

A C++ szabványos sablonkönyvtár (STL) adapter tárolói

Gyakorlatorientált szoftverfejlesztés C++ nyelven

Visual Studio Community fejlesztőkörnyezetben

@Katona József – Kővári Attila
2016

Publio Kiadó

Felelős kiadó: Publio Kiadó Kft.

www.publio.hu

Minden jog fenntartva!

DOI: 10.18395/szabvanyos.stl.adapter.2016

Előszó

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

E könyv a C++ nyelven történő objektumorientált szoftverfejlesztést megismertető sorozat ötödik 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 adapter tárolóit ismerteti. A könyv olvasása során megismerjük az adapter 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, Kratzmajer Zoltánnak szakmai tanácsaiért és módszertani javaslataiért.

A Szerzők

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

A könyvsorozat előző köteteinek témái nélkülözhetetlenek ahhoz, hogy megértsük az **STL (Standard Template Library)**, magyarul Szabványos Sablonkönyvtár elemeinek paraméterezhető osztályait és függvényeit. A sablonkönyvtár számos gyűjteményt, illetve komponenst biztosít számunkra, annak érdekében, hogy minél hatékonyabb, gyorsabb és elegánsabb megoldást tudjunk nyújtani egy programozási feladatra.

Konténerek (tárolók)

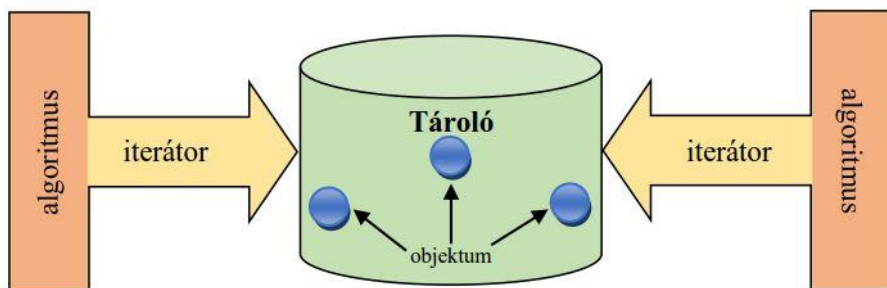
Olyan adatstruktúrák – például objektumok – amelyek más, azonos típusú objektumokat (elemeket) tárolnak, az adatok memóriában történő tárolását teszik lehetővé és tárolási lehetőségeik mellett karbantartó műveleteket is biztosítanak. (array – statikus tömb, vector – dinamikus tömb, deque – kétvégű sor, list – kétszeresen láncolt lista, set - halmaz, map – asszociatív tömb stb.). **A szekvenciális konténerek alaposabb bemutatása a könyvsorozat 4. kötetében történt.**

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 kínálva 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, de ezzel visszafelé lehet haladni stb.) **Az iterátorok részletezése a könyvsorozat 7. kötetében található.**



A sablonkönyvtár komponensek

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. **Terjedelmi korlátok miatt a jelenlegi kötet kizárólag a konténer adaptációkat ismerteti részletekbe menően.**

Függvényobjektumok (funktörök, 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.

funktör alkalmazása

```
std::mismatch(array0, array0+2, array1, std::equal_to<int>());
```

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

Néhány a <functional> fejláományban 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 sablonokkal (A sablonokról, mint például osztály- vagy függvénysablonok, részletesen a C++ sorozat 3. kötetében olvashat, továbbá elsajátíthatja azok használatát.) **(template)** dolgoznak, így bármilyen típusal képesek együtt működni.

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

```
#include <vector>
```

```
#include <list>
```

```
#include <map>
```

A fentebbi szabványos fejláomány 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:

- 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 névtérrel 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 destruktoral, amelyek a tároló osztályok objektumpéldányainak létrehozásában, másolatában és megszüntetésében játszanak nagy szerepet, továbbá minden tároló képes lehet egy másik tároló résztartományának eltárolására. A különböző tárolók elemeinek mindig van egy meghatározott sorrendje. A leggyakrabban használt tároló elemeket az alábbi táblázat foglalja össze.

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	Kettős végű 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 beszúrását, továbbá a kivételét az elejéről.
<stack>	Adapter	Verem egy olyan tároló, amelyből mindig az utoljára elhelyezett elemet tudjuk

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

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

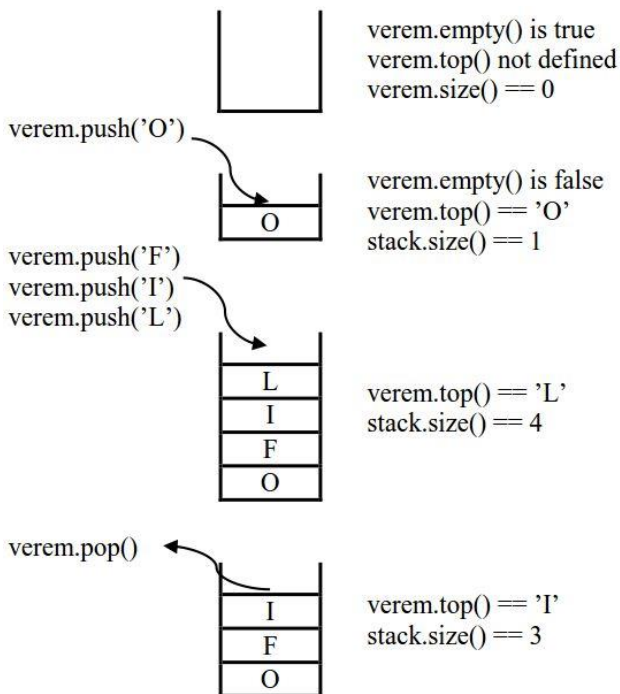
Adapter tárolók / Idomított sorozatok

Az STL könyvtár által biztosított alapvető sorozatokból (**array** – statikus tömb, **vector** – dinamikus tömb, **deque** – kétvégű sor, **list** – kétszeresen láncolt lista, **forward_list** – egyszeresen láncolt lista) – a megfelelő módon kialakított felülettel származnak – az alábbi adapter tárolók (**container adapter**), más neveken idomított sorozatok, sorozat-átalakítók (**sequence adapter**) vagy egyszerűen csak átalakítók (**adapter**): **stack**<AdatTípus> – verem, **queue**<AdatTípus> – sorozat, **priority-queue**<AdatTípus> – prioritásos sor.

A verem (**stack**, **LIFO** - **Last In-First Out**), a sor (**queue**, **FIFO** - **First In-First Out**) és a prioritásos sor (**priority_queue**) a korábbi adatszerkezetekre épülve valósítanak meg egy új adatszerkezetet. A verem és a sor nem külön adatszerkezetek, hanem más adatszerkezet segítségével egy viselkedés megvalósítása. Az implementáció során mindkét tároló a kétvégű sort használja, de ettől el lehet térni. Például egy dinamikus vektor esetében nem helyénvaló a sor használata, mivel nem túl hatékony a kivesz/eltávolít (**pop**) művelet, ami a vektor elejéről távolít el egy elemet, ennek következményeként a vektor teljes tartalma másolásra kerül. Az olyan sorozatok, mint a dinamikus tömb, kétszeresen láncolt lista vagy a kétvégű sor nem megvalósíthatók jelentős hatékonyságromlás nélkül, ezzel szemben például a verem és a sor kifinomult módon implementálható a fentebbi három sorozattípus segítségével. Ennek következményeként a két sorozatot leíró osztályt nem teljesen független tárolóként valósították meg, hanem az alaptároló átalakítóiként. Egy tároló átalakítója egy korlátozott felületet biztosít az adott tárolóhoz. Mivel a korlátozott felület elegendő szolgáltatást képes nyújtani a céljuk megvalósításához, így az adapterek nem biztosítanak bejárókat.

std::stack

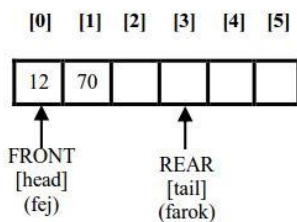
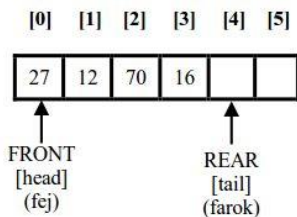
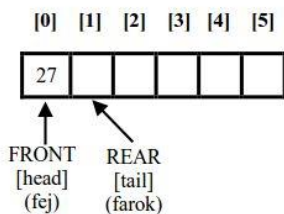
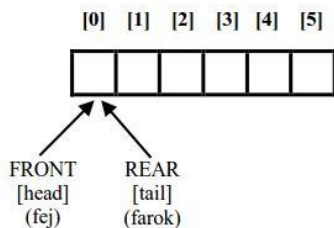
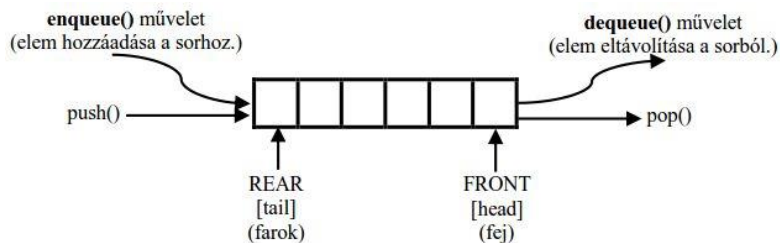
Az **STL**-ben a verem (**stack**) típus egy olyan adatszerkezet, amelyet a **stack** osztály implementál és a **<stack>** fejlécben írt le. A tároló tetejére új elemet helyezhetünk, illetve kizárólag a tároló tetején szereplő elemet vagyunk képesek elérni és kivenni. A verem mélyebben elhelyezkedő elemeik rejtettek a külvilág elől, mérete tetszőleges módon növelhető, de lehetőség van a méret korlátozására (szükség esetén) is.



A verem működésének ábrája

std::queue

A sorozat (**queue**) típus egy olyan adatszerkezet, amelyet a `<queue>` fejlánc ír le és olyan felületet kínál egy tárolóhoz, amely lehetőséget biztosít az adatszerkezet végére történő beszúráshoz, valamint kivételt annak elejéről.



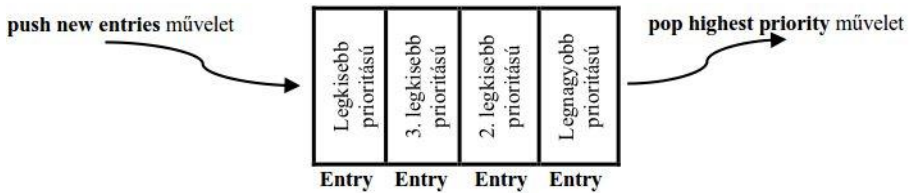
Elemek hozzáadása a sorhoz.

Elem eltávolítása a sorból

A sor működésének ábrája

std::priority_queue

A prioritásos sor (**priority_queue**) egy olyan adatszerkezet, amelyet a **<queue>** fejlármányban deklaráltak, az elemek fontossági értéket kapnak, és ez az érték szabályozza, hogy az elemek milyen sorrendben vehetők ki. Alapértelmezés szerint a prioritásos sor az elemeket egyszerűen a **<** operátor segítségével hasonlítja össze, és a **top()** mindig a legnagyobb elemet adja vissza. A prioritásos sor egyik hatékony megvalósítási módja a faszerkezet, mellyel az elemek egymáshoz viszonyított helyét könnyedén beállíthatjuk.



A prioritásos sor működésének ábrája

Műveletek	Jelentés verem (stack) esetén	Jelentés sor (queue) esetén
<code>bool empty() const</code>	Logikai igazgal tér vissza, ha a verem üres.	Logikai igazgal tér vissza, ha a sor üres.
<code>unsigned size() const</code>	Hány elem található a veremben?	Hány elem található a sorban?
<code>T& top()</code>	A verembe legutoljára beletett elem referenciáját adja vissza.	-
<code>push(const T&)</code>	Egy új elemet helyez el a verembe.	Egy új elemet helyez el a sorban.
<code>void pop()</code>	A verembe legutoljára beletett elemet eltávolítja.	A sorban legrégebben beletett elemet eltávolítja.
<code>T& front()</code>	-	A sorba legrégebben beletett elem referenciáját adja vissza.

A verem illetve a sor főbb műveletei

Minta példa a verem (stack) tároló használatára Visual Studio Community fejlesztőkörnyezetben

Írjunk egy StackDemo (*StackDemo*) osztályt, amely különböző, verem specifikus műveleteket ír le, majd egy objektumpéldány segítségével ezeket a műveleteket végrehajtatjuk. 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 **push()**.

Az osztály adatai a következők:

- *stack*, egy verem jellegű adapter tároló, amely 10 db egész szám tárolására képes.

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

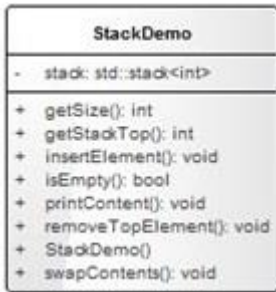
Valósítsuk meg az osztály metódusait:

- Írjuk meg az osztály konstruktorát, amely segítségével inicializáljuk a verem tárolót 1-től 10-ig. (**push()**).
- Írjunk egy *isEmpty()* függvényt, amely megvizsgálja, hogy a verem tartalmaz-e elemeket, majd annak függvényében vissza tér egy logikai értékkel. Igazzal, ha található elem a veremben, hamissal, ha üres. (**empty()**)
- Írjunk egy *getSize()* függvényt, amely a verem tároló aktuális méretét adja vissza. (**size()**)
- Írjunk egy *getStackTop()* függvényt, amely segítségével a verem legfelső elemét kérhetjük le. (**top()**)
- Írjunk egy *removeTopElement()* függvényt, amely a verem legfelső elemét távolítja el. (**pop()**)
- Írjunk egy *insertElement()* függvényt, amely a veremhez történő hozzáfűzést teszi lehetővé, a hozzáfűzendő érték 100. (**emplace()**)
- Írjunk egy *swapContents()* függvényt, amely két verem tartalmát cseréli meg. (**swap()**)
- Írjunk egy *printContent()* függvényt, amely kiírja a verem elemeit.

Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén [**heap**]) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az

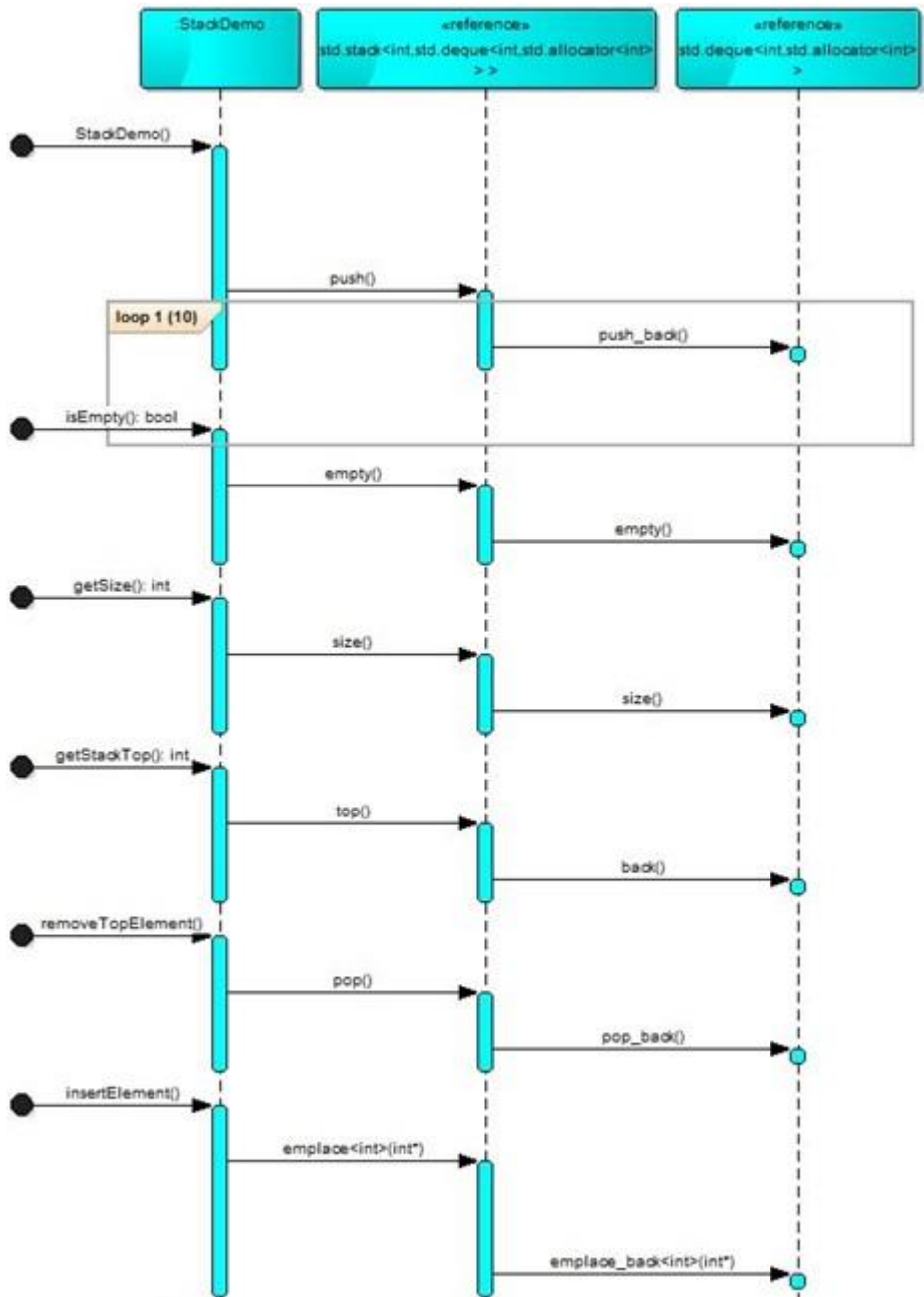
osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektum példányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. Mielőtt a feladatnak neki látnánk, tervezzük meg az osztály- és szekvencia diagramot. **Figyelem! A könyvben ábrázolt szekvenciadiagramok terjedelmi korlátok miatt a teljesség igénye nélkül készültek!**

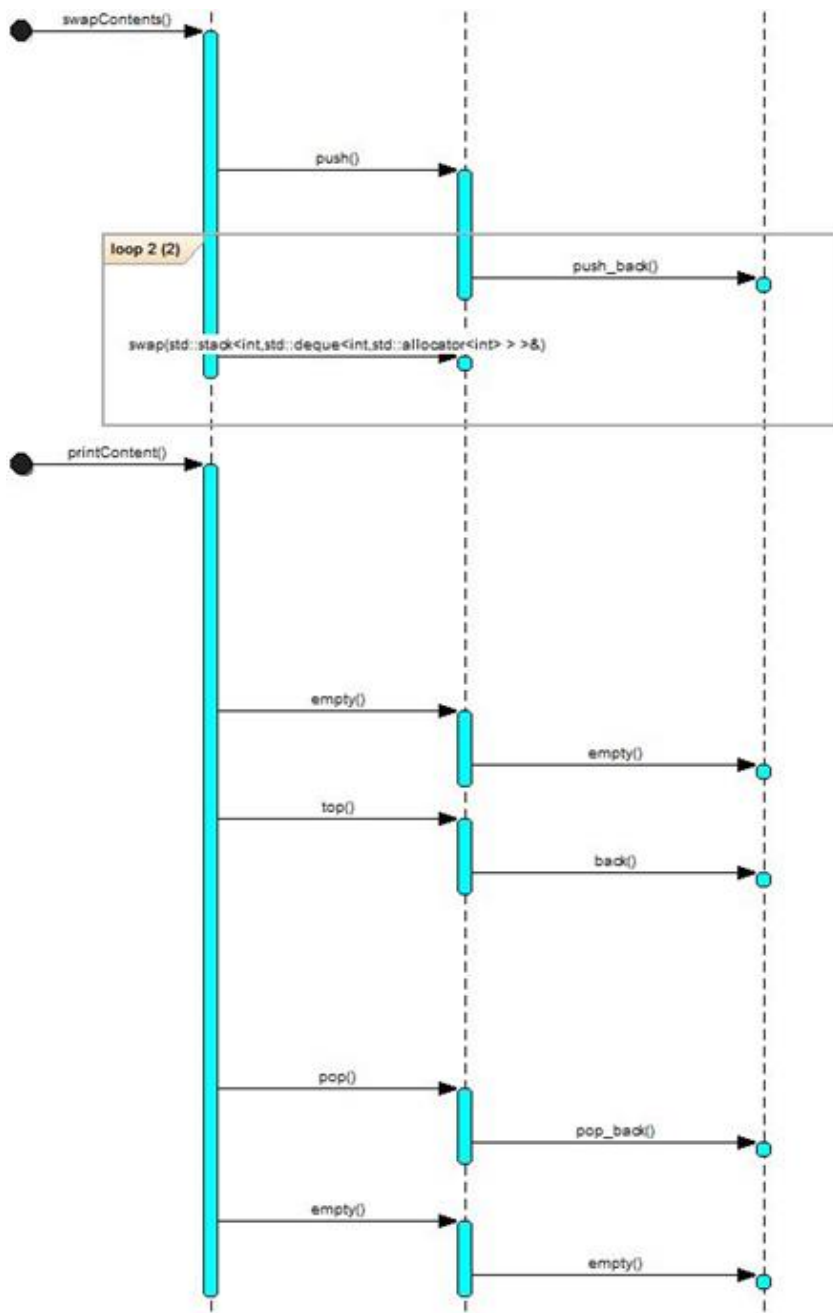
Statikus tervezés



Osztálydiagram

Dinamikus diagram



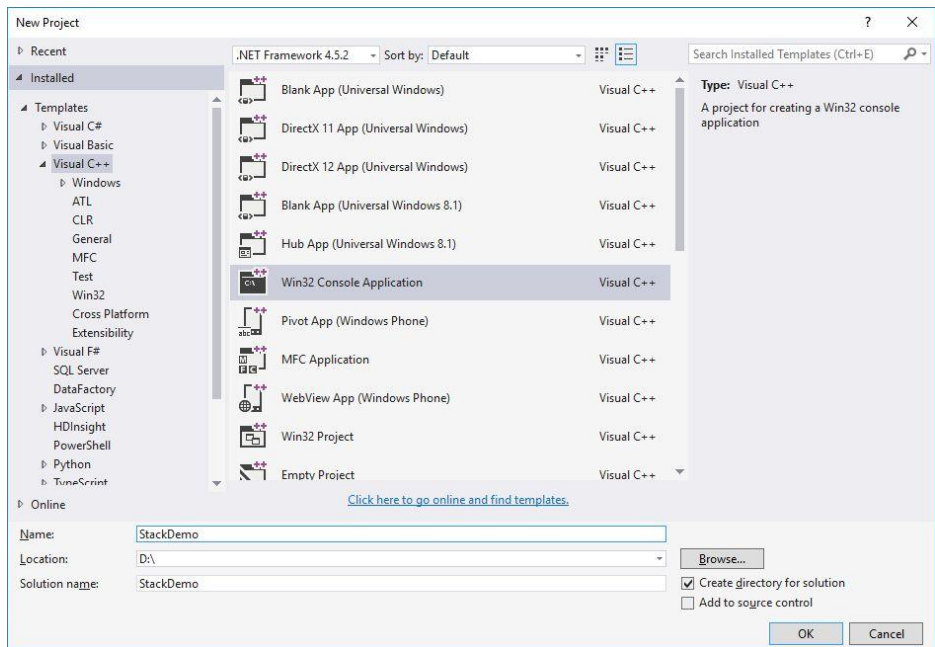


Szekvenciadiagram

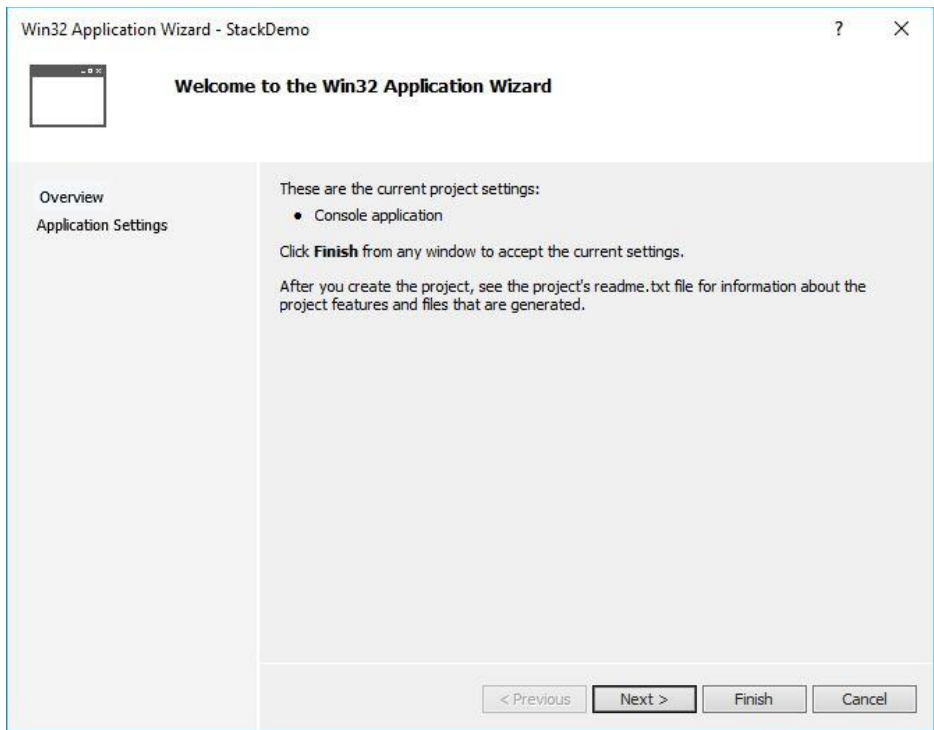
A projekt létrehozása

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.

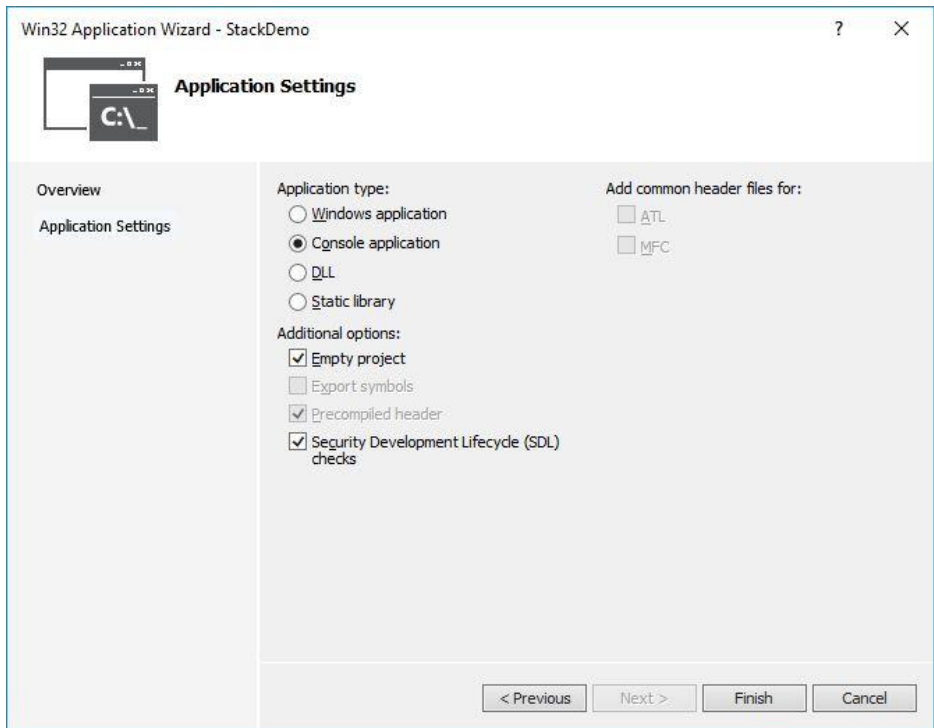
Új projekt a **File** menü **New** menüpontjában hozható létre, a **Project** almenüt kiválasztva, vagy akár a **Ctrl+Shift+N** gyorsindító gombkombinációval. Új projekt kiválasztása után megjelenő ablakban a **Visual C++ Win32** sablont (**Template**) és a **Win32 Console Application**-t kell kiválasztani,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



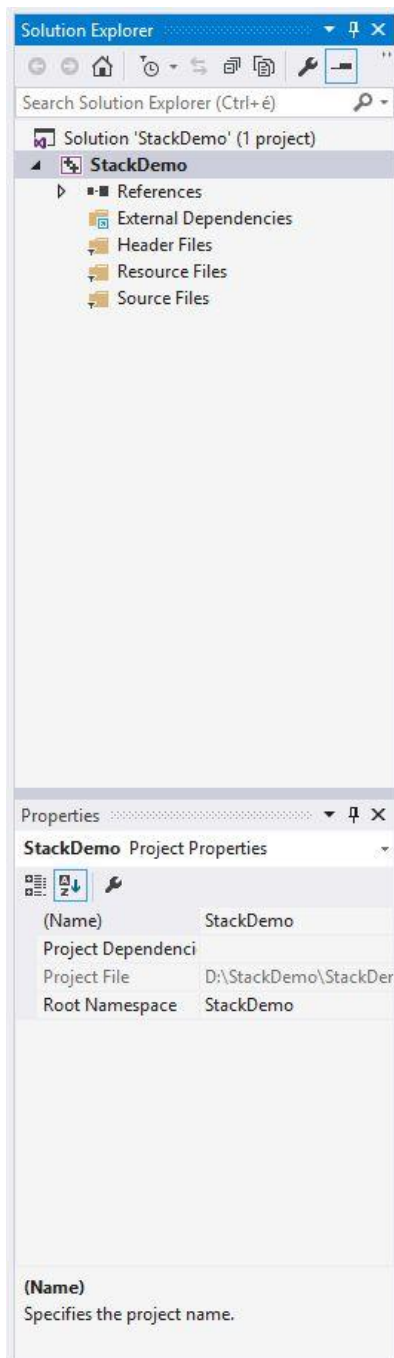
Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően a **StackDemo** név került megadásra. Ebben a részben lehet módosítani a projekthez kapcsolódó fájlok mentési helyét is (**Location**).



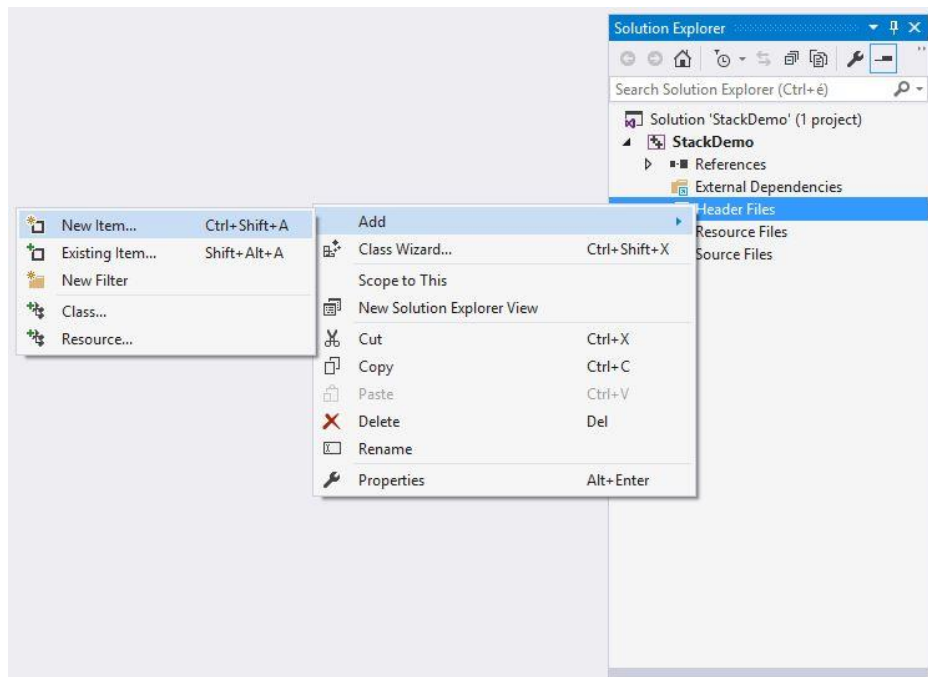
A **Next** gombra kattintva a beállítások ablak jelenik meg, itt megadhatjuk az alkalmazás típusát (**Console Application**), és néhány kiegészítő opciót. Mivel egyszerű alkalmazást készítünk, ezért célszerű az üres projekt (**Empty project**) opció kiválasztása, ahogy az alábbi képen figyelhető meg.



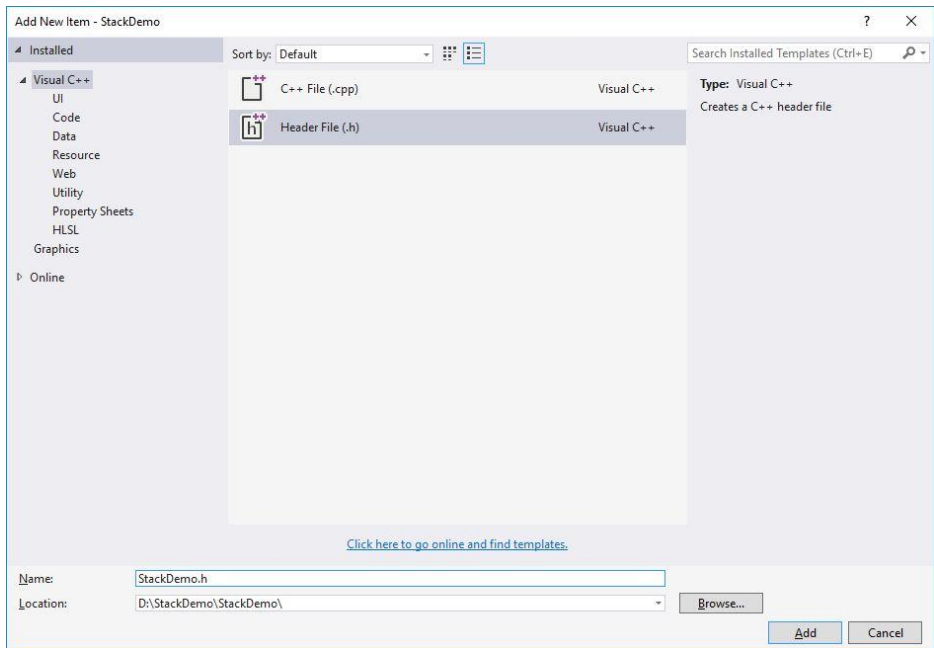
A **Finish** megnyomását követően a képernyő jobb szélén, a megoldás ablaka (**Solution Explorer**), alatta a beállítások ablak jelenik meg, ahogy a következő ábrán látható.



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



Az új fájl hozzáadásával a képernyő középső részén megjelenik a forráskód szerkesztő.

StackDemo.h

```
#ifndef STACKDEMO_H  
#define STACKDEMO_H
```

```
#include <stack>
```

```
class StackDemo  
{  
private:  
    std::stack<int> stack;  
public:  
    StackDemo();
```

```

bool isEmpty();

int getSize();

int getStackTop();

void insertElement();

void removeTopElement();

void swapContents();

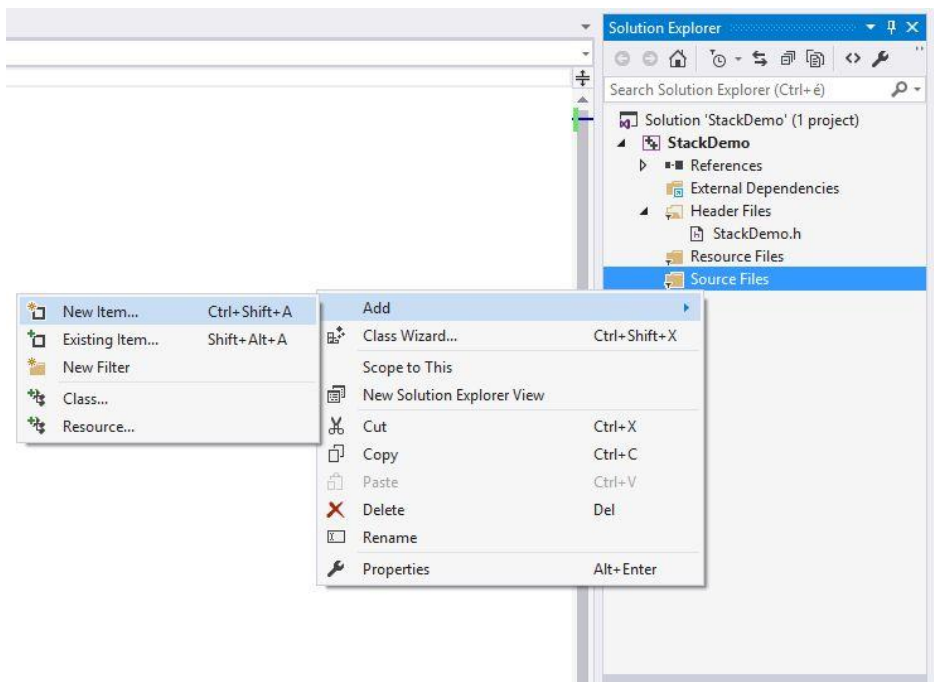
void printContent();

};

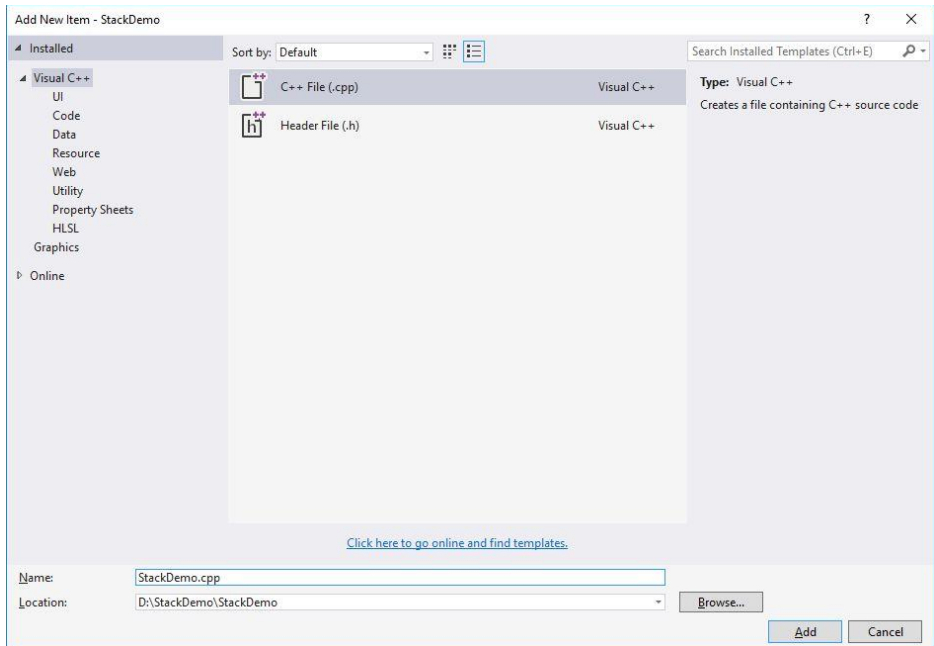
#endif

```

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



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



StackDemo.cpp

```
#include <iostream>
```

```
#include "StackDemo.h"
```

```
StackDemo::StackDemo()
```

```
{
```

```
for (int i = 1; i <= 10; ++i)
```

```
stack.push(i);
```

```
}
```

```
bool StackDemo::isEmpty()
```

```
{
```

```
return stack.empty();
```

```
}
```

```
int StackDemo::getSize()
```

```

{
return stack.size();
}

int StackDemo::getStackTop()
{
return stack.top();
}

void StackDemo::insertElement()
{
stack.emplace(100);
}

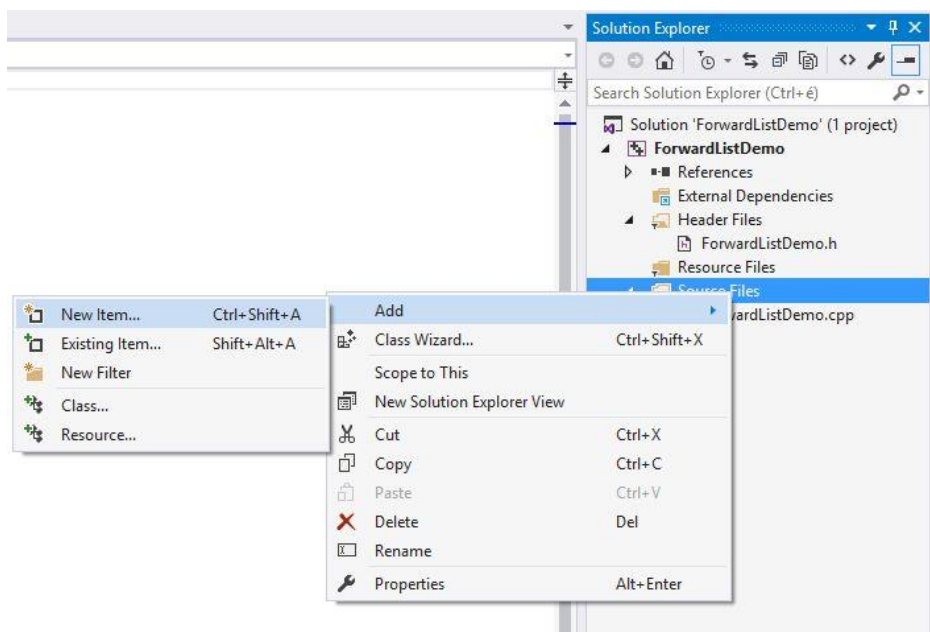
void StackDemo::removeTopElement()
{
stack.pop();
}

void StackDemo::swapContents()
{
std::stack<int> stack2;
stack2.push(11);
stack2.push(22);
stack.swap(stack2);
}

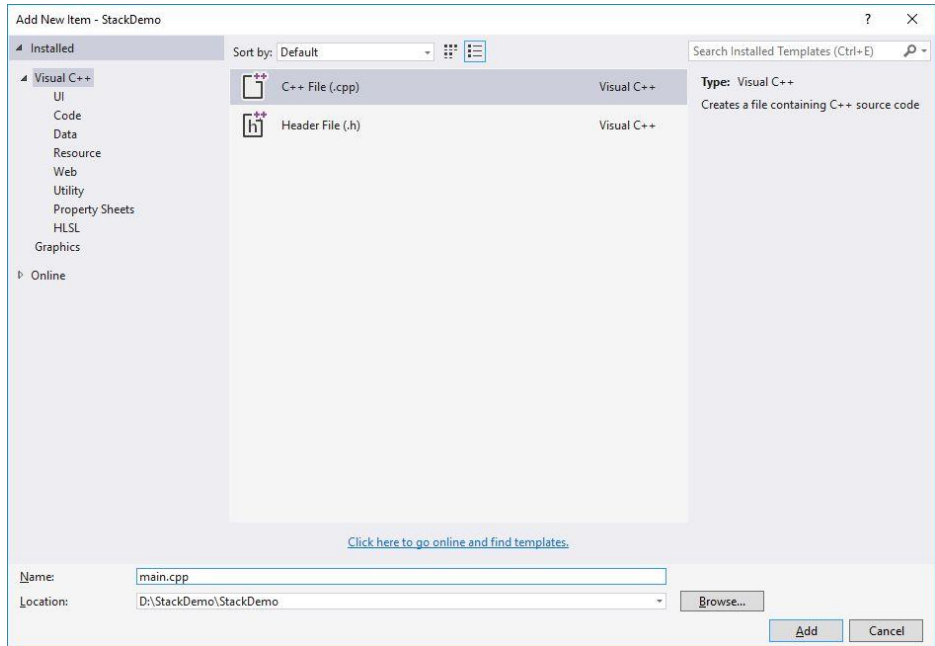
void StackDemo::printContent()
{
std::cout << "A verem elemei: ";
while (!stack.empty())
{
std::cout << ' ' << stack.top();
stack.pop();
}
std::cout << "\n";
}

```

}



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



main.cpp

```
#include <iostream>
```

```
#include <conio.h>
```

```
#include "StackDemo.h"
```

```
int main()
```

```
{
```

```
    StackDemo *StackDemoObject = new StackDemo();
```

```
    std::cout << "MUVELET => StackDemo() #konstruktor hivasa, verem feltoltese..."
```

```
    << std::endl;
```

```
    std::cout << "MUVELET => empty() #uresseg vizsgalata..." << std::endl;
```

```
    std::cout << "EREDMENY: "; std::cout << StackDemoObject->isEmpty() <<
```

```
    std::endl;
```

```
    std::cout << "MUVELET => size() #meret lekerdezese..." << std::endl;
```

```
    std::cout << "EREDMENY: "; std::cout << StackDemoObject->getSize() <<
```

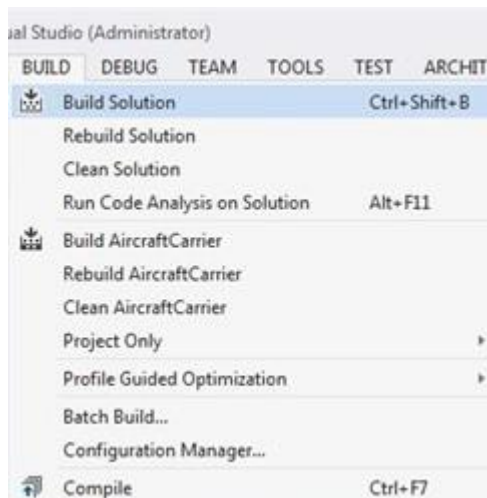
```

std::endl;
std::cout << "MUVELET => top() #legfelso elem lekerdezese..." << std::endl;
std::cout << "EREDMENY: "; std::cout << StackDemoObject->getStackTop();
std::cout << std::endl;
std::cout << "MUVELET => pop() #legfelso elem torlese..." << std::endl;
StackDemoObject->removeTopElement();
std::cout << "MUVELET => emplace() #hozzafuzes..." << std::endl;
StackDemoObject->insertElement();
std::cout << "MUVELET => swap() #veremek elemeinek megcserelese..." <<
std::endl;
StackDemoObject->swapContents();
std::cout << "MUVELET => pop() A verem tarolo tartalma..." << std::endl;
std::cout << "EREDMENY: "; StackDemoObject->printContent();

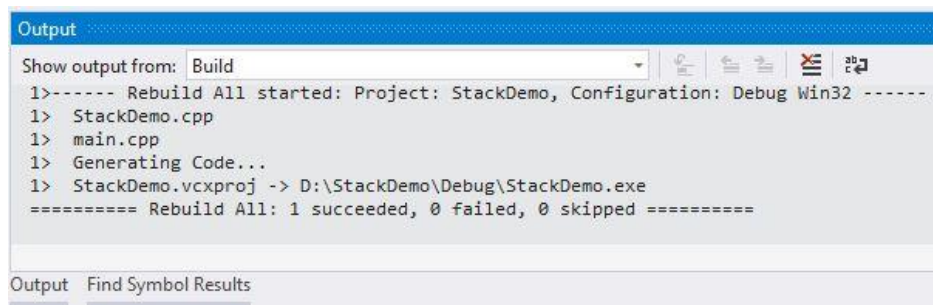
delete StackDemoObject;
_getch();
return 0;
}

```

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.



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



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

```
D:\StackDemo\Debug\StackDemo.exe
MUVELET => StackDemo() #konstruktor hívása, verem feltöltése...
MUVELET => empty() #üresesség vizsgálata...
EREDMENY: 0
MUVELET => size() #meret lekerdezes...
EREDMENY: 10
MUVELET => top() #legfelso elem lekerdezes...
EREDMENY: 10
MUVELET => pop() #legfelso elem torlese...
MUVELET => emplace() #hozzafuzes...
MUVELET => swap() #veremek elemeinek megcserelése...
MUVELET => pop() A verem tarolo tartalma...
EREDMENY: A verem elemei: 22 11
```

A feladathoz kapcsolódó magyarázatok

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

private:

```
std::stack<int> stack;
```

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:

```
StackDemo();
```

A *getSize()* tagfüggvények a leírásban megfogalmazott verem aktuális méret lekérdezését biztosítja, amíg a *getStackTop()* a verem legfelső elemét adja vissza. A láthatósági szintjük nyilvános (**public**), visszatérési típusuk egész szám (**integer**) és az argumentumlistájuk üres:

```
int getSize();  
int getStackTop();
```

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

```
bool isEmpty();
```

Az *insertElement()* tagfüggvény segítségével új elemeket fűztünk a veremhez a *removeTopElement()* tagfüggvény feladata a verem legfelső elemének eltávolítása. illetve a *swapContents()* tagfüggvény két verem elemét cseréli meg. Végül megfogalmazásra került egy *printContent()*, tagfüggvény, amely egy ciklus segítségével a képernyőn megjeleníti a vektor aktuális elemeit. A fentebb említett négy tagfüggvény visszatérési típusaik **void**, láthatósági szintjeik nyilvánosak (**public**), valamint az argumentumlistájuk üres.

```
void insertElement();  
void removeTopElement();  
void swapContents();  
void printContent();
```

Az **StackDemo.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 verem minden elemének 1-től 10-ig történő feltöltése, amelyet egyszerűen **push()** tagfüggvény segítségével hívásával valósítottunk meg.

```
StackDemo::StackDemo()  
{  
    for (int i = 1; i <= 10; ++i)  
        stack.push(i);
```

```
}
```

Az aktuális méret lekérdezését végrehajtó függvények implementációja során az **stack** objektumon keresztül egyszerűen hívásra került a **size()** tagfüggvény, amíg a verem legfelső elemének lekérdezése során a **top()** metódust hívtuk meg.

```
int StackDemo::getSize()
{
    return stack.size();
}
int StackDemo::getStackTop()
{
    return stack.top();
}
```

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

```
bool StackDemo::isEmpty()
{
    return stack.empty();
}
```

Új elem elhelyezése a veremben az **emplace()** metódus használatával történt, amíg a legfelső elem kivételét a **pop()**-al végeztük. Két verem tároló értékeinek megcserélését a **swap()** függvénnyel hajtottuk végre. A verem tartalmának konzolon történő megjelenítésért egy olyan ciklusszervezés felel amely több **stack** objektumon keresztüli tagfüggvény hívást hajt végre.

```
void StackDemo::insertElement()
{
    stack.emplace(100);
}
```

```

}
void StackDemo::removeTopElement()
{
    stack.pop();
}
void StackDemo::swapContents()
{
    std::stack<int> stack2;
    stack2.push(11);
    stack2.push(22);
    stack.swap(stack2);
}
void StackDemo::printContent()
{
    std::cout << "A verem elemei: ";
    while (!stack.empty())
    {
        std::cout << ' ' << stack.top();
        stack.pop();
    }
    std::cout << "\n";
}

```

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.

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

Írjunk egy QueueDemo (QueueDemo) osztályt, amely különböző, sor specifikus műveleteket ír le, majd egy objektumpéldány segítségével ezeket a műveleteket végrehajtatjuk. 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 **push()**. Az osztály adatait a következők:

- *queue*, egy sor jellegű adapter 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 sor tárolót 1-től 10-ig. (**push()**)

- Írjunk egy *isEmpty()* függvényt, amely megvizsgálja, hogy a sor tartalmaz-e elemeket, majd annak függvényében vissza tér egy logikai értékkel. Igazgal, ha található elem a sorban, hamissal, ha üres. (**empty()**)

- Írjunk egy *getSize()* függvényt, amely a sor tároló aktuális méretét adja vissza. (**size()**)

- Írjunk egy *getFirstElement()* függvényt, amellyel a sor legelső elemét kérhetjük le. (**front()**)

- Írjunk egy *getLastElement()* függvényt, amellyel a sor legutolsó elemét kérhetjük le. (**back()**)

- Írjunk egy *removeFirstElement()* függvényt, amely segítségével a sor legelső elemét távolíthatjuk el. (**pop()**)

- Írjunk egy *insertElement()* függvényt, amely a sorhoz történő hozzáfűzést teszi lehetővé, a hozzáfűzendő érték 100. (**emplace()**)

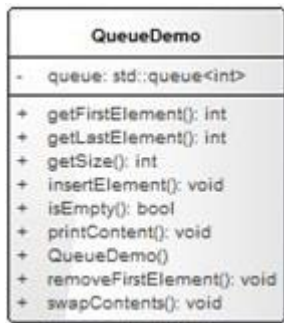
- Írjunk egy *swapContents()* függvényt, amely két sor tartalmát cseréli meg. (**swap**)

- Írjunk egy *printContent()* függvényt, amely egy ciklus segítségével kiírja a sor elemeit.

- Ne feledkezzünk meg az objektumpéldány által lefoglalt erőforrás felszabadításáról, a destruktor megírásáról sem.

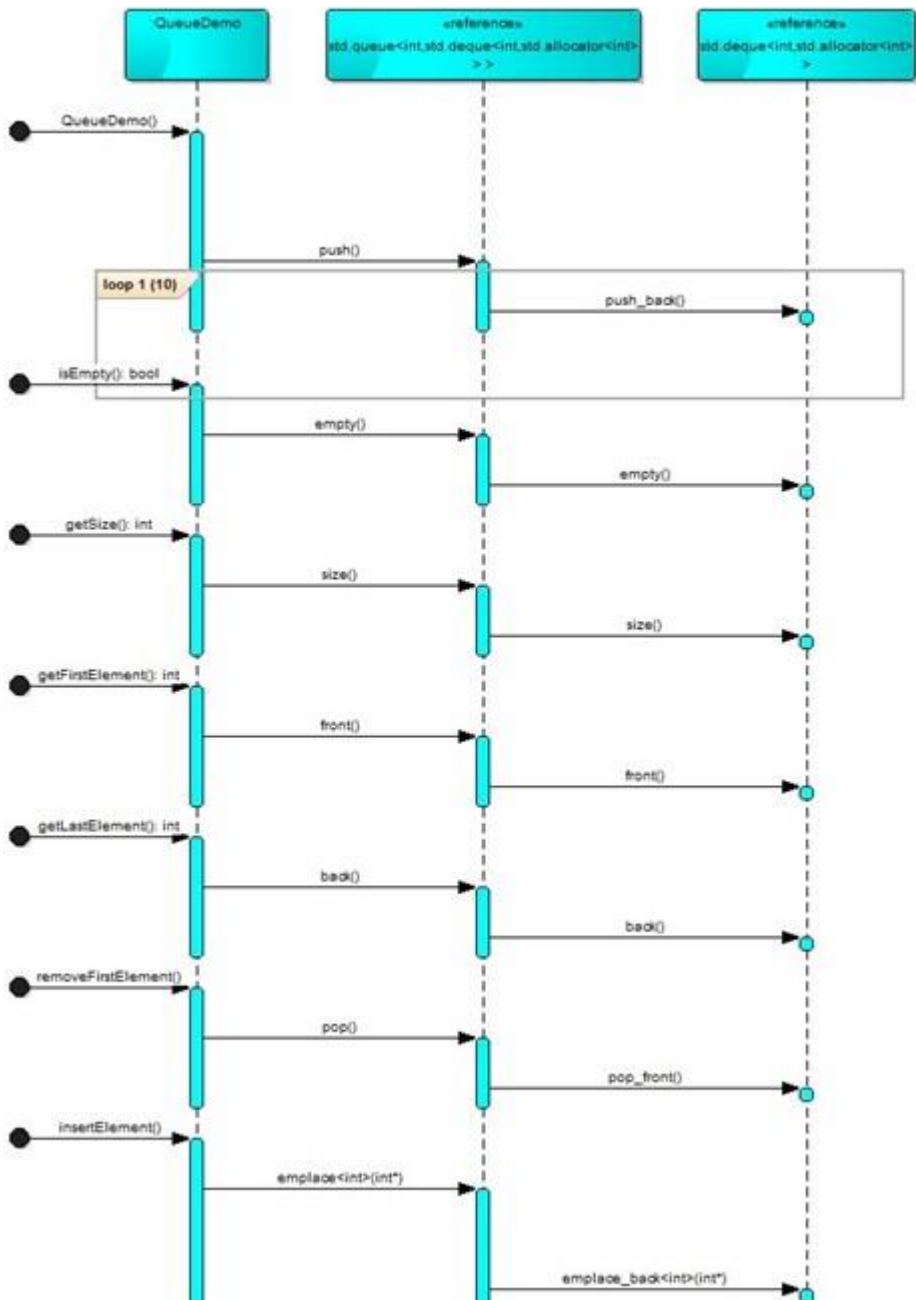
Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén **[heap]**) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektum példányra, gondoskodjunk annak megszüntetéséről, elkerülve ezzel a memóriaszivárgást. 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!**

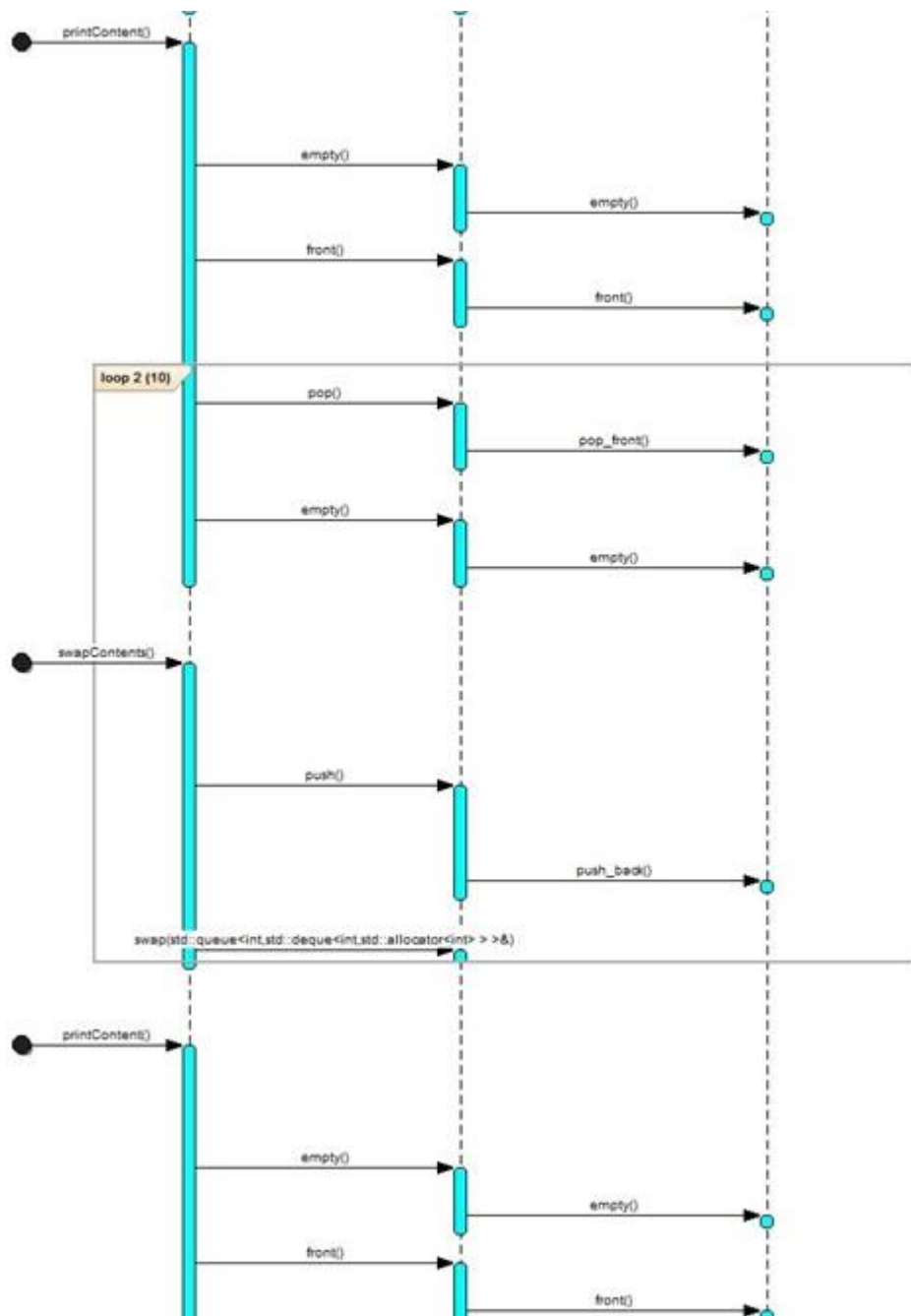
Statikus tervezés

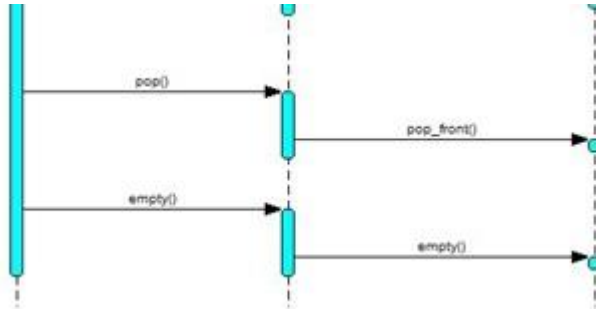


Osztálydiagram

Dinamikus diagram





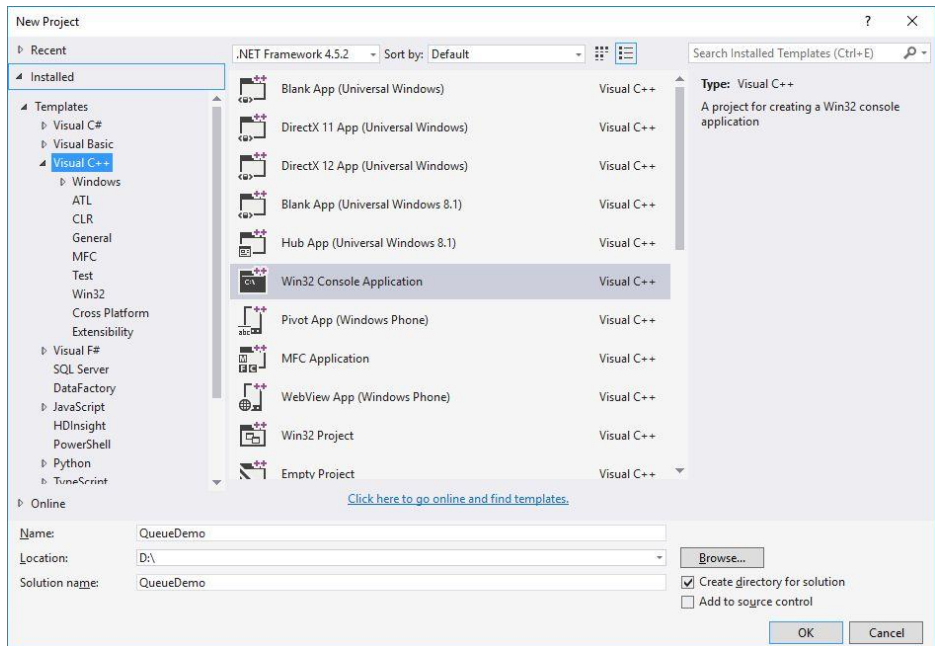


Szekvenciadiagram

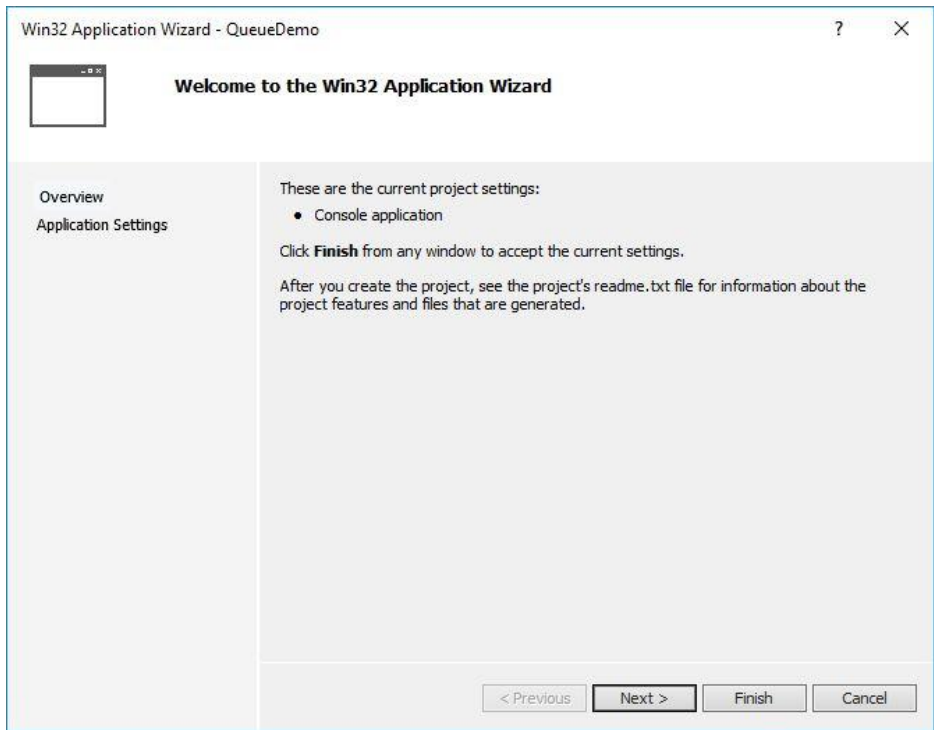
A projekt létrehozása

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.

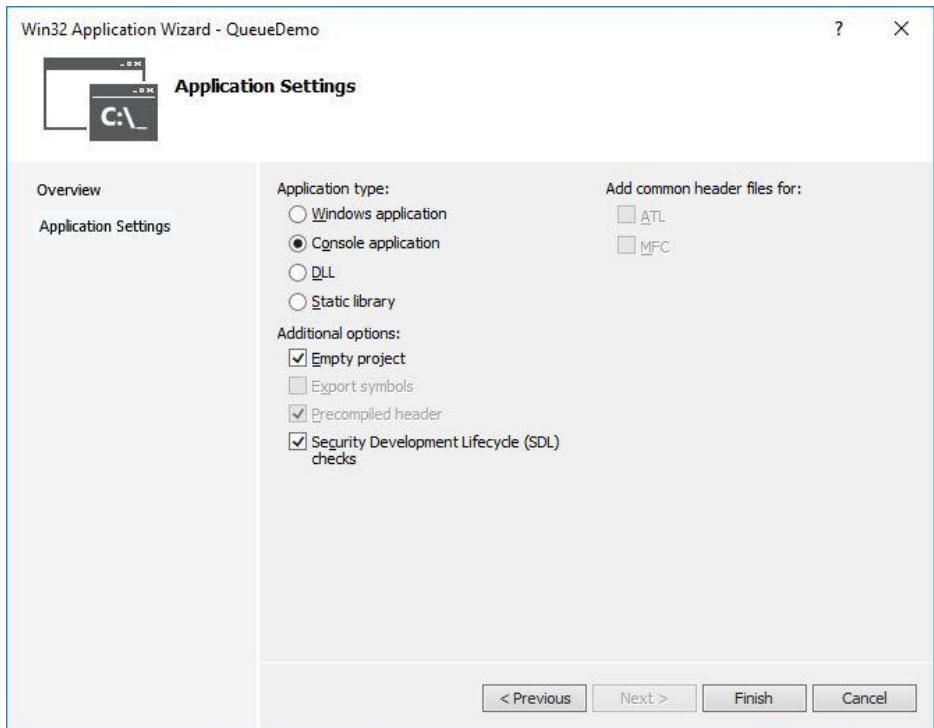
Új projekt a **File** menü **New** menüpontjában hozható létre, a **Project** almenüt kiválasztva, vagy akár a **Ctrl+Shift+N** gyorsindító gombkombinációval. Új projekt kiválasztása után megjelenő ablakban a **Visual C++ Win32** sablont (**Template**) és a **Win32 Console Application**-t kell kiválasztani,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



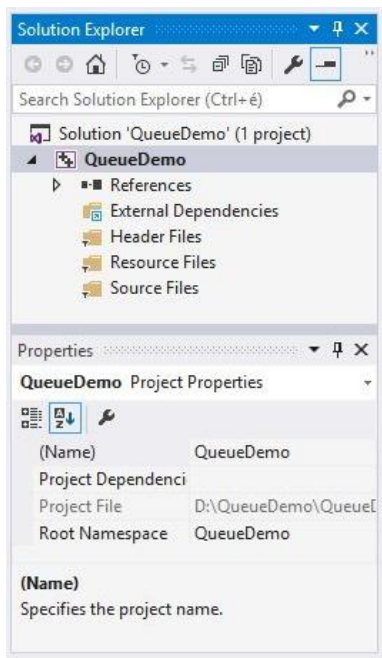
Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően a **QueueDemo** név került megadásra. Ebben a részben lehet módosítani a projekthez kapcsolódó fájlok mentési helyét is (**Location**). Egy összetett program általában több projektből tevődik össze, ezeket egységes rendszerré a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



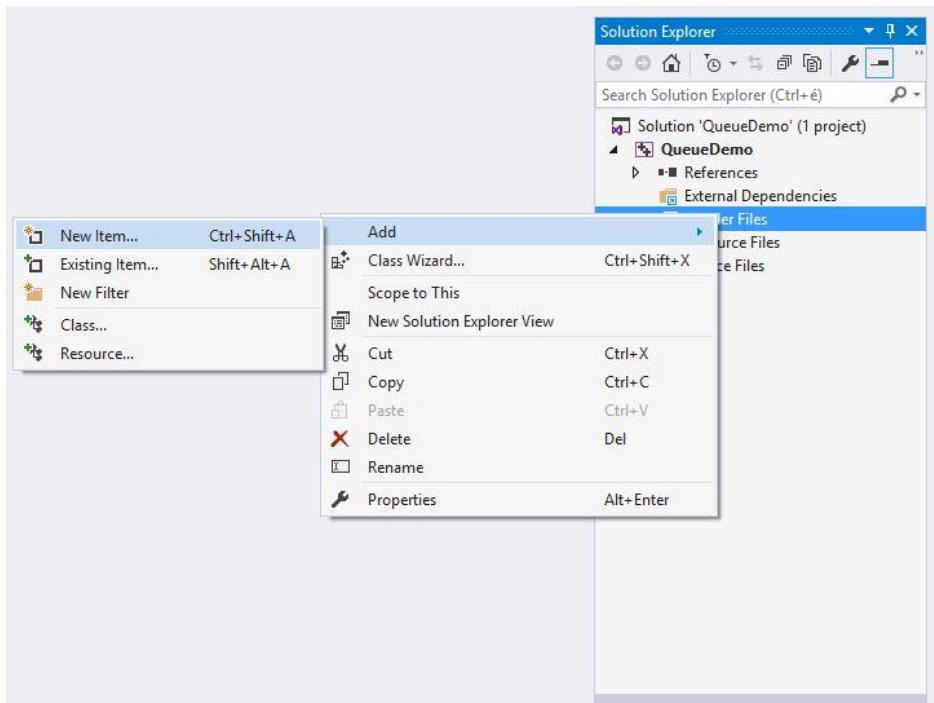
A **Next** gombra kattintva a beállítások ablak jelenik meg, itt megadhatjuk az alkalmazás típusát (**Console Application**), és néhány kiegészítő opciót. Mivel egyszerű alkalmazást készítünk, ezért célszerű az üres projekt (**Empty project**) opció kiválasztása, ahogy az alábbi képen megfigyelhető.



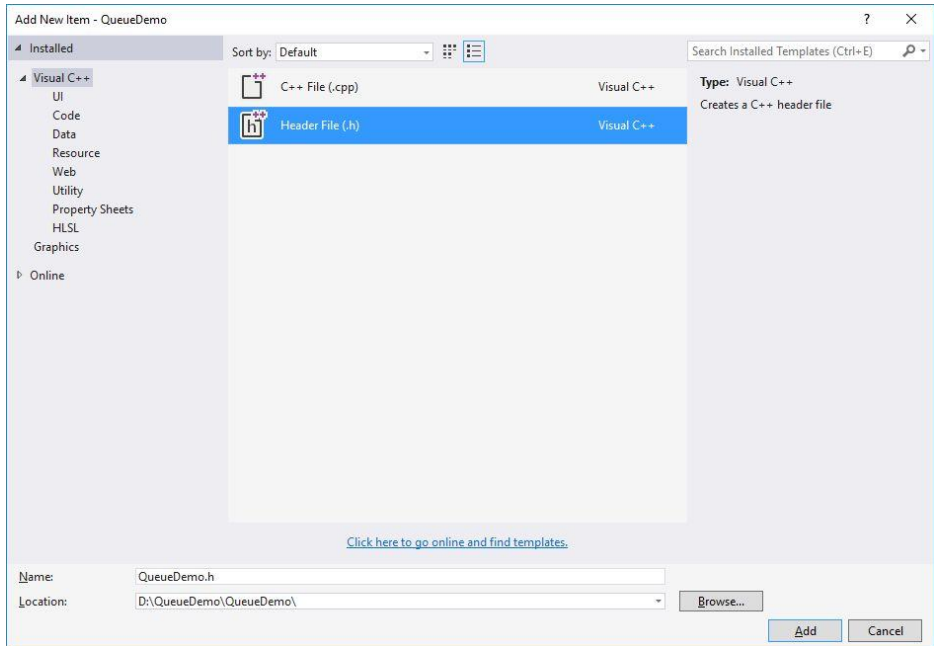
A **Finish** megnyomását követően a képernyő jobb szélén, a megoldás ablaka (**Solution Explorer**), alatta a beállítások ablak jelenik meg, ahogy a következő ábrán látható.



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



Az új fájl hozzáadásával a képernyő középső részén megjelenik a forráskód szerkesztő.

QueueDemo.h

```
#ifndef QUEUEDEMO_H
#define QUEUEDEMO_H

#include <queue>

class QueueDemo
{
private:
    std::queue<int> queue;

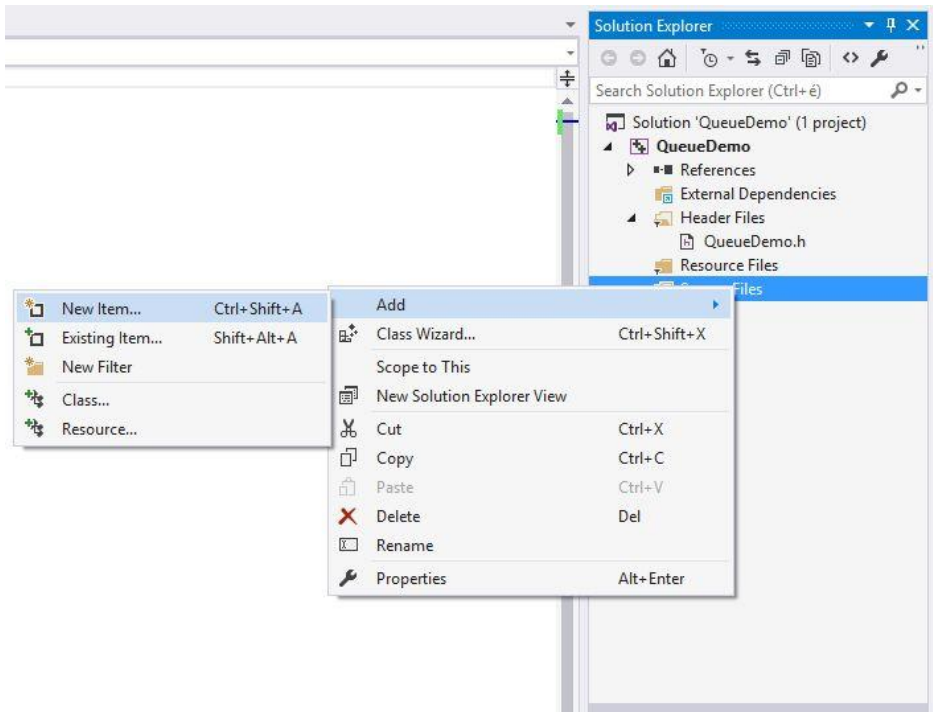
public:
```

```

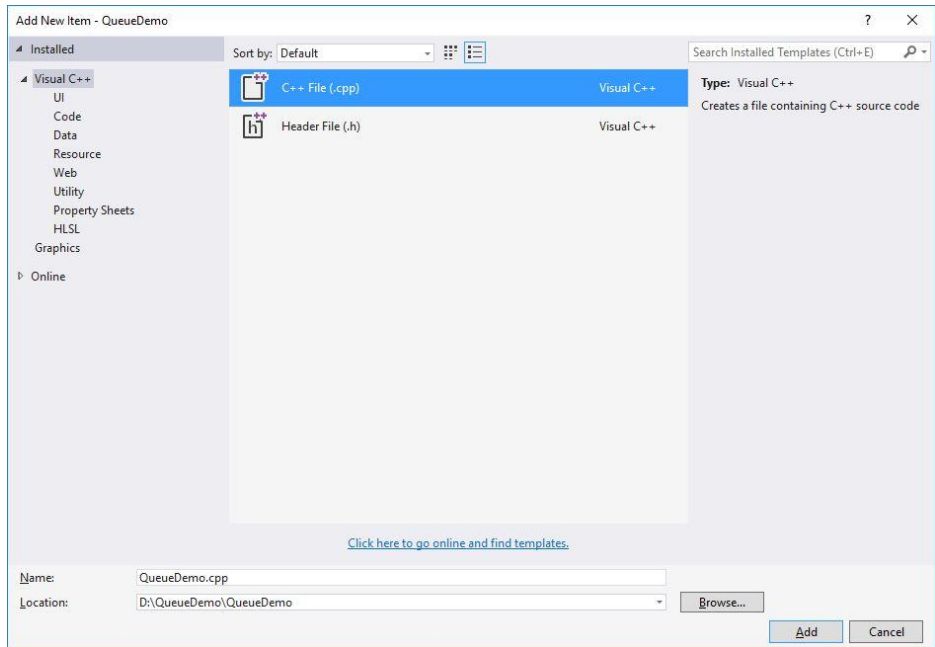
    QueueDemo();
    bool isEmpty();
    int getSize();
    int getFirstElement();
    int getLastElement();
    void insertElement();
    void removeFirstElement();
    void swapContents();
    void printContent();
};
#endif

```

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



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



QueueDemo.cpp

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

QueueDemo::QueueDemo()
{
    for (int i = 1; i <= 10; ++i)
        queue.push(i);
}

bool QueueDemo::isEmpty()
{

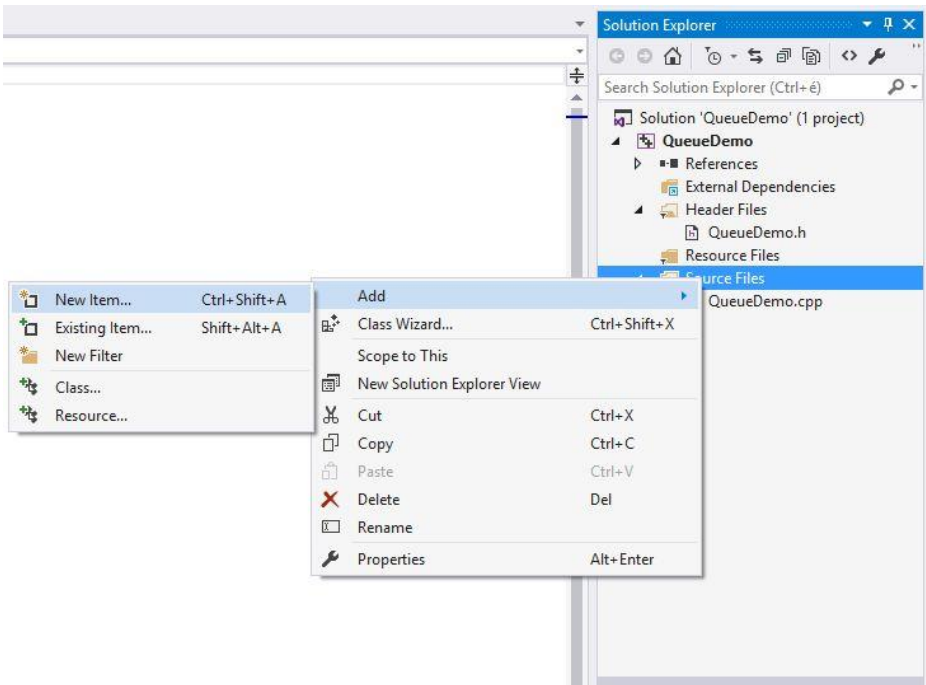
```

```
        return queue.empty();
    }
    int QueueDemo::getSize()
    {
        return queue.size();
    }
    int QueueDemo::getFirstElement()
    {
        return queue.front();
    }
    int QueueDemo::getLastElement()
    {
        return queue.back();
    }
    void QueueDemo::insertElement()
    {
        queue.emplace(100);
    }
    void QueueDemo::removeFirstElement()
    {
        queue.pop();
    }
    void QueueDemo::swapContents()
    {
        std::queue<int> queue2;
        queue2.push(11);
        queue2.push(22);
        queue.swap(queue2);
    }
    void QueueDemo::printContent()
    {
```

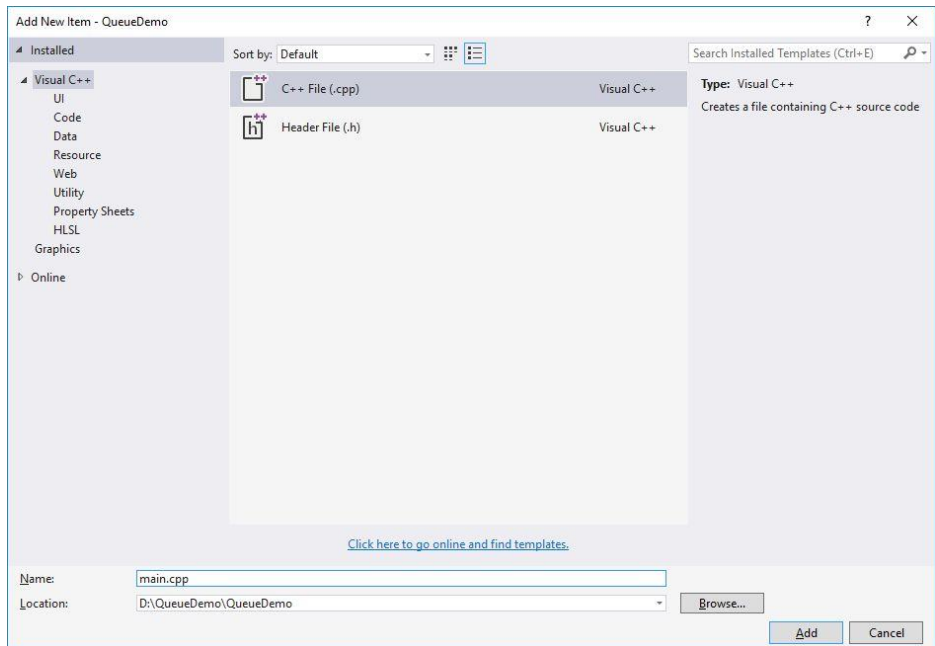
```

std::cout << "A sor elemei: ";
while (!queue.empty())
{
    std::cout << ' ' << queue.front();
    queue.pop();
}
std::cout << "\n";
}

```



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



main.cpp

```
#include <iostream>
```

```
#include <conio.h>
```

```
#include "QueueDemo.h"
```

```
int main()
```

```
{
```

```
    QueueDemo *QueueDemoObject = new QueueDemo();
```

```
    std::cout << "MUVELET => QueueDemo() #konstruktor hivas, verem feltoltese..."
```

```
    << std::endl;
```

```
    std::cout << "MUVELET => empty() #uresseg vizsgalata..." << std::endl;
```

```
    std::cout << "EREDMENY: "; std::cout << QueueDemoObject->isEmpty() <<
```

```
    std::endl;
```

```
    std::cout << "MUVELET => size() #meret lekerdezese..." << std::endl;
```

```
    std::cout << "EREDMENY: "; std::cout << QueueDemoObject->getSize()
```

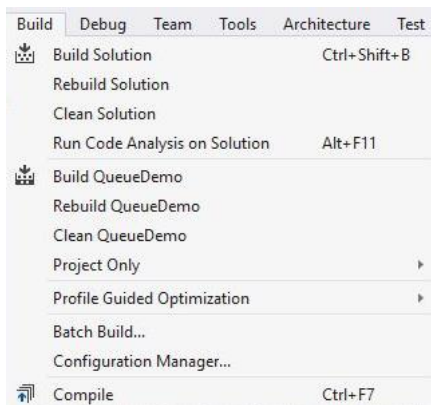
```

<< std::endl;
std::cout << "MUVELET => front() #legelso elem lekerdezese..." << std::endl;
std::cout << "EREDMENY: ";          std::cout << QueueDemoObject-
>getFirstElement(); std::cout << std::endl;
std::cout << "MUVELET => back() #legutolso elem lekerdezese..." << std::endl;
std::cout << "EREDMENY: ";          std::cout << QueueDemoObject-
>getLastElement(); std::cout << std::endl;
std::cout << "MUVELET => pop() #legelso elem torlese..." << std::endl;
QueueDemoObject->removeFirstElement();
std::cout << "MUVELET => emplace() #hozzafuzes..." << std::endl;
QueueDemoObject->insertElement();
std::cout << "EREDMENY: "; QueueDemoObject->printContent();
std::cout << "MUVELET => swap() #veremek elemeinek megcserelese..." <<
std::endl;
QueueDemoObject->swapContents();
std::cout << "EREDMENY: "; QueueDemoObject->printContent();

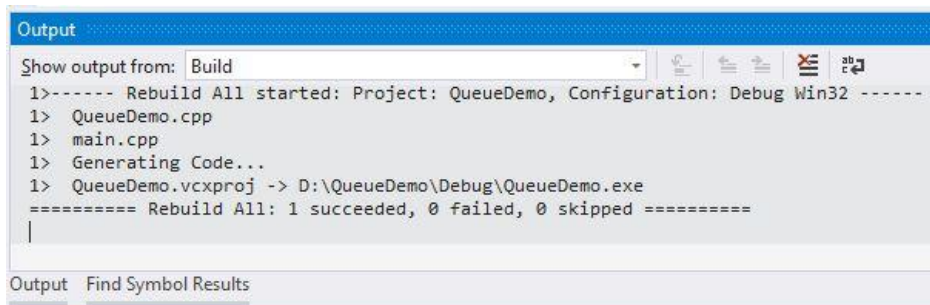
delete QueueDemoObject;
_getch();
return 0;
}

```

A forrásfájl 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.



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



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

```
D:\QueueDemo\Debug\QueueDemo.exe
>> MUVELET: Konstruktor hívása, a sor tarolo feltoltese
>> MUVELET: A sor uressegének vizsgálata empty() fuggveny hasznalataval
0
>> MUVELET: A sor tarolo merete size() fuggveny hasznalataval
10
>> MUVELET: A sor legelso elemenek lekerdezese front() fuggveny hasznalataval
1
>> MUVELET: A sor legutolso elemenek lekerdezese back() fuggveny hasznalataval
10
>> MUVELET: A sor tarolo legelso elemenek torlese pop() fuggveny segitsegevel
>> MUVELET: A sor tarolohoz torteno hozzafuzes emplace(100) fuggveny segitsegevel
>> MUVELET: A sor tarolo tartalma
A sor elemei: 2 3 4 5 6 7 8 9 10 100
>> MUVELET: Ket sor tarolo elemenek megcserelese swap() fuggveny segitsegevel
>> MUVELET: A sor tarolo tartalma
A sor elemei: 11 22
```

A feladathoz kapcsolódó magyarázatok

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

private:

```
std::queue<int> queue;
```

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:

```
QueueDemo();
```

A *getSize()* tagfüggvények a leírásban megfogalmazott sor aktuális méret lekérdezését biztosítja, amíg a *getFirstElement()* és a *getLastElement()* a sor legelső elemét, illetve legutolsó elemét adja vissza. A láthatósági szintjük nyilvános (**public**), visszatérési típusuk egész szám (**integer**) és az argumentumlistájuk üres:

```
int getSize();  
int getFirstElement();  
int getLastElement();
```

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

```
bool isEmpty();
```

Az *insertElement()* tagfüggvény segítségével új elemeket fűztünk a sorhoz a *removeFirstElement()* tagfüggvény feladata a sor legelső elemének eltávolítása, illetve a *swapContents()* tagfüggvény két sor elemét cseréli meg. Végül megfogalmazásra került egy *printContent()*, tagfüggvény, amely egy ciklus segítségével a képernyőn megjeleníti a sor aktuális elemeit. A fentebb említett négy tagfüggvény visszatérési típusa **void**, láthatósági szintjeik nyilvánosak (**public**), valamint az argumentumlistájuk üres.

```
void insertElement();  
void removeFirstElement();  
void swapContents();  
void printContent();
```

Az **QueueDemo.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 sor minden elemének 1-től 10-ig történő feltöltése, amelyet egyszerűen **push()** tagfüggvény segítségével hívásával valósítottunk meg.

```
QueueDemo::QueueDemo()  
{
```

```

    for (int i = 1; i <= 10; ++i)
        queue.push(i);
}

```

Az aktuális méret lekérdezését végrehajtó függvények implementációja során az **queue** objektumon keresztül egyszerűen hívásra került a **size()** tagfüggvény, amíg a sor legfelső és legutolsó elemének lekérdezése során a **front()** és a **back()** metódusokat hívtuk meg.

```

int QueueDemo::getSize()
{
    return queue.size();
}

int QueueDemo::getFirstElement()
{
    return queue.front();
}

int QueueDemo::getLastElement()
{
    return queue.back();
}

```

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

```

bool QueueDemo::isEmpty()
{
    return queue.empty();
}

```

Új elem elhelyezése a sorban az **emplace()** metódus használatával történt, amíg a legutolsó elem kivételét a **pop()**-al végeztük. Két sor tároló értékeinek

megcserélését a **swap()** függvénnyel hajtottuk végre. A sor tartalmának konzolon történő megjelenítésért egy olyan ciklusszervezés felel amely több **queue** objektumon keresztüli tagfüggvény hívást hajt végre.

```
void QueueDemo::insertElement()
{
    queue.emplace(100);
}
void QueueDemo::removeFirstElement()
{
    queue.pop();
}
void QueueDemo::swapContents()
{
    std::queue<int> queue2;
    queue2.push(11);
    queue2.push(22);
    queue.swap(queue2);
}
void QueueDemo::printContent()
{
    std::cout << "A sor elemei: ";
    while (!queue.empty())
    {
        std::cout << ' ' << queue.front();
        queue.pop();
    }
    std::cout << "\n";
}
```

A (**main.cpp**) állomány tartalmazza a **main()** függvényt, a program belépési pontját. A fő függvényben történő dinamikus példányosítás után

hívjuk meg a feladat teljesítéséhez szükséges függvényeket. A műveletek végrehajtását követően, gondoskodunk a lefoglalt erőforrások felszabadításáról az alapértelmezett explicit destruktor hívásának segítségével.

Minta példa a prioritásos sor (priority queue) tároló használatára Visual Studio Community fejlesztőkörnyezetben

Írjunk `PriorityQueueDemo` (*PriorityQueueDemo*) osztályt, amely különböző, prioritásos sor specifikus műveleteket ír le, majd egy objektumpéldány segítségével ezeket a műveleteket végrehajtatjuk. 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 **push()**. Az osztály adatai a következők:

- *priorityQueue*, a prioritásos sor jellegű adapter 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!

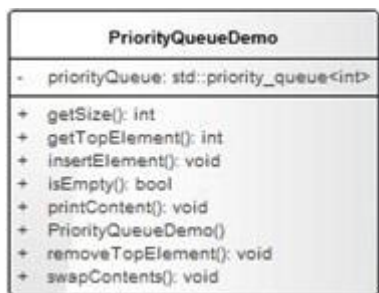
Valósítsuk meg az osztály metódusait:

- Írjuk meg az osztály konstruktorát, amely segítségével inicializáljuk a prioritásos sor tárolót 1-től 10-ig. (**push()**)
- Írjunk egy *isEmpty()* függvényt, amely megvizsgálja, hogy a prioritásos sor tartalmaz-e elemeket, majd annak függvényében térjen vissza egy logikai értékkel. Igazzal, ha található elem a prioritásos sorban, hamissal, ha üres. (**empty()**)
- Írjunk egy *getSize()* függvényt, amely a prioritásos sor tároló aktuális méretét adja vissza. (**size()**)
- Írjunk egy *getTopElement()* függvényt, amellyel a prioritásos sor legutolsó elemét kérhetjük le. (**top()**)
- Írjunk egy *removeTopElement()* függvényt, amellyel a prioritásos sor legutolsó elemét távolíthatjuk el. (**pop()**)
- Írjunk egy *insertElement()* függvényt, amely a prioritásos sorhoz történő hozzáfűzést teszi lehetővé, a hozzáfűzendő érték 100. (**emplace()**)
- Írjunk egy *swapContents()* függvényt, amely két prioritásos sor tartalmát cseréli meg. (**swap()**)

- Írjunk egy `printContent()` függvényt, amely egy ciklus segítségével kiírja a prioritásos sor elemeit.
- Ne feledkezzünk meg az objektumpéldány által lefoglalt erőforrás felszabadításáról, a destruktork megírásáról sem.

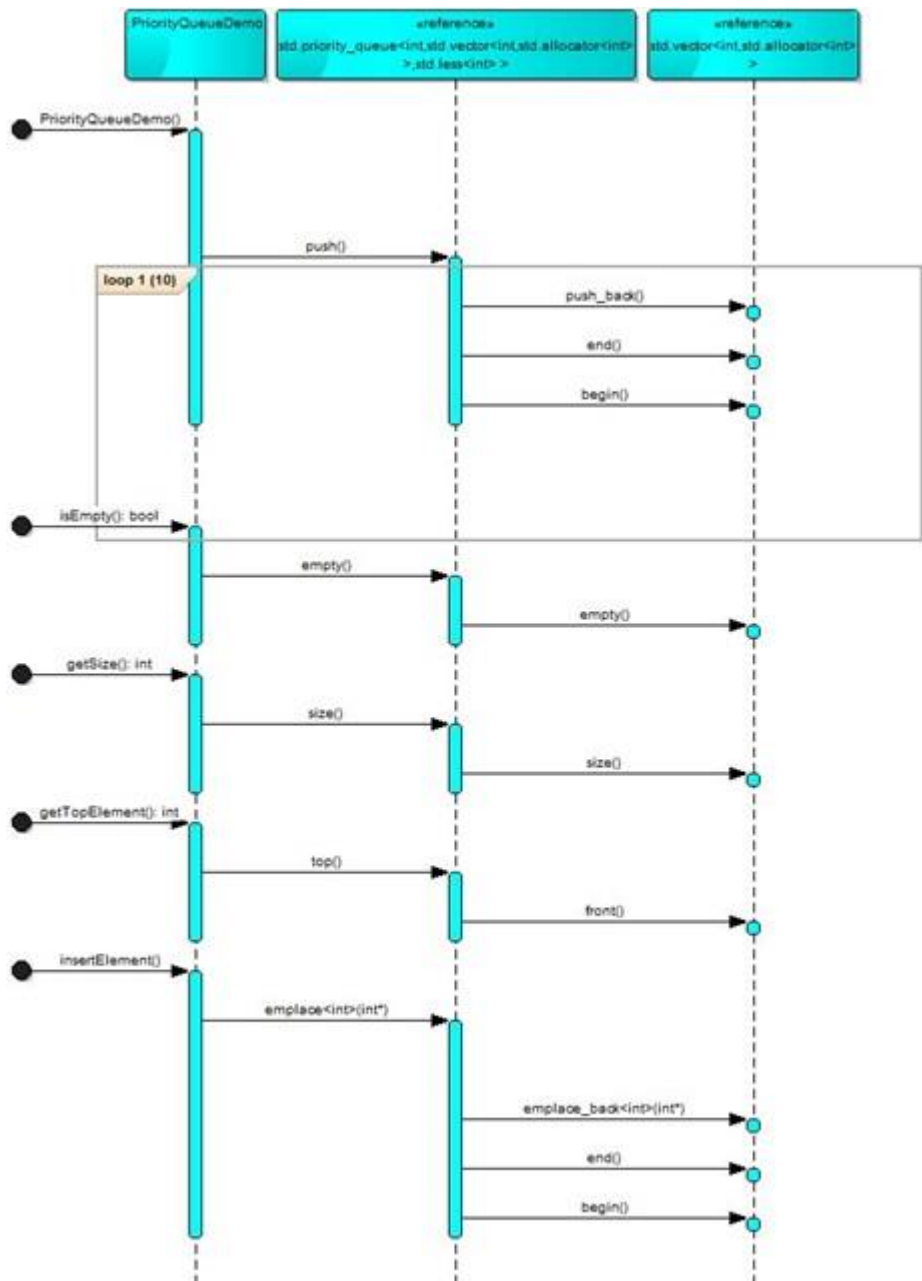
Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén **[heap]**) készítünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektum példányra, gondoskodjunk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

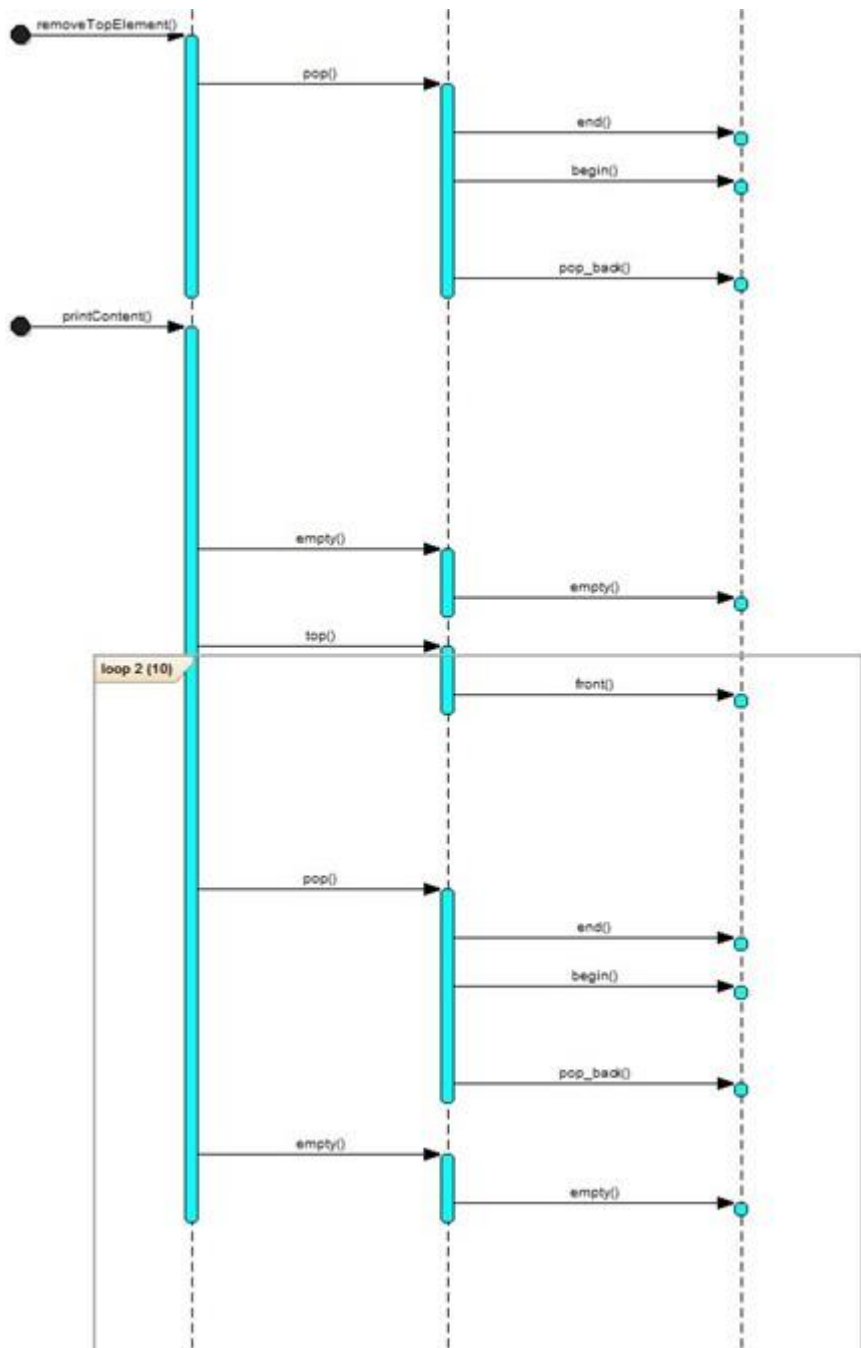
Statikus tervezés

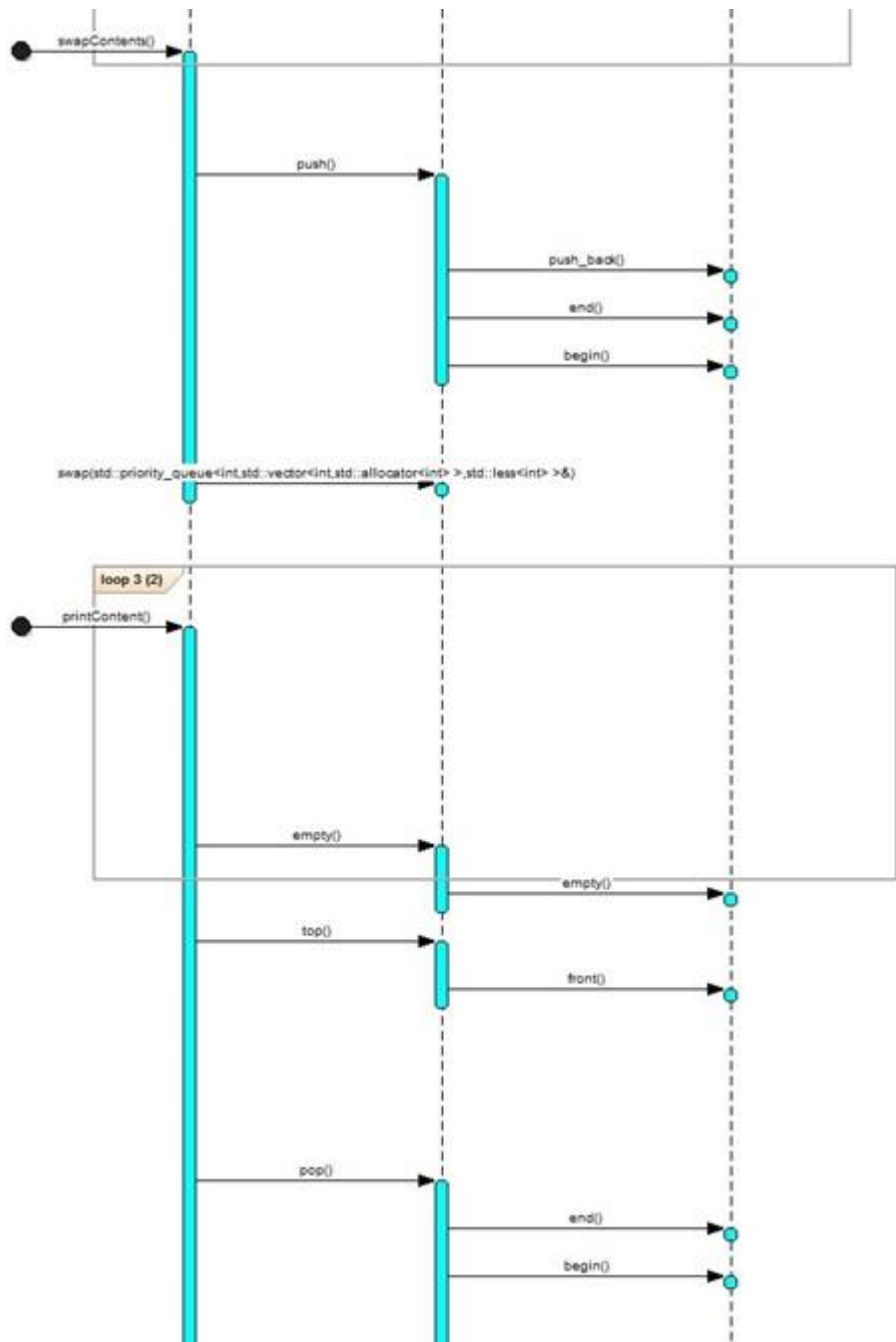


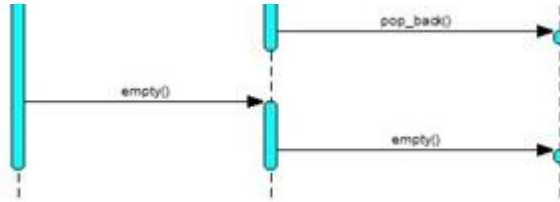
Osztálydiagram

Dinamikus diagram







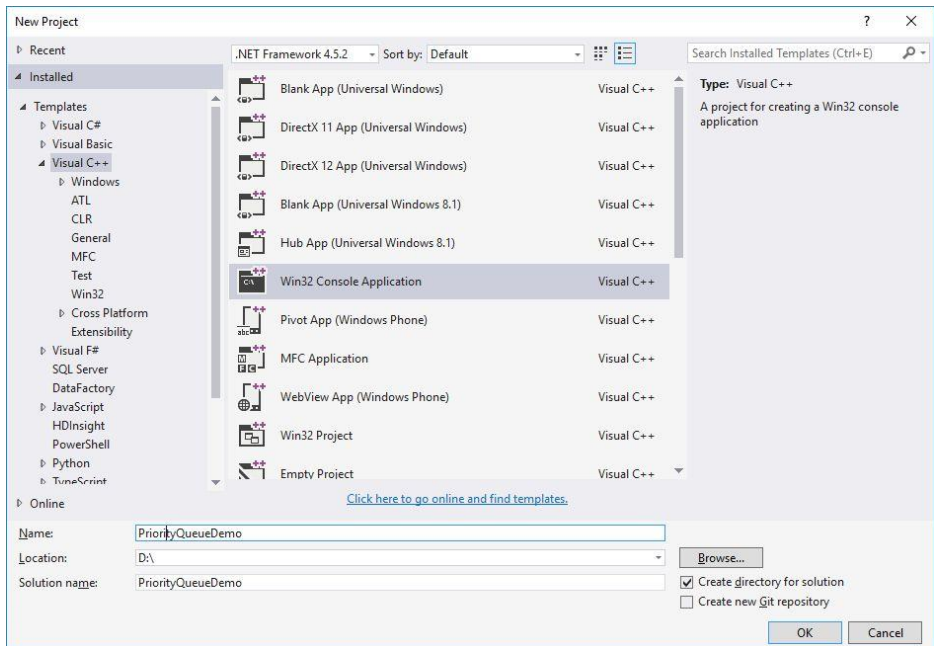


Szekvenciadiagram

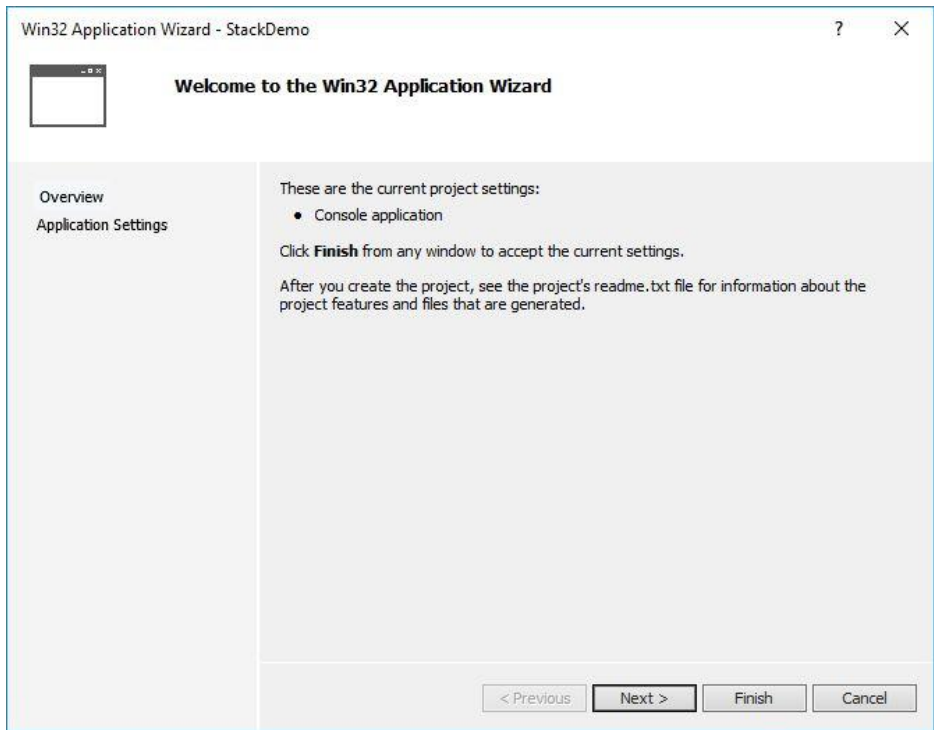
A projekt létrehozása

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.

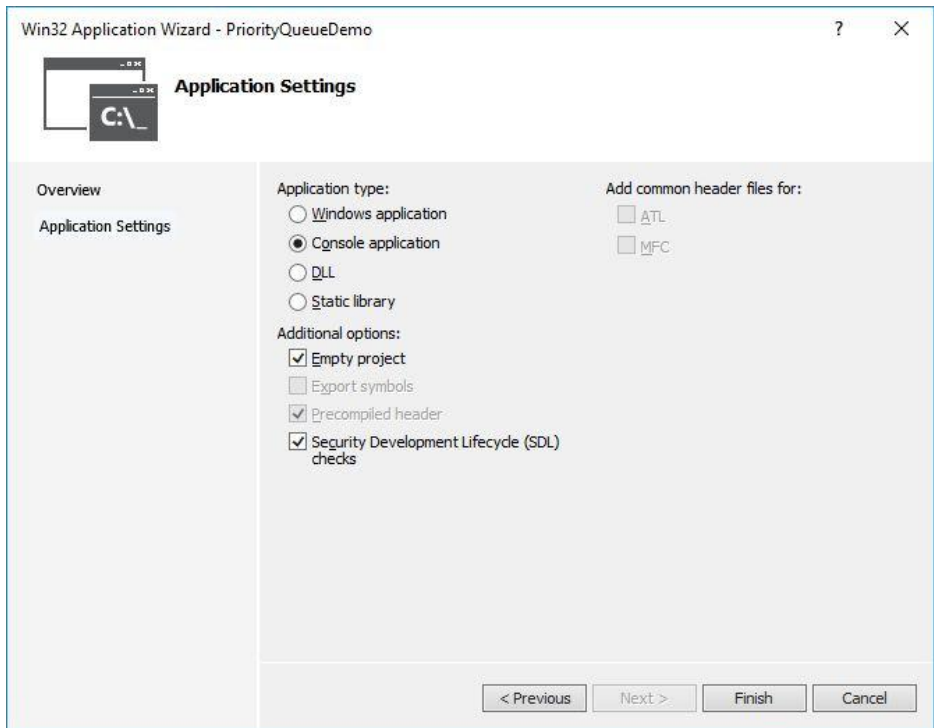
Új projekt a **File** menü **New** menüpontjában hozható létre, a **Project** almenüt kiválasztva, vagy akár a **Ctrl+Shift+N** gyorsindító gombkombinációval. Új projekt kiválasztása után megjelenő ablakban a **Visual C++ Win32** sablont (**Template**) és a **Win32 Console Application**-t kell kiválasztani,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



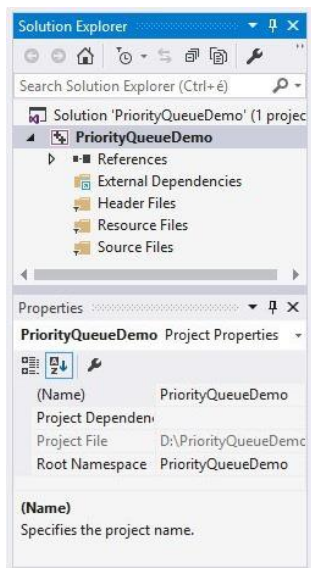
Az ablak alsó részén a projekt nevének (**Name**) a példának megfelelően a **PriorityQueueDemo** név került megadásra. Ebben a részben lehet módosítani a projekthez kapcsolódó fájlok mentési helyét is (**Location**). Egy összetett program általában több projektből tevődik össze, ezeket egységes rendszerré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



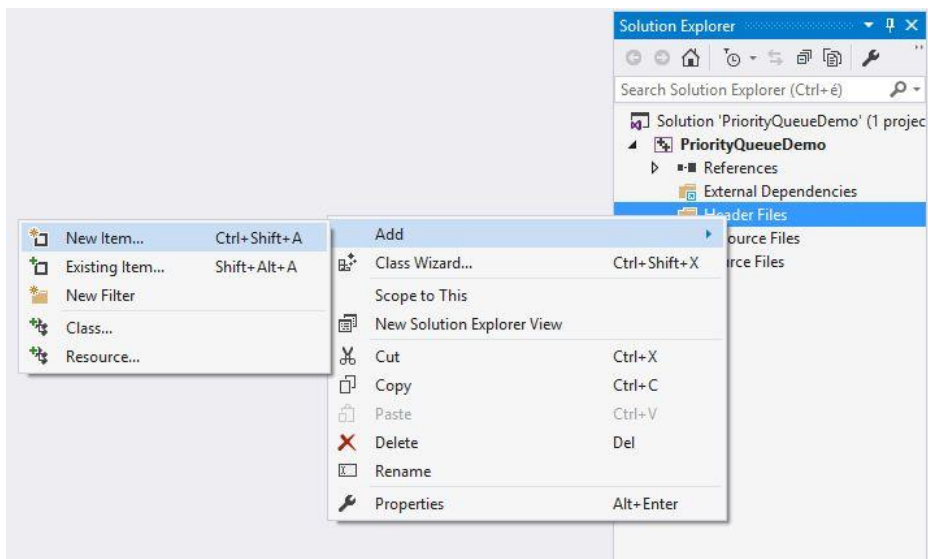
A **Next** gombra kattintva a beállítások ablak jelenik meg, itt megadhatjuk az alkalmazás típusát (**Console Application**), és néhány kiegészítő opciót. Mivel egyszerű alkalmazást készítünk, ezért célszerű az üres projekt (**Empty project**) opció kiválasztása, ahogy az alábbi képen figyelhető meg.



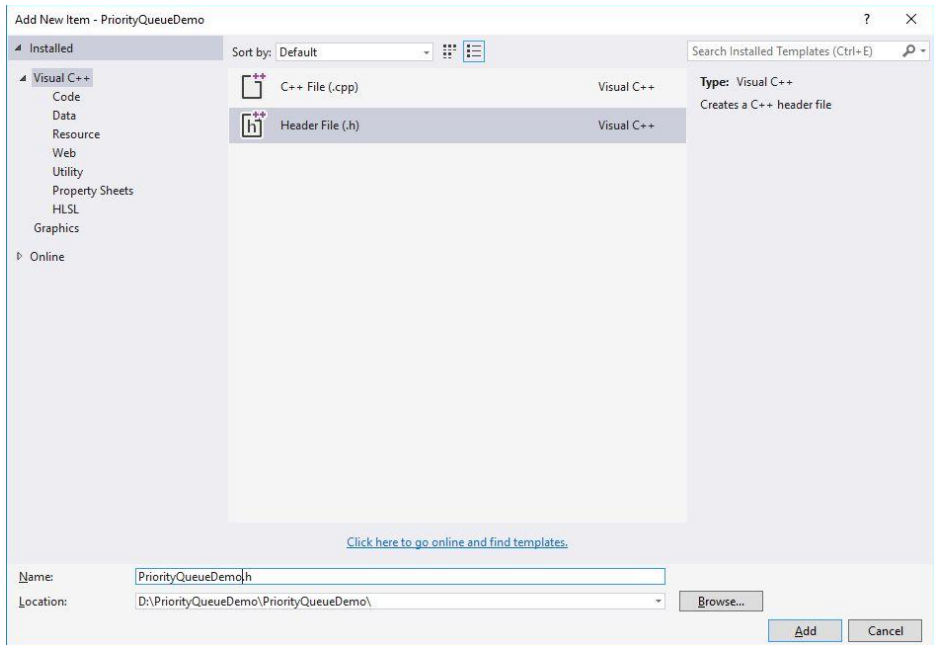
A Finish megnyomását követően a képernyő jobb szélén, a megoldás ablaka (**Solution Explorer**), alatta a beállítások ablak jelenik meg, ahogy a következő ábrán látható.



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



Az új fájl hozzáadásával a képernyő középső részén megjelenik a forráskód szerkesztő.

PriorityQueueDemo.h

```
#ifndef PRIORITYQUEUEDEMO_H  
#define PRIORITYQUEUEDEMO_H
```

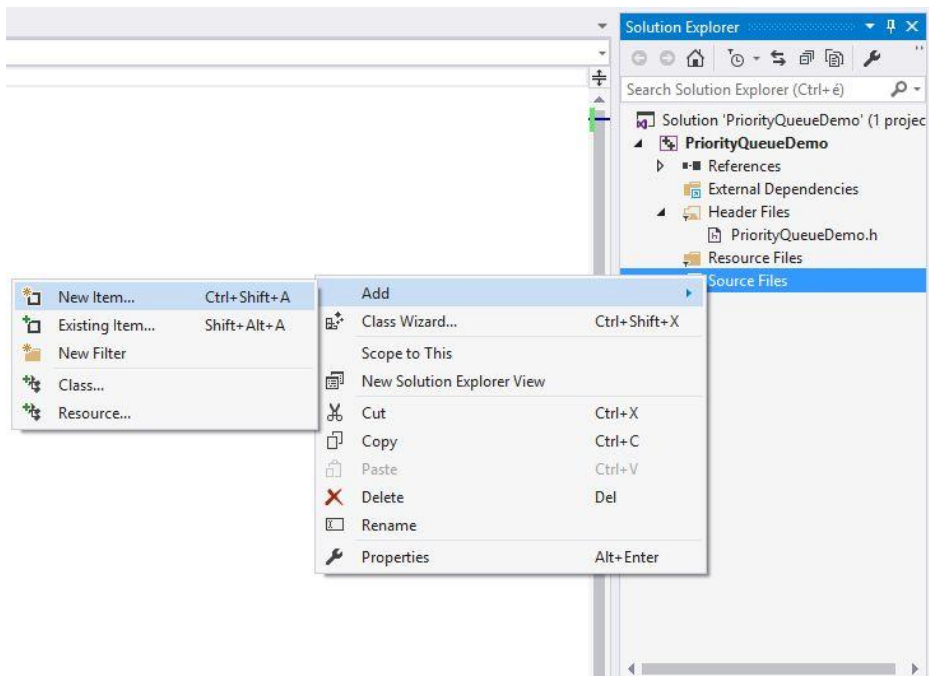
```
#include <queue>
```

```
class PriorityQueueDemo  
{
```

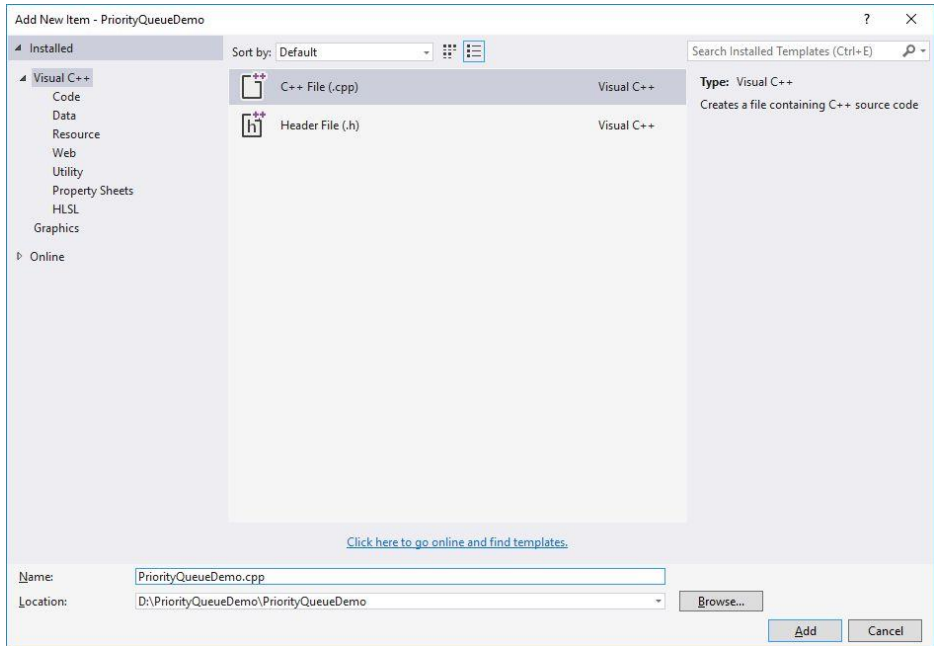
```
private:
    std::priority_queue<int> priorityQueue;

public:
    PriorityQueueDemo();
    bool isEmpty();
    int getSize();
    int getTopElement();
    void insertElement();
    void removeTopElement();
    void swapContents();
    void printContent();
};
#endif
```

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



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



PriorityQueueDemo.cpp

```
#include <iostream>

#include "PriorityQueueDemo.h"

PriorityQueueDemo::PriorityQueueDemo()
{
    for (int i = 1; i <= 10; ++i)
        priorityQueue.push(i);
}

bool PriorityQueueDemo::isEmpty()
{
    return priorityQueue.empty();
}

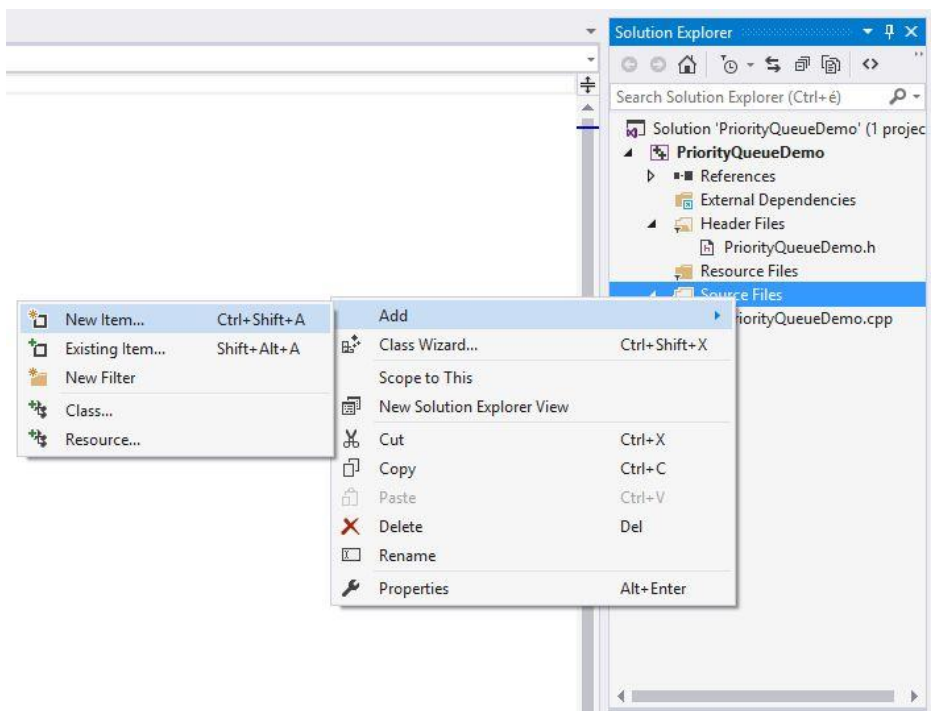
int PriorityQueueDemo::getSize()
{

```

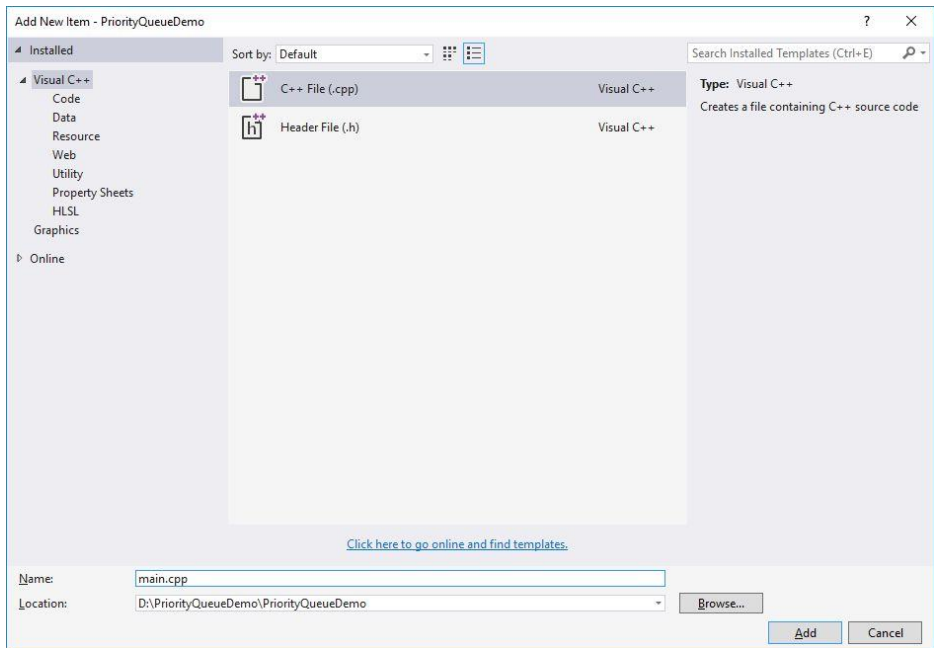
```

        return priorityQueue.size();
    }
    int PriorityQueueDemo::getTopElement()
    {
        return priorityQueue.top();
    }
    void PriorityQueueDemo::insertElement()
    {
        priorityQueue.emplace(100);
    }
    void PriorityQueueDemo::removeTopElement()
    {
        priorityQueue.pop();
    }
    void PriorityQueueDemo::swapContents()
    {
        std::priority_queue<int> PriorityQueue2;
        PriorityQueue2.push(11);
        PriorityQueue2.push(22);
        priorityQueue.swap(PriorityQueue2);
    }
    void PriorityQueueDemo::printContent()
    {
        std::cout << "A prioritások sor elemei: ";
        while (!priorityQueue.empty())
        {
            std::cout << ' ' << priorityQueue.top();
            priorityQueue.pop();
        }
        std::cout << "\n";
    }
}

```



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



main.cpp

```
#include <iostream>
```

```
#include <conio.h>
```

```
#include "PriorityQueueDemo.h"
```

```
int main()
```

```
{
```

```
    PriorityQueueDemo *PriorityQueueDemoObject = new    PriorityQueueDemo();
```

```
    std::cout << "MUVELET => PriorityQueueDemo() #Konstruktor hivasa, a  
    prioritások sor feltöltése..." << std::endl;
```

```
    std::cout << "MUVELET => empty() #üreség vizsgálat..." << std::endl;
```

```
    std::cout << "EREDMENY: "; std::cout << PriorityQueueDemoObject->isEmpty()  
    << std::endl;
```

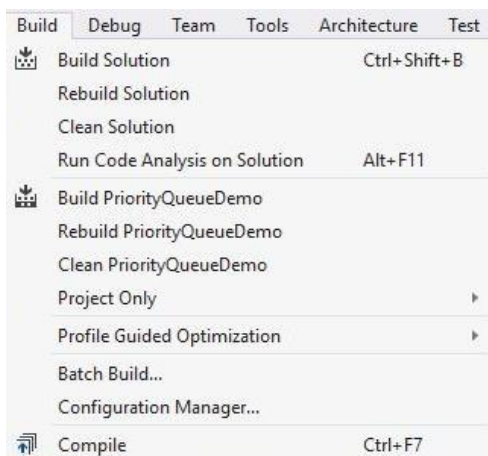
```
    std::cout << "MUVELET => size() #méret lekérdezés..." << std::endl;
```

```

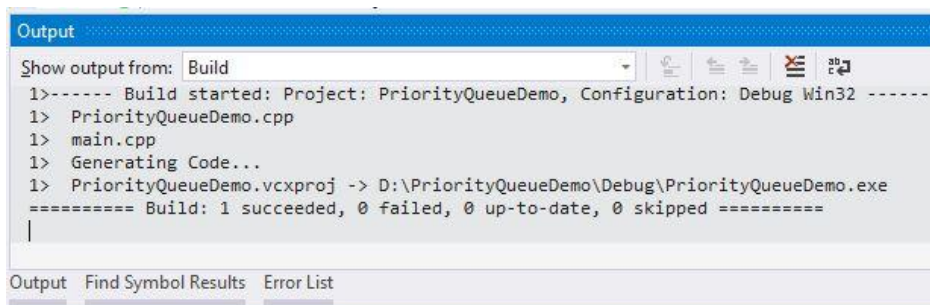
std::cout << "EREDMENY: ";          std::cout << PriorityQueueDemoObject-
>getSize() << std::endl;
std::cout << "MUVELET => top() #legutolso elem lekerdezese..." << std::endl;
std::cout << "EREDMENY: ";          std::cout << PriorityQueueDemoObject-
>getTopElement(); std::cout << std::endl;
std::cout << "MUVELET => emplace() #hozzafuzes..." << std::endl;
PriorityQueueDemoObject->insertElement();
std::cout << "MUVELET => pop() #legelso elem torlese..." << std::endl;
PriorityQueueDemoObject->removeTopElement();
std::cout << "MUVELET => pop() A prioritasos sor tarolo tartalma..." << std::endl;
std::cout << "EREDMENY: "; PriorityQueueDemoObject->printContent();
std::cout << "MUVELET => swap() #prioritasos sorok elemeinek megcserelese..."
<< std::endl;
PriorityQueueDemoObject->swapContents();
std::cout << "MUVELET => pop() A prioritasos sor tarolo tartalma..." << std::endl;
std::cout << "EREDMENY: "; PriorityQueueDemoObject->printContent();
delete PriorityQueueDemoObject;
_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.



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



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

```
D:\PriorityQueueDemo\Debug\PriorityQueueDemo.exe
MUVELET => PriorityQueueDemo() #Konstruktor hívása, a prioritásos sor feltöltése...
MUVELET => empty() #üreség vizsgálata...
EREDMENY: 0
MUVELET => size() #méret lekérdezése...
EREDMENY: 10
MUVELET => top() #legelső elem lekérdezése...
EREDMENY: 10
MUVELET => emplace() #hozzafüzes...
MUVELET => pop() #legelső elem törlése...
MUVELET => pop() A prioritásos sor taroló tartalma...
EREDMENY: A prioritásos sor elemei: 10 9 8 7 6 5 4 3 2 1
MUVELET => swap() #prioritásos sorok elemeinek megcserélése...
MUVELET => pop() A prioritásos sor taroló tartalma...
EREDMENY: A prioritásos sor elemei: 22 11
```

A feladathoz kapcsolódó magyarázatok

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

private:

```
std::priority_queue<int> priorityQueue;
```

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:

```
PriorityQueueDemo();
```

A *getSize()* tagfüggvények a leírásban megfogalmazott prioritásos sor aktuális méret lekérdezését biztosítja, amíg a *getTopElement()* a prioritásos sor legutolsó elemét adja vissza. A láthatósági szintjük nyilvános (**public**), visszatérési típusuk egész szám (**integer**) és argumentumlistájuk üres:

```
int getSize();  
int getTopElement();
```

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

```
bool isEmpty();
```

Az *insertElement()* tagfüggvény segítségével új elemeket fűztünk a prioritásos sorhoz a *removeTopElement()* tagfüggvény feladata a prioritásos sor legutolsó elemének eltávolítása, illetve a *swapContents()* tagfüggvény két prioritásos sor elemét cseréli meg. Végül megfogalmazásra került egy *printContent()*, tagfüggvény, amely egy ciklus segítségével a képernyőn megjeleníti a prioritásos sor aktuális elemeit. A fentebb említett négy tagfüggvény visszatérési típusaik **void**, láthatósági szintjeik nyilvánosak (**public**), valamint az argumentumlistájuk üres.

```
void insertElement();  
void removeTopElement();  
void swapContents();  
void printContent();
```

A **PriorityQueueDemo.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 prioritásos sor minden elemének 1-től 10-ig történő feltöltése, amelyet egyszerűen **push()** tagfüggvény segítségével hívásával valósítottunk meg.

```
PriorityQueueDemo::PriorityQueueDemo()  
{  
    for (int i = 1; i <= 10; ++i)  
        priorityQueue.push(i);
```

```
}
```

Az aktuális méret lekérdezését végrehajtó függvények implementációja során az **priorityQueue** objektumon keresztül egyszerűen hívásra került a **size()** tagfüggvény, amíg a sor legutolsó elemének lekérdezése során a **top()** metódusokat hívtuk meg.

```
int PriorityQueueDemo::getSize()
{
    return priorityQueue.size();
}

int PriorityQueueDemo::getTopElement()
{
    return priorityQueue.top();
}
```

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

```
bool PriorityQueueDemo::isEmpty()
{
    return priorityQueue.empty();
}
```

Új elem elhelyezése a sorban az **emplace()** metódus használatával történt, amíg a legutolsó elem kivételét a **pop()**-al végeztük. Két prioritásos sor tároló értékeinek megcserélését a **swap()** függvénnyel hajtottuk végre. A prioritásos sor tartalmának konzolon történő megjelenítésért egy olyan ciklusszervezés felel amely több **priorityQueue** objektumon keresztüli tagfüggvény hívást hajt végre.

```
void PriorityQueueDemo::insertElement()
{
```

```

    priorityQueue.emplace(100);
}
void PriorityQueueDemo::removeTopElement()
{
    priorityQueue.pop();
}
void PriorityQueueDemo::swapContents()
{
    std::priority_queue<int> PriorityQueue2;
    PriorityQueue2.push(11);
    PriorityQueue2.push(22);
    priorityQueue.swap(PriorityQueue2);
}
void PriorityQueueDemo::printContent()
{
    std::cout << "A prioritásos sor elemei: ";
    while (!priorityQueue.empty())
    {
        std::cout << ' ' << priorityQueue.top();
        priorityQueue.pop();
    }
    std::cout << "\n";
}

```

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

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.

Írjunk egy vermet szimuláló osztályt (*Stack*), amelynek az objektumpéldánya, az osztályban leírtak alapján, képes tetszőleges típusokkal (osztály sablon), veremre jellemző viselkedések és műveletek elvégzésére. Az osztály adatai a következők:

- Egy sablon típusú (*T*) adattag (*values*), amely a verem elemeinek az értékeit tárolja.
- Egy *int* típusú adattag (*top*), amely a verem legfelső elemét tárolja.
- Egy *int* típusú adattag (*maxSize*), amely a verem maximális méretét tárolja.

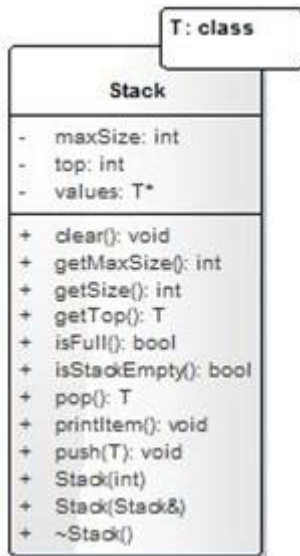
Ügyeljünk az egységbezárás elvére, tehát az adattagok **private** láthatóságu szinttel rendelkezzenek!

- Írjuk meg az osztály paraméteres konstruktorát (*Stack*), amely létrehoz egy üres, maximálisan 5 elem tárolására képes vermet.
- Írjuk meg az osztály másoló konstruktorát (*Stack(const Stack<T>& object)*), amely a paraméterül kapott verem objektum elemeiről egy másolatot hoz létre a memória dinamikus területén.
- Írjunk egy *isEmpty()* függvényt, amely a **top** érték segítségével eldönti, hogy a verem üres-e vagy sem, majd ennek megfelelően térjen vissza egy logikai értékkel.
- Írjunk egy *isFull()* függvényt, amely a **top** és **maxSize** változók értékeinek segítségével eldönti a veremről, hogy tele van-e vagy sem.
- Írjunk egy *push(T value)* függvényt, amely a paraméterül kapott új értéket a verembe helyezi. A sikeres verembe történő helyezést követően ne feledkezzünk meg a **top** változó értékének megváltoztatásáról sem.
- Írjunk egy *pop()* függvényt, amely a veremből kiveszi az elemeket. A sikeres veremből történő kivételt követően ne feledkezzünk meg a **top** változó értékének megváltoztatásáról sem.
- Írjunk egy *getTop()* függvényt, amely lekérdezi a verem tetején lévő elemet.

- Írjunk egy *clear()* függvényt, amely törli a veremből az elemeket és mindent kezdő pozícióba állít (*top=0*).
- Írjunk egy *getSize()* függvényt, amely lekérdezi az aktuális verem méretét.
- Írjunk egy *getMaxSize()* függvényt, amely lekérdezi az aktuális verem maximális méretét.
- Írjunk egy *printItem()* függvényt, amely a verem elemeit megjeleníti a konzolon.
- Ne feledkezzünk meg az objektumpéldány által lefoglalt erőforrás felszabadításáról, a destruktor megírásáról sem.
- Írjunk egy egyszerű *main* függvényt, ahol létrehozunk az objektumpéldányokat, majd az objektumon keresztül szimuláljuk egy verem viselkedését.

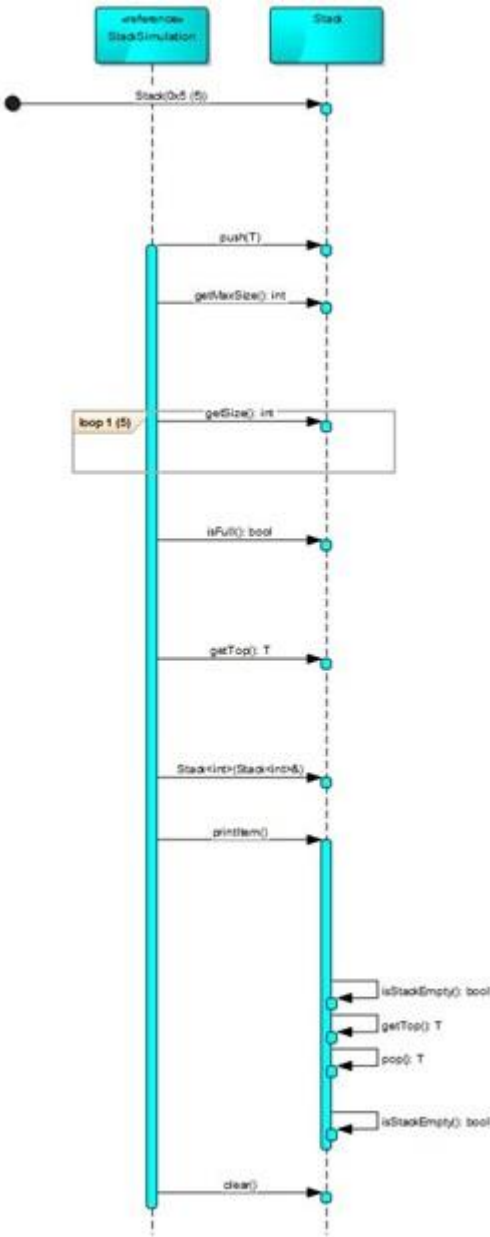
Az osztály implementálását követően dinamikus módszerrel (a memória dinamikus területén [**heap**]) készítsünk egy objektumpéldányt. Az objektumpéldány segítségével a feladat leírásnak megfelelően hívjuk meg az osztály egyes tagfüggvényeit. Amennyiben nincs tovább szükség az objektum példányra, gondoskodjunk 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 tervezés



Osztálydiagram

Dinamikus diagram



Szekvenciadiagram

Megoldás

StackSimulation.h

```
#ifndef STACKSIMULATION_H
#define STACKSIMULATION_H

#include <iostream>

template<class T> class Stack
{
    private:
        T* values;
        int top;
        int maxSize;

    public:
        Stack(int size = 5);
        Stack(const Stack& object);
        bool isEmpty();
        bool isFull();
        void push(T value);
        T pop();
        T getTop();
        void clear();
        int getSize();
        int getMaxSize();
        void printItem();
        ~Stack();
};
```

```
#endif
```

StackSimulation.cpp

```
#include <iostream>
```

```
#include "StackSimulation.h"
```

```
using namespace std;
```

```
template <class T> Stack<T>::Stack(int size)
```

```
{  
    if (size < 1)  
    {  
        cout << "A verem mérete hibás."  
    }  
    values = new T[size];  
    top = 0;  
    maxSize = size;  
}
```

```
template <class T> Stack<T>::Stack(const Stack<T>& object)
```

```
{  
    values = new T[object.maxSize];  
    top = object.top;  
    maxSize = object.maxSize;  
  
    for (int i = 0; i < top; ++i)  
    {  
        values[i] = object.values[i];  
    }  
}
```

```
template <class T> bool Stack<T>::isEmpty()
{
    return (top == 0) ? true : false;
}
```

```
template <class T> bool Stack<T>::isFull()
{
    return (top == maxSize) ? true : false;
}
```

```
template <class T> void Stack<T>::clear()
{
    top = 0;
}
```

```
template <class T> int Stack<T>::getSize()
{
    return top;
}
```

```
template <class T> int Stack<T>::getMaxSize()
{
    return maxSize;
}
```

```
template <class T> void Stack<T>::push(T value)
{
    if (top < maxSize)
    {
        values[top] = value;
        ++top;
    }
}
```

```
    }  
    else  
    {  
        cout << "A verem megtelt."  
    }  
}
```

```
template <class T> T Stack<T>::pop()  
{  
    if (top > 0)  
    {  
        --top;  
        return values[top + 1];  
    }  
    else  
    {  
        cout << "A verem üres."  
    }  
}
```

```
template <class T> T Stack<T>::getTop()  
{  
    if (top > 0)  
    {  
        return values[top - 1];  
    }  
    else  
    {  
        cout << "A verem üres."  
    }  
}
```

```

template <class T> void Stack<T>::printItem()
{
    cout << "A veremben talalhato elemek:" << endl;
    while (!isEmpty())
    {
        T topElement = getTop();
        cout << topElement << endl;
        pop();
    }
}

```

```

template <class T> Stack<T>::~~Stack()
{
    delete[] values;
}

```

main.cpp

```

#include <iostream>
#include <conio.h>
#include "StackSimulation.cpp"

```

```

using namespace std;

```

```

int main()
{
    Stack<int> *stackObject = new Stack<int>();
    cout << "A verem feltoltese..." << endl;
    stackObject->push(3);
    stackObject->push(2);
    stackObject->push(5);
}

```

```

stackObject->push(7);
stackObject->push(8);

cout << "A verem maximalis merete? : " << stackObject->getMaxSize() << endl;
cout << "A verem aktualis merete? : " << stackObject->getSize() << endl;
cout << "A verem tele? : " << stackObject->isFull() << endl;
cout << "A verem tetejen levo elem? : " << stackObject->getTop() << endl;
Stack<int> *extendedStackObject = new Stack<int>(*stackObject);

stackObject->printItem();
extendedStackObject->printItem();
extendedStackObject->clear();
stackObject->clear();

delete extendedStackObject;
delete stackObject;

_getch();
return 0;
}

```

Program futásának eredménye:

```
D:\StackSimulation\Debug\StackSimulation.exe
A verem feltoltese...
A verem maximalis merete? : 5
A verem aktualis merete? : 5
A verem tele? : 1
A verem tetejen levo elem? : 8
A veremben talalhato elemek:
8
7
5
2
3
A veremben talalhato elemek:
8
7
5
2
3
```

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://calmarius.net/?lang=hu&page=programming%2Fcpp_standard_library#stl_containers

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