ ~DequeDemo()
+ emplaceBack(int): void
+ emplaceFront(int): void
+ eraseElements(int): void
+ getBack(): int
+ getFront(): int
+ getMaxSize(): int
+ getSize(): int
+ insertElements(int, int): void
+ isEmpty(): bool
+ printElements(): void
+ shrinkToFit(): void

Osztálydiagram

Dinamikus diagram





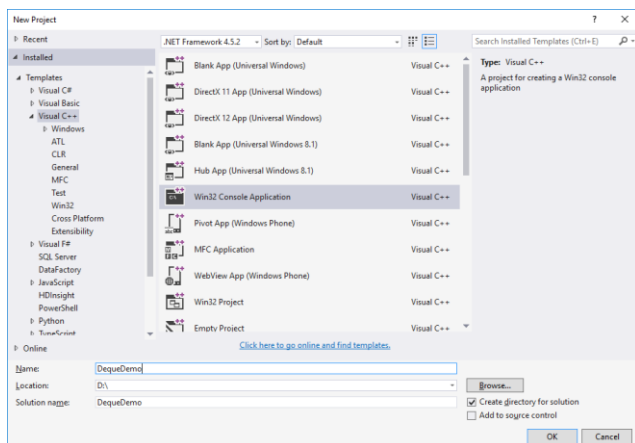
Néhány tagfüggvény szekvenciadiagramja

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

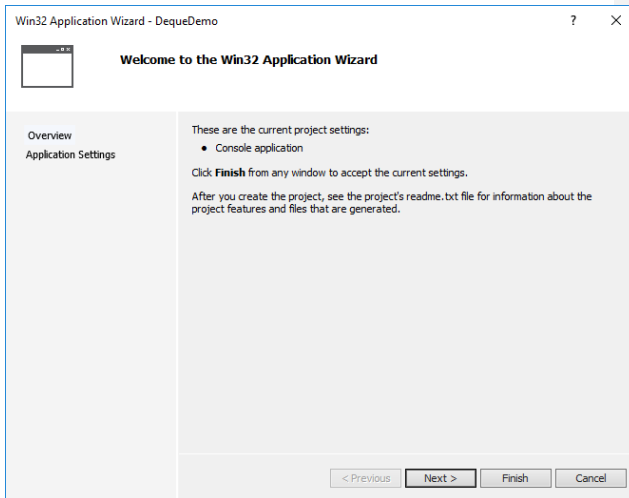
megfogalmazott minta példa kerül implementálásra és összeállításra, továbbá Windows operációs rendszeren futtatható kóddá történő lefordításra Visual Studio Community fejlesztőkörnyezet felhasználásával. A lefordított 32 bites alkalmazás *konzolban* kerül futtatásra.

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 közül a **Visual C++ Win32** sablont (**Template**) kell kiválasztani és a **Win32 Console Application**-t, mely az alábbi ábrán látható.



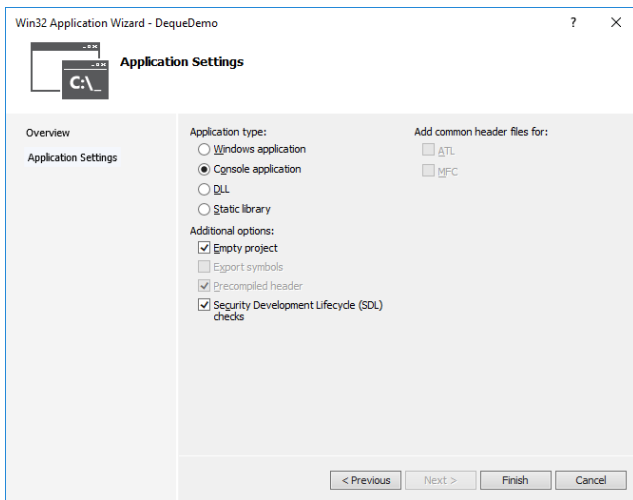
Win32 Console Application projekt létrehozása

Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően a **DequeDemo** név került megadásra. Az OK gombra kattintást követően elindul az alkalmazás varázsló, mely a következő ábrán látható.



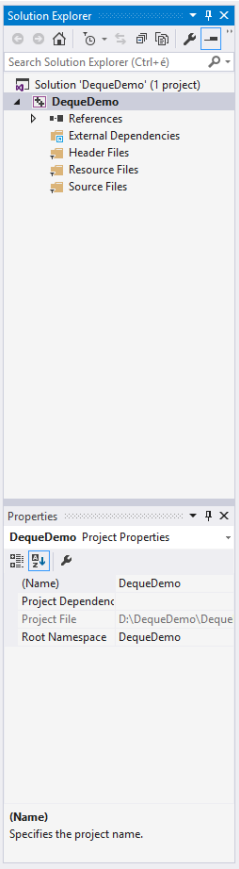
Win32 konzol alkalmazás varázsló

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 látható.



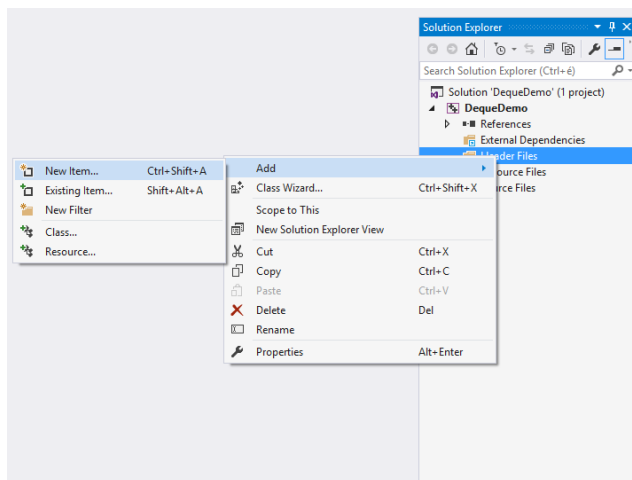
Alkalmazás beállításai

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



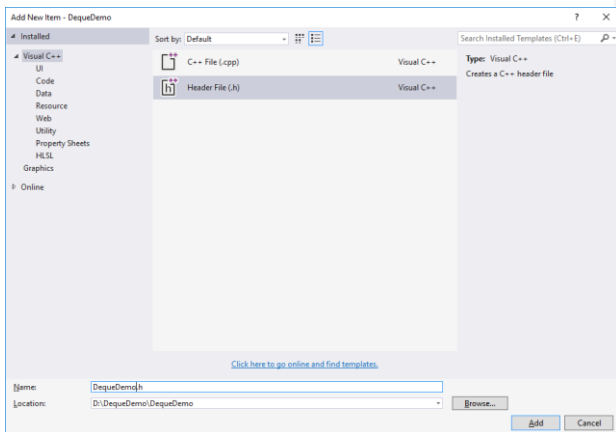
Megoldás és beállítások ablak

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.



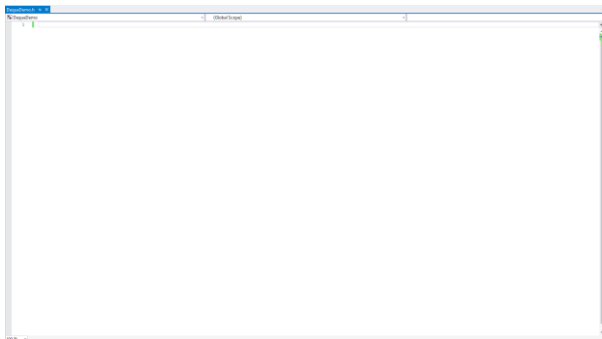
Új header fájl hozzáadása a projekthez

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



Új header fájl hozzáadása a projekthez, név megadása

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



Forráskód szerkesztő

DequeDemo.h

```
#ifndef DEQUEDEMO_H
#define DEQUEDEMO_H

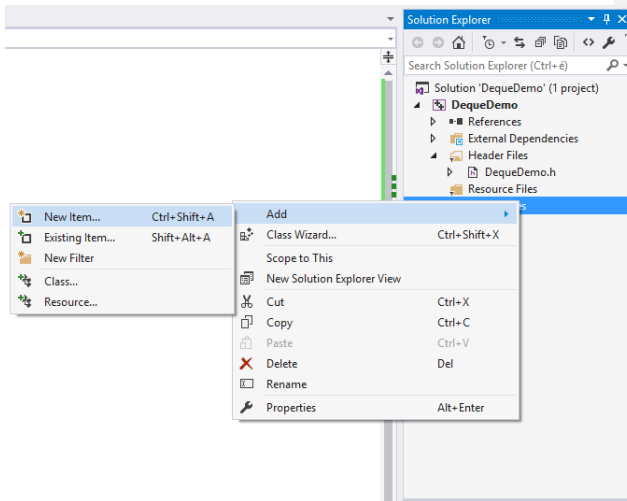
#include <deque>

class DequeDemo
{
private:
    std::deque<int> deque;

public:
    DequeDemo();
    int getSize();
    void clearDeque();
    int getMaxSize();
    void changeSize(int size);
    bool isEmpty();
    void shrinkToFit();
    void accessElementWithAt();
    int getFront();
    int getBack();
    void assignValues();
    void addElementAtTheEnd(int element);
    void addElementAtBeginning(int element);
    void deleteLastElement();
    void deleteFirstElement();
    void insertElements(int index, int value);
    void eraseElements(int length);
    void replaceFront(int value);
    void replaceBack(int value);
    void printElements();
    ~DequeDemo();
};

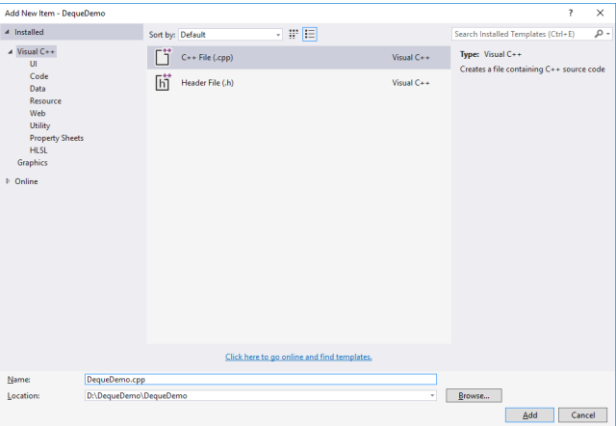
#endif
```

A deklarált osztályok megvalósításait a **DequeDemo.cpp** forrásfájl tartalmazza, melyeket a **header** fájlokhoz hasonlóan, de forrásfájl (**Source File**) hozzáadásával hozhatunk létre.



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása

DequeDemo.cpp

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

DequeDemo::DequeDemo()
{
    for (unsigned int i = 0; i < 10; ++i)
        deque.push_back(i*i);
}

int DequeDemo::getSize()
{
    return deque.size();
}

void DequeDemo::clearDeque()
{
    return deque.clear();
}

int DequeDemo::getMaxSize()
{
    return deque.max_size();
}

void DequeDemo::changeSize(int size)
{
    deque.resize(size);
}

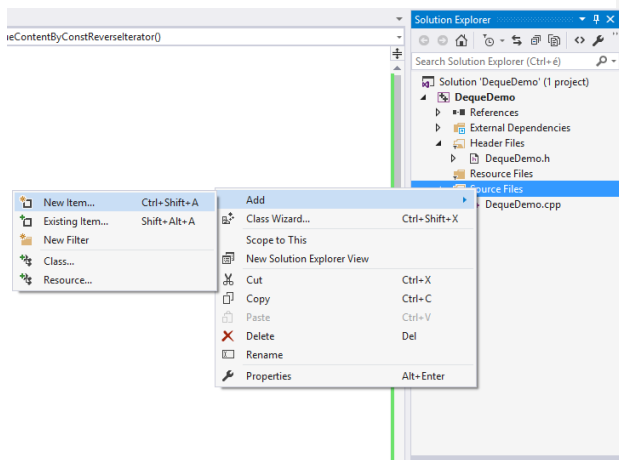
bool DequeDemo::isEmpty()
{
    return deque.empty();
}

void DequeDemo::shrinkToFit()
{
    deque.shrink_to_fit();
}

void DequeDemo::accessElementWithAt()
{
    int firstTermOfTribonacciSequence = 0, secondTermOfTribonacciSequence =
    0, thirdTermOfTribonacciSequence = 1;
    for (unsigned int i = 0; i < deque.size(); ++i)
    {
        if (i <= 1)
        {
            deque.at(i) = i;
        }
        else
        {
            deque.at(i) = firstTermOfTribonacciSequence +
            secondTermOfTribonacciSequence + thirdTermOfTribonacciSequence;
            firstTermOfTribonacciSequence = secondTermOfTribonacciSequence;
            secondTermOfTribonacciSequence = thirdTermOfTribonacciSequence;
            thirdTermOfTribonacciSequence = deque.at(i);
        }
    }
}
```

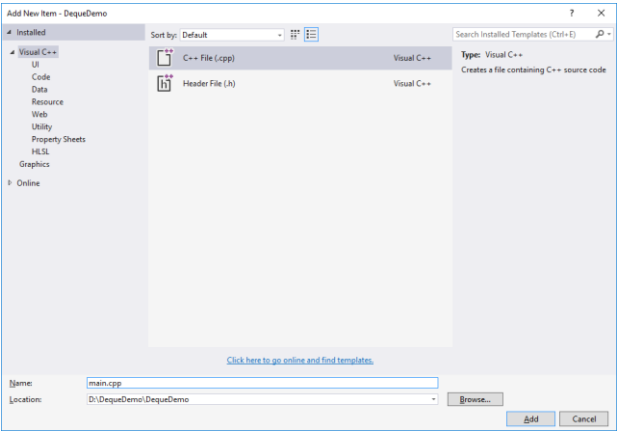
```
int DequeDemo::getFront()
{
    return deque.front();
}
int DequeDemo::getBack()
{
    return deque.back();
}
void DequeDemo::assignValues()
{
    deque.assign(3, 100);
}
void DequeDemo::addElementAtTheEnd(int value)
{
    deque.push_back(value);
}
void DequeDemo::addElementAtBeginning(int value)
{
    deque.push_front(value);
}
void DequeDemo::deleteFirstElement()
{
    deque.pop_front();
}
void DequeDemo::deleteLastElement()
{
    deque.pop_back();
}
void DequeDemo::insertElements(int index, int value)
{
    std::deque<int>::iterator iterator = deque.begin();
    deque.insert(iterator, index, value);
}
void DequeDemo::eraseElements(int length)
{
    deque.erase(deque.begin(), deque.begin() + length);
}
void DequeDemo::emplaceFront(int value)
{
    deque.emplace_front(value);
}
void DequeDemo::emplaceBack(int value)
{
    deque.emplace_back(value);
}
void DequeDemo::printElements()
{
    for (auto constIterator = deque.cbegin(); constIterator != deque.cend();
        ++constIterator)
    {
        std::cout << ' ' << *constIterator;
    }
}
```

```
DequeDemo::~DequeDemo() {}
```



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása

main.cpp

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

int main()
{
    DequeDemo *DequeDemoObject = new DequeDemo();

    std::cout << "MUVELET => DequeDemo() #konstruktor hivasa, deque tarolo
feltoltese..." << std::endl;
    std::cout << "ALLAPOT: "; DequeDemoObject->printElements(); std::cout
<< std::endl;
    std::cout << "MUVELET => empty() #uresseg vizsgalat..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << DequeDemoObject->isEmpty() <<
std::endl;
    std::cout << "MUVELET => at() #(ujra)feltoltes..." << std::endl;
    DequeDemoObject->accessElementWithAt();
    std::cout << "ALLAPOT: "; DequeDemoObject->printElements(); std::cout
<< std::endl;
    std::cout << "MUVELET => pop_back() #utolso elem torles..." <<
std::endl;
    DequeDemoObject->deleteLastElement();
    std::cout << "MUVELET => pop_front() #elso elem torles..." <<
std::endl;
    DequeDemoObject->deleteFirstElement();
    std::cout << "ALLAPOT: "; DequeDemoObject->printElements(); std::cout
<< std::endl;
    std::cout << "MUVELET => size() #tarolo meret lekerdezes..." <<
std::endl;
    std::cout << "EREDMENY: "; std::cout << DequeDemoObject->getSize();
std::cout << std::endl;
    std::cout << "MUVELET => max_size() #tarolo maximalis meretenek
lekerdezes..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << DequeDemoObject->getMaxSize();
std::cout << std::endl;
    std::cout << "MUVELET => insert() #beszuras..." << std::endl;
    DequeDemoObject->insertElements(3, 7);
    std::cout << "ALLAPOT: "; DequeDemoObject->printElements(); std::cout
<< std::endl;
    std::cout << "MUVELET => erase() #torles..." << std::endl;
    DequeDemoObject->eraseElements(3);
    std::cout << "ALLAPOT: "; DequeDemoObject->printElements(); std::cout
<< std::endl;
    std::cout << ">> MUVELET => emplace_front() #a tarolo elejehez torteno
hozzafuzes..." << std::endl;
    DequeDemoObject->emplaceFront(0);
    std::cout << ">> MUVELET => emplace_back() #a tarolo vegehez torteno
hozzafuzes..." << std::endl;
    DequeDemoObject->emplaceBack(81);
    std::cout << "ALLAPOT: "; DequeDemoObject->printElements(); std::cout
<< std::endl;
    std::cout << "MUVELET => assign() #(ujra)feltoltes..." << std::endl;
```

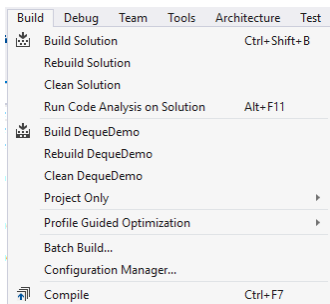
```

DequeDemoObject->assignValues();
std::cout << "ALLAPOT: "; DequeDemoObject->printElements(); std::cout
<< std::endl;
std::cout << ">> MUVELET => push_back() #tarolo vegehez torteno
hozzafuzes..." << std::endl;
DequeDemoObject->addElementAtTheEnd(101);
std::cout << ">> MUVELET => push_front() #tarolo elejehez torteno
hozzafuzes..." << std::endl;
DequeDemoObject->addElementAtBeginning(99);
std::cout << "ALLAPOT: "; DequeDemoObject->printElements(); std::cout
<< std::endl;
std::cout << ">> MUVELET => end() #tarolo utolso elemenek
lekerdezese..." << std::endl;
std::cout << "EREDMENY: "; std::cout << DequeDemoObject->getBack();
std::cout << std::endl;
std::cout << ">> MUVELET => front() #tarolo elso elemenek
lekerdezese..." << std::endl;
std::cout << "EREDMENY: "; std::cout << DequeDemoObject->getFront();
std::cout << std::endl;
std::cout << "MUVELET => resize() #tarolo ujrameretezese..." <<
std::endl;
DequeDemoObject->changeSize(50);
std::cout << "MUVELET => size() #tarolo meret lekerdezese..." <<
std::endl;
std::cout << "EREDMENY: "; std::cout << DequeDemoObject->getSize();
std::cout << std::endl;
std::cout << "MUVELET => clear() #tarolo uritese..." << std::endl;
DequeDemoObject->clearDeque();
std::cout << "MUVELET => size() #tarolo meret
lekerdezese..." << std::endl;
std::cout << "EREDMENY: "; std::cout << DequeDemoObject->getSize();
std::cout << std::endl;
std::cout << "MUVELET => shrink_to_fit() #ujrameretezes a tartalomhoz
igazodva..." << std::endl;
DequeDemoObject->shrinkToFit();
std::cout << "MUVELET => size() #tarolo meret lekerdezese..." <<
std::endl;
std::cout << "EREDMENY: "; std::cout << DequeDemoObject->getSize();
std::cout << std::endl;

delete DequeDemoObject;
_getch();
return 0;
}

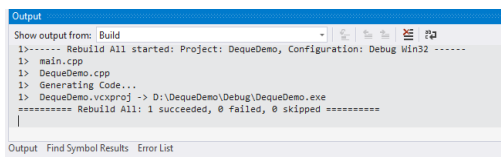
```

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.



Program fordítása

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



Kimenet ablak

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

```
D:\DequeDemo\Debug\DequeDemo.exe
MUNELET -> DequeDemo() #konstruktor hívása, deque tároló feltöltése...
ALLAPOT: 0 1 4 9 16 25 36 49 64 81
MUNELET -> empty() üresség vizsgálat...
EREDMENY: 0
MUNELET -> at() #újra)feltöltés...
ALLAPOT: 0 1 1 2 4 7 13 24 44 81
MUNELET -> pop_back() #utolsó elem törölés...
ALLAPOT: 1 1 2 4 7 13 24 44
MUNELET -> size() #tároló méret lekérdezése...
EREDMENY: 8
MUNELET -> max_size() #tároló maximális méretének lekérdezése...
EREDMENY: 1673741823
MUNELET -> insert() #beszúrás...
ALLAPOT: 7 7 7 1 1 2 4 7 13 24 44
MUNELET -> erase() #törölés...
ALLAPOT: 1 1 2 4 7 13 24 44
MUNELET -> emplace_front() #a tároló elejéhez történő hozzáfűzés...
MUNELET -> emplace_back() #a tároló végéhez történő hozzáfűzés...
ALLAPOT: 0 1 1 2 4 7 13 24 44 81
MUNELET -> assign() #(újra)feltöltés...
ALLAPOT: 100 100 100
MUNELET -> push_back() #tároló végéhez történő hozzáfűzés...
MUNELET -> push_front() #tároló elejéhez történő hozzáfűzés...
ALLAPOT: 99 100 100 100 101
MUNELET -> end() #tároló utolsó elemének lekérdezése...
EREDMENY: 101
MUNELET -> front() #tároló első elemének lekérdezése...
EREDMENY: 99
MUNELET -> resize() #tároló újraméretezése...
MUNELET -> size() #tároló méret lekérdezése...
EREDMENY: 50
MUNELET -> clear() #tároló ürítése...
MUNELET -> size() #tároló méret lekérdezése...
EREDMENY: 0
MUNELET -> shrink_to_fit() #újraméretezés a tartalomhoz igazodva...
MUNELET -> size() #tároló méret lekérdezése...
EREDMENY: 0
```

A program eredménye

A feladathoz kapcsolódó magyarázatok

Az osztály deklarációját a (**DequeDemo.h**) állományban adjuk meg. A deklarációban egy magán (**private**) jellegű, egész szám típusú (**integer**) adatok tárolására képes, kétvégű sor került megfogalmazásra, amelyet az **std** névtéren keresztül értünk el:

```
private:
    std::deque<int> deque;
```

A konstruktor neve tulajdonságaiból fakadóan megegyezik az osztály nevével, láthatósági szintje nyilvános (**public**), visszatérési értéke nincs, még **void** sem, továbbá az adott feladatban paraméter nélküli formában jelent meg:

```
public:
    DequeDemo();
```

A **getSize()**, **getMaxSize()**, **getFront()** és **getBack()** tagfüggvények a leírásban megfogalmazott kétvégű sor aktuális- és maximális méretének, valamint az első és utolsó elemének lekérdezését biztosítják. A láthatósági szintjük nyilvános (**public**), visszatérési típusuk egész szám (**integer**) és argumentumlistájuk üres:

```
int getSize();
int getMaxSize();
int getFront();
int getBack();
```

Az *isEmpty()* tagfüggvény a kétvégű sor ürességének vizsgálatát hajtja végre, láthatósági szintje nyilvános (**public**), visszatérési típusa logikai (**bool**), argumentumlistája üres:

```
bool isEmpty();
```

A *changeVector(int size)* tagfüggvény a kétvégű sor aktuális méretének átméretezését végzi, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a kétvégű sor jövőben használni kívánt méretének értéke:

```
void changeSize(int size);
```

A *shrinkToFit()* tagfüggvény a kétvégű sor aktuális méretének átméretezését végzi a tartalomhoz legjobban illeszkedően, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void shrinkToFit();
```

Az *accessElementWithAt()* tagfüggvény feladata ellenőrzött indexeléssel érje el a kétvégű sor egyes elemeit, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void accessElementWithAt();
```

Az *assignValues()* tagfüggvény iterátorokkal megadott tartományba helyez be elemeket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void assignValues();
```

Az *addElementAtTheEnd(int element)* tagfüggvény az utolsó pozícióba a paraméterül kapott elemet fűzi a kétvégű sorhoz, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely a beszúrni kívánt érték:

```
void addElementAtTheEnd(int element);
```

Az `addElementAtBeginning(int element)` tagfüggvény az első pozícióba a paraméterül kapott elemet fűzi a kétvégű sorhoz, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely a beszúrni kívánt érték:

```
void addElementAtBeginning(int element);
```

A `deleteLastElement()` tagfüggvény a kétvégű sor utolsó elemét törli, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void deleteLastElement();
```

A `deleteFirstElement()` tagfüggvény a kétvégű sor első elemét törli, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void deleteFirstElement();
```

Az `insertElements(int index, int value)` tagfüggvény egy megadott pozícióba n alkalommal beszúr egy értéket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint két egész szám típusú (**integer**) értéket vár paraméterül, ahol az első paraméter határozza meg a beszúrandó tartomány hosszát, a második argumentum a beszúrni kívánt értéket:

```
void insertElements(int index, int value);
```

Az `eraseElements(int length)` tagfüggvény egy megadott pozíciótól n hosszan törli az értéket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a kétvégű sorból törölni kívánt tartomány hossza:

```
void eraseElements(int length);
```

Az `emplaceFront(int value)` tagfüggvény az első pozícióba új elemet fűz a kétvégű sorhoz, visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a kétvégű sorhoz hozzáfűzni kívánt érték:

```
void emplaceFront(int value);
```

Az `emplaceBack(int value)` tagfüggvény az utolsó utáni pozícióba új elemet fűz a kétvégű sorhoz, visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a kétvégű sorhoz hozzáfűzni kívánt érték:

```
void emplaceBack(int value);
```

A `printElements()` tagfüggvény automatikusan kiválasztott típusú iterátor segítségével megjeleníti a konzolon a kétvégű sor aktuális elemeit, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void printElements();
```

A **DequeDemo.cpp** állomány tartalmazza az osztály teljeskörű implementációját. Az implementáció során az egyes tagfüggvények neve előtt a `::` (hatókör) operátor került alkalmazásra. azon célból, hogy pontosan meghatározzuk melyik osztály tagfüggvényét valósítottuk meg.

A konstruktor feladata, hogy a ciklusváltozó szorzatával feltöltse a kétvégű sort, amelyet a **push_back()** módosító függvény hívásával valósítottunk meg.

```
DequeDemo::DequeDemo()
{
    for (unsigned int i = 0; i < 10; ++i)
        deque.push_back(i*i);
}
```

Az aktuális- és maximális méret, valamint az első és utolsó elem lekérdezését végrehajtó függvények implementációja során a **deque** objektumon keresztül egyszerűen hívásra kerültek a **size()**, **max_size()**, illetve a **front()** és **back()** metódusok.

```
int DequeDemo::getSize()
{
    return deque.size();
}
int DequeDemo::getMaxSize()
{
    return deque.max_size();
}
int DequeDemo::getFront()
{
    return deque.front();
}
```

```
int DequeDemo::getBack()
{
    return deque.back();
}
```

Az üresség vizsgálata során a méret lekérdezéséhez hasonlóan a **deque** objektumon keresztül hívjuk meg a szükséges tagfüggvényt, amely ebben az esetben az **empty()** metódus.

```
bool DequeDemo::isEmpty()
{
    return deque.empty();
}
```

A kétvégű sor elemeinek törlését, a **clear()** tagfüggvény hívásával hajthatjuk végre. *Figyelem! A **clear()** tagfüggvény hívását követően a kétvégű sor ugyan nem tartalmaz már elemeket, de az elemek memória allokációja továbbra is megmarad, így a már nem használt memória felszabadítása érdekében ajánlott meghívni az alább is használt **shrink_to_fit()** tagfüggvényt!*

```
void DequeDemo::clearDeque()
{
    return deque.clear();
}
```

A méret módosító függvények közül az újra méretezéshez a **resize()**-t használtuk, amely a paraméterben kapott nagyságú értékre állította a kétvégű sor aktuális méretét, amíg a méret nagyságának a kétvégű sor méretéhez történő igazításhoz a **shrink_to_fit()** függvényt alkalmaztuk.

```
void DequeDemo::changeSize(int size)
{
    deque.resize(size);
}
void DequeDemo::shrinkToFit()
{
    deque.shrink_to_fit();
}
```

A kétvégű sor elemeinek elérését több módon is végezhetjük, azonban ebben a példában, kizárólag az indexhatár ellenőrzést végző **at()** tagfüggvényt alkalmaztuk. Az **at()** alkalmazása során a tribonacci sorozat 0, 1, 1, 2, 4, 7, 13, 24, 44 és 81 elemeit adtuk értékül a kétvégű sornak.

```

void DequeDemo::accessElementWithAt()
{
    int firstTermOfTribonacciSequence = 0, secondTermOfTribonacciSequence = 0,
    thirdTermOfTribonacciSequence = 1;
    for (unsigned int i = 0; i < deque.size(); ++i)
    {
        if (i <= 1)
        {
            deque.at(i) = i;
        }
        else
        {
            deque.at(i) = firstTermOfTribonacciSequence +
            secondTermOfTribonacciSequence + thirdTermOfTribonacciSequence;
            firstTermOfTribonacciSequence = secondTermOfTribonacciSequence;
            secondTermOfTribonacciSequence = thirdTermOfTribonacciSequence;
            thirdTermOfTribonacciSequence = deque.at(i);
        }
    }
}

```

A kétvégű sor módosító függvények közül az **assign()** alkalmazása során a 100-as értéket N-szer helyeztük a kétvégű sorba, ahol N=3.

```

void DequeDemo::assignValues()
{
    deque.assign(3, 100);
}

```

A kétvégű sor első elemének eltávolításához a **pop_front()**, amíg az utolsó elemének eltávolításához a **pop_back()** tagfüggvényt használtuk.

```

void DequeDemo::deleteFirstElement()
{
    deque.pop_front();
}
void DequeDemo::deleteLastElement()
{
    deque.pop_back();
}

```

A kétvégű sorba történő beszúrást az **insert()** tagfüggvény segítségével végeztük el ahol az első paraméter a kétvégű sor elejét jelzi (**begin()**), a második azt az index helyet, ameddig a beszúrást kívánjuk elvégezni, amíg a harmadik a beszúrandó értéket.

```

void DequeDemo::insertElements(int index, int value)
{
    std::deque<int>::iterator iterator = deque.begin();
    deque.insert(iterator, index, value);
}

```

A kétvégű sor első pozíciójába új elemet szúrhatunk be a **push_front()** tagfüggvény segítségével, amíg az utolsó pozíciójába új elemet a **push_back()** segítségével helyezhetünk.

```

void DequeDemo::addElementAtTheEnd(int value)
{
    deque.push_back(value);
}
void DequeDemo::addElementAtBeginning(int value)
{
    deque.push_front(value);
}

```

Az **erase()** tagfüggvényt használva a paraméterben megadott tartományból töröltük az elemeket, ahol a tartomány a kétvégű sor elejétől kezdődik és a függvény paraméterében átadott hosszig tart.

```

void DequeDemo::eraseElements(int length)
{
    deque.erase(deque.begin(), deque.begin() + length);
}

```

Az **emplace_front()** segítségével az első pozícióba a 99-es elemet fűztük a kétvégű sorhoz, amíg az **emplace_back()** segítségével az utolsó utáni pozícióba a 101-es elemet fűztük a kétvégű sorhoz.

```

void DequeDemo::emplaceFront(int value)
{
    deque.emplace_front(value);
}
void DequeDemo::emplaceBack(int value)
{
    deque.emplace_back(value);
}

```

A kétvégű sor elemek megjelenítéséhez automatikusan kiválasztott iterátor típust alkalmaztunk. A **cbegin()** tagfüggvény a kétvégű sor elejét (első elem), addig a **cend()** a kétvégű sor végét (utolsó utáni elem) reprezentálja egy konstans elérésű bejáróval.

```

void DequeDemo::printElements()
{
    for (auto constIterator = deque.cbegin(); constIterator != deque.cend();
        ++constIterator)
    {
        std::cout << ' ' << *constIterator;
    }
}

```

A **(main.cpp)** állomány tartalmazza a **main()** függvényt, a program belépési pontját. A fő függvényben történő dinamikus példányosítása után hívjuk meg a szükséges függvényeket. A műveletek végrehajtását követően, gondoskodjunk a lefoglalt erőforrások felszabadításáról az alapértelmezett implicit destruktor hívásának segítségével.

std::list

A kétszeresen vagy más néven kétirányú láncolt lista egy olyan szekvenciális tároló, amely nem csak a tároló két végén hajtja végre hatékonyan az elemek beszúrását és törlését (művelet költsége konstans). Ennek oka, hogy az elemek nem egy folytonos memória területen helyezkednek el, ellentétben például a dinamikus tömbbel, így az elemeket nem kell odébb csúsztatni a memóriában vagy egy újabb folytonos területet foglalni, hanem kizárólag a mutatókat kell átállítani. Ebből következőleg egy lista járulékos memóriaigénye két mutató, ahol az egyik a lista előző a másik a következő elemére mutat. Ezeknek a következménye az is, hogy nem különböztetünk meg kapacitást, illetve méretet. Mivel az ilyen típusú tárolóra nem értelmezett az indexelés-operátor, ezért az STL megalkotóinak olyan koncepciót kellett kitalálniuk, amely mind a dinamikus tömbre, mind a listákra, mind egyéb tárolókra alkalmazható. ez a közös koncepció az iterátor. A véletlenelem-hozzáférés itt nem támogatott, továbbá az általános algoritmusokhoz képest a keresés sem gyorsítható, ezért nincs is erre külön metódus. A kétszeresen láncolt lista deklarálásának általános szintaktikája:

```
list< ObjectType > ListName;
```

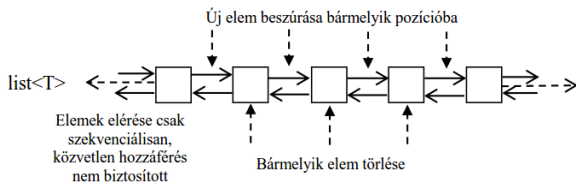
Egy kétszeresen láncolt listát a következő módon hozhatunk létre:

```

#include <list>

int main()
{
    std::list<int> ListName{1, 2, 3};
}

```



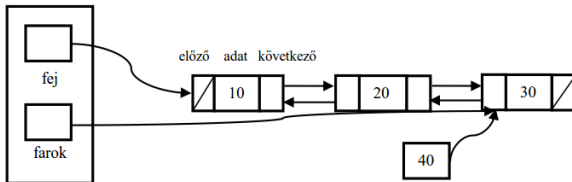
A List tároló sematikus ábrája elem hozzáfűzése és törlése esetén

Az **insert()** tagfüggvény segítségével a kétszeresen láncolt lista egy specifikus pozíciójába szúrhatunk be egy elemet. A függvény három típusát különböztethetjük meg:

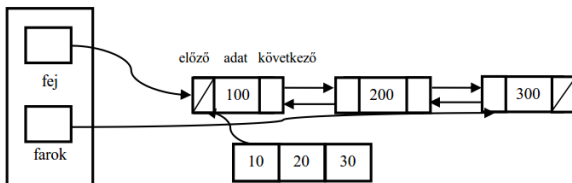
- **insert(iterator, element):** A listába egy értéket (**element**) szúr be a megadott iterátor (**iterator**) elé.
- **insert(iterator, count, element):** A listába egy értéket (**element**) szúr be a megadott iterátor (**iterator**) elé, n darabszor (**count**).
- **insert(iterator, start_iterator, end_iterator):** A paraméterben megadott tartományon tárolt elemeket **start_iterator**-tól és **end_iterator-1**-ig szúrja be a listába.

```
std::list<int> NameOfList1{10, 20, 30};
list<int>::iterator it = NameOfList1.begin();
NameOfList1.insert(it+2, 40);
std::list<int> NameOfList2{100, 200, 300};
NameOfList2.insert(NameOfList2.begin(), NameOfList1.begin(),
NameOfList1.end());
```

NameOfList1

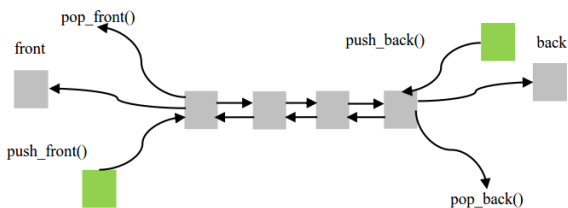


NameOfList2



A kétszeresen láncolt lista beszúrás műveletének ábrája

A kétszeresen láncolt lista esetében a **front** az első lista elemre mutató referenciát, amíg a **back** a lista végéhez tartozó referenciát adja vissza. Amennyiben a lista üres, úgy a fentebbi két referencia érvénytelen lesz. A lista végéhez történő elem beszúrása a **push_back()** metódus segítségével hajtható végre, ahol paraméterben megadhatjuk a tárolóhoz fűzendő értéket. A lista elejéhez történő beszúráshoz használhatjuk a **push_front()** tagfüggvényt, amíg az onnan való törléshez a **pop_front()** tagfüggvényt. A lista végétől történő elem törléséhez a **pop_back()** tagmetódus használható:



A kétszeresen láncolt lista néhány műveletének ábrája

A **splice()** az úgynevezett átláncolás függvény egy megadott lista elemeit vagy annak egy részét mozgatja át egy aktuális objektumba. A listából eltávolított elemekre mutató bejárók érvénytelenné válnak az átláncolás után.

```
#include <iostream>
#include <list>

int main()
{
    std::list<int> List1;
    std::list<int> List2;
    std::list<int>::iterator it;

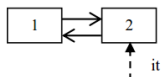
    for (int i = 1; i <= 2; ++i)
        List1.push_back(1);

    for (int i = 1; i <= 3; ++i)
        List2.push_back(1 * 10);

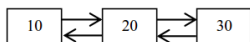
    it = List1.begin();
    ++it;

    List1.splice(it, List2);
    List2.splice(List2.begin(), List1, it);
    return 0;
}
```

List1

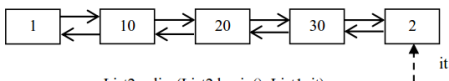


List2



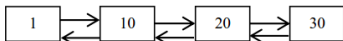
```
it = List1.begin();  
++it;  
List1.splice(it, list2);
```

List1



```
List2.splice(List2.begin(), List1, it);
```

List1



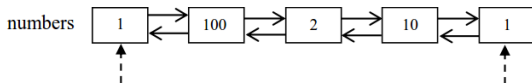
List2



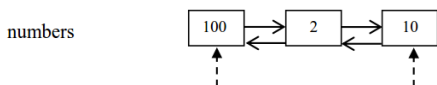
A Kétszeresen láncolt lista elemeinek átmozgatása a splice() függvénnyel

Az elemek eltávolításához a tároló tagmetódusokat biztosít: a **remove()** a listából minden olyan elemet eltávolít, amely megegyezik **remove()**-nak átadott értékkel. A **remove_if()** metódus függvényobjektumot vár paraméterül, amely a megfelelő feltételt tartalmazza.

```
#include <list>  
#include <iostream>  
  
int main()  
{  
    std::list<int> numbers = { 1,100,2,10,1,-1 };  
  
    numbers.remove(1);  
    numbers.remove_if([](int n) { return n > 9; });  
    return 0;  
}
```



numbers.remove(1)



numbers.remove_if([](int n) { return n > 9; });

numbers



A Kétszeresen láncolt lista elemeinek törlése a remove() és remove_if() függvénnyel

A lista tároló rendező metódusokat is tartalmaz. A **reverse()** névre hallgató tagmetódus egyszerűen megfordítja az elemek sorrendjét. A **sort()** tagfüggvény a tároló elemeit rendezi < operátora segítségével. A **merge()** metódus egy már rendezett listát összefűz egy korábban rendezett listával oly módon, hogy az elemek az összefűzés után is rendezettek maradnak. Ebben az esetben is használható az összehasonlító művelet.

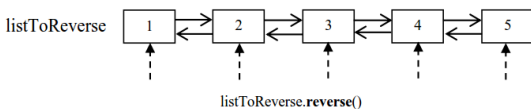
```

#include <iostream>
#include <list>

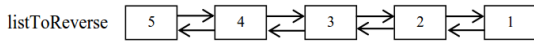
int main()
{
    std::list<int> listToReverse;

    for (int i = 1; i<=5; ++i)
        listToReverse.push_back(i);
    listToReverse.reverse();
    return 0;
}

```



listToReverse.reverse()



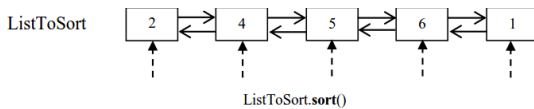
A Kétszeresen láncolt lista elem sorrendjeinek megfordítása a reverse() függvény segítségével.

```

#include <iostream>
#include <list>

int main()
{
    std::list<int> ListToSort = { 2,4,5,6,1 };
    ListToSort.sort();
    return 0;
}

```



A Kétszeresen láncolt lista elem rendezése a sort() függvény segítségével.

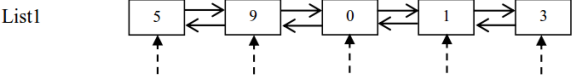
```

#include <iostream>
#include <list>

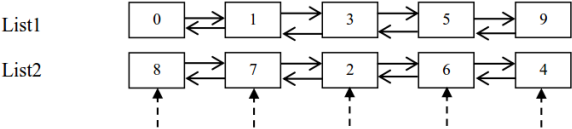
int main()
{
    std::list<int> List1 = { 5,9,0,1,3 };
    std::list<int> List2 = { 8,7,2,6,4 };

    List1.sort();
    List2.sort();
    List1.merge(List2);
    return 0;
}

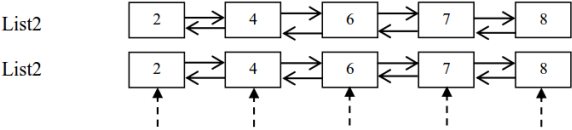
```



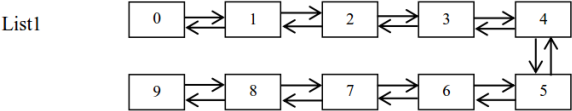
List1.sort()



List2.sort()



List1.merge(list2)



A Kétszeresen láncolt listák elemeinek összefűsülése a merge() függvény segítségével.

Elem hozzáférés	Általános, rövid leírás
front	Az első elemre mutató referenciát adja vissza.
back	Az utolsó elemre mutató referenciát adja vissza.
Kapacitás	Általános, rövid leírás

empty	Üresség vizsgálata. Logikai igaz értékkel tér vissza, ha a lista tároló üres, ellenkező esetben logikai hamis lesz az érték.
size	Az elemek számával tér vissza.
max_size	A lehetséges maximális elemszámmal visszatér.

Módosítók	Általános, rövid leírás
assign	Új tartalmat rendel a listához.
emplace	Specifikus pozícióba egy új elemet helyez.
emplace_front	Az első pozícióba egy új elemet helyez.
emplace_back	Az utolsó pozícióba egy új elemet helyez.
push_front	Elemet töröl a lista elejéről.
push_back	Elemet ad a lista végéhez.
pop_front	A lista elejéről töröl elemet.
pop_back	Törli a lista utolsó elemét.
insert	A specifikusan meghatározott pozíció elé szúr be egy új elemet.
erase	Elemek törlése a listából.
swap	Két lista tartalmát megcseréli, oly módon, hogy a bejárók érvényesek maradnak.
resize	A listát újraméretezi.
clear	Elemek törlése, minden elemre mutató bejáró érvénytelen lesz.

Műveletek	Általános, rövid leírás
splice	Több előfordulása ismert. Az első változat során a függvény paraméterben megadott lista elemeit átmozgatja az aktuális objektumba. Egy második változat esetében a lista harmadik, függvény paraméterként megadott iterátor által mutatott elemét mozgatja át. A harmadik változat során a lista harmadik függvény paramétereként megadott

	bejárója és a negyedikként megadott bejárója által bezárt, félig nyílt tartományt mozgatja át.
remove	A függvény paramétereként megadott értékeket eltávolítja a listából.
remove_if	Olyan elemeket távolít el a listából, amelyek a megadott feltételeknek eleget tesznek.
unique	Elsősorban rendezett listák esetén lehet hasznos, mivel egymás melletti azonos értékű elemek közül csak 1 db-ot hagy meg.
merge	Két rendezett listát fűszül össze. A művelet vége után a paraméterben átadott lista üres lesz.
sort	A lista elemeit úgy rendezi, hogy a bejárók érvényesek maradnak.
reverse	Úgy fordítja meg a lista elemek sorrendjét, hogy a bejárók érvényesek maradnak.

Bejárók	Általános, rövid leírás
begin	A kétszeresen láncolt lista elejét (első elemét) reprezentálja egy tetszőleges elérésű bejáróval.
end	A kétszeresen láncolt lista utolsó utáni helyét (utolsó utáni elem – <i>past-the-end</i>) reprezentálja egy tetszőleges elérésű bejáróval.
rbegin	A kétszeresen láncolt lista végét (utolsó elem) reprezentálja egy tetszőleges elérésű, fordított bejáróval.
rend	A kétszeresen láncolt lista első elem előtti helyét (első előtti elem) reprezentálja egy tetszőleges elérésű, fordított bejáróval.
cbegin	A kétszeresen láncolt lista elejét (első elem) reprezentálja egy konstans bejáróval.
cend	A kétszeresen láncolt lista utolsó utáni helyét (utolsó utáni elem – <i>past-the-end</i>) reprezentálja egy konstans bejáróval.
crbegin	A kétszeresen láncolt lista végét (utolsó elem) reprezentálja egy konstans, fordított bejáróval.

crend A kétszeresen láncolt lista első elem előtti helyét (első előtti elem) reprezentálja egy konstans, fordított bejáróval.

Kétszeresen láncolt lista objektum tagfüggvényei

Minta példa a kétszeresen láncolt lista (list) tároló használatára Visual Studio Community fejlesztőkörnyezetben

Írjunk egy `ListDemo` (*ListDemo*) osztályt, amely különböző, kétszeresen láncolt lista specifikus műveleteket ír le, majd egy objektumpéldány segítségével ezeket a műveleteket hajtassuk végre. Minden egyes tagfüggvény feladatának megfogalmazását, zárójelben követi az adott tárolóhoz tartozó tagfüggvény, amelyeket kötelező jelleggel kell használni a feladat helyes megoldásának elérése érdekében, például a konstruktor esetében ilyen a **`push_back()`** metódus. Az osztály adatai a következők:

- *list*, egy kétszeresen láncolt lista jellegű szekvenciális tároló, amely 15 db egész szám tárolására képes.

Ügyeljünk az egységbezáráshoz elvére, így a fentebb megfogalmazott adat(ok) **private**, az alábbi tagfüggvények **public** láthatósági szintekkel bírnak!

Valósítsuk meg az osztály metódusait:

- Írjuk meg az osztály konstruktorát, amely 15 mínusz a ciklusváltozó értékével inicializálja a kétszeresen láncolt lista tárolót. (**`push_back()`**)
- Írjunk egy *isEmpty()* függvényt, amely megvizsgálja, hogy a kétszeresen láncolt lista tartalmaz-e elemeket, majd annak függvényében visszatér egy logikai értékkel. Igazzal, ha található elem a kétszeresen láncolt listában, hamissal, ha üres. (**`empty()`**)
- Írjunk egy *getSize()* függvényt, amely a kétszeresen láncolt lista aktuális méretét adja vissza. (**`size()`**)
- Írjunk egy *getMaxSize()* függvényt, amely a kétszeresen láncolt maximális méretének értékével tér vissza. (**`max_size()`**)
- Írjunk egy *resizeList(int size)* függvényt, amely paraméterül vár egy egész szám típusú értéket, majd ezzel az értékkel átméretezi a kétszeresen láncolt listát. (**`resize()`**)
- Írjunk egy *clearList()* függvényt, amely törli a lista elemeit. (**`clear()`**)
- Írjunk egy *getFront()* függvényt, amely a kétszeresen láncolt lista első elemét adja vissza. (**`front()`**)

- Írjunk egy *getBack()* függvényt, amely a kétszeresen láncolt lista utolsó elemét adja vissza. (**back()**)
- Írjunk egy *assignValues()* függvényt, amely a kétszeresen láncolt listát újra feltölti 3 hosszan 100-as értékekkel (**assign()**)
- Írjunk egy *addElementAtTheEnd(int value)* függvényt, amely a paraméterül kapott értéket a kétszeresen láncolt lista végéhez fűzi. (**push_back()**)
- Írjunk egy *addElementAtBeginningOfList(int value)* függvényt, amely a lista elejéhez fűz egy új elemet a függvény paramétereként kapott értékkel. (**push_front()**)
- Írjunk egy *deleteFirstElement()* függvényt, amely a kétszeresen láncolt lista elejéről törli az első elemet. (**pop_front()**)
- Írjunk egy *deleteLastElement()* függvényt, amely a kétszeresen láncolt lista végéről törli az utolsó elemet. (**pop_back()**)
- Írjunk egy *insertElement(int index, int value)* függvényt, amely a kétszeresen láncolt lista elejétől, a függvény paramétereként kapott *index* határig beszúrja a paraméterként kapott *value* értéket. (**insert()**)
- Írjunk egy *eraseElements(int length)* függvényt, amely a kétszeresen láncolt lista elejétől, (**begin()**) a függvény paramétereként megkapott *index* határig, törli az értékeket. (**erase()**)
- Írjunk egy *emplace(int value)* függvényt, amely a kétszeresen láncolt lista egy specifikus pozíciójába beszúrja a paraméterben kapott számértéket. (**emplace()**)
- Írjunk egy *emplaceFront(int value)* függvényt, amely a kétszeresen láncolt lista elejéhez hozzáfűzi a paraméterben kapott számértéket. (**emplace_front()**)
- Írjunk egy *emplaceBack(int value)* függvényt, amely a kétszeresen láncolt lista végéhez hozzáfűzi a paraméterben kapott számértéket. (**emplace_back()**)
- Írjunk egy *spliceList()* függvényt, amely egy kétszeresen láncolt lista tartalmát áttölti egy másik listába. (**splice()**)
- Írjunk egy *removeSpecificElement()* függvényt, amely a kétszeresen láncolt lista tároló 7-es értékét törli. (**remove()**)
- Írjunk egy *removeIfLessThan20()* függvényt, amely a kétszeresen láncolt lista azon elemeit törli, amelyek kisebbek, mint 20. (**remove_if()**)
- Írjunk egy *removeDuplicateValue()* függvényt, amely a kétszeresen láncolt lista duplum értékeit törli. (**unique()**)
- Írjunk egy *sortList()* függvényt, amely a kétszeresen láncolt lista

elemeit rendez. (**sort()**)

- Írjunk egy *reverseList()* függvényt, amely a kétszeresen láncolt lista elemek sorrendjének megfordítását hajtja végre. (**reverse()**)
- Írjunk egy *printContent()* függvényt, amely kiírja egy konstans iterátor felhasználásával a kétszeresen láncolt lista minden egyes elemét, a **cbegin()** és a **cend()** metódusok felhasználásával.

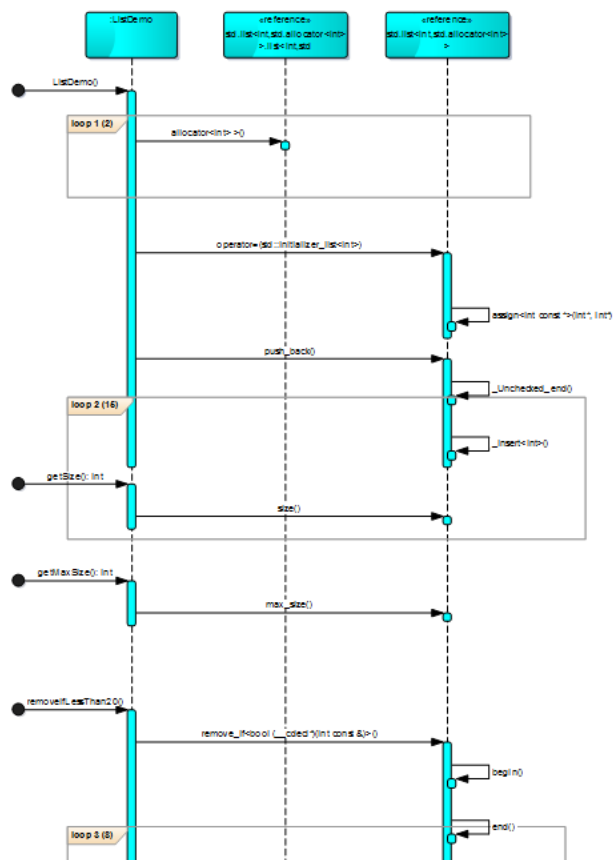
Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén [**heap**]) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektum példányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. Figyelem az **auto** kulcsszóval a fordítóra is bízhatjuk az iterátorok típusának felismerését! Mielőtt a feladatnak neki látnánk, tervezzük meg az osztály- és szekvencia diagramot. **Figyelem! A könyvben ábrázolt szekvenciadiagramok terjedelmi korlátok miatt a teljesség igénye nélkül készültek!**

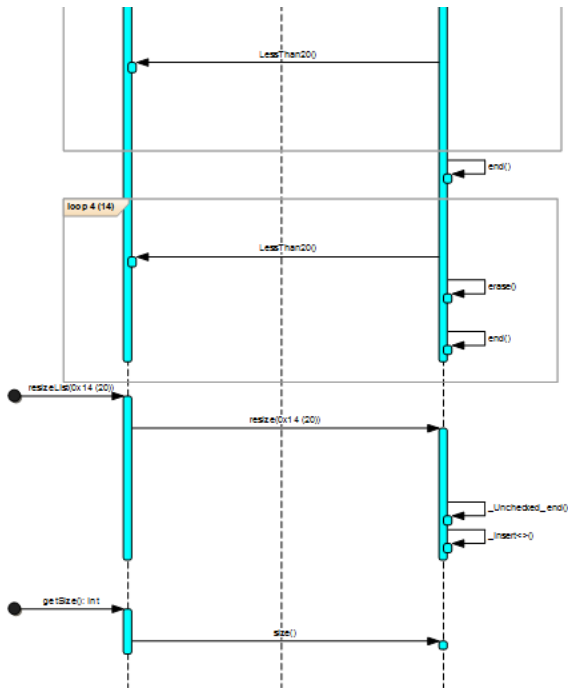
Statikus tervezés

ListDemo
- list: std::list<int> - tetranacci: std::list<int>
+ addElementAtBeginning(int): void + addElementAtTheEnd(int): void + assignValues(): void + clearList(): void + deleteFirstElement(): void + deleteLastElement(): void + emplace(int): void + emplaceBack(int): void + emplaceFront(int): void + eraseElements(): void + getBack(): int + getFront(): int + getMaxSize(): int + getSize(): int + insertElement(int, int): void + isEmpty(): bool + ListDemo() + printContent(): void + removeDuplicateValue(): void + removeIfLessThan20(): void + removeSpecificElement(): void + resizeList(int): void + reverseList(): void + sortList(): void + spliceList(): void

Osztálydiagram

Dinamikus diagram





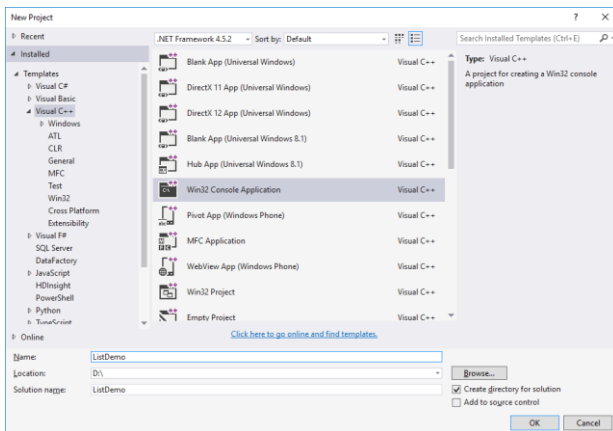
Néhány tagfüggvény szekvenciadiagramja

A projekt létrehozása

A projekt megvalósítása során konzolalkalmazást készítettünk, 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

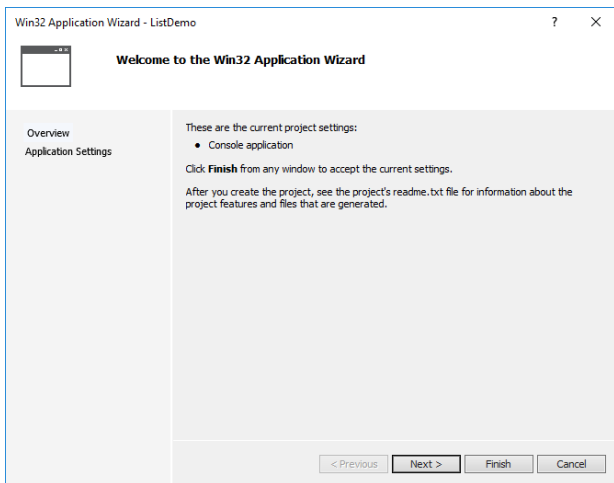
megfogalmazott minta példa kerül implementálásra és összeállításra, továbbá Windows operációs rendszeren futtatható kóddá történő lefordításra Visual Studio Community fejlesztőkörnyezet felhasználásával. A lefordított 32 bites alkalmazás *konzolban* kerül futtatásra.

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 közül a **Visual C++ Win32** sablont (**Template**) kell kiválasztani és a **Win32 Console Application**-t, mely az alábbi ábrán látható.



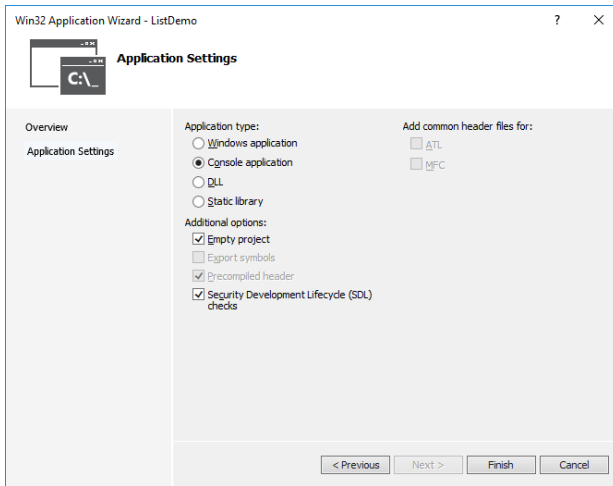
Win32 Console Application projekt létrehozása

Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően a **ListDemo** név került megadásra. Az OK gombra kattintást követően elindul az alkalmazás varázsló, mely a következő ábrán látható.



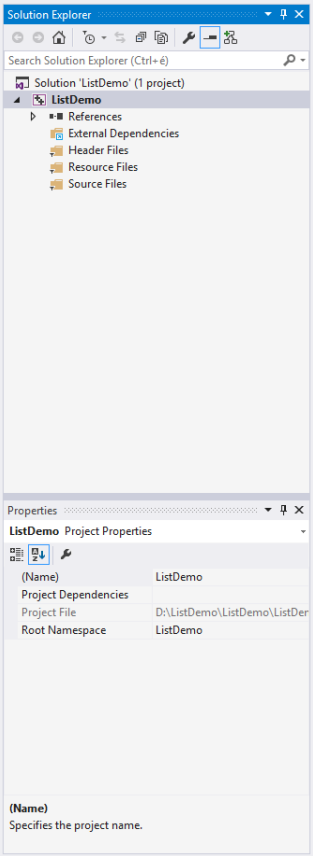
Win32 konzol alkalmazás varázsló

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.



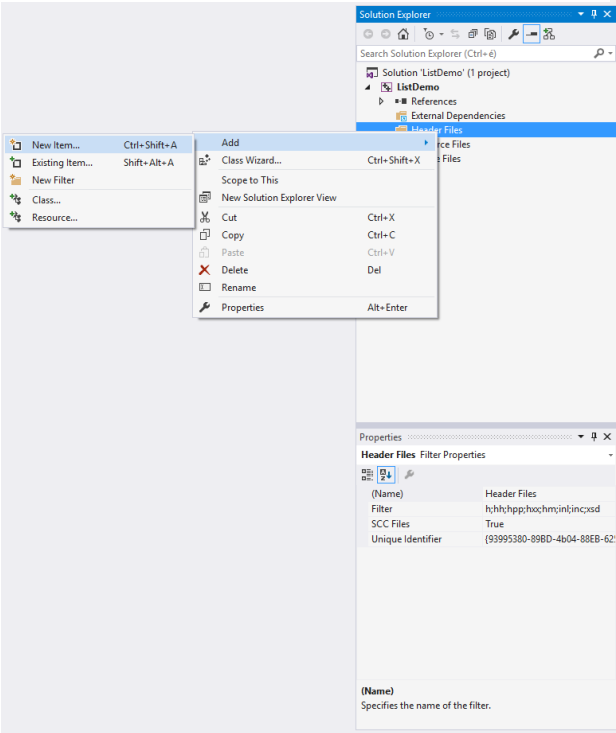
Alkalmazás beállításai

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



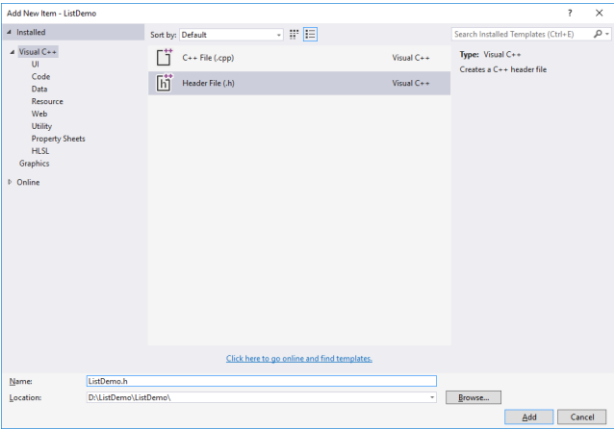
Megoldás és beállítások ablak

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.



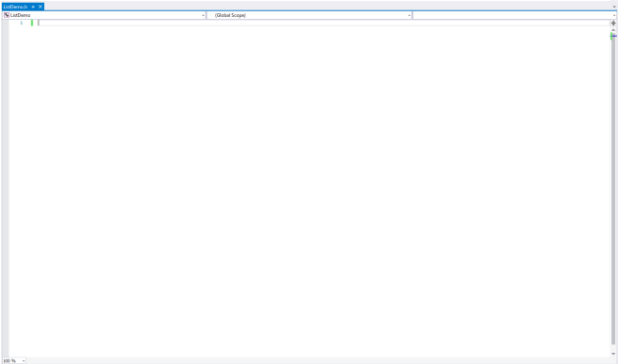
Új header fájl hozzáadása a projekthez

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



Új header fájl hozzáadása a projekthez, név megadása

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



Forráskód szerkesztő

ListDemo.h

```
#ifndef LISTDEMO_H
#define LISTDEMO_H

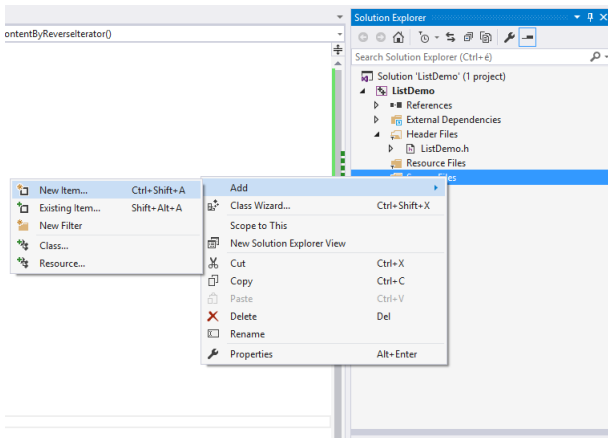
#include <list>

class ListDemo
{
private:
    std::list<int> list;
    std::list<int> tetranacci;

public:
    ListDemo();
    int getSize();
    int getMaxSize();
    void resizeList(int size);
    bool isEmpty();
    void clearList();
    int getFront();
    int getBack();
    void assignValues();
    void addElementAtTheEnd(int value);
    void addElementAtBeginning(int value);
    void deleteLastElement();
    void deleteFirstElement();
    void insertElement(int index, int value);
    void eraseElements();
    void emplace(int value);
    void emplaceFront(int value);
    void emplaceBack(int value);
    void spliceList();
    void removeSpecificElement();
    void removeIfLessThan20();
    void removeDuplicateValue();
    void sortList();
    void reverseList();
    void printContent();
};

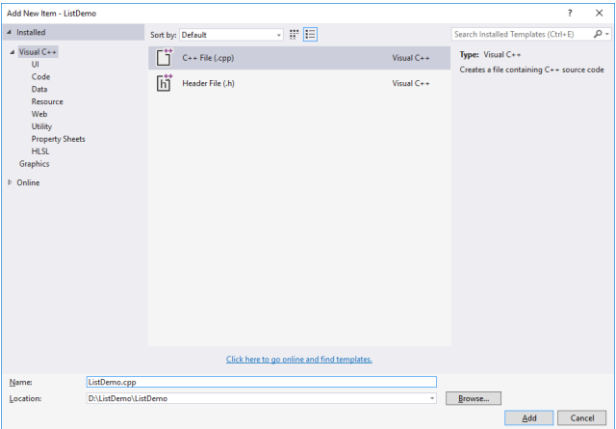
#endif
```

A deklarált osztályok megvalósításait a **ListDemo.cpp** forrásfájl tartalmazza, melyeket a **header** fájlokhoz hasonlóan, de forrásfájl (**Source File**) hozzáadásával hozhatunk létre.



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása

ListDemo.cpp

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

ListDemo::ListDemo()
{
    tetranacci = { 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401, 773, 1490 };
    for (unsigned int i = 0; i < 15; ++i)
        list.push_back(15 - i);
}

int ListDemo::getSize()
{
    return list.size();
}

int ListDemo::getMaxSize()
{
    return list.max_size();
}

void ListDemo::resizeList(int size)
{
    list.resize(size);
}

bool ListDemo::isEmpty()
{
    return list.empty();
}

void ListDemo::clearList()
{
    list.clear();
}

int ListDemo::getFront()
{
    return list.front();
}

int ListDemo::getBack()
{
    return list.back();
}

void ListDemo::assignValues()
{
    list.assign(3, 100);
}

void ListDemo::addElementAtTheEnd(int value)
{
    list.push_back(value);
}

void ListDemo::addElementAtBeginning(int value)
{
    list.push_front(value);
}

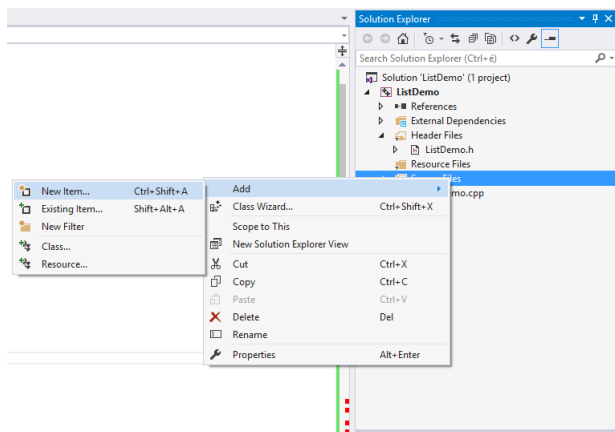
void ListDemo::deleteFirstElement()
{
}
```

```
    list.pop_front();
}
void ListDemo::deleteLastElement()
{
    list.pop_back();
}
void ListDemo::insertElement(int index, int value)
{
    list.insert(list.begin(), index, value);
}
void ListDemo::eraseElements()
{
    list.erase(list.begin(), list.end());
}
void ListDemo::emplace(int value)
{
    list.emplace(list.begin(), value);
}
void ListDemo::emplaceFront(int value)
{
    list.emplace_front(value);
}
void ListDemo::emplaceBack(int value)
{
    list.emplace_back(value);
}
void ListDemo::spliceList()
{
    list.splice(list.end(), tetranacci);
}
void ListDemo::removeSpecificElement()
{
    list.remove(7);
}
bool LessThan20(const int& value) { return (value < 20); }
void ListDemo::removeIfLessThan20()
{
    list.remove_if(LessThan20);
}
void ListDemo::removeDuplicateValue()
{
    list.unique();
}
void ListDemo::sortList()
{
    list.sort();
}
void ListDemo::reverseList()
{
    list.reverse();
}
```

```

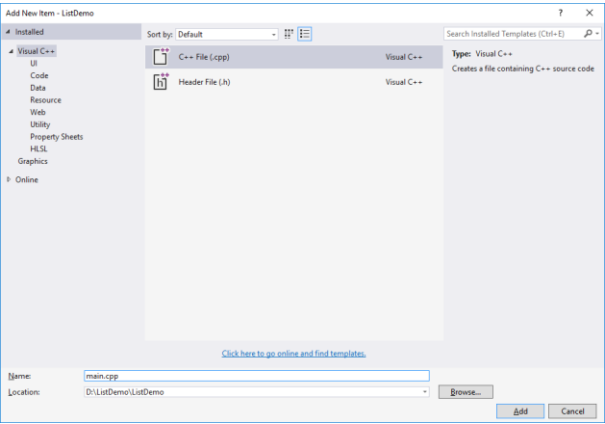
void ListDemo::printContent()
{
    for (auto constIterator = list.cbegin(); constIterator != list.cend();
        ++constIterator)
        std::cout << ' ' << *constIterator;
}

```



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása

main.cpp

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

int main()
{
    ListDemo *ListDemoObject = new ListDemo();

    std::cout << "MUVELET => ListDemo() #konstruktor hivasa, list tarolo
feltoltese" << std::endl;
    std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
    std::endl;
    std::cout << "MUVELET => empty() #uresseg vizsgalat..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << ListDemoObject->isEmpty() <<
    std::endl;
    std::cout << "MUVELET => pop_back() #utolso elem torles..." <<
    std::endl;
    ListDemoObject->deleteLastElement();
    std::cout << "MUVELET => pop_front() #első elem torles..." << std::endl;
    ListDemoObject->deleteFirstElement();
    std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
    std::endl;
    std::cout << "MUVELET => size() #tarolo meret lekerdezese..." <<
    std::endl;
    std::cout << "EREDMENY: "; std::cout << ListDemoObject->getSize();
    std::cout << std::endl;
    std::cout << "MUVELET => max_size() #tarolo maximalis meretenek
lekerdezese..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << ListDemoObject->getMaxSize();
    std::cout << std::endl;
    std::cout << "MUVELET => insert() #beszuras..." << std::endl;
    ListDemoObject->insertElement(3, 7);
    std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
    std::endl;
    std::cout << "MUVELET => back() #utolso elem lekerdezese..." <<
    std::endl;
    std::cout << "EREDMENY: "; std::cout << ListDemoObject->getBack() <<
    std::endl;
    std::cout << "MUVELET => front() #első elem lekerdezese..." <<
    std::endl;
    std::cout << "EREDMENY: "; std::cout << ListDemoObject->getFront() <<
    std::endl;
    std::cout << "MUVELET => push_back() #hozzafuzes a tarolo vegehez..." <<
    std::endl;
    ListDemoObject->addElementAtBeginning(15);
    std::cout << "MUVELET => push_front() #hozzafuzes a tarolo elejehez..."
    << std::endl;
    ListDemoObject->addElementAtTheEnd(1);
    std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
    std::endl;
    std::cout << "MUVELET => remove() #egy specifikus eleme torlese..." <<
```

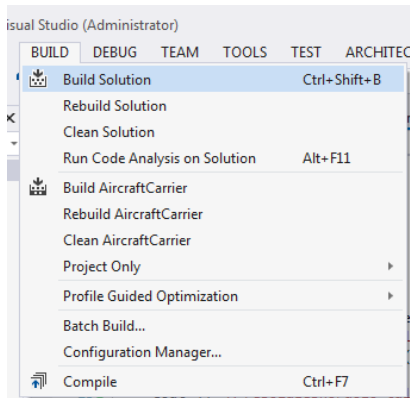
```

std::endl;
ListDemoObject->removeSpecificElement();
std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
std::endl;
std::cout << "MUVELET => resize() #atmeretezes..." << std::endl;
std::cout << "MUVELET => emplace_front() #hozzafuzes a tarolo
elejehhez..." << std::endl;
ListDemoObject->emplaceFront(99);
std::cout << "MUVELET => emplace_back() #hozzafuzes a tarolo vegehez..."
<< std::endl;
ListDemoObject->emplaceBack(9);
std::cout << "MUVELET => emplace() #beszuras specifikus pozicioba..." <<
std::endl;
ListDemoObject->emplace(5);
std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
std::endl;
std::cout << "MUVELET => splice() #tartalom attoltese a list
taroloba..." << std::endl;
ListDemoObject->spliceList();
std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
std::endl;
std::cout << "MUVELET => sort() #rendezes..." << std::endl;
ListDemoObject->sortList();
std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
std::endl;
std::cout << "MUVELET => reverse() #sorrend megforditasa..." <<
std::endl;
ListDemoObject->reverseList();
std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
std::endl;
std::cout << "MUVELET => unique() #duplikatumok torlese..." <<
std::endl;
ListDemoObject->removeDuplicateValue();
std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
std::endl;
std::cout << "MUVELET => remove() #20-nal kisebb elemek torlese..." <<
std::endl;
ListDemoObject->removeIfLessThan20();
std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
std::endl;
std::cout << "MUVELET => clear() #elemek torlese..." << std::endl;
ListDemoObject->clearList();
std::cout << "MUVELET => assign() #(ujra)feltoltes..." << std::endl;
ListDemoObject->assignValues();
std::cout << "ALLAPOT: "; ListDemoObject->printContent(); std::cout <<
std::endl;
ListDemoObject->resizeList(20);
std::cout << "MUVELET => size() #tarolo meret lekerdez..." <<
std::endl;
std::cout << "EREDMENY: "; std::cout << ListDemoObject->getSize();
std::cout << std::endl;
std::cout << "MUVELET => erase() #torles..." << std::endl;

```

```
ListDemoObject->eraseElements();  
  
delete ListDemoObject;  
_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.



Program fordítása

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

```
Output
Show output from: Build
1>----- Rebuild All started: Project: ListDemo, Configuration: Debug Win32 -----
1> main.cpp
1> ListDemo.cpp
1> Generating Code...
1> ListDemo.vcxproj -> D:\ListDemo\Debug\ListDemo.exe
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped =====

Output Find Symbol Results Error List
```

Kimenet ablak

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

```
WT D:\ListDemo\Debug\ListDemo.exe
MUNELET -> listdemo() #struktur: lista, list tarolo feltoltese
ALLAPOT: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
MUNELET -> empty() #uresseg vizsgalat...
EREDMENY: 0
MUNELET -> pop_back() #utolso elem torles...
MUNELET -> pop_front() #első elem torles...
ALLAPOT: 14 13 12 11 10 9 8 7 6 5 4 3 2
MUNELET -> size() #tarolo meret lekerdezese...
EREDMENY: 13
MUNELET -> max_size() #tarolo maximalis meretnek lekerdezese...
EREDMENY: 357913041
MUNELET -> insert() #beszuras...
ALLAPOT: 7 7 7 14 13 12 11 10 9 8 7 6 5 4 3 2
MUNELET -> back() #utolso elem lekerdezese...
EREDMENY: 2
MUNELET -> front() #első elem lekerdezese...
EREDMENY: 7
MUNELET -> push_back() #hozzafuzes a tarolo vegéhez...
MUNELET -> push_front() #hozzafuzes a tarolo elejéhez...
ALLAPOT: 15 7 7 14 13 12 11 10 9 8 7 6 5 4 3 2 1
MUNELET -> remove() #egy specifikus elem torlese...
ALLAPOT: 15 14 13 12 11 10 9 8 6 5 4 3 2 1
MUNELET -> resize() #atmeretezese...
MUNELET -> emplace_front() #hozzafuzes a tarolo elejéhez...
MUNELET -> emplace_back() #hozzafuzes a tarolo vegéhez...
MUNELET -> emplace() #beszuras specifikus pozicioba...
ALLAPOT: 5 99 15 14 13 12 11 10 9 8 6 5 4 3 2 1 9 1 2 4 8 15 29 56 100 200 401 773 1490
MUNELET -> splice() #tartalom attoltese a list taroloha...
ALLAPOT: 1 1 1 2 2 3 4 4 5 5 6 8 8 9 9 10 11 12 13 14 15 15 29 56 99 100 200 401 773 1490
MUNELET -> sort() #rendezes...
MUNELET -> reverse() #sorrend megforditasa...
ALLAPOT: 1490 773 401 200 100 99 56 29 15 15 14 13 12 11 10 9 9 8 6 5 5 4 4 3 2 2 1 1 1
MUNELET -> unique() #szamliketmek torlese...
ALLAPOT: 1490 773 401 200 100 99 56 29 15 14 13 12 11 10 9 8 6 5 4 3 2 1
MUNELET -> remove() #20-nal kisebb elemek torlese...
ALLAPOT: 1490 773 401 200 100 99 56 29
MUNELET -> assign() #ujra feltoltese...
ALLAPOT: 100 100 100
MUNELET -> size() #tarolo meret lekerdezese...
EREDMENY: 20
MUNELET -> erase() #torles...
```

A program futásának eredménye

A feladathoz kapcsolódó magyarázatok

Az osztály deklarációját a (**ListDemo.h**) állományban adjuk meg. A deklarációban két magán (**private**) jellegű, egész szám típusú (**integer**) adatok tárolására képes kétszeresen láncolt lista kerül megfogalmazásra, amelyet az std névtéren keresztül érünk el:

```
private:
    std::list<int> list;
    std::list<int> tetranacci;
```

A konstruktor neve tulajdonságaiból fakadóan megegyezik az osztály nevével, láthatósági szintje nyilvános (**public**), visszatérési értéke nincs, még **void** sem, továbbá az adott feladatban paraméter nélküli formában jelenik meg:

```
public:
    ListDemo();
```

A *getSize()*, *getMaxSize()*, *getFront()* és *getBack()* tagfüggvények a leírásban megfogalmazott kétszeresen láncolt lista aktuális- és maximális méretének, valamint az első és utolsó elemének lekérdezését biztosítják. A láthatósági szintjük nyilvános (**public**), visszatérési típusuk egész szám (**integer**) és argumentumlistájuk üres:

```
int getSize();
int getMaxSize();
int getFront();
int getBack();
```

Az *isEmpty()* tagfüggvény a kétszeresen láncolt lista ürességének vizsgálatát hajtja végre, láthatósági szintje nyilvános (**public**), visszatérési típusa logikai (**bool**), argumentumlistája üres:

```
bool isEmpty();
```

A *resizeList(int size)* tagfüggvény a kétszeresen láncolt lista aktuális méretének átméretezését végzi, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a kétszeresen láncolt lista jövőben használni kívánt méretének értéke:

```
void resizeList(int size);
```

Az `clearList()` tagfüggvény a kétszeresen láncolt lista elemeit törli, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void clearList();
```

Az `assignValues()` tagfüggvény iterátorokkal megadott tartományba helyez be elemeket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void assignValues();
```

Az `addElementAtTheEnd(int element)` tagfüggvény az utolsó pozícióba a paraméterül kapott elemet fűzi a kétszeresen láncolt listához, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely a beszúrni kívánt érték:

```
void addElementAtTheEnd(int value);
```

Az `addElementAtBeginning(int element)` tagfüggvény az első pozícióba a paraméterül kapott elemet fűzi a kétszeresen láncolt listához, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely a beszúrni kívánt érték:

```
void addElementAtBeginning(int value);
```

A `deleteLastElement()` tagfüggvény a kétszeresen láncolt lista utolsó elemét törli, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void deleteLastElement();
```

A `deleteFirstElement()` tagfüggvény a kétszeresen láncolt lista első elemét törli, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void deleteFirstElement();
```

Az `insertElements(int index, int value)` tagfüggvény egy megadott pozícióba n alkalommal beszúr egy értéket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint két egész szám típusú (**integer**) értéket vár paraméterül, ahol az első paraméter határozza meg a beszúrandó tartomány hosszát, a második argumentum a beszúrni kívánt értéket:

```
void insertElements(int index, int value);
```

Az `eraseElements()` tagfüggvény egy megadott pozíciótól n hosszan törli az értéket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void eraseElements();
```

Az `emplace(int value)` tagfüggvény specifikus pozícióba új elemet ad a kétszeresen láncolt listához, visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a kétszeresen láncolt listához hozzáfűzni kívánt érték:

```
void emplace(int value);
```

Az `emplaceFront(int value)` tagfüggvény az első pozícióba új elemet fűz a kétszeresen láncolt listához, visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a kétszeresen láncolt listához hozzáfűzni kívánt érték:

```
void emplaceFront(int value);
```

Az `emplaceBack(int value)` tagfüggvény az utolsó utáni pozícióba új elemet fűz a kétszeresen láncolt listához, visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték a kétszeresen láncolt listához hozzáfűzni kívánt érték:

```
void emplaceBack(int value);
```

A `printContent()` tagfüggvény konstans típusú iterátor segítségével megjeleníti a konzolon a kétszeresen láncolt lista aktuális elemeit, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void printContent();
```

A `spliceList()` tagfüggvény új elemeket tölt át a kétszeresen láncolt listába, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void spliceList();
```

A `removeSpecificElement()` tagfüggvény egy specifikus elemet töröl a kétszeresen láncolt listából, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void removeSpecificElement();
```

A `removeIfLessThan20()` tagfüggvény a 20-tól kisebb értékeket törli a kétszeresen láncolt listából, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void removeIfLessThan20();
```

A `removeDuplicateValue()` tagfüggvény a duplumokat törli a kétszeresen láncolt listából, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void removeDuplicateValue();
```

A `sortList()` tagfüggvény sorba rendezi a kétszeresen láncolt lista elemeit, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void sortList();
```

A `reverseList()` tagfüggvény a kétszeresen láncolt listában az elemek aktuális sorrendjét megfordítja, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void reverseList();
```

A **ListDemo.cpp** állomány tartalmazza az osztály teljeskörű implementációját. Az implementáció során az egyes tagfüggvények neve előtt a `::` (hatókör) operátor került alkalmazásra. azon célból, hogy pontosan meghatározzuk melyik osztály tagfüggvényét valósítottuk meg.

A konstruktor feladata, hogy az adattagokat kezdőértékkel inicializálja. A *list* azonosítójú kétszeresen láncolt lista adattag 15-től 1-ig csökkenő sorrendben megkapja a ciklusváltozó értékeit. A *tetranocci* azonosítójú kétszeresen láncolt lista adattag forráskód szintű feltöltése során a tetranocci sorozat 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401, 773, 1490 elemeit kapja meg. A kezdőérték adásnál a **push_back()** módosító függvényt alkalmaztuk.

```
ListDemo::ListDemo()
{
    tetranacci = { 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401, 773, 1490 };
    for (unsigned int i = 0; i < 15; ++i)
        list.push_back(15 - i);
}
```

Az aktuális- és maximális méret, valamint az első és utolsó elem lekérdezését végrehajtó függvények implementációja során a **list** objektumon keresztül egyszerűen hívásra kerültek a **size()**, **max_size()**, illetve a **front()** és **back()** metódusok.

```
int ListDemo::getSize()
{
    return list.size();
}
int ListDemo::getMaxSize()
{
    return list.max_size();
}
int ListDemo::getFront()
{
    return list.front();
}
int ListDemo::getBack()
{
    return list.back();
}
```

Az üresség vizsgálata során a méret lekérdezéséhez hasonlóan a **list** objektumon keresztül hívjuk meg a szükséges tagfüggvényt, amely ebben az esetben az **empty()** metódus.

```
bool ListDemo::isEmpty()
{
    return list.empty();
}
```

A méret módosító függvények közül az újra méretezéshez a **resize()**-t használtuk, amely a paraméterben kapott nagyságú értékre állította a kétszeresen láncolt lista aktuális méretét.

```
void ListDemo::resizeList(int size)
{
    list.resize(size);
}
```

A kétszeresen láncolt lista elemeinek törlését, a **clear()** tagfüggvény hívásával hajthatjuk végre.

```
void ListDemo::clearList()
{
    list.clear();
}
```

A kétszeresen láncolt lista módosító függvények közül az **assign()** alkalmazása során a 100-as értéket N-szer helyeztük a kétszeresen láncolt listába, ahol az N=3-al.

```
void ListDemo::assignValues()
{
    list.assign(3, 100);
}
```

A kétszeresen láncolt lista első pozíciójába új elemet szúrhatunk be a **push_front()** és **emplace_front()** tagfüggvény segítségével, amíg az utolsó pozíciójába új elemet a **push_back()** és **emplace_back()** segítségével helyezhetünk. Az **emplace()** lehetőséget biztosít a kétszeresen láncolt lista egy specifikus pozíciójába történő elem beszúrásához.

```

void ListDemo::addElementAtTheEnd(int value)
{
    list.push_back(value);
}
void ListDemo::addElementAtBeginning(int value)
{
    list.push_front(value);
}
void ListDemo::emplace(int value)
{
    list.emplace(list.begin(), value);
}
void ListDemo::emplaceFront(int value)
{
    list.emplace_front(value);
}
void ListDemo::emplaceBack(int value)
{
    list.emplace_back(value);
}

```

A kétszeresen láncolt lista első elemének eltávolításához a **pop_front()**, amíg az utolsó elemének eltávolításához a **pop_back()** tagfüggvényt használtuk.

```

void ListDemo::deleteFirstElement()
{
    list.pop_front();
}
void ListDemo::deleteLastElement()
{
    list.pop_back();
}

```

A kétszeresen láncolt listába történő beszúrást az **insert()** tagfüggvény segítségével végeztük el ahol az első paraméter a kétszeresen láncolt list elejét jelzi (**begin()**), a második azt az index helyet, ameddig a beszúrást kívánjuk elvégezni, amíg a harmadik a beszúrandó értéket.

```

void ListDemo::insertElement(int index, int value)
{
    list.insert(list.begin(), index, value);
}

```

Az **erase()** tagfüggvényt felhasználva a paraméterben megadott tartományból töröltük az elemeket, ahol a tartomány a kétszeresen láncolt lista elejétől kezdődik és a függvény paraméterében átadott hosszíig tart.

```

void ListDemo::eraseElements()
{
    list.erase(list.begin(), list.end());
}

```

Egy kétszeresen láncolt lista tartalmát (a tetranocci sorozat első néhány eleme) egy másik kétszeresen láncolt listába a **splice()** függvény segítségével került áttöltésre, amíg specifikus elem törlését a **remove()** függvénnyel valósítottuk meg.

```

void ListDemo::spliceList()
{
    list.splice(list.end(), tetranacci);
}
void ListDemo::removeSpecificElement()
{
    list.remove(7);
}

```

A kétszeresen láncolt listából feltételnek megfelelő érték halmazt a **remove_if()** tagfüggvénnyel távolítottunk el, a feladatban ez a feltétel a 20-tól kisebb értékek voltak. A duplumok törlését a **unique()** függvény segítségével, amíg a sorrendbe történő rendezést a **sort()** metódus segítségével, a sorrendiség megfordítását a **reverse()**-el végeztük el.

```

bool LessThan20(const int& value) { return (value < 20); }
void ListDemo::removeIfLessThan20()
{
    list.remove_if(LessThan20);
}
void ListDemo::removeDuplicateValue()
{
    list.unique();
}
void ListDemo::sortList()
{
    list.sort();
}
void ListDemo::reverseList()
{
    list.reverse();
}

```

A kétszeresen láncolt lista elemek megjelenítéséhez automatikusan kiválasztott iterátor típust alkalmaztunk. A **cbegin()** tagfüggvény a kétszeresen láncolt lista elejét (első elem), addig a **cend()** a kétszeresen láncolt lista végét (utolsó utáni elem) reprezentálja egy konstans elérésű bejáróval.

```

void ListDemo::printContent()
{
    for (auto constIterator = list.cbegin(); constIterator != list.cend();
        ++constIterator)
        std::cout << ' ' << *constIterator;
}

```

A **(main.cpp)** állomány tartalmazza a **main()** függvényt, a program belépési pontját. A fő függvényben történő dinamikus példányosítása után hívjuk meg a szükséges függvényeket. A műveletek végrehajtását követően, gondoskodjunk a lefoglalt erőforrások felszabadításáról az alapértelmezett explicit destruktor hívásának segítségével.

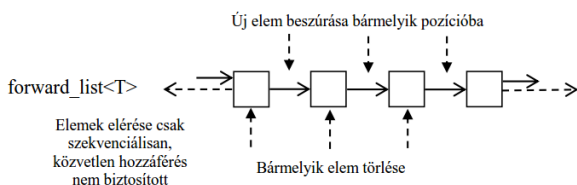
std::forward_list

Egy olyan szekvenciális tároló, amely egyirányú (egyszeresen láncolt) listakezelést tesz lehetővé. Lehetőséget kínál az elemek gyors beszűrésére és törlésére a konténeren belül bárhova, ráadásul ezeknek a műveleteknek az időigény pozíciótól függetlenül konstans és nem függ az elemek számától. Habár a tároló nem támogatja a gyors, véletlenszerű hozzáférést, az elemein – az elsőől az utolsóig – gyorsan végig lehet iterálni. Az egyirányú lista minden elemének a mutatója a következő elem pozícióját tárolja. A kétszeresen láncolt listával ellentétben az egyszeresen láncolt lista kizárólag egy közvetlen mutatót tárol, amely a következő elem helyére mutat, amíg a kétszeresen láncolt lista esetében két mutató kerül eltárolásra, amely a következő elem pozíciója mellett az előbbi elem helyére is mutat. Így elmondható, hogy a kétszeresen láncolt lista ugyan hatékonyabb iterációt tesz lehetővé (előre és hátra), de az elemek és hozzájuk tartozó mutatók tárolási költségét tekintve az egyszeresen láncolt lista a kedvezőbb, főleg abban az esetben, ha sok elemet tartalmaz a tároló. Más szekvenciális tárolókkal (**array**, **vector** és **deque**) összevetve a **forward_list** lényegesen hatékonyabb beszűrés és törlést valósít meg, ráadásul a konténeren belüli mozgatás is sokkal költséghatékonyabb. Gondoljunk csak azokra az algoritmusokra, amelyek intenzív elem mozgatást végeznek (rendező algoritmusok). Az egyszeresen láncolt listának is meg van azonban a hátránya a fentebb említett három tárolóval szemben. Az elemekhez történő közvetlen hozzáférés itt sem biztosított, így, ha például a 10. helyen szereplő elem értékét szeretnénk elérni, akkor azt kizárólag a lista elejéről kezdve minden egyes elemén végig iterálva tehetjük meg, továbbá a következő elemre mutató tárhely igénye nagy listák esetében nagy memória költséget jelenthet. Az egyszeresen láncolt lista deklarálásának általános szintaktikája:

```
forward_list< ObjectType > ForwardListName;
```

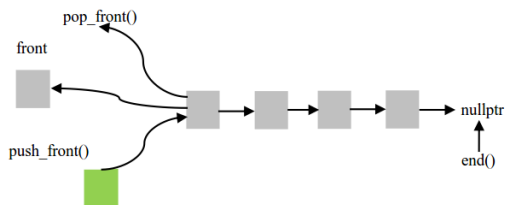
Egy kétvégű sort a következő módon hozhatunk létre:

```
#include <forward_list>
int main()
{
    std::forward_list<int> ForwardListName{1, 2, 3};
}
```



Az egyszerűen láncolt lista sematikus ábrája elem hozzáfűzése és törlése esetén

Az egyszerűen láncolt lista esetében, összehasonlítva a kétszeresen láncolt listával, nem értelmezett a **back()** és a **push_back()** tagfüggvény. Az egyszerűen láncolt lista elérhető adattagjai, tagfüggvényei és iterátorai, illetve azok működés elvei csaknem teljesen megegyeznek a kétszeresen láncolt lista tárolóknál olvasottakkal, ezért itt most nem kerülnek külön részletezésre, azok egy összefoglalását a fentebb tekinthetjük át.



Az egyszerűen láncolt lista néhány műveletének ábrája

Elem hozzáférés	Általános, rövid leírás
front	Az első elemre mutató referenciát adja vissza.
Kapacitás	Általános, rövid leírás
empty	Üresség vizsgálata. Logikai igaz értékkel tér vissza, ha a lista tároló üres, ellenkező esetben logikai hamis lesz az érték.
max_size	A lehetséges, maximális elemszám.
Módosítók	Általános, rövid leírás
assign	Új tartalmat rendel a sorhoz.
emplace_front	Az első pozícióba egy új elemet helyez.
emplace_after	A specifikusan meghatározott pozíció után egy elemet szúr be.
push_front	Elemet szúr be a sor elejére.
pop_front	A sor első elemét törli.
insert_after	A specifikusan meghatározott pozíció után egy elemet szúr be.
erase_after	Elemek törlése a sorból, megadott, specifikus pozíció után vagy tartományból.
swap	Két sor tartalmát megcseréli.
resize	A sor átméretezése.
clear	Elemek törlése a sorból.
Bejárók	Általános, rövid leírás
end	A sor egy tetszőleges elérésű bejárójával tér vissza, amely az utolsó utáni helyre (<i>past-the-end</i>) mutat.
before_begin	A sor egy tetszőleges elérésű bejárójával tér vissza, amely az első előtti helyre mutat.

begin	A sor egy tetszőleges elérésű bejárójával tér vissza, amely az első helyre mutat.
cend	A sor egy konstans bejárójával tér vissza, amely az utolsó utáni helyre (<i>past-the-end</i>) mutat.
cbefore_begin	A sor egy konstans bejárójával tér vissza, amely az első előtti helyre mutat.
cbegin	A sor egy konstans bejárójával tér vissza, amely az első helyre mutat.

Műveletek	Általános, rövid leírás
splice_after	Több előfordulása ismert. Az első változat során a függvényben megadott sor elemeit átmozgatja az aktuális objektumba. Egy második változat esetében a sor harmadik függvény paramétereként megadott iterátor által mutatott elemét mozgatja át. A harmadik változat során a sor harmadik függvény paramétereként megadott bejárója és a negyedikként megadott bejárója által bezárt félig nyílt tartományt mozgatja át.
remove	A függvény paramétereként megadott értékeket eltávolítja a sorból.
remove_if	Olyan elemeket távolít el a sorból, amelyek a megadott feltételeknek eleget tesznek.
unique	Elsősorban rendezett sor esetén lehet hasznos, mivel egymás melletti azonos értékű elemek közül csak 1db-ot hagy meg.
merge	Két rendezett sort fűszöl össze. A művelet vége után a paraméterben átadott sor üres lesz.
sort	A sor elemeit úgy rendezi, hogy a bejárók érvényesek maradnak.
reverse	Úgy fordítja meg a sor elemek sorrendjét, hogy a bejárók is érvényesek maradnak.

Egyszeresen láncolt lista objektum műveletek

Minta példa az egyszerűen láncolt lista (**forward_list**) tároló használatára Visual Studio Community fejlesztőkörnyezetben

Írjunk egy `ForwardListDemo` (*ForwardListDemo*) osztályt, amely különböző, egyszerűen láncolt lista specifikus műveleteket ír le, majd egy objektumpéldány segítségével ezeket a műveleteket hajtjuk végre. Minden egyes tagfüggvény feladatának megfogalmazását, zárójelben követi az adott tárolóhoz tartozó tagfüggvény, amelyeket kötelező jelleggel kell használni a feladat helyes megoldásának elérése érdekében, például a konstruktor esetében ilyen a **`push_back()`** metódus. Az osztály adatai a következők:

- *hexanacci*, egy egyszerűen láncolt lista jellegű szekvenciális tároló, amely egész szám típusú adatok tárolására képes.

Ügyeljünk az egységbezárás elvére, így a fentebb megfogalmazott adat(ok) **`private`**, az alábbi tagfüggvények **`public`** láthatósági szintekkel bírnak!

Valósítsuk meg az osztály metódusait:

- Írjuk meg az osztály konstruktorát, amely a hexanacci számsorozat néhány értékével inicializálja az egyszerűen láncolt lista tárolót. (**`push_back()`**)
- Írjunk egy *`isEmpty()`* függvényt, amely megvizsgálja, hogy az egyszerűen láncolt lista tartalmaz-e elemeket, majd annak függvényében visszatér egy logikai értékkel. Igazzal, ha található elem az egyszerűen láncolt listában, hamissal, ha üres. (**`empty()`**)
- Írjunk egy *`getMaxSize()`* függvényt, amely az egyszerűen láncolt lista maximális méretének értékével tér vissza. (**`max_size()`**)
- Írjunk egy *`assignValues()`* függvényt, amely az egyszerűen láncolt listát újra feltölti 7 hosszan 100-as értékekkel. (**`assign()`**)
- Írjunk egy *`emplaceFront(int value)`* függvényt, amely az egyszerűen láncolt lista elejéhez hozzáfüzi a paraméterben kapott számértéket. (**`emplace_front()`**)
- Írjunk egy *`addElementAtBeginningOfList(int value)`* függvényt, amely a lista elejéhez fűz egy új elemet a függvény paramétereiként kapott értékkel. (**`push_front()`**)
- Írjunk egy *`deleteFirstElement()`* függvényt, amely az egyszerűen láncolt lista elejéről törli az első elemet. (**`pop_front()`**)
- Írjunk egy *`emplaceAfter(int value)`* függvényt, amely az egyszerűen láncolt lista végéhez hozzáfüzi a paraméterben kapott számértéket.

(emplace_back())

- Írjunk egy `insertElementAfter(int index, int value)` függvényt, amely az egyszerűen láncolt lista elejétől, a függvény paramétereként kapott `index` határig beszúrja a paraméterként kapott `value` értéket. **(insert_after())**
- Írjunk egy `eraseElementsAfter()` függvényt, amely az egyszerűen láncolt lista elejétől, **(begin())** a függvény paramétereként megkapott `index` határig, törli az értékeket. **(erase_after())**
- Írjunk egy `resize(int size)` függvényt, amely paraméterül vár egy egész szám típusú értéket, majd ezzel az értékkel átméretezi a sort. **(resize())**
- Írjunk egy `clear()` függvényt, amely az egyszerűen láncolt lista elemeit törli. **(clear())**
- Írjunk egy `getFront()` függvényt, amely az egyszerűen láncolt lista első elemének értékével tér vissza. **(front())**
- Írjunk egy `removeSpecificElement()` függvényt, amely az egyszerűen láncolt lista 99-es értékét törli. **(remove())**
- Írjunk egy `removeIfLessThan20()` függvényt, amely az egyszerűen láncolt lista azon elemeit törli, amelyek kisebbek, mint 20. **(remove_if())**
- Írjunk egy `removeDuplicateValue()` függvényt, amely a kétszeresen láncolt lista duplum értékeit törli. **(unique())**
- Írjunk egy `sortList()` függvényt, amely az egyszerűen láncolt lista elemeit rendezi. **(sort())**
- Írjunk egy `reverseList()` függvényt, amely az egyszerűen láncolt lista elemek sorrendjének megfordítását hajtja végre. **(reverse())**
- Írjunk egy `printContent()` függvényt, amely kiírja egy konstans iterátor felhasználásával az egyszerűen láncolt lista minden egyes elemét, a **cbegin()** és a **cend()** metódusok felhasználásával.
- 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 szükség az objektum példányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. Figyelem az **auto** kulcsszóval a fordítóra is bízhatjuk az iterátorok típusának felismerését! Mielőtt a feladatnak neki látnánk, tervezzük meg az osztály- és szekvencia diagramot. **Figyelem! A könyvben ábrázolt**

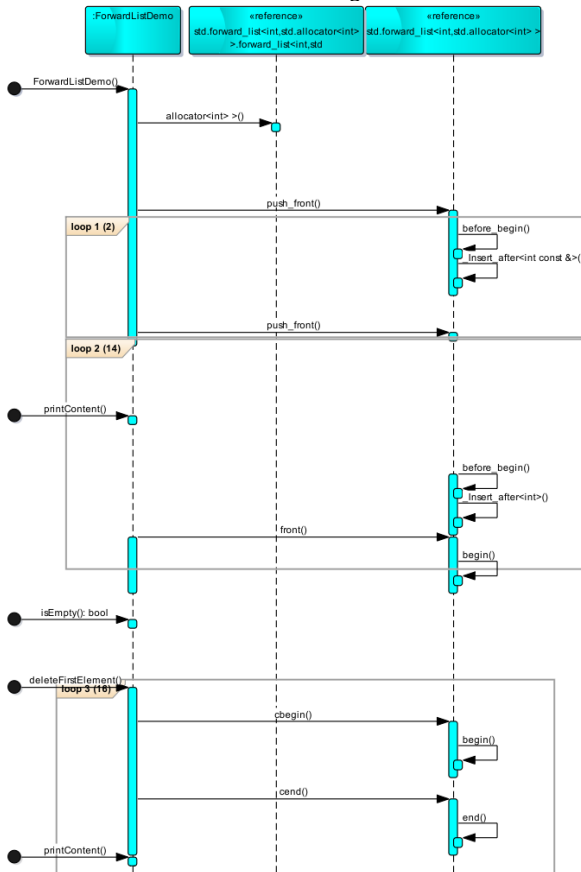
szekvenciadiagramok terjedelmi korlátok miatt a teljesség igénye nélkül készültek!

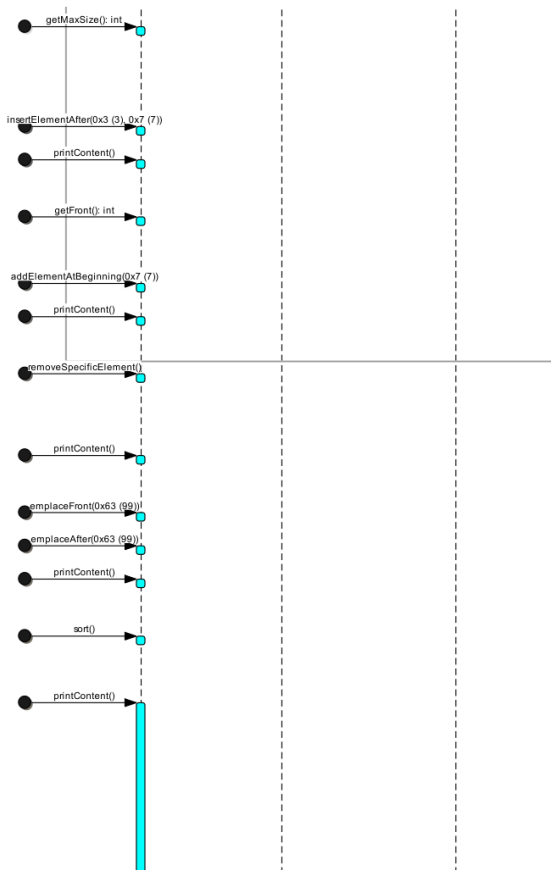
Statikus tervezés

ForwardListDemo
- hexanacci: std::forward_list<int>
+ addElementAtBeginning(int): void
+ assignValues(): void
+ clear(): void
+ deleteFirstElement(): void
+ emplaceAfter(int): void
+ emplaceFront(int): void
+ eraseElementsAfter(): void
+ ForwardListDemo()
+ ~ForwardListDemo()
+ getFront(): int
+ getMaxSize(): int
+ insertElementAfter(int, int): void
+ isEmpty(): bool
+ printContent(): void
+ removeDuplicateValue(): void
+ removeIfLessThan20(): void
+ removeSpecificElement(): void
+ resize(int): void
+ reverse(): void
+ sort(): void

Osztálydiagram

Dinamikus diagram



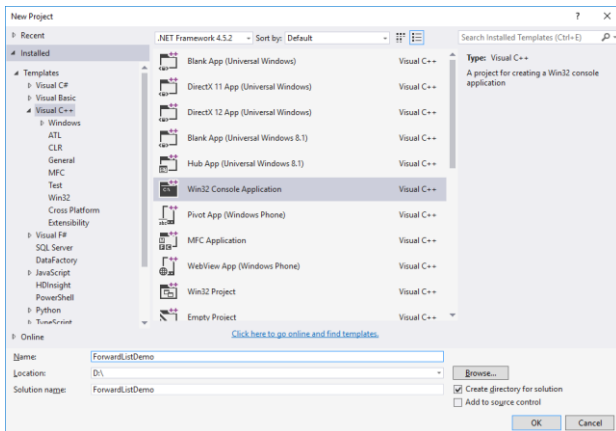


Néhány tagfüggvény szekvenciadiagramja

A projekt létrehozása

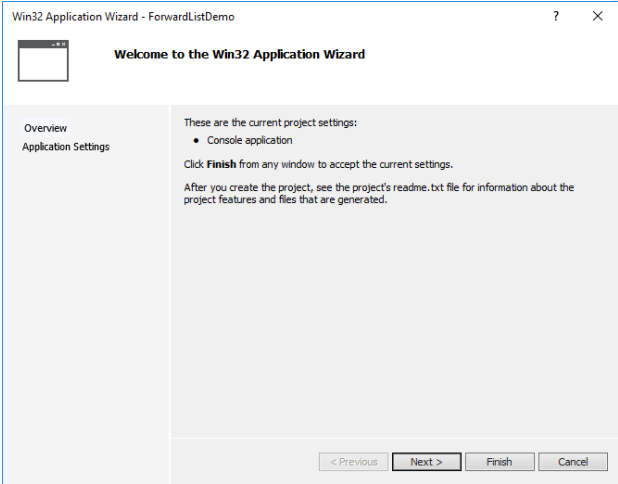
A projekt megvalósítása során konzolalkalmazást készítünk, ahol a Visual Studio, szabványos C++-nak megfelelő szintaktikát alkalmaz. A projekt három fájlban kerül implementálásra. Az alkalmazás belépési pontja a **main** függvény, amely a három fájl közül az egyikben fog helyet foglalni. Az előzőekben 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 közül a **Visual C++ Win32** sablont (**Template**) kell kiválasztani és a **Win32 Console Application**-t, mely az alábbi ábrán látható.



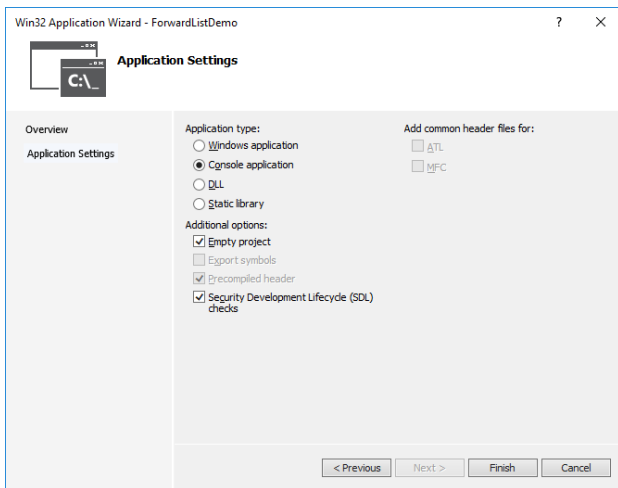
Win32 Console Application projekt létrehozása

Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően a **ForwardListDemo** 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.



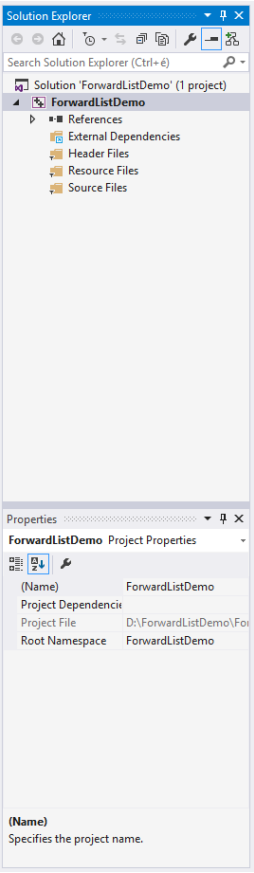
Win32 konzol alkalmazás varázsló

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 látható.

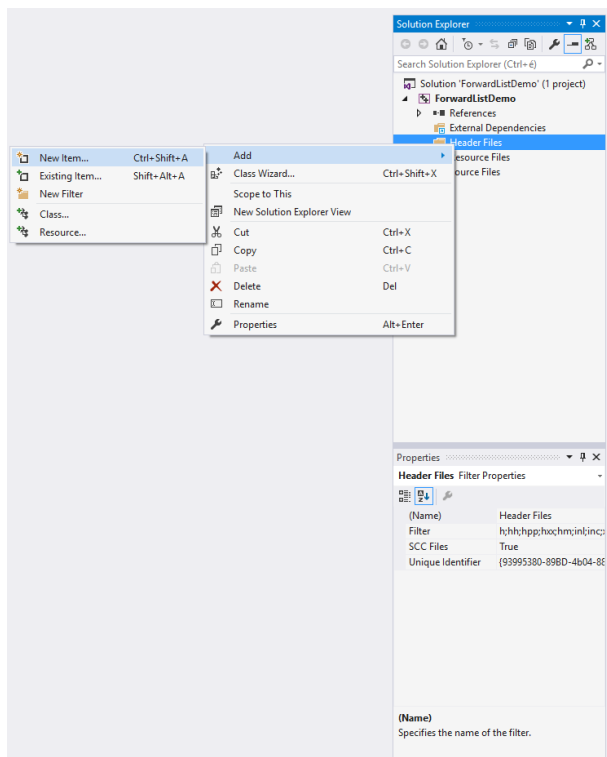


Alkalmazás beállításai

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

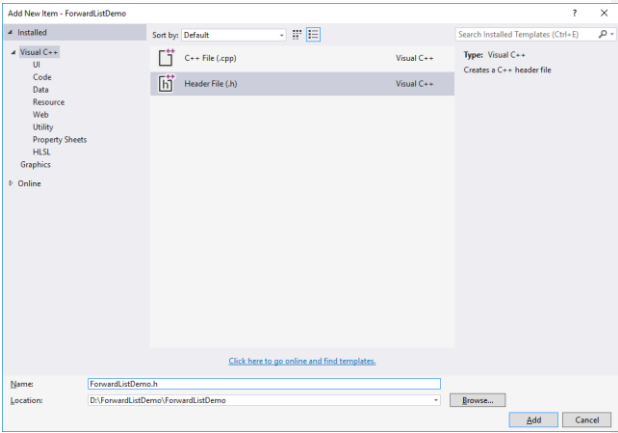


Megoldás és beállítások ablak



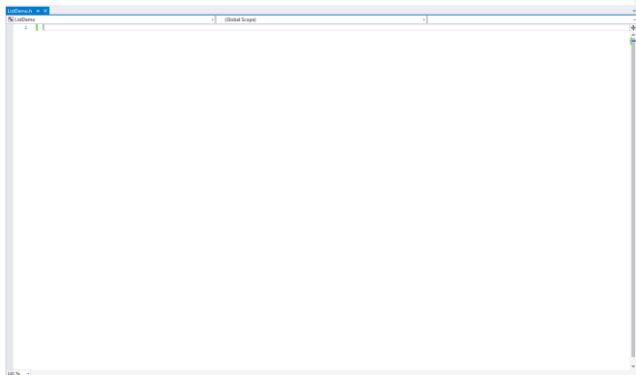
Új header fájl hozzáadása a projekthez

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



Új header fájl hozzáadása a projekthez, név megadása

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



Forráskód szerkesztő

ForwardListDemo.h

```
#ifndef FORWARDLISTDEMO_H
#define FORWARDLISTDEMO_H

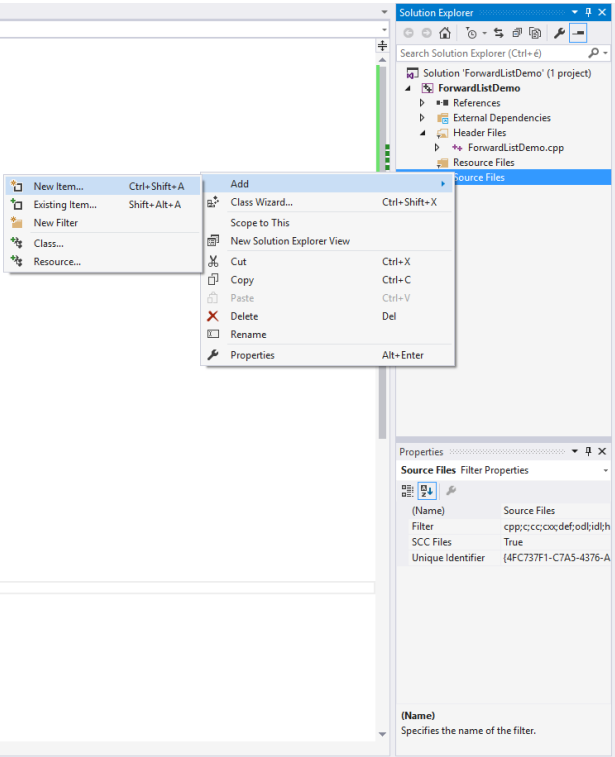
#include <forward_list>

class ForwardListDemo
{
private:
    std::forward_list<int> hexanacci;

public:
    ForwardListDemo();
    int getMaxSize();
    bool isEmpty();
    void assignValues();
    void emplaceFront(int value);
    void addElementAtBeginning(int value);
    void deleteFirstElement();
    void emplaceAfter(int value);
    void insertElementAfter(int index, int value);
    void eraseElementsAfter();
    void resize(int value);
    void clear();
    int getFront();
    void removeSpecificElement();
    void removeIfLessThan20();
    void removeDuplicateValue();
    void sort();
    void reverse();
    void printContent();
    ~ForwardListDemo();
};

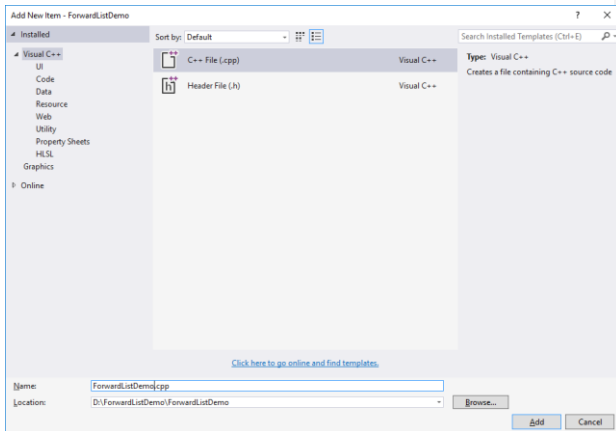
#endif
```

A deklarált osztályok megvalósításait a **ForwardListDemo.cpp** forrásfájl tartalmazza, melyeket a **header** fájlokhoz hasonlóan, de forrásfájl (**Source File**) hozzáadásával hozhatunk létre.



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása

ForwardListDemo.cpp

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

ForwardListDemo::ForwardListDemo()
{
    int firstTermOfHexanacciSequence = 0,
        secondTermOfHexanacciSequence = 1,
        thirdTermOfHexanacciSequence = 1,
        fourthTermOfHexanacciSequence = 2,
        fifthTermOfHexanacciSequence = 4,
        sixthTermOfHexanacciSequence = 8;
    for (unsigned int i = 0; i <= 15; ++i)
    {
        if (i <= 1)
        {
            hexanacci.push_front(1);
        }
        else
        {
            hexanacci.push_front(firstTermOfHexanacciSequence +
                secondTermOfHexanacciSequence +
                thirdTermOfHexanacciSequence +
                fourthTermOfHexanacciSequence +
                fifthTermOfHexanacciSequence +
                sixthTermOfHexanacciSequence);
            firstTermOfHexanacciSequence = secondTermOfHexanacciSequence;
            secondTermOfHexanacciSequence = thirdTermOfHexanacciSequence;
            thirdTermOfHexanacciSequence = fourthTermOfHexanacciSequence;
            fourthTermOfHexanacciSequence = fifthTermOfHexanacciSequence;
            fifthTermOfHexanacciSequence = sixthTermOfHexanacciSequence;
            sixthTermOfHexanacciSequence = hexanacci.front();
        }
    }
}

bool ForwardListDemo::isEmpty()
{
    return hexanacci.empty();
}

int ForwardListDemo::getMaxSize()
{
    return hexanacci.max_size();
}

void ForwardListDemo::assignValues()
{
    hexanacci.assign(7, 100);
}

void ForwardListDemo::emplaceFront(int value)
{
    hexanacci.emplace_front(value);
}

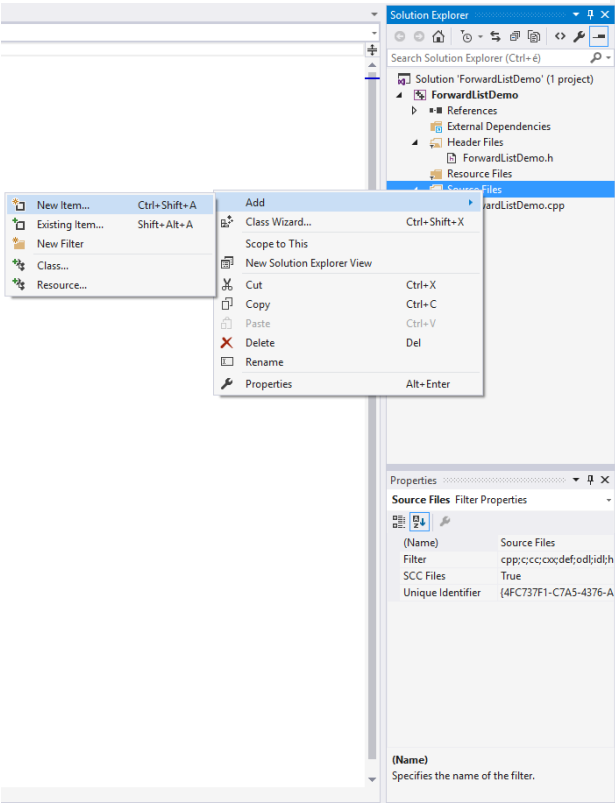
void ForwardListDemo::addElementAtBeginning(int value)
```

```

{
    hexanacci.push_front(value);
}
void ForwardListDemo::deleteFirstElement()
{
    hexanacci.pop_front();
}
void ForwardListDemo::emplaceAfter(int value)
{
    hexanacci.emplace_after(hexanacci.before_begin(), value);
}
void ForwardListDemo::insertElementAfter(int index, int value)
{
    hexanacci.insert_after(hexanacci.before_begin(), index, value);
}
void ForwardListDemo::eraseElementsAfter()
{
    hexanacci.erase_after(hexanacci.begin(), hexanacci.end());
}
void ForwardListDemo::resize(int value)
{
    hexanacci.resize(value);
}
void ForwardListDemo::clear()
{
    hexanacci.clear();
}
int ForwardListDemo::getFront()
{
    return hexanacci.front();
}
void ForwardListDemo::removeSpecificElement()
{
    hexanacci.remove(99);
}
bool lessThan20(const int& value) { return (value < 20); }
void ForwardListDemo::removeIfLessThan20()
{
    hexanacci.remove_if(lessThan20);
}
void ForwardListDemo::removeDuplicateValue()
{
    hexanacci.unique();
}
void ForwardListDemo::sort()
{
    hexanacci.sort();
}
void ForwardListDemo::reverse()
{
    hexanacci.reverse();
}
void ForwardListDemo::printContent()

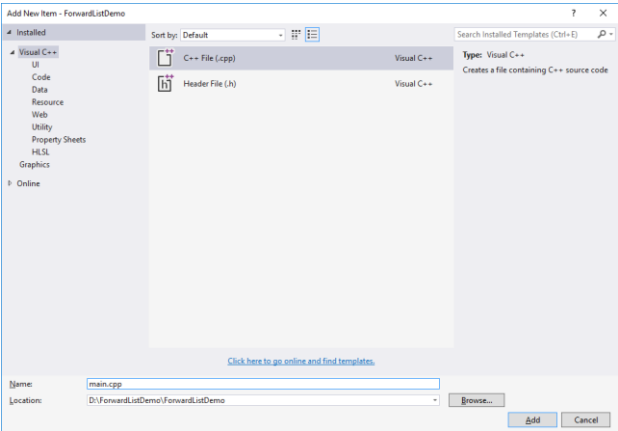
```

```
for (std::forward_list<int>::const_iterator constIterator =  
    hexanacci.cbegin(); constIterator != hexanacci.cend(); ++constIterator)  
{  
    std::cout << ' ' << *constIterator;  
}  
}  
ForwardListDemo::~ForwardListDemo() {}
```



Új forrásfájl hozzáadása a projekthez

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



Új forrásfájl hozzáadása a projekthez, név megadása

main.cpp

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

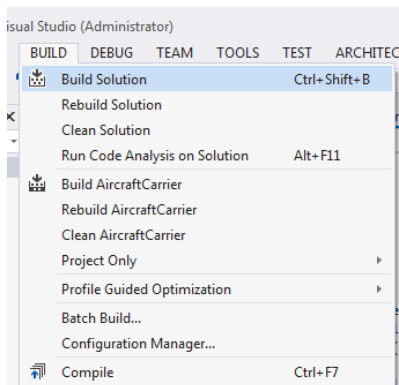
int main()
{
    std::cout << "MUVELET => ForwardListDemo() #konstruktor hivasa, a
    forward_list tarolo feltoltese" << std::endl;
    ForwardListDemo *ForwardListDemoObject = new ForwardListDemo();
    std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
    std::cout << std::endl;
    std::cout << "MUVELET => empty() #uresseg vizsgalat..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << ForwardListDemoObject->isEmpty()
    << std::endl;
    std::cout << "MUVELET => pop_front() #also elem torlese..." << std::endl;
    ForwardListDemoObject->deleteFirstElement();
    std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
    std::cout << std::endl;
    std::cout << "MUVELET => max_size() #tarolo maximalis meretenek
    lekerdezese..." << std::endl;
    std::cout << "EREDMENY: "; std::cout << ForwardListDemoObject-
    >getMaxSize(); std::cout << std::endl;
    std::cout << "MUVELET => insert() #beszuras..." << std::endl;
    ForwardListDemoObject->insertElementAfter(3, 7);
    std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
    std::cout << std::endl;
    std::cout << "MUVELET => front() #also elem lekerdezese..." <<
    std::endl;
    std::cout << "EREDMENY: "; std::cout << ForwardListDemoObject-
    >getFront() << std::endl;
    std::cout << "MUVELET => push_front() #hozzafuzes a tarolo elejehez..."
    << std::endl;
    ForwardListDemoObject->addElementAtBeginning(7);
    std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
    std::cout << std::endl;
    std::cout << "MUVELET => remove() #egy specifikus eleme torlese..." <<
    std::endl;
    ForwardListDemoObject->removeSpecificElement();
    std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
    std::cout << std::endl;
    std::cout << "MUVELET => emplace_front() #hozzafuzes a tarolo
    elejehez..." << std::endl;
    ForwardListDemoObject->emplaceFront(99);
    std::cout << "MUVELET => emplace_back() #hozzafuzes a tarolo vegehez..."
    << std::endl;
    ForwardListDemoObject->emplaceAfter(99);
    std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
    std::cout << std::endl;
    std::cout << "MUVELET => sort() #rendezes..." << std::endl;
    ForwardListDemoObject->sort();
    std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
    std::cout << std::endl;
```

```

std::cout << "MUVELET => reverse() #sorrend megforditasa..." <<
std::endl;
ForwardListDemoObject->reverse();
std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
std::cout << std::endl;
std::cout << "MUVELET => unique() #duplikatumok torlese..." <<
std::endl;
ForwardListDemoObject->removeDuplicateValue();
std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
std::cout << std::endl;
std::cout << "MUVELET => remove() #20-nal kisebb elemek torlese..." <<
std::endl;
ForwardListDemoObject->removeIfLessThan20();
std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
std::cout << std::endl;
std::cout << "MUVELET => clear() #elemek torlese..." << std::endl;
ForwardListDemoObject->clear();
std::cout << "MUVELET => assign() #(ujra)feltoltese..." << std::endl;
ForwardListDemoObject->assignValues();
std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
std::cout << std::endl;
std::cout << "MUVELET => erase_after() #elem torles..." << std::endl;
ForwardListDemoObject->eraseElementsAfter();
std::cout << "ALLAPOT: "; ForwardListDemoObject->printContent();
std::cout << std::endl;
ForwardListDemoObject->resize(20);
delete ForwardListDemoObject;
_getch();
return 0;
}

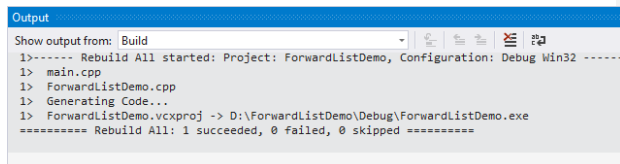
```

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.



Program fordítása

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



Kimenet ablak

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

```
D:\ForwardListDemo\Debug\ForwardListDemo.exe
MOVELET -> ForwardListDemo() #konstruktor hívása, a forward_list tároló feltöltése
ALLAPOT: 117910 59448 29970 15109 7617 3840 1936 976 492 248 125 63 32 16 1 0
MOVELET -> empty() #tárolás ülszágát...
EREDMENY: 0
MOVELET -> pop_front() #első elem törlése...
ALLAPOT: 59448 29970 15109 7617 3840 1936 976 492 248 125 63 32 16 1 0
MOVELET -> max_size() #tároló maximális méretének lekérdezése...
EREDMENY: 538476915
MOVELET -> insert() #beszúrás...
ALLAPOT: 7 7 7 59448 29970 15109 7617 3840 1936 976 492 248 125 63 32 16 1 0
MOVELET -> front() #első elem lekérdezése...
EREDMENY: 7
MOVELET -> push_front() #hozzáfűzés a tároló elejéhez...
ALLAPOT: 7 7 7 7 59448 29970 15109 7617 3840 1936 976 492 248 125 63 32 16 1 0
MOVELET -> remove() #egy specifikus elem törlése...
ALLAPOT: 7 7 7 59448 29970 15109 7617 3840 1936 976 492 248 125 63 32 16 1 0
MOVELET -> emplace_front() #hozzáfűzés a tároló elejéhez...
MOVELET -> emplace_back() #hozzáfűzés a tároló végéhez...
ALLAPOT: 99 99 7 7 7 59448 29970 15109 7617 3840 1936 976 492 248 125 63 32 16 1 0
MOVELET -> sort() #rendezés...
ALLAPOT: 0 1 7 7 7 16 32 63 99 99 125 248 492 976 1936 3840 7617 15109 29970 59448
MOVELET -> reversed() #sorrend megfordítása...
ALLAPOT: 59448 29970 15109 7617 3840 1936 976 492 248 125 99 99 63 32 16 7 7 7 1 0
MOVELET -> unique() #duplikátumok törlése...
ALLAPOT: 59448 29970 15109 7617 3840 1936 976 492 248 125 99 63 32 16 7 1 0
MOVELET -> remove() #20-nál kisebb elemek törlése...
ALLAPOT: 59448 29970 15109 7617 3840 1936 976 492 248 125 99 63 32
MOVELET -> clear() #elemek törlése...
MOVELET -> assign() #(ujra)töltés...
ALLAPOT: 100 100 100 100 100 100 100 100
MOVELET -> erase_after() #elem törlése...
ALLAPOT: 100
```

A program eredménye

A feladathoz kapcsolódó magyarázatok

Az osztály deklarációját a (**ForwardListDemo.h**) állományban adjuk meg. A deklarációban egy magán (**private**) jellegű, egész szám típusú (**integer**) adatok tárolására képes egyszerűen láncolt lista került megfogalmazásra, amelyet az **std** névtéren keresztül értünk el:

```
private:
    std::forward_list<int> hexanacci;
```

A konstruktor neve tulajdonságaiból fakadóan megegyezik az osztály nevével, láthatósági szintje nyilvános (**public**), visszatérési értéke nincs, még **void** sem, továbbá az adott feladatban paraméter nélküli formában jelenik meg:

```
public:
    ForwardListDemo();
```

A **getMaxSize()**, **getFront()** tagfüggvények a leírásban megfogalmazott egyszerűen láncolt lista maximális méretének, valamint az első elemének lekérdezését biztosítják. A láthatósági szintjük nyilvános (**public**), visszatérési típusuk egész szám (**integer**) és argumentumlistájuk üres:

```
int getMaxSize();
int getFront();
```

Az `isEmpty()` tagfüggvény az egyszerűen láncolt lista ürességének vizsgálatát hajtja végre, láthatósági szintje nyilvános (**public**), visszatérési típusa logikai (**bool**), argumentumlistája üres:

```
bool isEmpty();
```

A `resize(int value)` tagfüggvény az egyszerűen láncolt lista aktuális méretének átméretezését végzi, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték az egyszerűen láncolt lista jövőben használni kívánt méretének értéke:

```
void resize(int value);
```

A `clear()` tagfüggvény az egyszerűen láncolt lista elemeit törli, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void clearList();
```

Az `assignValues()` tagfüggvény iterátorokkal megadott tartományba helyez be elemeket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void assignValues();
```

Az `addElementAtBeginning(int value)` tagfüggvény az első pozícióba a paraméterül kapott elemet fűzi az egyszerűen láncolt listához, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely a beszúrni kívánt érték:

```
void addElementAtBeginning(int value);
```

A `deleteFirstElement()` tagfüggvény az egyszerűen láncolt lista első elemét törli, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void deleteFirstElement();
```

Az `insertElementAfter(int index, int value)` tagfüggvény egy megadott pozícióba n alkalommal beszúr egy értéket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), valamint két egész szám típusú (**integer**) értéket vár paraméterül, ahol az első paraméter határozza meg a beszúrandó tartomány hosszát, a második argumentum a beszúrni kívánt értéket:

```
void insertElementAfter(int index, int value);
```

Az `eraseElementsAfter()` tagfüggvény egy megadott pozíciótól n hosszan törli az értéket, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void eraseElementsAfter();
```

Az `emplaceFront(int value)` tagfüggvény az első pozícióba új elemet fűz az egyszerűen láncolt listához, visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték az egyszerűen láncolt listához hozzáfűzni kívánt érték:

```
void emplaceFront(int value);
```

Az `emplaceBack(int value)` tagfüggvény az utolsó utáni pozícióba új elemet fűz az egyszerűen láncolt listához, visszatérési típusa nincs (**void**), valamint egy egész szám típusú (**integer**) értéket vár paraméterül, amely érték az egyszerűen láncolt listához hozzáfűzni kívánt érték:

```
void emplaceBack(int value);
```

A `removeSpecificElement()` tagfüggvény egy specifikus elemet töröl az egyszerűen láncolt listából, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void removeSpecificElement();
```

A `removeIfLessThan20()` tagfüggvény a 20-tól kisebb értékeket törli az egyszerűen láncolt listából, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void removeIfLessThan20();
```

A `removeDuplicateValue()` tagfüggvény a duplumokat törli az egyszerűen láncolt listából, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void removeDuplicateValue();
```

A `sort()` tagfüggvény sorba rendezi az egyszerűen láncolt lista elemeit, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void sort();
```

A `reverseList()` tagfüggvény az egyszerűen láncolt listában az elemek aktuális sorrendjét megfordítja, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void reverseList();
```

A `printContent()` tagfüggvény konstans típusú iterátor segítségével megjeleníti a konzolon az egyszerűen láncolt lista aktuális elemeit, láthatósági szintje nyilvános (**public**), visszatérési típusa nincs (**void**), argumentumlistája üres:

```
void printContent();
```

A **ForwardListDemo.cpp** állomány tartalmazza az osztály teljeskörű implementációját. Az implementáció során az egyes tagfüggvények neve előtt a `::` (hatókör) operátor került alkalmazásra. azon célból, hogy pontosan meghatározzuk melyik osztály tagfüggvényét valósítottuk meg.

A konstruktor feladata, hogy az adattagokat kezdőértékkel inicializálja. A *hexanacci* azonosítójú egyszerűen láncolt lista adattag értékül kapja a hexanacci számsorozat első néhány értékét. A kezdőérték adásnál a **push_front()** módosító függvényt alkalmaztuk.

```

ForwardListDemo::ForwardListDemo()
{
    int firstTermOfHexanacciSequence = 0,
    secondTermOfHexanacciSequence = 1,
    thirdTermOfHexanacciSequence = 1,
    fourthTermOfHexanacciSequence = 2,
    fifthTermOfHexanacciSequence = 4,
    sixthTermOfHexanacciSequence = 8;
    for (unsigned int i = 0; i <= 15; ++i)
    {
        if (i <= 1)
        {
            hexanacci.push_front(i);
        }
        else
        {
            hexanacci.push_front(firstTermOfHexanacciSequence +
            secondTermOfHexanacciSequence +
            thirdTermOfHexanacciSequence +
            fourthTermOfHexanacciSequence +
            fifthTermOfHexanacciSequence +
            sixthTermOfHexanacciSequence);
            firstTermOfHexanacciSequence = secondTermOfHexanacciSequence;
            secondTermOfHexanacciSequence = thirdTermOfHexanacciSequence;
            thirdTermOfHexanacciSequence = fourthTermOfHexanacciSequence;
            fourthTermOfHexanacciSequence = fifthTermOfHexanacciSequence;
            fifthTermOfHexanacciSequence = sixthTermOfHexanacciSequence;
            sixthTermOfHexanacciSequence = hexanacci.front();
        }
    }
}

```

Az aktuális méret, valamint az első elem lekérdezését végrehajtó függvények implementációja során a **hexanacci** objektumon keresztül egyszerűen hívásra kerültek a **size()**, illetve a **front()** metódusok.

```

int ForwardListDemo::getMaxSize()
{
    return hexanacci.max_size();
}
int ForwardListDemo::getFront()
{
    return hexanacci.front();
}

```

Az üresség vizsgálata során a méret lekérdezéséhez hasonlóan a **hexanacci** objektumon keresztül hívjuk meg a szükséges tagfüggvényt, amely ebben az esetben az **empty()** metódus.

```
bool ForwardListDemo::isEmpty()
{
    return hexanacci.empty();
}
```

A méret módosító függvények közül az újra méretezéshez a **resize()**-t használtuk, amely a paraméterben kapott nagyságú értékre állította az egyszerűen láncolt lista aktuális méretét.

```
void ForwardListDemo::resize(int value)
{
    hexanacci.resize(value);
}
```

Az egyszerűen láncolt lista elemeinek törlését, a **clear()** tagfüggvény hívásával hajthatjuk végre.

```
void ForwardListDemo::clear()
{
    hexanacci.clear();
}
```

Az egyszerűen láncolt lista módosító függvények közül az **assign()** alkalmazása során a 100-as értéket N-szer helyeztük az egyszerűen láncolt listába, ahol az N=7-al.

```
void ForwardListDemo::assignValues()
{
    hexanacci.assign(7, 100);
}
```

Az egyszerűen láncolt lista első pozíciójába új elemet szúrhatunk be a **push_front()** és **emplace_front()** tagfüggvény segítségével, amíg az utolsó pozíciójába új elemet az **emplace_after()** segítségével helyezhetünk.

```

void ForwardListDemo::addElementAtBeginning(int value)
{
    hexanacci.push_front(value);
}
void ForwardListDemo::emplaceFront(int value)
{
    hexanacci.emplace_front(value);
}
void ForwardListDemo::emplaceAfter(int value)
{
    hexanacci.emplace_after(hexanacci.before_begin(), value);
}

```

Az egyszerűen láncolt lista első elemének eltávolításához a **pop_front()** tagfüggvényt használtuk.

```

void ForwardListDemo::deleteFirstElement()
{
    hexanacci.pop_front();
}

```

Az egyszerűen láncolt listába történő beszúrást az **insert_after()** tagfüggvény segítségével végeztük el ahol az első paraméter az egyszerűen láncolt list elejét jelzi (**before_begin()**), a második azt az index helyet, ameddig a beszúrás kívánjuk elvégezni, amíg a harmadik a beszúrandó értéket.

```

void ForwardListDemo::insertElementAfter(int index, int value)
{
    hexanacci.insert_after(hexanacci.before_begin(), index, value);
}

```

Az **erase_after()** tagfüggvényt felhasználva a paraméterben megadott tartományból töröltük az elemeket, ahol a tartomány az egyszerűen láncolt lista elejétől kezdődik és a függvény paraméterében átadott hosszig tart.

```

void ForwardListDemo::eraseElementsAfter()
{
    hexanacci.erase_after(hexanacci.begin(), hexanacci.end());
}

```

Specifikus elem törlését a **remove()** függvénnyel valósítottuk meg.

```

void ForwardListDemo::removeSpecificElement()
{
    hexanacci.remove(99);
}

```

Az egyszerűen láncolt listából feltételnek megfelelő érték halmaza a **remove_if()** tagfüggvénnyel távolítottunk el, a feladatban ez a feltétel a 20-tól kisebb értékek voltak. A duplumok törlését a **unique()** függvény segítségével, amíg a sorrendbe történő rendezést a **sort()** metódus segítségével, a sorrendiség megfordítását a **reverse()**-el végeztük el.

```
bool lessThan20(const int& value) { return (value < 20); }
void ForwardListDemo::removeIfLessThan20()
{
    hexanacci.remove_if(lessThan20);
}
void ForwardListDemo::removeDuplicateValue()
{
    hexanacci.unique();
}
void ForwardListDemo::sort()
{
    hexanacci.sort();
}
void ForwardListDemo::reverse()
{
    hexanacci.reverse();
}
```

Az egyszerűen láncolt lista elemek megjelenítéséhez konstans iterátor típust alkalmaztunk. A **cbegin()** tagfüggvény az egyszerűen láncolt lista elejét (első elem), addig a **cend()** az egyszerűen láncolt lista végét (utolsó utáni elem) reprezentálja egy konstans elérésű bejáróval.

```
void ForwardListDemo::printContent()
{
    for (std::forward_list<int>::const_iterator constIterator =
        hexanacci.cbegin(); constIterator != hexanacci.cend(); ++constIterator)
    {
        std::cout << ' ' << *constIterator;
    }
}
```

A **(main.cpp)** állomány tartalmazza a **main()** függvényt, a program belépési pontját. A fő függvényben történő dinamikus példányosítása után hívjuk meg a szükséges függvényeket. A műveletek végrehajtását követően, gondoskodjunk a lefoglalt erőforrások felszabadításáról az alapértelmezett implicit destruktor hívásának segítségével.

A szekvenciális tárolók összegzése

Röviden összegezve a szekvenciális tárolókat, elmondható, hogy a **vector**-t akkor érdemes használni, ha dinamikus méretű tömböt szeretnénk alkalmazni, és tudjuk azt, hogy műveletekre (beszúrás, törlés) csak a tároló végén lesz szükségünk. A **deque**-tároló használata, akkor célszerű, ha a sor elején és végén végzünk műveleteket (beszúrás, törlés). Amennyiben felmerül az igény, arra hogy ne csak a tároló végeken hajtsuk végre hatékonyan műveleteket (beszúrás, törlés), akkor a leggyorsabb megoldást a kétszeresen vagy az egyszeresen láncolt lista jelentheti, memória kapacitástól függően. Ha ezeket az egyszerű szabályokat figyelembe vesszük, akkor biztos, hogy mindig a leghatékonyabb megoldást fogjuk használni.

Logikai térkép a megfelelő STL tároló kiválasztásához²

² Forrás: Tóth Bertalan: C++ Programozás STL Konténerekkel, 2016.

Gyakorló példa

A gyakorló feladat sikeres implementálásához elengedhetetlen a könyv sorozat eddigi számaiban leírt ismeretek elsajátítása és megértése, mivel a feladat erősen épül az ott leírtakra. Az egyes függvények logikái és algoritmikus megértését néhány ábrával támogatjuk.

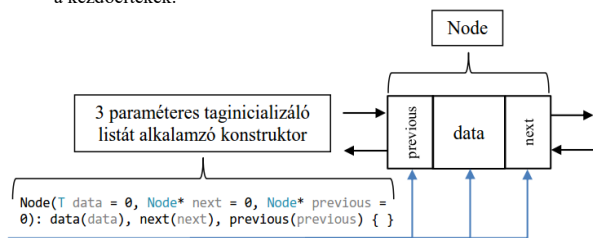
Írjunk egy kétszeresen láncolt listát szimuláló osztályt (*List*), amelynek az objektumpéldánya, az osztályban leírtak alapján, képes tetszőleges típusokkal (osztály sablon), kétszeresen láncolt listára jellemző viselkedések és műveletek elvégzésére. A feladat megvalósítása során kettő darab osztályt implementáljunk.

Az első osztály (*Node*) a lista egy elemét írja le, az adatai a következők:

- Egy sablon típusú (*T*) adattag (*data*), amely a list elemének az értékét tárolja.
- Egy *Node*<*T*> típusú adattag mutató (*previous*), amely a lista egy előző elemének címére mutat.
- Egy *Node*<*T*> típusú adattag mutató (*next*), amely a lista egy következő elemének címére mutat.

Valósítsuk meg az osztály egyetlen speciális metódusát:

- Írjuk meg az osztály konstruktorát, de ebben az esetben ne a már jól ismert konstruktor törzsében alkalmazott értékadásos módszert válasszuk, hanem az úgynevezett taginicializáló lista módszerét alkalmazzuk. Az inicializáló listát közvetlenül a konstruktor feje után kettősponttal elválasztva adjuk meg. A vesszővel tagolt lista elemei az osztály adatai (*data*, *next*, *previous*), melyeket zárójelben követnek a kezdőértékek.

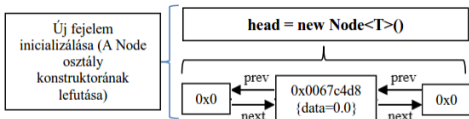


A második osztály (*List*) a kétszeresen láncolt lista viselkedését és annak lehetséges műveleteit írja le, az adatai a következők:

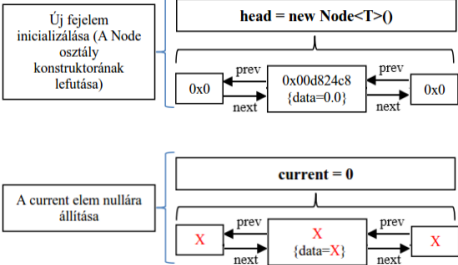
- Egy *Node<T>* típusú adattag mutató (*head*), amely a lista első elemére mutat.
- Egy *Node<T>* típusú adattag mutató (*current*), amely a lista aktuális elemére mutat.
- Egy *int* típusú adattag (*size*), amely a lista méretét tárolja.

Ügyeljünk az egységbezárás elvére, tehát az adatok **private** láthatóságú szintekkel rendelkezzenek!

- Írjuk meg az osztály paraméter nélküli konstruktorát (*List*), amely létrehoz egy üres, kétszeresen láncolt listát.

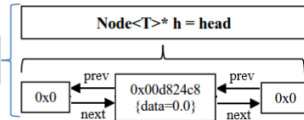


- Írjuk meg az osztály másoló konstruktorát (*List(const List<T>& object)*), amely a paraméterül kapott lista objektum elemeiről egy másolatot hoz létre a memória dinamikus területén. Az alábbi folyamat szemlélteti a szükséges másoló konstruktor működését:

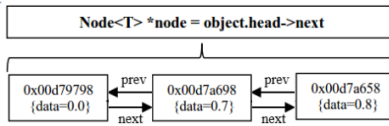


Ezt egy ciklusban addig ismételjük, amíg végig értünk a paraméterül kapott objektum elemein.

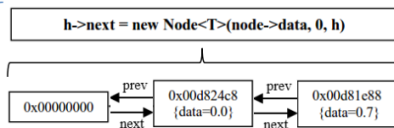
Új fejlelem inicializált értékeinek átadása



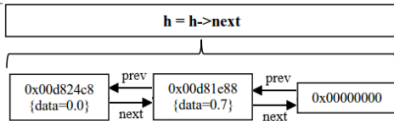
A paraméterül kapott objektumban lévő **head** elemének **next** mutatójának kinyerése.



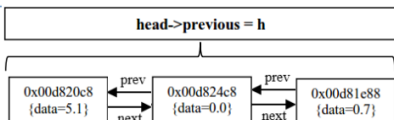
A **h** node-nak **next** mutatója egy új **node**-al lesz egyenlő, melynek **data** értéke az objektumból kinyert aktuális értékkel (0.0) egyenlő, a **previous** mutatója mindig a 0x0 helyre fog mutatni, amíg a **next** mutatója az aktuális **h** memória területére.



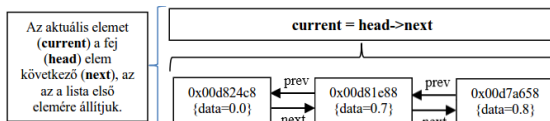
A paraméterül kapott objektumban lévő **head** elemének **next** mutatójának kinyerése.



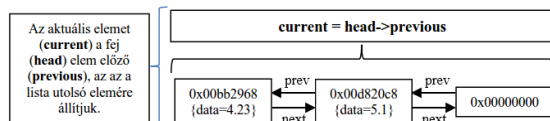
A ciklus lefutását követően a **head** **previous** mutatóját a ciklus utolsó „kinyert” értékre állítjuk.



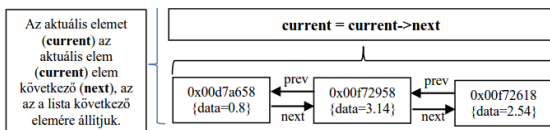
- Írjunk egy *isEmpty()* függvényt, amely a **head->next** mutató segítségével eldönti, hogy a lista üres-e vagy sem, majd ennek megfelelően térjen vissza egy logikai értékkel.
- Írjunk egy *getSize()* függvényt, amely lekérdezi az aktuális lista méretét.
- Írjunk egy *positionToFirstItem()* függvényt, amely az aktuális elemet az első elemre állítja. A program írja ki, ha a lista üres.



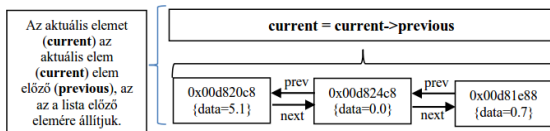
- Írjunk egy *positionToLastItem()* függvényt, amely az aktuális elemet az utolsó elemre állítja. A program írja ki, ha a lista üres.



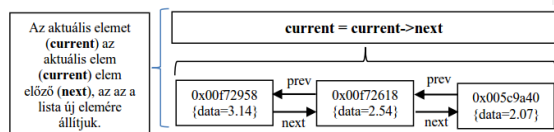
- Írjunk egy *positionToNextItem()* függvényt, amely az aktuális elemet a következő elemre állítja. A program írja ki, ha a lista üres.



- Írjunk egy *positionToPreviousItem()* függvényt, amely az aktuális elemet az előző elemre állítja. A program írja ki, ha a lista üres.

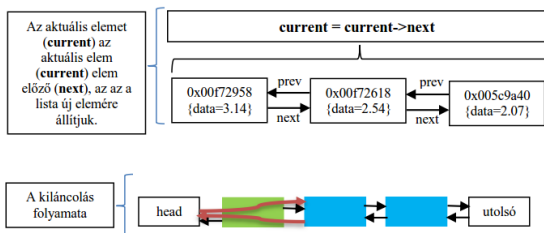


- Írjunk egy `insertItem(T newItem)` függvényt, amely a paraméterül kapott új értéket az aktuális elem után szúrja be. Ha a listában még nincs elem, akkor hívjuk meg azt a függvényt, amely az első helyre történő beszúrásért felel. Amennyiben az utolsó elem az aktuális, akkor az utolsó helyre történő beszúrásért felelős függvényt hívjuk meg. A sikeres beszúrást követően ne feledkezzünk meg a lista méretének növeléséről.



- Írjunk egy `insertItemToFirstPosition(T newItem)` függvényt, amely a paraméterül kapott új értéket az első elem helyére szúrja be. Ha a listában csak fejelem létezik, akkor az újonnan beszúrt elem a fejelemre fog visszamutatni, mindezek mellett az utolsó és az aktuális elem is ez az elem lesz, egyébként a fejelem következő mutatója az új elem referencia helyére fog mutatni és a régi első elem visszamutatóját az új első elemre kell állítani.
- Írjunk egy `insertItemToLastPosition(T newItem)` függvényt, amely a paraméterül kapott új értéket a lista végére szúrja be. Ha a listában csak fejelem létezik (nincs még elem), akkor hívjuk meg azt a függvényt, amely az első helyre szúrja be az elemet, egyébként a fejelem előző mutatója az új elem referencia helyére fog mutatni és a régi utolsó előremutatóját az új elemre kell állítani.
- Írjunk egy `modifyActualItem(T newItem)` függvényt, az aktuális értéket a paraméterül kapott új értékre állítja. Ha az aktuális érték 0, akkor írjuk ki, hogy a lista még üres.

- Írjunk egy `getCurrentValue()` függvényt, amely az aktuális lista elem értékével tér vissza. Ha az aktuális elem egyenlő nullával, akkor a lista üres.
- Írjunk egy `isItemFirst()` függvényt, amely azt vizsgálja, hogy az aktuális elem a lista első eleme-e (**head**->**next**), a vizsgálat eredményének megfelelően adjon vissza egy logikai értéket. Ha az aktuális elem egyenlő nullával, akkor a lista üres.
- Írjunk egy `isItemLast()` függvényt, amely azt vizsgálja, hogy az aktuális elem a lista utolsó eleme-e (**head**->**previous**), a vizsgálat eredményének megfelelően adjon vissza egy logikai értéket. Ha az aktuális elem egyenlő nullával, akkor a lista üres.
- Írjunk egy `removeCurrentItem()` függvényt, amely az aktuális lista elemet kiveszi a listából. Ha az aktuális elem egyenlő nullával, a lista üres, ezt írjuk ki a konzolra. Amennyiben a lista nem üres, úgy az aktuális elem értékét adjuk egy átmeneti változó értékének. Ha az aktuális elem következő mutatójának referenciája nulla, akkor hívjuk meg azt a metódus, amelynek feladata az utolsó elem eltávolítása, mivel ekkor épp az utolsó elemről állunk, egyébként az aktuális elemet adjuk egy lista elem (node) mutatónak, majd az aktuális elemet láncoljuk ki a listából. A sikeres kiláncolást követően töröljük a lista elem mutatóját, majd csökkentjük a lista tároló méretét és töröljük az átmeneti tárolót is.



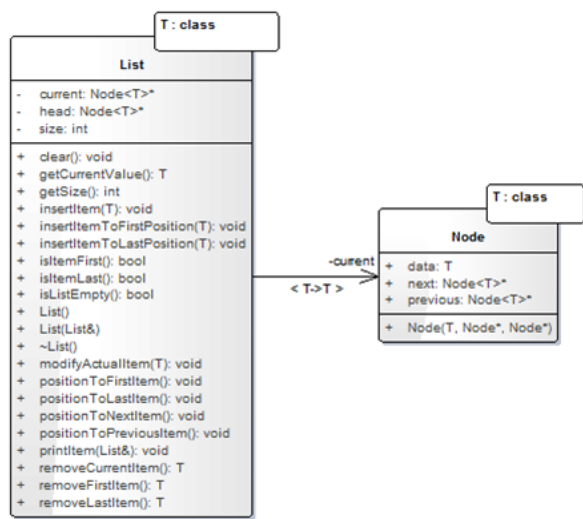
- Írjunk egy `removeFirstItem()` függvényt, amely a lista első elemét kiveszi a listából. Ha a fej következő mutató értéke nulla, a lista üres, ezt írjuk ki a konzolra. Amennyiben a lista nem üres, úgy a fej következő mutatójának adat értékét adjuk értékül egy átmeneti tárolónak. Abban az esetben, ha a lista egyelemű, akkor a fej következő

mutatójának értékét töröljük, majd állítsuk nullára a fej következő és előző mutatóját, valamint az aktuális elemet, egyébként láncoljuk ki az első elemet. Abban az esetben, ha az első elem az aktuális, akkor azt átállítjuk a fej következő mutatójának értékéül. Utolsó lépésként csökkentünk a lista méretét és töröljük az átmeneti tárolót.

- Írjunk egy *removeLastItem()* függvényt, amely a lista utolsó elemét kiveszi a listából. Ha a fej előző mutató értéke nulla, a lista üres, ezt írjuk ki a konzolra. Amennyiben a lista nem üres, úgy a fej előző mutatójának adat értékét adjuk értékül egy átmeneti tárolónak. Abban az esetben, ha a lista egyelemű, akkor hívjuk meg az első elem törléséért felelős függvényt. Egyébként láncoljuk ki az utolsó elemet. Abban az esetben, ha az utolsó elem az aktuális, akkor azt átállítjuk a fej előző mutatójának értékéül. Utolsó lépésként csökkentünk a lista méretét és töröljük az átmeneti tárolót.
- Írjunk egy *clear()* függvényt, amely törli a listából az elemeket és mindent kezdő pozícióba állít (*head->previous=0*, *head->next=0*, *size=0*).
- Írjunk egy *printItem()* függvényt, amely a paraméterül kapott lista elemeit megjeleníti a konzolon.
- Ne feledkezzünk meg az objektumpéldány által lefoglalt erőforrás felszabadításáról, a destruktorki írásáról sem.
- Írjunk egy egyszerű *main* függvényt, ahol létrehozunk az objektumpéldányokat, majd az objektumon keresztül szimuláljuk egy kétszeresen láncolt lista viselkedését.

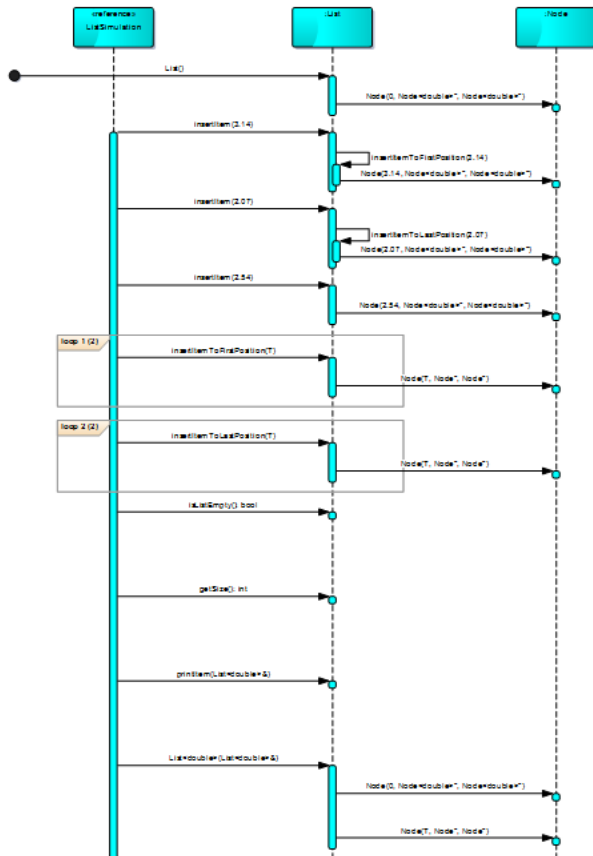
Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén [**heap**]) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektum példányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. Mielőtt a feladatnak neki látnánk, tervezzük meg a statikus osztály modellt és a dinamikus szekvencia diagramot, amely az implementálás során nagy segítséget nyújthat.

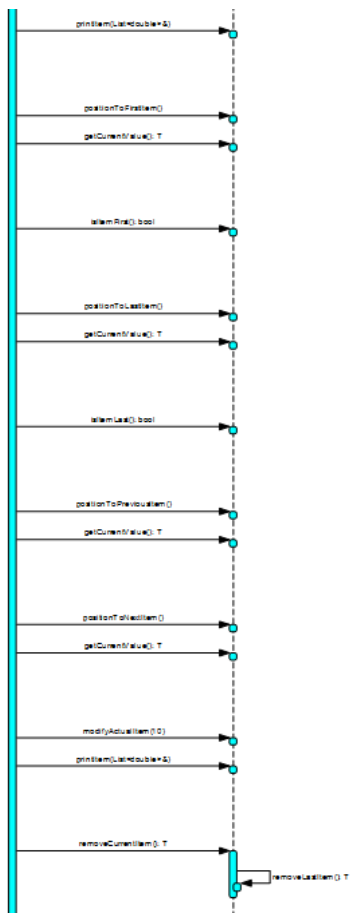
Statikus diagram

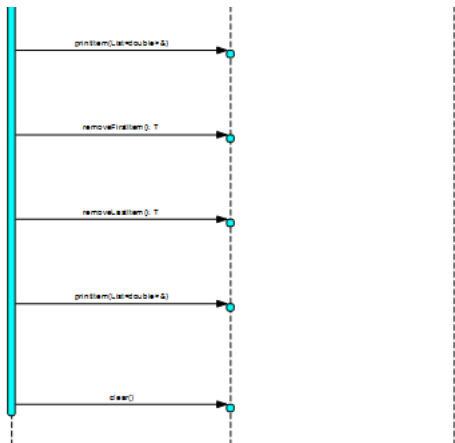


Osztálydiagram

Dinamikus diagram







Szekvenciadiagram

Megoldás

ListSimulation.h

```
#ifndef LISTSIMULATION_H
#define LISTSIMULATION_H

#include <iostream>

template<class T> class Node
{
public:
    T data;
    Node<T>* previous;
    Node<T>* next;
    Node(T data = 0, Node* next = 0, Node* previous = 0)
        : data(data), next(next), previous(previous) { }
};

template<class T> class List
{
private:
    Node<T>* head;
    Node<T>* current;
    int size;

public:
    List();
    List(const List& object);
    bool isListEmpty();
    int getSize();
    void positionToFirstItem();
    void positionToLastItem();
    void positionToPreviousItem();
    void positionToNextItem();
    void insertItem(T newItem);
    void insertItemToFirstPosition(T newItemToFirstPosition);
    void insertItemToLastPosition(T newItemToLastPosition);
    void modifyActualItem(T currentItem);
    T getCurrentValue();
    bool isItemFirst();
    bool isItemLast();
    T removeCurrentItem();
    T removeFirstItem();
    T removeLastItem();
    void clear();
    void printItem(const List& object);
    ~List();
};
#endif
```

ListSimulation.cpp

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

using namespace std;

template <class T> List<T>::List()
{
    head = new Node<T>();
    current = 0;
    size = 0;
}

template <class T> List<T>::List(const List<T>& object)
{
    head = new Node<T>();
    current = 0;

    Node<T>* h = head;
    for (Node<T> *node = object.head->next; node != 0; node = node->next)
    {
        h->next = new Node<T>(node->data, 0, h);
        h = h->next;

        if (object.current == node)
        {
            current = h;
        }
    }
    head->previous = h;
}

template <class T> bool List<T>::isListEmpty()
{
    return (head->next == 0) ? true : false;
}

template <class T> int List<T>::getSize()
{
    return size;
}

template <class T> void List<T>::positionToFirstItem()
{
    if (head->next == 0)
    {
        cout << "A lista üres";
    }
    else
    {
        current = head->next;
    }
}
```

```

    }
}

template <class T> void List<T>::positionToLastItem()
{
    if (head->previous == 0)
    {
        cout << "A lista üres";
    }
    else
    {
        current = head->previous;
    }
}

template <class T> void List<T>::positionToNextItem()
{
    if (head->next == 0)
    {
        cout << "A lista üres";
    }
    else if (head != current)
    {
        current = current->next;
    }
}

template <class T> void List<T>::positionToPreviousItem()
{
    if (head->previous == 0)
    {
        cout << "A lista üres";
    }
    else if (current != head)
    {
        current = current->previous;
    }
}

template <class T> void List<T>::insertItem(T newItem)
{
    if (current == 0)
    {
        insertItemToFirstPosition(newItem);
    }
    else
    {
        if (current == head->previous)
        {
            insertItemToLastPosition(newItem);
        }
    }
}

```

```

        else
        {
            current->next = new Node<T>(newItem, current->next, current);
            current = current->next;
            ++size;
        }
    }
}

template <class T> void List<T>::insertItemToFirstPosition(T newItem)
{
    if (head->next == 0)
    {
        head->next = new Node<T>(newItem, 0, head);
        head->previous = head->next;
        current = head->next;
    }
    else
    {
        head->next = new Node<T>(newItem, head->next, head);
        head->next->next->previous = head->next;
    }
    ++size;
}

template <class T> void List<T>::insertItemToLastPosition(T newItem)
{
    if (head->previous == 0)
    {
        insertItemToFirstPosition(newItem);
    }
    else
    {
        head->previous = new Node<T>(newItem, 0, head->previous);
        head->previous->previous->next = head->previous;
    }
    ++size;
}

template <class T> void List<T>::modifyActualItem(T newItem)
{
    if (current == 0)
    {
        cout << "A lista üres.";
    }
    else
    {
        current->data = newItem;
    }
}

```

```

template <class T> T List<T>::getCurrentValue()
{
    if (current == 0)
    {
        cout << "A lista üres";
    }
    else
    {
        return current->data;
    }
}

template <class T> bool List<T>::isItemFirst()
{
    if (current == 0)
    {
        cout << "A lista üres";
    }
    else
    {
        return (head->next==current ? true : false);
    }
}

template <class T> bool List<T>::isItemLast()
{
    if (current == 0)
    {
        cout << "A lista üres";
    }
    else
    {
        return (head->previous==current ? true : false);
    }
}

template <class T> T List<T>::removeCurrentItem()
{
    if (current == 0)
    {
        cout << "A lista üres";
    }
    T itemToDelete = current->data;
    if (current->next == 0)
    {
        removeLastItem();
    }
    else
    {
        Node<T>* node = current;
        current->next->previous = current->previous;
    }
}

```

```

        current->previous->next = current->next;
        current = current->next;
        delete node;
    }
    --size;
    return itemToDelete;
}

template <class T> T List<T>::removeFirstItem()
{
    if (head->next == 0)
    {
        cout << "A lista üres";
    }
    T itemToDelete = head->next->data;
    if (head->next == head->previous)
    {
        delete head->next;
        head->next = 0;
        head->previous = 0;
        current = 0;
    }
    else
    {
        Node<T>* node = head->next;
        head->next = node->next;
        head->next->previous = head;

        if (current == node)
        {
            current = head->next;
        }
        delete node;
    }
    --size;
    return itemToDelete;
}

template <class T> T List<T>::removeLastItem()
{
    if (head->previous == 0)
    {
        cout << "A lista üres";
    }
    T itemToDelete = head->previous->data;
    if (head->next == head->previous)
    {
        removeFirstItem();
    }
    else
    {

```

```

Node<T>* node = head->previous;
node->previous->next = 0;
head->previous = node->previous;

if (current == node)
{
    current = head->previous;
}
delete node;
}
--size;
return itemToDelete;
}

template <class T> void List<T>::clear()
{
    Node<T>* node;
    current = head->next;
    while (current != 0)
    {
        node = current;
        current = current->next;
        delete node;
    }
    head->previous = 0;
    head->next = 0;
    size = 0;
}

template <class T> void List<T>::printItem(const List<T>& object)
{
    if (current == 0)
    {
        cout << "list<T>";
    }
    else
    {
        cout << "list<T>: ";
        Node<T> *node;
        for (node = object.head->next; node->next!=0; node=node->next)
        {
            cout << node->data << "->";
        }
        if (node != 0)
        {
            cout << node->data << endl;
        }
    }
}
}

```

```
template <class T> List<T>::~List()
{
    delete head;
}
```

main.cpp

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

using namespace std;

int main()
{
    List<double> *doubleLinkedListObject = new List<double>();
    doubleLinkedListObject->insertItem(3.14);
    doubleLinkedListObject->insertItem(2.07);
    doubleLinkedListObject->insertItem(2.54);
    doubleLinkedListObject->insertItemToFirstPosition(0.8);
    doubleLinkedListObject->insertItemToFirstPosition(0.7);
    doubleLinkedListObject->insertItemToLastPosition(4.23);
    doubleLinkedListObject->insertItemToLastPosition(5.10);
    cout << "a lista ures? : " << doubleLinkedListObject->isEmpty() << endl;
    cout << "az első lista elemeinek száma: " << doubleLinkedListObject->getSize() << endl;
    doubleLinkedListObject->printItem(*doubleLinkedListObject);

    List<double> *extendedDoubleLinkedListObject = new List<double>(*doubleLinkedListObject);
    extendedDoubleLinkedListObject->printItem(*extendedDoubleLinkedListObject);
    extendedDoubleLinkedListObject->positionToFirstItem();
    cout << "az aktuális elem? : " << extendedDoubleLinkedListObject->getCurrentValue() << endl;
    cout << "a lista első elemén állunk? : " << extendedDoubleLinkedListObject->isItemFirst() << endl;
    extendedDoubleLinkedListObject->positionToLastItem();
    cout << "az aktuális elem? : " << extendedDoubleLinkedListObject->getCurrentValue() << endl;
    cout << "a lista utolsó elemén állunk? : " << extendedDoubleLinkedListObject->isItemLast() << endl;
    extendedDoubleLinkedListObject->positionToPreviousItem();
    cout << "a lista utolsó előtti elem? : " << extendedDoubleLinkedListObject->getCurrentValue() << endl;
    extendedDoubleLinkedListObject->positionToNextItem();
    cout << "a lista utolsó elem? : " << extendedDoubleLinkedListObject->getCurrentValue() << endl;
}
```

```

extendedDoubleLinkedListObject->modifyActualItem(10);
extendedDoubleLinkedListObject-
>printItem(*extendedDoubleLinkedListObject);
cout << "a lista aktualis elemenek eltavolitasa: " <<
extendedDoubleLinkedListObject->removeCurrentItem() << endl;
extendedDoubleLinkedListObject-
>printItem(*extendedDoubleLinkedListObject);
cout << "a lista első elemenek eltavolitasa: " <<
extendedDoubleLinkedListObject->removeFirstItem() << endl;
cout << "a lista utolsó elemenek eltavolitasa: " <<
extendedDoubleLinkedListObject->removeLastItem() << endl;
extendedDoubleLinkedListObject-
>printItem(*extendedDoubleLinkedListObject);
extendedDoubleLinkedListObject->clear();
doubleLinkedListObject->clear();

delete extendedDoubleLinkedListObject;
delete doubleLinkedListObject;
_getch();
return 0;
}

```

Program futásának eredménye:

```

D:\JatSimulacio\Debug\ListSimulation.exe
a lista urrez : 0
az első lista elemének száma: 7
list<T>: 0.7->0.8->3.14->2.54->2.07->4.23->5.1
list<T>: 0.7->0.8->3.14->2.54->2.07->4.23->5.1
az aktualis elem? : 0.7
a lista első elemének allunk? : 1
az aktualis elem? : 5.1
a lista utolsó elemének allunk? : 1
a lista utolsó előtti elem? : 4.23
a lista utolsó elem? : 5.1
list<T>: 0.7->0.8->3.14->2.54->2.07->4.23->10
a lista aktualis elemének eltavolitasa: 10
list<T>: 0.7->0.8->3.14->2.54->2.07->4.23
a lista első elemének eltavolitasa: 0.7
a lista utolsó elemének eltavolitasa: 4.23
list<T>: 0.8->3.14->2.54->2.07

```

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: *Objektumorientált szoftverfejlesztés*. ComputerBooks, 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.
- http://calmarious.net/?lang=hu&page=programming%2Fc++_standard_library#stl_containers
- Tóth Bertalan: *C++ Programozás STL Konténerekkel*, 2016.

Tartalom

A C++ szabványos sablonkönyvtár (STL) szekvenciális tárolói.....	1
Előszó	2
STL (Standard Template Library) – A C++ nyelv szabványos sablonkönyvtára	3
Konténerek, tárolók (containers)	7
Szekvenciális tárolók	9
std::array.....	9
Minta példa a statikus tömb (array) tároló használatára Visual Studio Community fejlesztőkörnyezetben	15
Statikus tervezés	16
Dinamikus diagram.....	17
A projekt létrehozása	19
A feladathoz kapcsolódó magyarázatok.....	34
std::vector	37
Minta példa a dinamikus tömb (vector) tároló használatára Visual Studio Community fejlesztőkörnyezetben	43
Statikus tervezés	45
Dinamikus diagram.....	46
A projekt létrehozása	49
A feladathoz kapcsolódó magyarázatok.....	65
std::deque	72
Minta példa a kétvégű sor (deque) tároló használatára Visual Studio Community fejlesztőkörnyezetben	75
Statikus tervezés	78
Dinamikus diagram.....	79
A projekt létrehozása	80
A feladathoz kapcsolódó magyarázatok.....	97
std::list.....	104

Minta példa a kétszeresen láncolt lista (list) tároló használatára Visual Studio Community fejlesztőkörnyezetben	114
Statikus tervezés	117
Dinamikus diagram	118
A projekt létrehozása.....	119
A feladathoz kapcsolódó magyarázatok.....	138
std::forward_list	146
Minta példa az egyszeresen láncolt lista (forward_list) tároló használatára Visual Studio Community fejlesztőkörnyezetben.....	150
Statikus tervezés	152
A projekt létrehozása.....	155
A feladathoz kapcsolódó magyarázatok.....	173
A szekvenciális tárolók összegzése	181
Gyakorlópélda	183
Megoldás	195
Felhasznált irodalom.....	204