

Katona József–Király Zoltán

# OBJEKTUMORIENTÁLT SZOFTVERFEJLESZTÉS C++ NYELVEN

STL-TÁROLÓK

STL (Standard Template Library) tárolókról általánosságban  
Tárolók, Iterátorok, Algoritmusok  
Térkép a legmegfelelőbb STL-tároló kiválasztásához



Dunakavics  
D•U•F PRESS

I  
N  
F  
O  
R  
M  
A  
T  
I  
K  
A  
I

K  
Ö  
N  
Y  
V  
E  
K

3.

Katona József–Király Zoltán

OBJEKTUMORIENTÁLT SZOFTVERFEJLESZTÉS C++ NYELVEN

STL-TÁROLÓK

© Katona József–Király Zoltán, 2015

Lektorálta: Dr. Kusper Gábor, tanszékvezető főiskolai docens

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

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

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

Készült a HTSART nyomdában.  
Felelős vezető: Halász Iván

ISBN 978-963-9915-54-1  
ISSN 2415-9115

Katona József–Király Zoltán

OBJEKTUMORIENTÁLT  
SZOFTVERFEJLESZTÉS  
C++ NYELVEN

STL-TÁROLÓK

STL (Standard Template Library) tárolókról általánosságban

Tárolók, Iterátorok, Algoritmusok

Térkép a legmegfelelőbb STL-tároló kiválasztásához



Dunakavics

D·U·F PRESS

Dunaújváros, 2015

I  
N  
F  
O  
R  
M  
A  
T  
I  
K  
A  
I

K  
Ö  
N  
Y  
V  
E  
K

3.



# Tartalom

## 1. Fejezet

STL- (Standard Template Library) tárolókról általánosságban	9
Tárolók	9
Szekvenciális tárolók (vector, list, deque)	10
Asszociatív tárolók (set, multiset, map, multimap)	10
Halmazok és multi-halmazok (set, multiset)	10
Asszociatív tömbök (map, multimap)	10
Iterátorok	13
Algoritmusok	15
Térkép a legmegfelelőbb STL-tároló kiválasztásához	15
Gyakran használt tároló tagfüggvények és felelősségeik	16
Dinamikus tömb (Vector) mintapélda	19
Kétszeresen láncolt lista (List) mintapélda	19
Kétfélgű sor (Deque) mintapélda	16
Halmaz (Set) mintapélda	19
Multi-halmaz (Multiset) mintapélda	19
Map (asszociatív tömb) mintapélda	19
Multimap mintapélda	19
Gyakorló feladatok	20
<b>2. Felhasznált irodalom</b>	<b>32</b>



# Előszó

Úgy gondoljuk, hogy ez a könyvsorozat hozzájárulhat ahhoz is, hogy különböző szakemberek vagy akár középiskolai diákok is kamatoztathassák tudásukat a témában.

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

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

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

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

*A Szerzők*

*\*OOP: angolul object-oriented programming egy programozási módszertan*



# *Objektumorientált programozás*

## *C++ nyelven*

*(STL-tárolók)*

### *1. Fejezet*

#### STL- (Standard Template Library) tárolókról általánosságban

Az úgynevezett **STL- (Standard Template Library)** tárolók az elmúlt évek legnagyobb újítása, amit a C++ nyelv területén alkotattak. A könyvben ismertetett példák alapján kezdő **STL**-programozóvá válhatunk. Az **STL**-ek olyan objektumok, amelyek képesek arra, hogy más objektumokat tároljanak. Számos olyan függvény-megvalósítást tartalmaz, amelyeket széles körben alkalmaznak különböző problémák megoldására.

A könyvtárban olyan adatszerkezeteket valósítottak meg, mint például a dinamikus vektorok, kétszeresen láncolt listák, halmazok, multi-halmazok, és az asszociatív tömbök.

Mivel az **STL**-ek sablonokkal (**template**) dolgoznak, bármilyen típussal képesek együttműködni. Az **STL**-könyvtár három fő részből épül fel:

- tárolók;
- iterátorok (egyfajta mutató);
- algoritmusok.

Ezek együttese számos megoldást kínál, különböző programozási problémákra.

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

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

## Tárolók

A gyakorlatban viszonylag ritkán írunk tároló osztályokat, hiszen azokat az **STL**-ben megírták és mindenki számára elérhetővé tették. A különböző tárolók elemeinek mindig van egy meghatározott sorrendje. A leggyakrabban a következő tárolóelemeket használjuk:

Tároló	Leírás
<b>vector</b>	dinamikus tömb
<b>set</b>	halmaz
<b>list</b>	kétszeresen
<b>map</b>	láncolt lista
	asszociatív tömb

A táblázatba foglalt tárolókat a további fejezetekben/alfejezetekben ismerhetjük meg alaposabban.

### SZEKVENCIÁLIS TÁROLÓK (VECTOR, LIST, DEQUE)

A szekvenciális tárolóknál a sorrendet a programozó határozza meg. Az **STL** három különböző szekvenciális tárolót tartalmaz (implementál): a **vector**, a **list** és a **deque**. Ezek a tárolók különböző módon valószínűleg 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.

– **vector**: A vektor lényegében egy dinamikus tömb, amelyek folyamatos memóriaterületen helyezkednek el. A már jól megszokott egyszerű tömböknél, itt is lehetőség van indexelt hozzáférésre. A vektor végéhez való hozzáfűzést vagy épp törlését nagyon gyorsan hajtja végre, abban az esetben, ha az elemeket odébb kell csúsztatni lassúvá, válhat, hiszen ez sok másolási művelettel jár. A vektornak két fontos mérőszáma van, az egyik a méret (**size**), a másik a kapacitás (**capacity**). 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. Tehát a vektor kapacitásnak mindig nagyobbak kell lennie, mint a méretének. Az első az úgynevezett (**capacity**) függvény adja vissza, a másodikat a (**size**). A kapacitást manuálisan a (**reserve**) függvény segítségével állíthatjuk be. Ha a tömb elejére szeretnénk beszúrni egy elemet, akkor a vektor (**insert**) függvényét kell meghívni, amely egy iterátort és egy beszúrandó értéket vár. A tagfüggvény mindig a megadott iterátorral jelzett elem elé szúrja be az új elem értékét. A vektorból való törlés többféle módon is történhet. Ha az utolsó elemet szeretnénk törölni, akkor a **pop\_back()** tagmetódus használható. Ha egy tetszőleges elem kitörlése a cél, akkor azt az **erase()** függvénnyel tehetjük meg, amely a törlendő elemre mutató iterátort várja paraméterül. Az összes elem törléséhez a **clear()** metódus áll rendelkezésre. A vektort, mint egy tömböt is használhatjuk, mivel folyamatos memóriaterületen tárolja az elemeit.

– **list**: Ezzel a tárolóval valósítjuk meg a kétszeresen láncolt listát, amely az jelenti, hogy a beszúrás és a törlés nemcsak a tároló két végén lehet gyors, hiszen nem kell a műveletek után a következő elemeket eggyel odébb csúsztatni a memóriában, illetve új folytonos területet foglalni. Ezért a lista esetében nem különböztetünk meg kapacitást, illetve méretet. Mivel a láncolt listára 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. Az elemek eltávolításához a tároló kiváló tagmetódusokat biztosít: a **remove()** a listából minden olyan elemet eltávolít, amely az elem **==** operátora megegyezik **remove()**-nak átadott értékkel. A **remove\_if()** metódus **==** operátor helyett függvényobjektumot vár paraméterül. A **list** 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 már rendezett listával oly módon, hogy az elemek az összefűzés után is rendezettek maradnak. Itt is használható az összehasonlító művelet.

– **deque (double ended queue, kétvégű sor)**: A **vector**tól 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. A sor eleji beszúrást és törlést 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, 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.

Röviden összegezve a szekvenciális tárolókat, elmondható, hogy a **vector** akkor érdemes használni, ha dinamikus tömbre van szükségünk, 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). Továbbá felmerülhet olyan igény, hogy nem csak a tárolók végén szeretnénk műveletet végezni. Ekkor a leggyorsabb megoldást a kétszeresen láncolt lista jelenti. Ha ezeket az egyszerű szabályokat figyelembe vesszük, akkor biztos, hogy mindig a leghatékonyabb megoldást fogjuk használni.

#### ASSZOCIATÍV TÁROLÓK (SET, MULTISSET, MAP, MULTIMAP)

Az úgynevezett asszociatív tárolók index-kulcsok segítségével rendkívül gyors elemhozzáférést tesznek lehetővé. Az **STL** négy ilyen tárolót biztosít számunkra: **set**, **multiset**, **map**, **multimap**. A **set** és a **map** tárolóknál egy kulcshoz csak egy érték tartozhat. Azonban a „multi” előtagot viselő tárolók lehetővé teszik azt, hogy egy kulcshoz, akár több érték is tartozzon. Az asszociatív tárolók kétirányú iterálással dolgoznak.

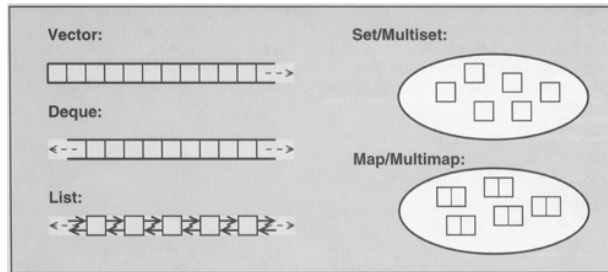
#### HALMAZOK ÉS MULTI-HALMAZOK (SET, MULTISSET)

A halmazok és a multi-halmazok rendezve tárolják az elemeiket. A két tároló különbözősége, hogy amíg a halmaznál egy kulcshoz csak egy érték tartozhat, addig a multi-halmazok megengedik az egy kulcshoz több érték hozzárendelést. A halmaztárolók az elemek sorrendjét saját maguk döntik el, mivel a felhasználónak egészen más elvárásai vannak, mint egy vektorral vagy listával szemben. Egy halmaz esetében például azt szeretnénk tudni, hogy egy adott eleme benne van-e vagy sem, továbbá szeretnénk beszúrni elemeket, viszont, ha egy elemet már tartalmaz a halmaz, akkor ne engedje ugyanakkor az elemnek a beszúrását.

#### ASSZOCIATÍV TÖMBÖK (MAP, MULTIMAP)

Az asszociatív tömböket a **map** és a **multimap** osztályokban implementálták. Az asszociatív tömbök és halmazok különbsége abból adódik, hogy amíg a halmazoknál az érték és a keresés alapjául szolgáló adatstruktúra egy és ugyanaz, addig ez az asszociatív tömböknél különvállik. Így lehetőségünk nyílik arra, hogy megadjuk a kulcsot, amely alapján rendezni szeretnénk, a kulcshoz tartozó adatokat. Fontos megjegyezni, hogy a kulcsot nem változtathatjuk, az adatot viszont igen. A **map** tárolóban nem lehet két egyforma kulcs, amíg a **multimap**ben igen. Az asszociatív tömbök esetében nem lehet „túlindexelni”, hiszen, ha a hivatkozott elem nem létezik, akkor létrehozza. Ez egyben hibalehetőséget is hordoz magában. A **map**ben történő beszúrás az (**insert**) függvénnyel történik, amely paraméterül várja a kulcs értéket és a kulcshoz tartozó értéket. Abban az esetben, ha meg akarunk keresni egy kulcsot a tárolóban, és nem akarjuk, hogy a tároló automatikusan létrehozza, ha nincs benne, akkor a **find()** metódus nyújthat megoldás.

1. ábra. STL-tároló típusok.



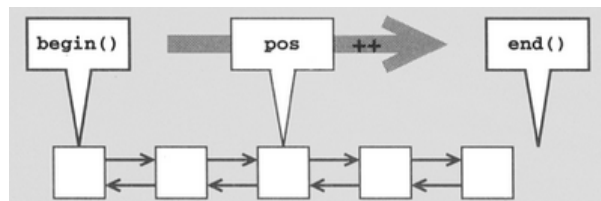
Forrás: <http://www.cplusplus.com/2014/01/STL-deque-example-c.html>

## Iterátorok

A pointerek szerencsés általánosításával, az iterátorokkal hasonlóan kezelhetjük a legkülönfélébb tárolókat. Ugyanakkor az egyes tárolótípusoknál különbségek is adódhatnak. A tárolók által visszaadott iterátorokra különböző műveletek értelmezettek, attól függően, hogy milyen tároló adta vissza őket. A dinamikus tömbnél hozzáadhattunk egy egész számot az iterátorhoz, a láncolt lista ezt érthető okokból nem támogatja. Így az iterátorokat is egyfajta csoportba sorolhatjuk, aszerint, hogy milyen műveletet értelmezünk rajtuk:

- beviteli iterátorok (**input iterators**);
- kimeneti iterátorok (**output iterators**);
- előreléptető iterátorok (**forward iterators**);
- kétirányú iterátorok (**bidirectional iterators**);
- véletlen hozzáférésű iterátorok (**random access iterators**).

2. ábra. Iterátor viselkedése egy kétszeresen láncolt lista esetében.



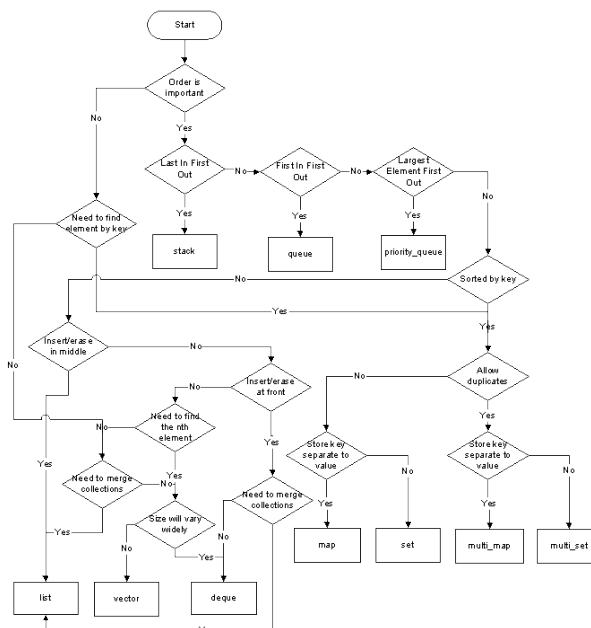
Forrás: [http://www.bogotobogo.com/cplusplus/STL3\\_iterators.php](http://www.bogotobogo.com/cplusplus/STL3_iterators.php)

## Algoritmusok

Az algoritmusok az iterátorok által kijelölt területen végeznek műveleteket. Nagy rugalmasságot tesznek lehetővé, hiszen számos előre megírt algoritmust tartalmaznak, amelyek absztrakt adatokkal képesek dolgozni, tehát mindig alkalmazkodnak a környezethez. Ilyen algoritmus lehet például a vektor legnagyobb vagy legkisebb eleme, egy adott elem keresése stb. További fontos megjegyzés, hogy **Managed C++**-ban az **STL**-tárolók az **std**-névtérben találhatóak. Szükséges még egy **include** direktíva beépítése is, amelynek neve megegyezik a tároló nevével.

### Térkép a legmegfelelőbb STL-tároló kiválasztásához

3. ábra. Útvonal (folyamat ábra) a megfelelő STL-tároló kiválasztásához.



Forrás: <http://stackoverflow.com/questions/471432/in-which-scenario-do-i-use-a-particular-STL-container>

GYAKRAN HASZNÁLT TÁROLÓ TAGFÜGGVÉNYEK ÉS FELELŐSSÉGEIK

A legtöbb tároló rendelkezik tagfüggvényekkel, az alábbi táblázatban a leggyakrabban alkalmazott függvényeket és felelősségeiket olvashatjuk.

Tagfüggvény	Felelősség
<b>push_front</b>	A tároló első elme elé beszúr egy új elemet. (A <b>vector</b> számára nem elérhető).
<b>pop_front</b>	A tároló első elemének törlése (A <b>vector</b> számára nem elérhető).
<b>push_back</b>	A tároló utolsó eleme után egy újabb elemet szúr be.
<b>pop_back</b>	A tároló utolsó elemének eltávolítása.
<b>empty</b>	Logikai igaz-értékkel tér vissza, amennyiben a tároló üres.
<b>size</b>	A tárolóban található elemek darabszámának az összegével tér vissza.
<b>insert</b>	A tárolóba történő új elem beszúrása, egy adott pozícióba.
<b>erase</b>	Egy elem eltávolítása a tárolóból, egy megadott pozícióból.
<b>clear</b>	Az összes elemet eltávolítja a tárolóból.
<b>resize</b>	A tároló átméretezése.
<b>front</b>	Az első elemre mutató referencia-értékkel tér vissza.
<b>back</b>	Az utolsó elemre mutató referencia-értékkel tér vissza.

Egyszerű, tárolókat szemléltető programok az alábbiakban találhatóak. Itt mindegyik tárolóra találhatunk egy-egy példát egy main függvénybe foglalva, Sajnos az STL-tárolók részletekbe menő magyarázata nem fér bele a könyv kereteibe, azonban részletes ismertetésük megtalálható a Benedek Zoltán, Levendovszky Tihamér által írt „*Szoftverfejlesztés C++ nyelven*” könyvében.

#### DINAMIKUS TÖMB (VECTOR) MINTAPÉLDA

```

/*****
 * A vector STL-tároló bemutatása
 * Demonstrációs célok:
 *     -> vector tároló létrehozása
 *     -> vector tároló tagfüggvényei
 *     -> iterátor létrehozása
 *
 * Katona József <katonaj@mail.duf.hu>
 *****/
#include <vector> //a vektor tároló behívása
#include <iostream> //C++ alapvető adatfolyam I/O rutinok
#include <conio.h> //DOS konzol I/O rutinok hívása

int main()
{
    std::vector<double> a;
    std::vector<double>::const_iterator i;

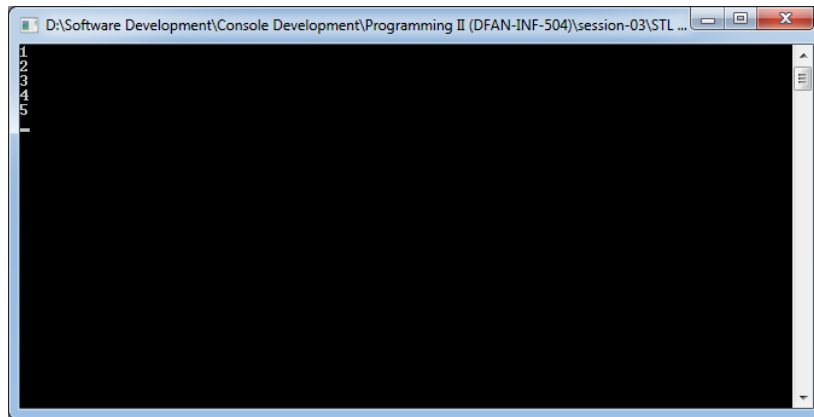
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);
    a.push_back(5);

    for(i=a.begin(); i!=a.end(); ++i)
        std::cout<<(*i)<<std::endl;

    _getch();
    return 0;
}

```

Vizsgáljuk meg a fentebbi példát, amely az **STL** által biztosított **vector** (dinamikus tömb) használatát illusztrálja. A forráskódot elemezve jól kivehető, hogy az **STL** az úgynevezett **std**-névtérben található. A névtér mellett szükséges egy **include** direktíva. Az **include**olni kívánt állomány neve megegyezik a tároló nevével. A továbbiakban egy **double** típusú elemekből álló dinamikus tömböt (**vector**) tárolót foglalmazzunk meg. A tároló mellett egy iterátort is létrehozunk, amely segítségével könnyedén végig tudunk lépkedni (iterálni) a tömb egyes elemein. Fontos megjegyezni, hogy mindenegyres tárolótípusra építhető egy iterátortípus is, tehát az esetünkben ez a típus **vector<double>::iterator**. A dinamikus tömb számos előre megírt tagfüggvényei közül egyet kiemelve (**push\_back**) folyamatos hozzáfűzéssel töltjük fel a **vector** egyes elemeit. A tárolón történő iterálás során, a konzolon megjelenítjük a **vector** egyes elemeit. A dinamikus tömb használata során ne feledkezzünk el arról, hogy a vektor végéhez való hozzáfűzés vagy épp törlés nagyon gyors, azonban, ha az elemeket odébb szeretnénk csúsztatni könnyen lassúvá, válhat, hiszen az sok másolási művelettel jár. Az alábbi ábra mutatja a dinamikus tömb mintapéldájának kimenetét:



```
D:\Software Development\Console Development\Programming II (DFAN-INF-504)\session-03\STL ...  
1  
2  
3  
4  
5  
  
_
```

KÉTSZERESEN LÁNCOLT LISTA (LIST) MINTAPÉLDA

```

/*****
* A list STL-tároló bemutatása
* Demonstrációs célok:
*         -> list tároló létrehozása
*         -> list tároló tagfüggvényei
*         -> iterátor létrehozása
*
* Katona József      <katonaj@mail.duf.hu>
*****/
#include <list>      //a list tároló behívása
#include <iostream>  //C++ alapvető adatfolyam I/O rutinok
#include <conio.h>   //DOS konzol I/O rutinok hívása

using namespace std;

int main()
{
    list<int> lst;
    int i;

    for(i = 0; i < 10; i++)
        lst.push_back(rand());

    cout << "Az eredeti lista: " << endl;

    list<int>::iterator p = lst.begin();
    while(p != lst.end())
    {
        cout << *p << " ";
        p++;
    }

    cout << endl << endl;

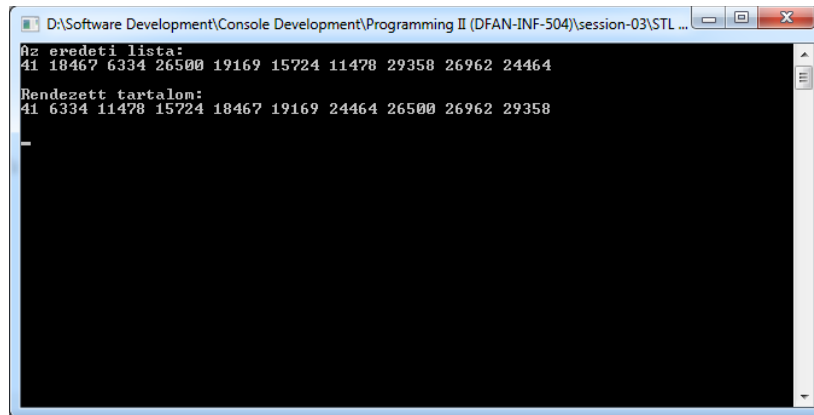
    lst.sort();

    cout << "Rendezett tartalom:\n";
    p = lst.begin();
    while(p != lst.end())
    {
        cout << *p << " ";
        p++;
    }
    cout << endl << endl;

    _getch();
    return (0);
}

```

Vizsgáljuk meg a fentebbi példát, amely az **STL** által biztosított **list** (kétszeresen láncolt lista) használatát mutatja be. A dinamikus tömbhöz hasonlóan itt is szükséges az úgynevezett **std**-névtér behívása. Itt is elengedhetetlen az `include` direktívára, ahol az `includeolni` kívánt állomány neve megegyezik a tároló nevével. A továbbiakban egy `int` típusú elemekből álló kétszeresen láncolt lista (**list**) tárolót fogalmazunk meg. A tároló mellett egy iterátort is létrehozunk, amely segítségével könnyedén végig tudunk lépkedni (iterálni) a lista egyes elemein. Az iterátor mellett egy ciklusváltozót is deklarálunk. A tároló számos tagfüggvényéből egyet felhasználva (**push\_back**) folyamatos hozzáfűzéssel töltjük fel a kétszeresen láncolt lista (**list**) egyes elemeit. A lista elemei véletlen egész számokból épülnek fel. A tárolón történő iterálás során, a konzolon megjelenítjük a kétszeresen láncolt lista (**list**) egyes elemeit. A véletlen értékkel feltöltött lista elemeit a tárolón alkalmazott **sort()** tagfüggvénnyel, egyszerűen rendezzük. A rendezést követően ismét megjelenítjük az immár rendezett lista elemeket. A kétszeresen láncolt lista során ne feledkezzünk el arról, hogy a beszúrás és a törlés nemcsak a lista két végén lehet gyors, ellentétben a dinamikus tömbbel. Az alábbi ábra mutatja a kétszeresen láncolt lista mintapéldájának kimenetét:



```
D:\Software Development\Console Development\Programming II (DFAN-INF-504)\session-03\STL ...
Az eredeti lista:
41 18467 6334 26500 19169 15724 11478 29358 26962 24464
Rendezett tartalom:
41 6334 11478 15724 18467 19169 24464 26500 26962 29358
```

KÉTVÉGŰ SOR (DEQUE) MINTAPÉLDA

```

/*****
* A deque STL-tároló bemutatása
* Demonstrációs célok:
*     -> deque tároló létrehozása
*     -> deque tároló tagfüggvényei
*     -> iterátor létrehozása
*
* Katona József <katona.jozsef.duf@freemail.hu>
*****/
#include <iostream> //C++ alapvető adatfolyam I/O rutinok
#include <conio.h> //DOS konzol I/O rutinok hívása
#include <deque> //deque tároló behívása

int main ()
{
    unsigned int i;

    std::deque<int> first;
    std::deque<int> second (4,100);
    std::deque<int> third (second.begin(),second.end());
    std::deque<int> fourth (third);

    int myints[] = {16,2,77,29};
    std::deque<int> fifth (myints, myints + sizeof(myints) /
sizeof(int) );

    std::cout << "A első deque tartalma: ";
    for (std::deque<int>::iterator it = first.begin();
it!=first.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << std::endl;

    std::cout << "A második deque tartalma: ";
    for (std::deque<int>::iterator it = second.begin();
it!=second.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << std::endl;

    std::cout << "A harmadik deque tartalma: ";
    for (std::deque<int>::iterator it = third.begin();
it!=third.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << std::endl;
}

```

```
std::cout << "A negyedik deque tartalma: ";
for (std::deque<int>::iterator it = fourth.begin();
it!=fourth.end(); ++it)
    std::cout << ' ' << *it;
std::cout << std::endl;

std::cout << "Az ötödik deque tartalma: ";
for (std::deque<int>::iterator it = fifth.begin();
it!=fifth.end(); ++it)
    std::cout << ' ' << *it;

std::cout << '\n';

_getch();
return 0;

}
```

Elemezzük a fentebbi forráskódot, amely az STL által biztosított **deque** (kétvégű sor) használatát mutatja be. Itt is szükséges az úgynevezett **std**-névtér behívása, továbbá nélkülözhetetlen az **include** direktíva, ahol az **include**olni kívánt állomány neve megegyezik maga a tároló nevével. A továbbiakban öt darab **int** típusú elemekből álló kétvégű sor (**deque**) tárolót fogalmazunk meg. A tároló mellett öt darab iterátort is létrehozunk, amelyek segítségével könnyedén végig tudunk lépkedni (iterálni) a sorok egyes elemein. A forráskód szekvenciális elemzése során megfigyelhető, hogy az első, deklarált, integer típusú, üres (értékek nélküli) kétvégű sort, majd négy darab egész típusú 100 kezdőértékkel „feltöltött” **dequet** definiáltunk. A harmadik kétvégű sor egyfajta végig iterálást végez a második **dequen**. A negyedik tárolónál egy másoló konstruktor biztosítja, hogy a harmadik kétvégű sor elemeit átmásoljuk. Az ötödik és egyben utolsó **deque** tárolónál az iterátor konstruktort használjuk arra, hogy tömböket másoljunk bele. A tárolók feltöltése után, az egyes iterátorok segítségével kiíratjuk azok értékeit. A kétvégű sor esetén ne feledkezzünk el arról, hogy a dinamikus tömbhöz képest a sor végén is gyorsan végrehajtnak a beszárási és törlési műveletek. Az alábbi ábra szemlélteti a kétvégű sor mintapéldájának kimenetét:

```

D:\Software Development\Console Development\Programming II (DFAN-INF-504)\session-03\STL ...
A első deque tartalma:
A második deque tartalma: 100 100 100 100
A harmadik deque tartalma: 100 100 100 100
A negyedik deque tartalma: 100 100 100 100
Az ötödik deque tartalma: 16 2 77 29

```

### HALMAZ (SET) MINTAPÉLDA

```

/*****
* A set STL-tároló bemutatása                                     *
* Demonstrációs célok:                                         *
*     -> set tároló létrehozása                                 *
*     -> set tároló tagfüggvényei                             *
*     -> iterátor létrehozása                                  *
*                                                                 *
* Katona József <katonaj@mail.duf.hu>                          *
*****/
#include <set>           //a set tároló behívása
#include <iostream>     //C++ alapvető adatfolyam I/O rutinok
#include <conio.h>      //DOS konzol I/O rutinok hívása

using namespace std;

void IsTrue(int x)
{
    cout << (x ? "True" : "False") << endl;
}

int main()
{
    set<int> s1;
    cout << "s1.insert(5)" << endl;
    s1.insert(5);
    cout << "s1.insert(8)" << endl;
    s1.insert(8);
}

```

```
cout << "s1.insert(12)" << endl;
s1.insert(12);

set<int>::iterator it;
cout << "it=find(8)" << endl;
it=s1.find(8);
cout << "it!=s1.end() returned ";
IsTrue(it!=s1.end());

cout << "it=find(6)" << endl;

it=s1.find(6);
cout << "it!=s1.end() returned ";
IsTrue(it!=s1.end());

_getch();
return 0;
}
```

Tanulmányozzuk a fentebbi forráskódot, amely az STL által biztosított **set** (halmaz) használatát mutatja be. Ennél a tárolótípusnál is szükséges az úgynevezett **std**-névtér behívása, továbbá nélkülözhetetlen az **include** direktívára, ahol az **include**olni kívánt állomány neve megegyezik a tároló nevével. A **main**-függvényt megelőzően létrehozásra került egy **IsTrue** nevezetű függvény, amely a paraméterül kapott egész típusú változótól függően, ternális kifejezés (feltételes operátor) segítségével, jeleníti meg a képernyőn a „True” vagy „False” kifejezést. A **main**-függvényen belül a **set**-tárolón alkalmazott **insert()** tagfüggvényt felhasználva a halmazba különböző értékek kerülnek beszúrásra. A beszúrásokat követően a **set**-objektumra épített iterátor segítségével és a **find()** tagfüggvény alkalmazásával a 8-as érték kerül keresésre. A **find()** metódus által visszaadott iterált értéket átadjuk az **IsTrue** nevezetű függvénynek, amely igaz értéket jelenít meg a képernyőn, mivel a 8-as érték megtalálható a halmazban. Szekvenciálisan tovább haladva a forráskódon, egy újabb keresési kísérlet történik egy adott érték – jelen esetben a 6-os szám – halmazbeli vizsgálatára. Ismét meghívásra kerül a **find()** függvény, majd a metódust felhasználva eldönti, hogy a keresett érték megtalálható-e a halmazban. Mivel 6-os érték nem került elhelyezésre a halmazban, így a függvény által megjelenített szöveg „False”. A halmaz esetén ne feledkezzünk el arról, hogy egy kulcshoz csak egy érték tartozhat, célszerű olyan problémáknál alkalmazni az ilyen jellegű tárolókat, ahol azt vizsgáljuk, hogy egy adott elemet már tartalmaz-e a tároló, mivel egy halmazban egy elem csak egyszer szerepelhet. Az alábbi ábra szemlélteti a halmaz mintapéldájának kimenetét:

```

D:\Software Development\Console Development\Programming II (DFAN-INF-504)\session-03\STL ...
s1.insert(5)
s1.insert(8)
s1.insert(12)
it=find(8)
it!=s1.end() returned True
it=find(6)
it!=s1.end() returned False
    
```

### MULTI-HALMAZ (MULTISET) MINTAPÉLDA

```

/*****
 * A multiset STL-tároló bemutatása
 * Demonstrációs célok:
 *     -> multiset tároló létrehozása
 *     -> multiset tároló tagfüggvényei
 *     -> iterátor létrehozása
 *
 * Katona József <katonaj@mail.duf.hu>
 *****/
#include <iostream>
#include <set>
#include <algorithm>
#include <conio.h>

using namespace std;

int main()
{
    int a[ 10 ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
    int aSize = sizeof(a) / sizeof(int);

    std::multiset< int, std::less< int > > intMultiset(a, a + aSize);

    cout << "Printing out all the values in the multiset : ";
    
```

```

        cout<<"\n\nFrequency of different names"<<endl;
        cout<<"Number of Phones for Joe = "<<phoneNums.count("Joe")<<endl;
        cout<<"Number of Phones for Will =
"<<phoneNums.count("Will")<<endl;
        cout<<"Number of Phones for Smith =
"<<phoneNums.count("Smith")<<endl;
        cout<<"Number of Phones for Zahid =
"<<phoneNums.count("Zahid")<<endl;

        pair<multiset<string,int>::iterator,
multiset<string,int>::iterator> ii;
        multiset<string, int>::iterator it;
        ii = phoneNums.equal_range("Joe");
        cout<<"\n\nPrinting all Joe and then erasing them"<<endl;
        for(it = ii.first; it != ii.second; ++it)
        {
            cout<<"Key = "<<it->first<<"    Value = "<<it->second<<endl;
        }
        phoneNums.erase(ii.first, ii.second);

        _getch();
        return 0;
    }

```

Vizsgáljuk meg a fentebbi példát, amely az STL által biztosított **multiset** (multi-halmaz) használatát mutatja be. A fentebbi tárolókhöz hasonlóan itt is szükséges az úgynevezett **std**-névtér behívása. Továbbra is elengedhetetlen az **include** direktíva alkalmazása, ahol az **include**olni kívánt állomány neve megegyezik a tároló nevével. A forráskód további vizsgálata során láthatjuk, hogy először egy egész típusú vektor vagy más néven tömb került definiálásra, amely 10 darab egész típusú változót képes tárolni. A tömb feltöltése forráskód szinten történt. A tömb létrehozása után egy egész típusú változó is definiálásra kerül, ahol a **sizeof()** metódus segítségével, először a tömb bájtban lefoglalt mérete került lekérdezésre, majd maga az **integer**-típus bájtban foglalt memóriamérete is lehívásra került. Az így kapott két érték elosztásra került egymással, majd ezt a kalkulált értéket kapja meg a változó. A következő lépésben egy multi-halmaz kerül definiálásra, ahol a **less<Key>** utasítás olyan, mintha a kisebb, mint-utasítás alkalmazása történt volna ( $a < b$ ), jelen esetben az értéknek nagyobbnak kell lennie, mint a kulcsértéknek. A **multiset**-tároló megfogalmazása mellett egy iterátor is létrehozásra került, amely segítségével a multi-halmazon végig iterálva kiírjuk a képernyőre annak értékeit. A következő lépésben a **count()** tagfüggvény segítségével megszámlálásra kerül, hogy hányszor szerepel a 15-ös érték a tárolóban. Mivel ilyen érték nem található, a multi-halmazban, ezért 0 érték kerül visszaadásra. Ezt követően az **insert()** függvény segítségével a 15-ös értéket a tárolóba történő beszúrása történik.

A beszúrást követően ismételten megszámlálásra kerül a 15-ös szám érték-előfordulásának gyakorisága. A kapott eredmény ezek után már kettő egység, hiszen a multi-halmaz egyik legfontosabb tulajdonságai közé sorolható, hogy a „hagyományos” halmazzal szemben egy érték többször is előfordulhat a tárolóban. Az alábbi ábra szemlélteti a halmaz mintapéldájának kimenetét:

```

D:\Software Development\Console Development\Programming II (DFAN-INF-504)\session-03\STL ...
Printing out all the values in the multiset : 1 7 9 13 18 22 22 30 85
100
There are currently 0 values of 15 in the multiset
After two inserts, there are currently 2 values of 15 in the multiset
Printing out all the values in the multiset : 1 7 9 13 15 15 18 22 22
30 85 100

```

### MAP (ASSZOCIATÍV TÖMB) MINTAPÉLDA

```

/*****
* A map STL-tároló bemutatása
* Demonstrációs célok:
*     -> map tároló létrehozása
*     -> map tároló tagfüggvényei
*     -> iterátor létrehozása
*
* Katona József <katonaj@mail.duf.hu>
*****/
#include <iostream>
#include <map>
#include <string>
#include <conio.h>
using namespace std;

int main()
{

```

```

map<string, int> freq;
string word;
int i = 0;
while (i<10) {
    cin >> word;
    freq[word]++;
    i++;
}

map<string, int>::const_iterator iter;
for (iter=freq.begin(); iter != freq.end(); ++iter) {
    cout << iter->second << " " << iter->first << endl;
}
getch();
return 0;
}

```

Vizsgáljuk meg a fentebbi példát, amely az **STL** által biztosított **map** (asszociatív tömb) használatát mutatja be. A fentebbi tárolókhoz hasonlóan itt is szükséges az úgynevezett **std**-névtér behívása. Továbbra is elengedhetetlen az **include**-direktíva alkalmazása, ahol az **include**olni kívánt állomány neve megegyezik a tároló nevével. A map megalkotását követően deklarációra kerül egy **string**-típusú változó, amelyben majd a bementben szereplő szavak kerülnek beolvasásra. A beolvasást egy **while** (elől tesztelő) vezérlési szerkezet segítségével került megvalósításra, ahol összesen tíz darab **string**-érték beolvasása történt meg. A beolvasást követően egy iterátor segítségével a map-tárolón egy végigiterálás hajtodik végre és megjelenítésre kerülnek a beolvasott szavak és azok előfordulásának gyakoriságai. Az alábbi ábra szemlélteti az asszociatív tömb mintapéldájának kimenetét:

```

teszt
teszt
első
második
harmadik
negyedik
ötödik
hatodik
hetedik
nyolcadik
1 első
1 harmadik
1 hatodik
1 hetedik
1 második
1 negyedik
1 nyolcadik
1 ötödik
2 teszt

```

MULTIMAP MINTAPÉLDA

```

/*****
* A multimap STL-tároló bemutatása
* Demonstrációs célok:
*         -> set tároló létrehozása
*         -> set tároló tagfüggvényei
*         -> iterátor létrehozása
*
* Katona József    <katonaj@mail.duf.hu>
*****/
#include <iostream>
#include <map>
#include <string>
#include <conio.h>

using namespace std;

int main()
{
    multimap<string, int> phoneNums;

    phoneNums.insert(pair<string, int>("Joe",123));
    phoneNums.insert(pair<string, int>("Will",444));

    phoneNums.insert(pair<string, int>("Joe",369));
    phoneNums.insert(pair<string, int>("Smith",567));
    phoneNums.insert(pair<string, int>("Joe",888));
    phoneNums.insert(pair<string, int>("Will",999));
    cout<<"\n\nFrequency of different names"<<endl;
    cout<<"Number of Phones for Joe = "<<phoneNums.count("Joe")<<endl;
    cout<<"Number of Phones for Will =
"<<phoneNums.count("Will")<<endl;
    cout<<"Number of Phones for Smith =
"<<phoneNums.count("Smith")<<endl;
    cout<<"Number of Phones for Zahid =
"<<phoneNums.count("Zahid")<<endl;

    pair<multimap<string,int>::iterator,
multimap<string,int>::iterator> ii;
    multimap<string, int>::iterator it;
    ii = phoneNums.equal_range("Joe");
    cout<<"\n\nPrinting all Joe and then erasing them"<<endl;
    for(it = ii.first; it != ii.second; ++it)
    {
        cout<<"Key = "<<it->first<<"    Value = "<<it->second<<endl;
    }
}

```

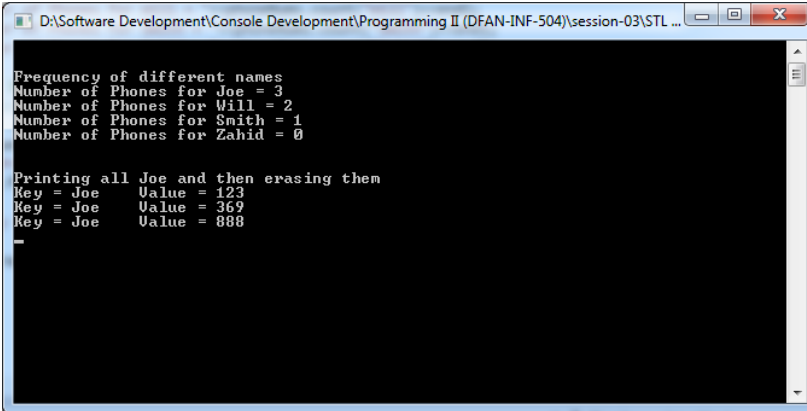
```

    }
    phoneNums.erase(ii.first, ii.second);

    _getch();
    return 0;
}

```

Vizsgáljuk meg a fentebbi példát, amely az **STL** által biztosított **multimap** (asszociatív tömb) használatát mutatja be. A fentebbi tárolókhöz hasonlóan itt is szükséges az úgynevezett **std**-névtér behívása. Továbbra is elengedhetetlen az **include**-direktíva alkalmazása, ahol az **include**olni kívánt állomány neve megegyezik a tároló nevével. A map megalkotását követően deklarálásra kerül egy **string**-típusú változó, amelyben majd a bementben szereplő szavak kerülnek beolvasásra. A beolvasást egy **while** (elől tesztelő) vezérlési szerkezet segítségével került megvalósításra, ahol összesen tíz darab **string**-érték beolvasása történt meg. A forráskód elemzése során látható, hogy először létrehozásra kerül egy **multimap** tároló, amelyben személyek nevei és azok telefonszámjai kerültek tárolásra. A tároló létrehozását követően az **insert()** függvény segítségével két érték beszúrásra került a tárolóba, majd ezek után duplumok kerültek megalkotásra. A beszúrásokat követően a **count()** függvény segítségével megszámlálásra került, hogy bizonyos személyekhez hány darab telefonszám tartozik. Tovább haladva a forráskódon iterátorok segítségével először kiíratásra került az összes Joe-hoz tartozó telefonszám, majd az **erase()** tagfüggvény segítségével törlésre kerültek a tárolóból. Az alábbi ábra szemlélteti az asszociatív tömb mintapéldájának kimenetét:



```

D:\Software Development\Console Development\Programming II (DFAN-INF-504)\session-03\STL ...
Frequency of different names
Number of Phones for Joe = 3
Number of Phones for Will = 2
Number of Phones for Smith = 1
Number of Phones for Zahid = 0

Printing all Joe and then erasing them
Key = Joe   Value = 123
Key = Joe   Value = 369
Key = Joe   Value = 888
-

```

## Gyakorló feladatok

**Valósítson meg egy sort láncolt lista sablonként (template). Az egyes Node-ok rendelkezzenek előre és hátra mutató pointerrel. A sor (FIFO-tároló) egyik végére lehessen berakni, másik végéről kivenni elemeket. Próbálja ki a template-et egy egyszerű (beépített) és egy összetett (class) típusú adattal is!**

**Írjon egy olyan programot, amely filenév-keresőként szolgál. Egy adott (konzolról bekért) mappának és minden almappájának (rekurzívan) összes file-nevét beolvassa, majd a file-nevek között keresni lehet:**

- teljes file-névre keresünk, válaszul visszaadja azokat a mappákat, amiben ilyen nevű file van.
- rész-stringre keresünk: válaszul visszaadja azokat a mappákat, amiben olyan file-nevek vannak, amik tartalmazzák az adott stringet (ennek egyik alosztala a megfelelő kiterjesztésre való keresés)

**A használt adatszerkezetet szabadon megválaszthatja, használhatja a C++ STL osztályait (pl. vector, map stb.) is! (GYF\_CA\_11)**

A Szerencsejáték Rt. honlapjáról ([www.szerencsejatek.hu](http://www.szerencsejatek.hu)) letölthető az eddigi ötöslottó nyerőszámok jegyzéke Excel formátumban (Comma Separated Value, csv-formában mentve ebből szöveges állomány készíthető). Írjon olyan programot, amely beolvassa a nyerőszámok jegyzékét csv-formában és tárolja azt valamilyen adatszerkezetben, majd több funkciót kínál:

- bekért 5 számról megadja, hogy melyik év melyik hetében lett volna ötösünk, négyesünk, hármassunk.
- játék: megadott 5 számmal + megadott időtartammal (pl. ezeket a számokat 1968. 10. hetétől 25. hetéig játszom meg) kiadja az ötöseinket, négyeseinket, hármassainkat (ha vannak).

A használt adatszerkezetet szabadon megválaszthatja, használhatja a C++ STL-osztályait (pl. vector, map stb.) is! (GYF\_CA\_12)

## *Felhasznált irodalom*

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

### *Segédanyagok*

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