

```
#include <forward_list>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;
```

```
void Listakiir(const string& s,
              const forward_list<int>& lista) {
    cout << s << "\t";
    copy(lista.cbegin(), lista.cend(),
          ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

```
int main() {
    forward_list<int> lista = { 1, 2, 3, 4 };
    Listakiir ("A kiinduló lista:", lista);

    // Elem beszúrása a lista elejére
    lista.insert_after(lista.before_begin(), 10);
    lista.push_front(11);
    lista.insert_after(lista.before_begin(),
                      { 21, 22, 23, 3, 11, 12 } );
    Listakiir ("8 új elem az elejére:", lista);

    lista.erase_after(lista.begin());
    lista.pop_front(); // 1. elem
    Listakiir ("Az első 2 elem törölve:", lista);

    lista.sort();
    lista.unique();
    Listakiir ("Rendezet");
}
```

TÓTH BERTALAN

**C++ PROGRAMOZÁS
STL KONTÉNEREKSEL**

Tóth Bertalan:

C++ programozás STL konténerekkel



Jelen dokumentumra a Creative Commons Nevezd meg! – Ne add el! – Ne változtasd meg! 3.0 Unported licenz feltételei érvényesek: a művet a felhasználó másolhatja, többszörözheti, továbbadhatja, amennyiben feltünteti a szerző nevét és a mű címét, de nem módosíthatja, és kereskedelmi forgalomba se hozhatja.

Lektorálta: Juhász Tibor

Tartalom

Tartalom	3
Bevezetés	6
1. Bevezetés az STL-be	7
1.1 Konténerek.....	8
1.1.1 Soros tárolók	8
1.1.2 Asszociatív tárolók.....	10
1.2 Bejárók (iterátorok).....	11
1.3 Függvényobjektumok (funktorok).....	12
1.4 Algoritmusok	13
1.5 Adapterek (adaptors)	14
1.6 Helyfoglalók (allocators)	14
2. Programozás az STL elemeivel.....	15
2.1 Adatok tárolása párokban (pair)	16
2.2 Az iterátorok (bejárók) használata	18
2.2.1 Az iterátorok kategorizálása	18
2.2.1.1 Az input iterátor	19
2.2.1.2 Az output iterátor	19
2.2.1.3 A forward iterátor	19
2.2.1.4 A bidirectional iterátor	20
2.2.1.5 A random-access iterator	20
2.2.2 Iterátor-jellemzők (iterator traits).....	21
2.2.3 Iterátorok a konténerekben tárolt elemekhez.....	21
2.2.4 Az iterátor függvénysablonok használata	22
2.2.5 Iterátor adapterek	22
2.2.5.1 Adatfolyam-iterátor adapterek	22
2.2.5.2 Beszűrő iterátor adapterek	23
2.2.5.3 Fordított (reverse) iterátor adapter	23
2.2.5.4 Áthelyező (move) iterátor adapter	23
2.2.6 A konténerek bejárása	24
2.3 Programozás függvényobjektumokkal	26
2.3.1 Függvénytűzők átadása az algoritmusoknak.....	26
2.3.2 Függvényobjektumok átadása az algoritmusoknak	26
2.3.3 Előre definiált függvényobjektumok	28
2.3.4 Függvényobjektumok előállítás programból.....	29
2.3.5 Hasító függvények készítése	31
2.4 A konténerekről programozói szemmel.....	32
2.4.1 A konténersablonok paraméterezése	32
2.4.2 Típusok a konténer osztályokban.....	33
2.4.3 Konténerek konstruálása és értékadása	35
2.4.3 A konténerek közös tagfüggvényei és operátorai.....	36
2.4.3.1 Iterátorok lekérdezése	36
2.4.3.2 A tárolók elemszáma	37
2.4.3.3 Az elemek közvetlen elérése.....	37
2.4.3.4 A konténerek módosítása	39

2.4.4 Soros konténerek alkalmazása	44
2.4.4.1 array.....	44
2.4.4.2 vector.....	45
2.4.4.3 deque.....	48
2.4.4.4 list	50
2.4.4.5 forward_list	51
2.4.4.6 A soros konténerek összehasonlítása.....	52
2.4.5 Programozás asszociatív konténerekkel.....	53
2.4.5.1 Rendezett halmaz konténerek (set, multiset)	53
2.4.5.2 Rendezett asszociatív tömb (szótár) konténerek	55
2.4.5.3 Rendezetlen halmaz és asszociatív tömb (szótár) konténerek	58
2.4.6 Programozás konténer adapterekkel	60
2.4.6.1 A verem adatstruktúra	60
2.4.6.2 A sor adatstruktúra.....	61
2.4.6.3 A prioritásos adatstruktúra	62
2.5 Ismerkedés az algoritmusokkal	64
2.5.1 Az algoritmusok végrehajtási ideje.....	64
2.5.2 Nem módosító algoritmusok	64
2.5.2.1 Adott művelet elvégzése az elemeken – <i>for_each()</i>	64
2.5.2.2 Elemek vizsgálata.....	66
2.5.2.3 Elemek számlálása – <i>count()</i>	66
2.5.2.4 Elemek keresése – a <i>find()</i> csoport.....	67
2.5.2.5 Azonosság (<i>equal()</i>) és eltérés (<i>mismatch()</i>) vizsgálata.....	69
2.5.2.6 Elemsorozat keresése (<i>search()</i>).....	70
2.5.3 Elemek sorrendjét módosító algoritmusok	71
2.5.3.1 Elemek átalakítása – <i>transform()</i>	71
2.5.3.2 Elemek másolása, áthelyezése	72
2.5.3.3 Az ismétlődő szomszédos elemek törlése a tartományból – <i>unique()</i>	73
2.5.3.4 Elemek eltávolítása a tartományból – <i>remove()</i>	74
2.5.3.5 Elemek lecserélése – <i>replace()</i>	75
2.5.3.6 Az elemek sorrendjének módosítása.....	76
2.5.3.7 Az elemek permutációja	77
2.5.3.8 Az elemek felosztása (particionálása)	78
2.5.3.9 Elemek inicializálása	79
2.5.3.10 A csere algoritmus – <i>swap()</i>	81
2.5.4 Rendezés és keresés	82
2.5.4.1 Rendezési algoritmusok.....	82
2.5.4.2 Bináris keresés rendezett tartományokban	85
2.5.4.3 Rendezett tartományok összefésülése.....	87
2.5.4.4 Halmazműveletek rendezett tartományokkal.....	88
2.5.4.5 Halomműveletek	89
2.5.4.6 Tartományok lexikografikus összehasonlítása	91
2.5.4.7 Legkisebb és legnagyobb kiválasztása	92
2.5.5 Numerikus algoritmusok	93
2.6 Helyfoglalás allokátorokkal.....	96

2.7 A konténerszerű osztályok használata	98
2.7.1 A C++ nyelv hagyományos tömbjei	98
2.7.2 Sztringek	99
2.7.2.1 A string és a vector<char> típusok azonos tagfüggvényei	99
2.7.2.2 A string típus speciális tagfüggvényei	100
2.7.2.3 Összehasonlító külső sztringműveletek	108
2.7.3 Bitkészletek – bitset	108
2.7.3.1 Bitkészletek létrehozása	109
2.7.3.2 Bitkészlet-műveletek	110
2.7.3.2 Konverziós műveletek	112
2.7.4 A vector<bool> specializáció	113
2.7.5 A valarray értéktömb	114
2.7.5.1 Az értéktömb létrehozása és értékadása	114
2.7.5.2 Az indexelés művelete	116
2.7.5.3 További műveletek	118
2.7.5.4 Mátrixműveletek	120
3. Utószó helyett	124

C++ programozás STL konténerekkel

Bevezetés

A *Hewlett-Packard Company* által fejlesztett Szabványos sablonkönyvtár (STL, *Standard Template Library*) a szabványos C++ nyelv könyvtára lett 1998-ban. A következő években ez jelentősen kibővült, és sok új elemmel gazdagodva a C++11/C++14 nyelvekben jelent meg. Ebben az anyagban az Szabványos sablonkönyvtárnak csak azon részeit ismertetjük, amelyek szükségesek a címben megfogalmazott célok eléréséhez. A tárgyalás során feltételezzük, hogy az Olvasó ismeri a C++11 nyelvet. A C++98 nyelven programozók számára egy másik kiadvány („*A C++11 nyelv új lehetőségeinek áttekintése*”) előzetes feldolgozása javasolt, amely a következő címen érhető el:

<http://www.zmgzeg.sulinet.hu/programozas/#ccpp>

A STL számunkra érdekes részei általánosított osztály- és függvénysablonokat tartalmaznak a leggyakrabban használt adatstruktúrák és algoritmusok megvalósításaként. A sablonkönyvtár az általánosított (*generic*) programozást konténerekkel (tárolókkal), iterátorokkal (bejárókkal, általánosított mutatókkal) valamint algoritmusokkal támogatja. A konténerek felhasználásával történő C++ programozáshoz kapcsolódó STL elemeket, valamint az elérésükhöz szükséges deklarációs állományokat az alábbi táblázatban foglaltuk össze:

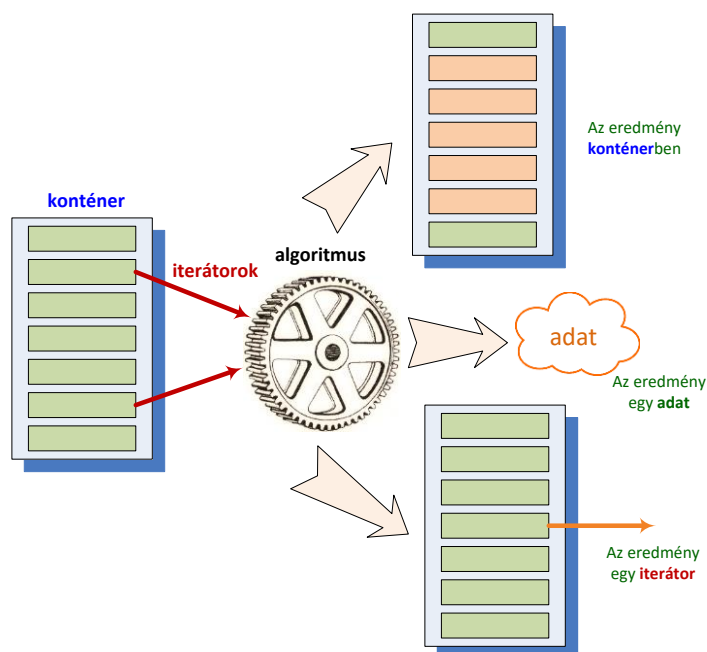
Rövid leírás	Fejállomány
Algoritmusok: rendezés, keresés, másolás stb.	<code><algorithm></code>
Asszociatív tárolók: rendezett halmazok (elemismétlődéssel – <i>multiset</i> , illetve elemismétlődés nélkül – <i>set</i>)	<code><set></code>
Asszociatív tárolók: kulcs/érték adatpárok kulcs szerint rendezett tárolása 1:1 (<i>map</i>), illetve 1:n (<i>multimap</i>) kapcsolatban	<code><map></code>
Asszociatív tárolók: nem rendezett halmazok (elemismétlődéssel – <i>unordered_multiset</i> , illetve elemismétlődés nélkül – <i>unordered_set</i>)	<code><unordered_set></code>
Asszociatív tárolók: kulcs/érték adatpárok nem rendezett tárolása (elemismétlődéssel – <i>unordered_multimap</i> , illetve elemismétlődés nélkül – <i>unordered_map</i>)	<code><unordered_map></code>
Függvényobjektumok (<i>function()</i> , <i>bind()</i>)	<code><functional></code>
Iterátorelemek, előre definiált iterátorok, adatfolyam-iterátorok	<code><iterator></code>
Memóriakezelés (<i>unique_ptr</i> , <i>shared_ptr</i> , <i>allocator</i> stb.)	<code><memory></code>
Műveleti elemek, <i>move()</i> , <i>swap()</i> , a <i>pair</i> (adattér) struktúra	<code><utility></code>
Numerikus műveletek a konténerekben tárolt adatokon	<code><numeric></code>
Soros tároló adapter: verem (<i>stack</i>)	<code><stack></code>
Soros tároló adapterek: sor (<i>queue</i>), prioritásos sor (<i>priority_queue</i>)	<code><queue></code>
Soros tároló: egydimenziós statikus tömb (<i>array</i>)	<code><array></code>
Soros tároló: egyirányú lineáris lista (<i>forward_list</i>)	<code><forward_list></code>
Soros tároló: kétirányú lineáris lista (<i>list</i>)	<code><list></code>
Soros tároló: kettős végű sor (<i>deque</i>)	<code><deque></code>
Soros tároló: egydimenziós dinamikus tömb (<i>vector</i>) és specializált változata <i>vector<bool></i>	<code><vector></code>
Soros tároló szerű: értéktömb (<i>valarray</i>)	<code><valarray></code>
Soros tároló szerű: inicializációs lista (<i>initializer_list</i>)	<code><initializer_list></code>
Soros tároló szerű: karaktersorozat (<i>basic_string</i>) és ennek specializált változatai: <i>basic_string<char></i> - <i>string</i> , <i>basic_string<wchar_t></i> - <i>wstring</i> stb.	<code><string></code>
Soros tároló szerű: rögzített méretű bittömb (<i>bitset</i>)	<code><bitset></code>

1. Bevezetés az STL-be

A C++ nyelv Szabványos Sablonkönyvtára (Standard Template Library - STL) osztály- és függvénysablonokat tartalmaz, amelyekkel elterjedt adatstruktúrákat (vektor, sor, lista, halmaz, szótár stb.) és algoritmusokat (rendezés, keresés, összefésülés stb.) építhetünk be a programunkba.

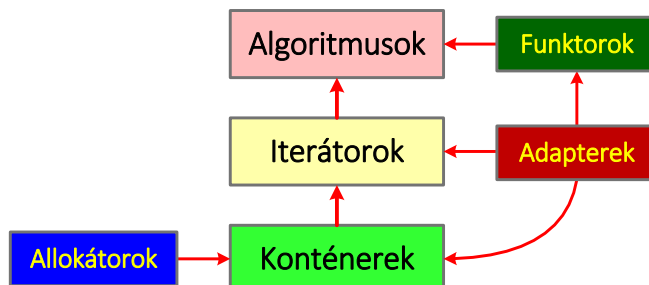
A sablonos megoldás lehetővé teszi, hogy az adott néven szereplő osztályokat és függvényeket (majdnem) minden típushoz felhasználhatjuk, a program igényeinek megfelelően.

Az STL alapvetően három csoportra épül, a konténerekre (tárolókra), az algoritmusokra és az iterátorokra (bejárókra). Egyszerűen megfogalmazva az algoritmusokat a konténerekben tárolt adatokon hajtjuk végre az iterátorok felhasználásával:



A végrehajtott algoritmus működésének eredményét többféleképpen is megkaphatjuk (konténerben, iterátorként vagy valamilyen egyéb adatként).

A konténerek, az iterátorok és az algoritmusok kiegészítéseként az STL-ben további szabványos elemeket is találunk: helyfoglalókat (*allocators*), illesztőket (*adapters*), függvényobjektumokat (*function objects – functors*) stb.



Ebben az anyagban az STL konténerekkel való programozást kívánjuk segíteni, így eltekintünk egy részletekbe menő leírástól. Ennek ellenére a három alappillér mellett röviden ki kell térnünk a további STL elemek bemutatására is, hiszen ezek szorosan kapcsolódnak a konténerekhez és az algoritmusokhoz.

1.1 Konténerek

A konténerek (kollekciók, tárolók) olyan objektumok, amelyekben más, azonos típusú objektumokat (elemeket) tárolhatunk. A tárolás módja alapján a konténereket három csoportba sorolhatjuk. Soros (*sequence*) tárolóról beszélünk, amikor az elemek sorrendjét a tárolás sorrendje határozza meg. Ezzel szemben az adatokat egy kulccsal azonosítva tárolják az asszociációs (*associative*) konténerek, melyeket tovább csoportosíthatjuk a kulcs alapján rendezett, illetve nem rendezett (*unordered*) tárolókra.

A konténerek sokféleképpen különböznek egymástól:

- a memóriahasználat hatékonysága,
- a tárolt elemek elérési ideje,
- új elem beszúrásának, illetve valamely elem törlésének időigénye,
- új elem konténer elejére, illetve végére történő beillesztésének ideje,
- stb.

Általában ezeket a szempontokat kell figyelembe vennünk, amikor a konténerekkel ismerkedünk, vagy amikor valamilyen tárolót kívánunk a programozási feladat megoldásához kiválasztani.

1.1.1 Soros tárolók

A soros tárolók jellemzője, hogy megőrzik az elemek beviteli sorrendjét. Az **array** kivételével tetszőleges pozícióra beszúrhatunk elemet, illetve törölhetünk onnan. Ezek a műveletek általában a tárolók végein a leggyorsabbak. A hagyományos C tömbök helyett az **array** és a **vector** típusok alkalmazása javasolt.

array<>

A sablonparaméterben megadott konstans elemszámmal létrejövő, egydimenziós tömbök osztálysablonja.



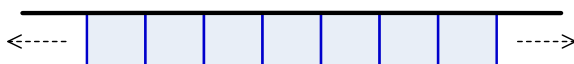
vector<>

A vektor **dinamikus tömbben** tárolódik folytonos memóriaterületen, amely a végén növekedhet. Az elemeket indexelve is elérhetjük konstans $O(1)$ idő alatt. Elem eltávolítása (**pop_back()**), illetve hozzáadása (**push_back()**) a vektor végéhez szintén $O(1)$ időt igényel, míg az elején vagy a közepén ezek a műveletek (**insert()**, **erase()**) $O(n)$ végrehajtású idejűek. Rendezetlen vektorban egy adott elem megkeresésének ideje szintén $O(n)$.



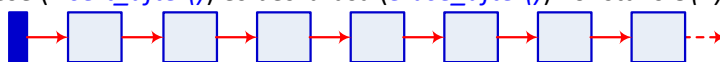
deque<>

Kettősvégű sor megvalósító osztálysablon, amely mindkét végén növelhető, egydimenziós tömböket tartalmazó listában tárolódik. Elemeket mindkét végén konstans $O(1)$ idő alatt adhatunk (**push_front()**, **push_back()**) a kettősvégű sorhoz, illetve távolíthatunk (**pop_front()**, **pop_back()**) el onnan. Az elemek index segítségével is elérhetők.



forward_list<>

Egyszeres láncolású lista, melyet csak az elején lehet bővíteni. Azonos elemtípus esetén az elemek helyigénye kisebb, mint a kettős láncolású listáé. Az elemek törlése (**insert_after()**) és beszúrása (**erase_after()**) konstans $O(1)$ időt igényel.

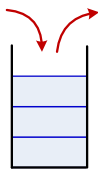


list<>

Kettős láncolású lista, melynek elemei nem érhetők el az indexelés operátorával. tetszőleges pozíció esetén a beszúrás (*insert()*) és a törlés (*erase()*) művelete gyorsan, konstans $O(1)$ idő alatt elvégezhető. A lista minkét végéhez adhatunk elemeket (*push_front()*, *push_back()*), illetve törölhetünk (*pop_front()*, *pop_back()*) onnan.



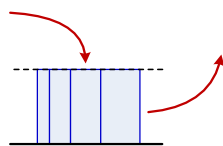
A soros tárolókra épülő, konténerillesztő osztálysablonok a tároló adapterek. Az alábbi konténer adapterek elemein nem lehet végiglépkedni, így semmilyen algoritmus hívásakor sem használhatjuk azokat.

stack<>

A **last in first out** (LIFO – az utoljára betett elemet vesszük ki először) működésű **verem** adatszerkezet típusa. A verem csak a legfelső (*top*) pozícióban lévő elem módosítását (felülírás, behelyezés, kivétel) engedi. Alapértelmezés szerint a **deque** konténerre épül, azonban a **vector** és a **list** is használható a megvalósításához.

queue<>

Sor adatszerkezetet megvalósító típus, amely csak az utolsó (*back*) pozícióra való beszúrást és az első (*front*) pozícióról való eltávolítást teszi lehetővé (**first in first out**, FIFO). Ezeken túlmenően az első és az utolsó elem lekérdezése és módosítása is megengedett. Az alapértelmezett **deque** mellett a **list** soros tárolóra épülve is elkészíthető.

priority_queue<>

A **prioritásos sorban** az elemek a $<$ (kisebb) operátorral hasonlítva, rendezetten tárolódnak. A prioritásos sort csak az egyik, a legnagyobb elemet tartalmazó végén érjük el (*top*). Ez az elem szükség esetén módosítható, vagy kivehető a sorból. Alapértelmezés szerint a **vector** konténer fölött jön létre, azonban a **deque** is alkalmazható.

Az alábbi „konténerszerű” osztályok és osztálysablonok nem tartoznak szorosan a tárolókhoz, azonban a konténerekéhez hasonló tagfüggvényekkel, illetve függvénysablonokkal (*begin()*, *end()*, *swap()*) rendelkezhetnek.

típus[n]**Egydimenziós C tömb**

basic_string<>
string, **wstring**,
u16string,
u32string

A **basic_string** sablon specializációjaként kialakított **karakter sorozat osztályok** a **char**, a **wchar_t**, a **char16_t** illetve a **char32_t** típusú elemek vektorára épülnek. Rendelkeznek a **vector** típusra jellemző tagfüggvényekkel és működéssel, azonban sztringkezelő műveletekkel ki is bővítik azokat.

bitset<>

Rögzített méretű bitkészlet tárolására használható osztálysablon. A bitkészlettel bitenkénti logika és biteltolás műveleteket végezhetünk, valamint számmá és sztringgé alakíthatjuk a tartalmát.

valarray<>

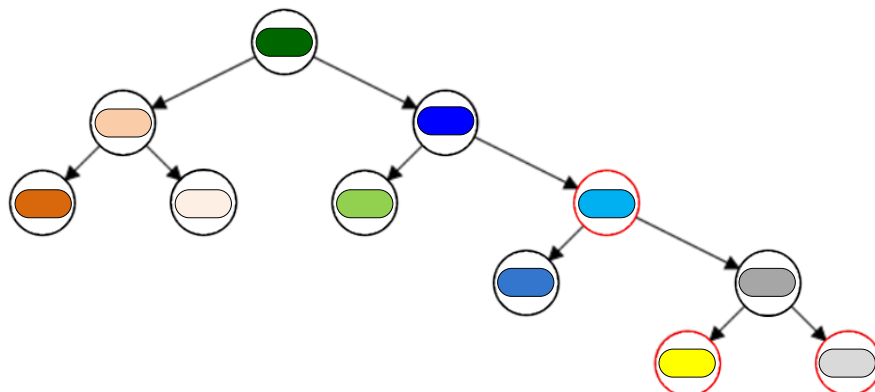
A számokat tároló **értéktömb** összes elemén egyetlen hívással elvégezhetünk bizonyos matematikai műveleteket.

vector<bool>

A **vector** osztálysablon **specializált változata** az elemeket bitek formájában tárolja. Ezzel ugyan helyet takarítunk meg, azonban az iterátorokkal csak korlátozottan férünk hozzá a tárolt adatokhoz.

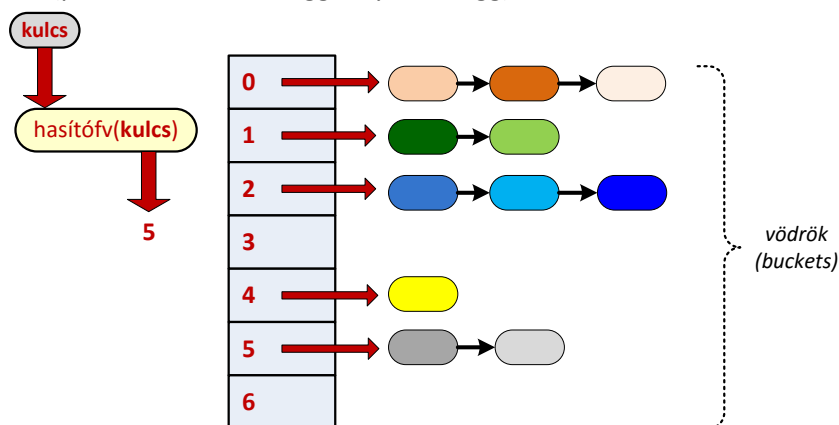
1.1.2 Asszociatív tárolók

Az asszociatív konténerekben az elemekhez való hozzáférés nem az elem pozíciója, hanem egy kulcs értéke alapján megy végbe. A **rendezett asszociatív tárolók** esetén biztosítani kell a rendezéshez használható kisebb (<) műveletet. Az elemek fizikailag egy önkiegyensúlyozó bináris keresőfa (*red-black tree*) adatstruktúrában helyezkednek el.



A rendezett konténerek esetén általában logaritmusos végrehajtási időt ($O(\log(n))$) igényelnek a műveletek, azonban a rendezettségnek köszönhetően hatékony algoritmusokkal dolgozhatunk. Ebbe a csoportba tartoznak az egyedi kulcsokkal működő halmaz (**set**) és a szótár (asszociatív tömb: **map**), valamint ezek kulcsismétlődést megengedő változataik: a **multiset** és a **multimap**. Megjegyezzük, hogy kulcsismétlődés esetén a keresés végrehajtási ideje lineáris ($O(n)$).

Más a helyzet a **rendezetlen (unordered)** asszociatív konténerek esetén. Ebben az esetben az elem gyors elérése érdekében minden elemhez egy hasító érték tárolódik egy *hash*-táblában. Az elemek elérésekor ismét kiszámítódik a hasító érték, és ez alapján majdnem konstans idő alatt lehet elérni az elemeket (persze ez a hasító függvénytől is függ).



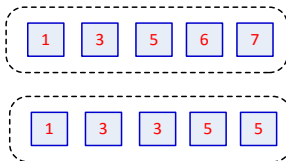
A hasító (kulcsztranszformációs) függvény a kulcsobjektumot egy indexszé (hasító kód) alakítja, amely a **hasító táblában** kijelöl egy elemet (indexeli azt). A *hash*-tábla minden eleme az objektumok egy csoportjára (*bucket* – vödör, kosár) hivatkozik, amennyiben az adott hash-kódhoz tartoznak objektumok. Kereséskor a megfelelő *bucket* objektumait egymás után a kulcshoz hasonlítva találjuk meg a kívánt objektumot, amennyiben az létezik. Jól látható, hogy a hasító tábla működésének hatékonysága nagyban függ a hasító függvénytől. Egy jó *hash*-függvény véletlenszerűen és egyenletesen osztja szét az objektumokat a „vödrökbe”, minimalizálva ezzel a lineáris keresés lépéseit a *bucket*-ekben.

A C++ nyelv alaptípusaihoz az STL biztosítja a megfelelő `hash()` függvényeket (`<functional>`). A fenti négy asszociatív konténer nem rendezett változatai az `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

Míg a halmaz konténerekben a tárolt adat jelenti a kulcsot, addig a szótár tárolókban (kulcs/érték) adatpárokat helyezhetünk el. Az adatpárok típusa a `pair` struktúrasablon, amely lehetővé teszi, hogy egyetlen objektumban két (akár különböző típusú) objektumot tároljunk. A tárolt objektumok közül az elsőre a `first`, míg a másodikra a `second` névvel hivatkozhatunk. (A szótárakban a `first` jelenti a kulcsot.)

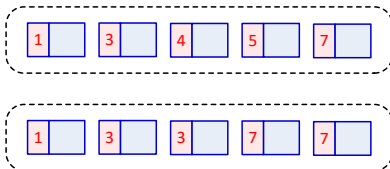
Az alábbi táblázatban röviden bemutatjuk az asszociatív konténereket.

`set<>`, `multiset<>`



Mindkét rendezett halmaz konténer a tárolt adatokat kulcsként használja. A `set`-ben a kulcsok (a tárolt adatok) egyediek kell, legyenek, míg a `multiset`-ben ismétlődhetnek. A két osztálysablon műveletei a `count()` és az `insert()` tagfüggvényektől eltekintve megegyeznek.

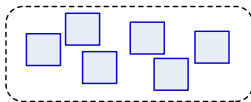
`map<>`, `multimap<>`



Mindkét szótár (asszociatív tömb) konténer elmei `pair` típusúak, és kulcs/érték adatpárokat tartalmaznak. A tárolók elemei a kulcs alapján rendezettek. A kulcsok a `map` esetén egyediek, míg a `multimap` esetén ismétlődhetnek. A halmazokhoz hasonlóan a két osztálysablonnak csak a `count()` és `insert()` tagfüggvényei különböznek egymástól.

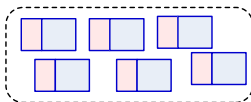
`unordered_set<>`

`unordered_multiset<>`



`unordered_map<>`

`unordered_multimap<>`



Az alábbi összehasonlítás megállja a helyét mind a négy rendezett, illetve nem rendezett asszociatív konténer esetén:

- egy rendezett konténer kevesebb memóriát foglal ugyanannyi tárolt elem esetén,
- kevés elem esetén a keresés gyorsabb lehet a rendezett tárolókban,
- a műveletek többsége gyorsabb a rendezetlen asszociatív konténerekkel,
- a rendezetlen konténerek nem definiálják a lexikografikus összehasonlítás műveleteit: `<`, `<=`, `>` és `>=`.

1.2 Bejárók (iterátorok)

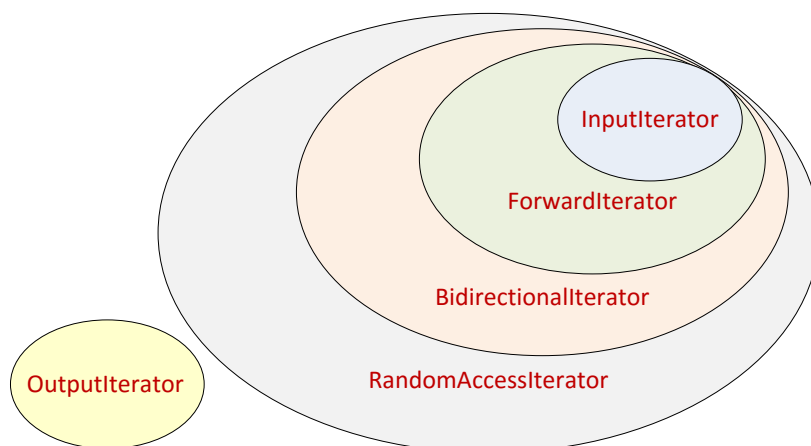
Az iterátorok (*iterators*) elkülönítik egymástól a konténerelemekhez való hozzáférés (és bejárás) módját a konténerek fajtájától. Ezzel lehetővé vált olyan általánosított algoritmusok készítése, amelyek függetlenek a konténerek eltérő elemhozzáférési megoldásaitól (`push_back()`, `insert()` stb.).

Mivel az iterátorosztályokat ellátták a hagyományos mutatók operátoraival, az iterátorokat paraméterként fogadó algoritmus függvénysablonok többsége C tömbökkel is működik.

Mivel a konténerek működése lényeges eltéréseket mutat, egyetlen általános iterátor helyett a szabvány négy egymásra épülő és egy ötödik, különálló iterátort vezetett be. A különböző algoritmusok is más és más kategóriájú iterátor(oka)t várnak paraméterként.

A legegyszerűbb a bemeneti (*input*) iterátor, amely mindössze két alapvető műveletet támogat: az aktuális érték kiolvasása a konténerből (`=*` művelet) valamint léptetés a következő elemre (`++` művelet). Ezeket a műveleteket bővíti ki a konténerbe való írással (`*=` művelet) az előrehaladó (*forward*)

iterátor. Tovább bővítve a műveletek sorát a dekrementálással (--), eljutunk a kétirányú (*bidirectional*) iterátorhoz. A sort az adott számú lépéssel előre és visszafelé is lépni tudó, valamint az indexelést (`[]` művelet) is támogató, tetszőleges elérés (*random-access*) iterátora zárja. (Ez utóbbi bejáró minden olyan műveletet támogat, ami a közönséges mutatókkal is elvégezhető.)



A különálló ötödik kategóriába tartozó kimenteti (*output*) iterátor segítségével írhatunk a konténer aktuális elemébe, illetve a következő elemre léphetünk. Az output iterátor előírásainak az előrehaladó, a kétirányú és a tetszőleges elérésű bejárók is eleget tesznek.

1.3 Függvényobjektumok (funktorkok)

Az STL sok szabványos algoritmust (bejárás, rendezés, keresés stb.) biztosít a programozók számára, amelyek feldolgozzák a konténerekben tárolt adatokat. Az algoritmusok működése függvényobjektumok (*function objects*, *functors*) megadásával testre szabható, hiszen így módon meghatározhatjuk, hogy milyen műveletek hajtódjanak végre a kollekció elemein. A függvényobjektum hagyományos függvénymutató is lehet, azonban az esetek többségében objektumokat alkalmazunk.

A függvényobjektum olyan típus, amely megvalósítja a függvényhívás `()` operátorát. A függvényobjektumoknak két lényeges előnye van a közönséges függvényekhez képest:

- a függvényobjektum megőrizheti a működési állapotát,
- mivel a függvényobjektum egy típus, megadhatjuk sablon paraméterként.

Az STL maga is erősen támaszkodik a függvényobjektumokra, melyek különböző megjelenési formáival találkozhatunk: függvényobjektumok, predikátum függvények, összeállított függvény objektumok, tagfüggvény adapterek stb.

Amikor az egyoperandusú (*unary*) függvényobjektum **bool** típusú értékkel tér vissza, **predikátumnak** nevezzük. **Bináris predikátumról** beszélünk, amikor egy kétoperandusú függvényobjektum ad vissza **bool** típusú értéket, a két paraméter összevetésének eredményeként.

A C++11-től használható lambda-kifejezések valójában névtelen függvényobjektumok.

A függvényobjektumokkal kapcsolatos STL osztálysablonokat a `<functional>` fejláblományban találjuk. Ugyanitt tárolódnak a **hash** függvényobjektum alaptípusokkal és néhány STL típussal (**string**, **bitset**, **vector<bool>** stb.) specializált változatai is.

1.4 Algoritmusok

Az STL algoritmusok igen hasznosak, hiszen felgyorsíthatják és megbízhatóvá tehetik a C++ programok fejlesztését. Napjainkban már közel 100 kipróbált függvénysablon áll a rendelkezésünkre. Az algoritmusok egy részét arra tervezték, hogy módosítsák egy kijelölt adatsor elemeit, azonban soha sem változtatják meg magukat az adatokat tároló konténerek.

Az algoritmusok nem taggfüggvényei a konténereknek, globális függvénysablonok, amelyek iterátorok segítségével férnek hozzá a konténerekben tárolt adatokhoz. Az algoritmusok teljesen függetlenek a konténerektől, a paraméterként megkapott iterátorok feladata a konténerek ismerete.

Az algoritmusok használatához az `<algorithm>` fejlécményt kell a programunkba beépíteni. A numerikus algoritmusok esetén a `<numeric>` deklarációs fájlra van szükségünk.

Az algoritmusok közötti eligazodásban segít, ha a különböző műveleteket a viselkedésük és a működésük alapján csoportokba soroljuk. Egy lehetséges kategorizálás – ahol egy algoritmus több csoportban is megjelenhet – az alábbiakban látható.

Nem módosító algoritmusok

Ezek az algoritmusok nem változtatják meg sem az adatelemeket, sem pedig azok tárolási sorrendjét.

<code>adjacent_find()</code>	<code>find_first_of()</code>	<code>max_element()</code>
<code>all_of()</code> , <code>any_of()</code> , <code>none_of()</code>	<code>for_each()</code>	<code>min_element()</code>
<code>count()</code> , <code>count_if()</code>	<code>lexicographical_compare()</code>	<code>minmax_element()</code>
<code>equal()</code>	<code>max()</code>	<code>mismatch()</code>
<code>find()</code> , <code>find_if()</code> , <code>find_if_not()</code>	<code>min()</code>	<code>search()</code>
<code>find_end()</code>	<code>minmax()</code>	<code>search_n()</code>

Módosító algoritmusok

Az adatmódosító algoritmusokat arra tervezték, hogy megváltoztassák a konténerben tárolt adatelemek értékét. Ez megtörténhet közvetlenül, magában a konténerben, vagy pedig az elemek más konténerbe való másolásával. Néhány algoritmus csupán az elemek sorrendjét módosítja, és ezért került ide.

<code>copy()</code> , <code>copy_if()</code>	<code>generate()</code>	<code>replace()</code> , <code>replace_if()</code>
<code>copy_backward()</code>	<code>generate_n()</code>	<code>replace_copy()</code> ,
<code>copy_n()</code>	<code>iter_swap()</code>	<code>replace_copy_if()</code>
<code>fill()</code>	<code>merge()</code>	<code>swap()</code>
<code>fill_n()</code>	<code>move()</code>	<code>swap_ranges()</code>
<code>for_each()</code>	<code>move_backward()</code>	<code>transform()</code>

Eltávolító algoritmusok

Ezek valójában módosító algoritmusok, azonban céljuk az elemek eltávolítása egy konténerből, vagy másolása egy másik tárolóba.

<code>remove()</code> , <code>remove_if()</code>	<code>remove_copy_if()</code>	<code>unique_copy()</code>
<code>remove_copy()</code> ,	<code>unique()</code>	

Átalakító algoritmusok

Ezek is módosító algoritmusok, azonban kimondottan az elemsorrend megváltoztatásával jár a működésük.

<code>is_partitioned()</code>	<code>partition_point()</code>	<code>rotate()</code>
<code>is_permutation()</code>	<code>prev_permutation()</code>	<code>rotate_copy()</code>
<code>next_permutation()</code>	<code>random_shuffle()</code> , <code>shuffle()</code>	<code>stable_partition()</code>
<code>partition()</code>	<code>reverse()</code>	
<code>partition_copy()</code>	<code>reverse_copy()</code>	

Rendező algoritmusok

Az itt található módosító algoritmusok feladata a teljes konténerben, vagy a tároló egy tartományában található elemek rendezése.

<code>is_heap()</code>	<code>nth_element()</code>	<code>push_heap()</code>
<code>is_heap_until()</code>	<code>partial_sort()</code>	<code>sort()</code>
<code>is_partitioned()</code>	<code>partial_sort_copy()</code>	<code>sort_heap()</code>
<code>is_sorted()</code>	<code>partition()</code>	<code>stable_partition()</code>
<code>is_sorted_until()</code>	<code>partition_copy()</code>	<code>stable_sort()</code>
<code>make_heap()</code>	<code>pop_heap()</code>	

Rendezett tartomány algoritmusok

Ezek az algoritmusok az elemek rendezettségét kihasználva igen hatékonyan működnek.

<code>binary_search()</code>	<code>lower_bound()</code>	<code>set_symmetric_difference()</code>
<code>equal_range()</code>	<code>merge()</code>	<code>set_union()</code>
<code>includes()</code>	<code>set_difference()</code>	<code>upper_bound()</code>
<code>inplace_merge()</code>	<code>set_intersection()</code>	

Numerikus algoritmusok

Számokat tároló konténerek elemein műveletek végző algoritmusok csoportja.

<code>accumulate()</code>	<code>inner_product()</code>	<code>partial_sum()</code>
<code>adjacent_difference()</code>	<code>iota()</code>	

Néhány tároló rendelkezik az algoritmusok némelyikével megegyező nevű tagfüggvénnyel. Ezek létezésnek oka, hogy kihasználva a konténerek speciális adottságait, hatékonyabb és biztonságosabb tagfüggvény készíthető, mint az általános algoritmus. Egyetemes szabályként megfogalmazható, hogy részesítsük előnyben a tagfüggvényeket a program készítése során.

1.5 Adapterek (adaptors)

A teljesség kedvéért megosztunk néhány gondolatot az adapter sablonokról. Általánosságban egy adapter valamit valamivé alakít. Mint láttuk a konténer adapterek néhány soros konténert olyan interfésszel látnak el, ami által megvalósulnak a verem, a sor és a prioritásos sor adatstruktúrák.

Az iterátoradapterek az algoritmusok és a tároló osztályok között képeznek egy interfészt, ami által egyszerűbbé válik az algoritmus hívásakor megkívánt típusú iterátor és a hivatkozott konténerobjektum összekapcsolása.

A C++98 nyelvben különböző adaptersablonok segítették a függvényobjektumok (funktorkok) kezelését, valamint az objektumtagok funktorként való elérését. A C++11 szabvány a korábbi függvényobjektum adaptereket elavultnak minősítette, és helyettük egyszerűbb, ún. burkoló (*wrapper*) osztálysablonokat vezetett be.

1.6 Helyfoglalók (allocators)

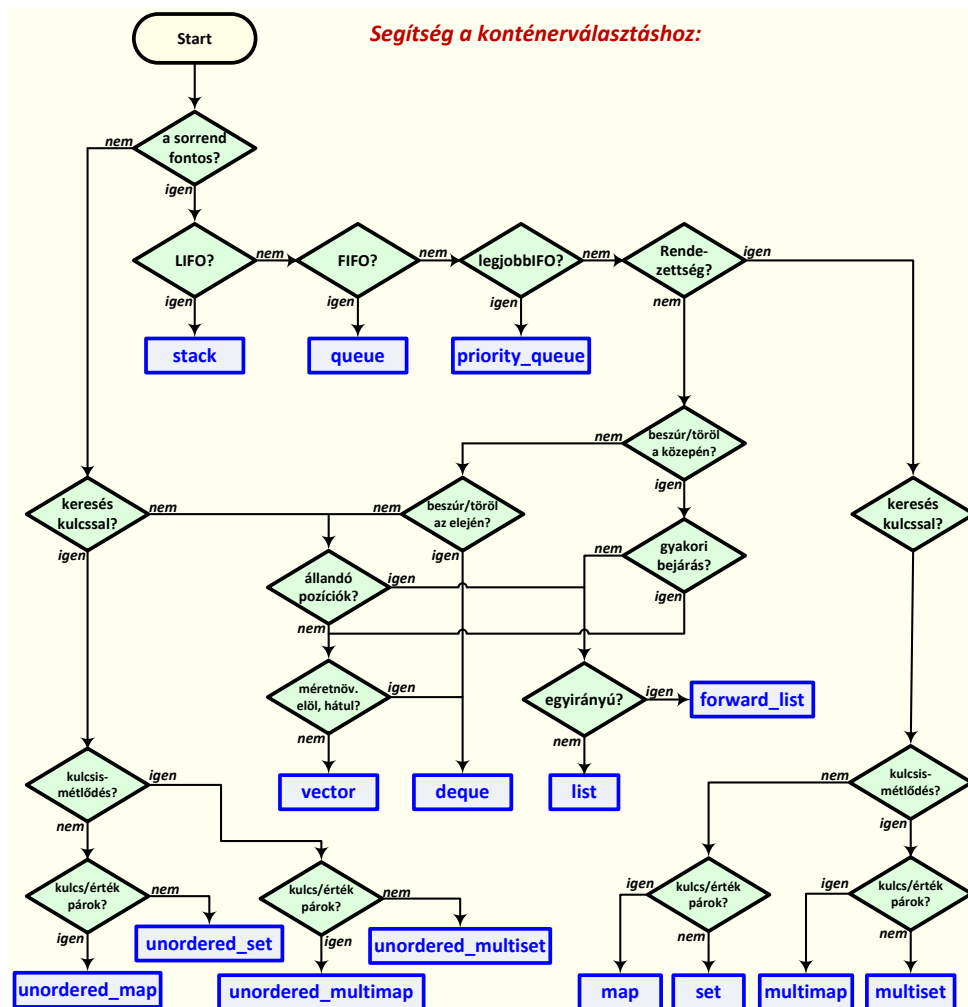
A helyfoglaló a konténerek számára lefoglalja a szükséges memóriaterületeket. Az `array` tároló kivételével minden konténer típus utolsó sablonparamétere az *Allocator*, mely alapértelmezés szerint az `allocator` osztálysablon (`<memory>`) példányát kapja értékül. Csak nagyon különleges esetekben lehet szüksége arra, hogy lecseréljük a tárolók alapértelmezés szerinti helyfoglalóját. A későbbiekben mi is csupán a technika bemutatására szorítkozunk.

2. Programozás az STL elemeivel

Egy szoftver tervezésekor többféle cél alapján optimalizálhatunk: memóriahasználatra, futási sebességre, fejlesztési időre, egyszerűsége, robosztusságra. Sokszor azonban az ismereteink és a programozási gyakorlatunk szabják meg a kiválasztott adatszerkezetet, vagy a felhasznált algoritmust. Például egy buborékrende­zés megírása C++ nyelven senkinek sem jelent problémát, azonban egy gyors­rende­zés hibátlan implementálása már sokaknak fejfájást okozhat. Hasonló a helyzet az adatszerkezetekkel is. Gyakran tesszük le a voksot a rögzített méretű tömbök mellet, mintsem a dinamikus megoldást választanánk, nem is beszélve a különböző listaszerkezetekről.

Egészen másképp áll egy feladathoz a programozó, ha az adatszerkezetek és az algoritmusok készen állnak a rendelkezésére a használt programozási nyelv könyvtárában. Napjainkban minden programozási nyelv gazdag, típusfüggetlen (*generic*, *template*) sablonkönyvtárral segíti a programfejlesztést.

Niklaus Wirth már 1975-ben megadta a programkészítés esszenciáját: „*Algoritmusok + adatszerkezetek = programok*”. Napjainkra az összefüggés valamelyest módosult, azonban a szoftverfejlesztés különböző szintjein felmerülő problémák megoldásánál jó használható ez a megközelítés. Ennek szellemében folytatjuk a Szabványos sablonkönyvtárral való ismerkedést. Számunkra a konténerek képviselik az adatszerkezeteket. A bennük tárolt adatokat általános algoritmusokkal dolgozhatjuk fel, iterátorok közvetítésével. Míg a bevezetésben általános fogalmakat, elveket tisztáztunk, a most következő részekben C++ programokon keresztül mutatjuk be az STL néhány alapvető elemének felhasználását.



2.1 Adatok tárolása párokban (pair)

A **pair** (pár) sablonosztály (`<utility>`) segítségével két objektumot egyetlen objektumként kezelhetünk. A **pair** objektumokban az első tárolt adathoz a **first**, míg a másodikhoz a **second** publikus adat-tagokkal férhetünk hozzá. Ezen tagok típusa mindig megegyezik a sablonosztály példányosítása során megadott típusokkal.

A **pair** sablonosztály a C++11 nyelvnek megfelelően rendelkezik alapértelmezett konstruktorral, amikor a tárolt objektumok az alapértékükkel inicializálódnak. Ugyancsak használhatjuk a paraméteres, a másoló és a mozgató (*move*) konstruktort.

```
pair<string, int> par1;
pair<string, int> par2("szin", 12);
pair<string, int> par3(par2);

pair<string, int> par4(move(par3));
string s1 = "magassag";
int m = 123;
pair<string, int> par5(move(s1), move(m));
```

Olvashatóbbá tehetjük a programjainkat, ha az adatpár típusát **typedef** segítségével állítjuk elő:

```
typedef pair<string, int> IntPropPair;
IntPropPair par1;
IntPropPair par2("szin", 12);
IntPropPair par3(par2);
IntPropPair par4(move(par3));
```

A **pair** osztálysanlon **operator=()** (értékadás) és **swap()** (csere) tagfüggvényei szintén fel vannak készítve a *move* szemantika alkalmazására. A konstruktorok mellett a **make_pair()** függvénysablon kényelmesebbé teszi az adatpárok inicializálását:

```
pair<string, int> par1 = pair<string, int>("index", 2);
pair<string, int> par2 = make_pair("index", 2);
```

A **pair** osztállysablon minden példánya a később bemutatásra kerülő konténer típusokhoz hasonlóan, rendelkezik belső (tag)típusokkal (**first_type**, **second_type**) melyekhez a hatókör operátorral (::) férhetünk hozzá.

```
#include <utility>
#include <iostream>
#include <string>
using namespace std;

int main() {
    pair<string, int> par = make_pair("index", 2);
    pair<string, int>::first_type nev = par.first;
    pair<string, int>::second_type adat = par.second;
    cout << nev << adat << endl; // index2
}
```

Az adatpárokból tárolt objektumokat a **get<0>(pár)** és a **get<1>(pár)** hívásokkal is elérhetjük:

```
pair<string, int>::first_type nev = get<0>(par);
pair<string, int>::second_type adat = get<1>(par);
```

Az adatpárokhöz relációs műveletek (**=**, **!=**, **<**, **<=**, **>**, **>=**) is tartoznak, amelyek ún. lexikografikus összehasonlítást végeznek. (Ez azt jelenti, hogy először összehasonlítják a párok **first** elemét, és csak akkor kerül sor a **second** elemek összevetésére, ha a **first** elemek azonosak.)

A **pair** típus kitüntetett szereppel bír az asszociatív tömbök (*maps*) létrehozásakor, hisz az adatpár első elemét (**first**) kulcsként, míg a másodikat (**second**) adatként használják.

A C++11 nyelvben bevezették a tetszőleges adategyüttes kialakítását lehetővé tevő **tuple** osztálysablon (`<tuple>`). Így ha több, akár különböző típusú objektumot kell konténerekben együtt tárolnunk, választhatunk a **pair/tuple** osztálysablonok és a saját osztálytípus kialakítása között. *Például, az alábbi programban komplex számok tárolására használjuk a **pair** osztálysablon:*

```
#include <utility>
#include <iostream>
#include <cmath>
using namespace std;

typedef pair<double, double> komplex;
#define re first
#define im second

komplex operator+(const komplex& a, const komplex& b) {
    komplex c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;
    return c;
}

komplex operator*(const komplex& a, const komplex& b) {
    komplex c;
    c.re = a.re*b.re-a.im*b.im;
    c.im = a.re*b.im + b.re*a.im;
    return c;
}

ostream& operator<< (ostream& out, const komplex& a) {
    out << a.re << ((a.im>0)? '+' : '-') << abs(a.im) << 'i';
    return out;
}

int main() {
    komplex x = komplex(12, 23), y=komplex(11, -11);
    komplex z = x + y;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    cout << "x+y = " << z << endl;
    cout << "x*y = " << x*y << endl;
    cout << "x*(1-1i) = " << x*komplex(1,-1) << endl;
    return 0;
}
```

```
x = 12+23i
y = 11-11i
x+y = 23+12i
x*y = 385+121i
x*(1-1i) = 35+11i
```

2.2 Az iterátorok (bejárók) használata

Az iterátor olyan objektum, amely képes egy adathalmaz bejárására. Az adathalmaz jelentheti valamely STL konténerben tárolt elemeket, vagy azok résztartományát. Az iterátor egy pozíciót határoz meg a tárolóban. Az iterátorok használatához az `<iterator>` fejlánc kell a programunkba beépíteni.

Az iterátorra, mint általánosított mutatóra tekintve, jobban megérthetjük annak működését. Ez alapján az iterátorokhoz rendelt alapvető műveletek működése egyszerűen értelmezhető, amennyiben p és q iterátorkat, n pedig nemnegatív egész számot jelölnek. (Felhívjuk a figyelmet arra, hogy az egyes műveletek alkalmazhatósága **függ az iterátor fajtájától** – lásd következő részt.)

- A $*p$ kifejezés megadja a konténer p által kijelölt pozícióján álló elemet. Amennyiben az elem egy objektum, akkor annak tagjaira a $(*p).tag$ vagy a $p->tag$ formában hivatkozhatunk.
- A $p[n]$ kifejezés megadja a konténer $p+n$ kifejezés által kijelölt pozícióján álló elemet - hatásában megegyezik a $*(p+n)$ kifejezéssel.
- A $p++$ illetve a $p--$ kifejezések hatására a p iterátor az aktuális pozíciót követő, illetve megelőző elemre lép. (A műveletek prefixus alakja is alkalmazható).
- A $p==q$ és a $p!=q$ kifejezések segítségével ellenőrizhetjük, hogy a p és q iterátorok a tárolón belül ugyanarra az elemre hivatkoznak-e vagy sem.
- A $p<q$, $p<=q$, $p>q$, $p>=q$ kifejezések segítségével ellenőrizhetjük, hogy a tárolón belül a p által kijelölt elem megelőzi-e a q által mutatott elemet, illetve fordítva.
- A $p+n$, $p-n$ kifejezésekkel a p által kijelölt elemhez képest n pozícióval távolabb álló elemre hivatkozhatunk előre (+), illetve visszafelé (-).
- A $p+=n$, $p-=n$ kifejezések kiértékelése után a p az eredeti pozíciójához képest n pozícióval távolabb álló elemre hivatkozik előre (+=), illetve visszafelé (-=).
- A $q-p$ kifejezés megadja a q és p iterátorok által kijelölt elemek pozíciókban mért távolságát egymástól.

2.2.1 Az iterátorok kategorizálása

A bejárókat öt alapvető kategóriába sorolhatjuk.

Iterátor-kategória	Leírás	Műveletek (p az iterátor)	Alkalmazási terület
Kimeneti (<i>output</i>)	írás a konténerbe, előre haladva	$*p, ++$	ostream
Bemeneti (<i>input</i>)	olvasás a konténerből, előre haladva	$=*p, -, ++, ==, !=$	istream
Előrehaladó (<i>forward</i>)	írás és olvasás, előre haladva	$*p, =*p, -, ++, ==, !=$	forward_list, unordered_set, unordered_multiset, unordered_map, unordered_multimap
Kétirányú (<i>bidirectional</i>)	írás és olvasás, előre vagy visszafelé haladva	$*p, =*p, -, ++, --, ==, !=$	list, set, multiset, map, multimap
Tetszőleges elérésű (<i>random-access</i>)	írás és olvasás, előre vagy visszafelé haladva, illetve indexelve is.	$*p, =*p, -, ++, --, ==, !=, [], +, -, +=, -=, <, >, <=, >=$	vector, deque, array

Fontos megjegyeznünk, hogy a *forward* iterátortól kezdve minden kategória helyettesítheti a megelőző kategóriákat. (A kategóriához tartozó új műveleteket bordó színnel emeltük ki.)

2.2.1.1 Az input iterátor

A legegyszerűbb iterátor, amely csak a konténerek olvasására használható. A bemeneti iterátorral csak az *istream_iterátor* osztálysablon tér vissza. Az input iterátor tetszőleges más iterátorral helyettesíthető, kivéve a kimeneti *iterátort*. Az alábbi példában 3, szóközzel tagolt egész számot olvasunk be:

```
#include <iostream>
#include <iterator>
using namespace std;

int main () {
    double adatok[3] = {0};
    cout << "Kerek 3 számot: ";

    istream_iterator<double> pOlvaso(cin);
    for (int i = 0; i < 3; i++) {
        adatok[i] = *pOlvaso;
        if(i<2) pOlvaso++;
    }

    for (int elem : adatok)
        cout << elem << "\t";
    cout << endl;
    return 0;
}
```

2.2.1.2 Az output iterátor

Az output iterátorral mindenütt találkozhatunk, ahol valamilyen adatfeldolgozás folyik az STL eszközeivel, például a másolás (*copy*), vagy az összefésülés (*merge*) algoritmusok. Kimeneti iterátort az output adatfolyam-iterátor adapter (*ostream_iterator*) és a beszűrő iterátor adapterek (*inserter*, *front_inserter* és *back_inserter*) szolgáltatnak. A kimeneti adatfolyam iterátorra való másolás az adatok kirását jelenti:

```
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

int main(void) {
    int adatok[] = {1, 2, 3, 12, 23, 34};
    copy(begin(adatok), end(adatok), ostream_iterator<int>(cout, "\t"));
    cout << endl;
    return 0;
}
```

2.2.1.3 A forward iterátor

Amennyiben egyesítjük a bemeneti és a kimeneti iterátorokat, megkapjuk az előrehaladó iterátort, amellyel a konténerben tárolt adatokon csak előre irányban lépkedhetünk. Az előrehaladó iterátor műveleteivel minden további nélkül készíthetünk elemeket új értékkel helyettesítő függvénysablont:

```
#include <iostream>
#include <iterator>
using namespace std;

template <class FwdIter, class Tipus>
void Cserel(FwdIter elso, FwdIter utolso, const Tipus& regi, const Tipus& uj) {
    while (elso != utolso) {
        if (*elso == regi)
            *elso = uj;
        ++elso;
    }
}
```

```
int main(void) {
    int adatok[] = {1, 2, 3, 12, 23, 34};
    Cserel(begin(adatok), end(adatok), 2, 22);
    Cserel(begin(adatok), end(adatok), 1, 111);
    copy(begin(adatok), end(adatok), ostream_iterator<int>(cout, "\\t"));
    cout << endl;
    return 0;
}
```

2.2.1.4 A bidirectional iterátor

A kétirányú iterátorral a konténerben tárolt adatokon előre és visszafelé is lépkedhetünk, köszönhetően a *decrement* (`--`) operátornak. Több algoritmus is kétirányú iterátorokat vár paraméterként, mint például az adatok sorrendjét megfordító [reverse\(\)](#) algoritmus.

```
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

int main(void) {
    int adatok[] = {1, 2, 3, 12, 23, 34};
    reverse(begin(adatok), end(adatok));
    copy(begin(adatok), end(adatok), ostream_iterator<int>(cout, "\\t"));
    cout << endl;
    return 0;
}
```

2.2.1.5 A random-access iterator

A tetszőleges elérésű iterátorok lehetőségei teljes egészében megegyeznek a normál mutatókéval. A [vector](#) és a [deque](#) tárolókon túlmenően a C tömbök esetén is ilyen iterátorokat használhatunk. Az *alábbi példaprogramban egy függvénysablont készítettünk a tetszőleges elérésű iterátorokkal kijelölt tartomány elemeinek véletlenszerű átrendezésére. (Az elempárok cseréjét az [iter_swap\(\)](#) algoritmus hívásával végezzük.)*

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <cstdlib>
#include <ctime>
using namespace std;

template <class RandIter>
void Kever(RandIter elso, RandIter utolso) {
    while (elso < utolso) {
        iter_swap(elso, elso + rand() % (utolso - elso));
        ++elso;
    }
}

int main(void) {
    int adatok[] = {1, 2, 3, 12, 23, 34};
    srand(unsigned(time(nullptr)));
    Kever(begin(adatok), end(adatok));
    copy(begin(adatok), end(adatok), ostream_iterator<int>(cout, "\\t"));
    cout << endl;
    return 0;
}
```

2.2.2 Iterátor-jellemzők (iterator traits)

Az `iterator_traits` osztálysablon egységes felületet biztosít a különböző típusú iterátorokhoz. Az osztálysablonban az `iterator` osztálynak megfelelő típusdefiniciókat találunk:

Típus	Leírás
<code>distance_type</code>	az iterátorok távolságát leíró típus,
<code>value_type</code>	az iterátor által elérhető adat típusa – ez <code>void</code> kimeneti bejárók esetén,
<code>pointer</code>	a bejárt (<code>value_type</code> típusú) adatokhoz tartozó mutatótípus,
<code>reference</code>	a bejárt (<code>value_type</code> típusú) adatokhoz tartozó referenciátípus,
<code>iterator_category</code>	az iterátor kategóriája, egyike az <code>input_iterator_tag</code> , az <code>output_iterator_tag</code> , a <code>forward_iterator_tag</code> , a <code>bidirectional_iterator_tag</code> , és a <code>random_access_iterator_tag</code> típusoknak.

A C++ nyelv mutatóihoz az `iterator_traits` osztálysablon specializált változatát használhatjuk. Az iterátor-jellemzőkre olyan sablonokban van szükség, amelyek iterátort várnak paraméterként. *Jól szemlélteti ezt az alábbi példaprogram, amelyben definiált függvénysablonnal a megadott tartomány elemeinek sorrendjét megfordítjuk:*

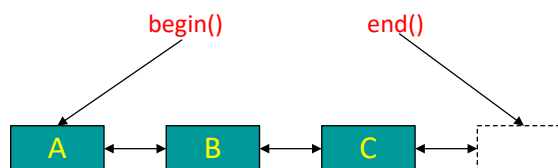
```
#include <iostream>
#include <iterator>
using namespace std;

template<class BiIter>
void Megfordit(BiIter elso, BiIter utolso) {
    typename iterator_traits<BiIter>::difference_type n = distance(elso, utolso);
    n--;
    while(n > 0) {
        typename iterator_traits<BiIter>::value_type adat = *elso;
        *elso++ = *--utolso;
        *utolso = adat;
        n -= 2;
    }
}

int main(void) {
    int adatok[] = {1, 2, 3, 12, 23, 34};
    Megfordit(begin(adatok), end(adatok));
    copy(begin(adatok), end(adatok), ostream_iterator<int>(cout, "\t"));
    cout << endl;
    return 0;
}
```

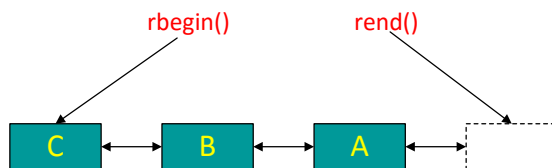
2.2.3 Iterátorok a konténerekben tárolt elemekhez

Normál és konstans iterátorral térnek vissza a konténerek `begin()` és `cbegin()` tagfüggvényei, míg az utolsó elem utáni pozícióra hivatkoznak az `end()` és `cend()` tagfüggvények. A bejáráshoz előre kell léptetnünk (++) a `begin()/cbegin()` tagok hívásával megkapott iterátorkat. (A függvények által visszaadott `iterator` és `const_iterator` típusú bejárók kategóriáját a konténer fajtája határozza meg.)



Az `array`, a `vector`, a `deque`, a `list`, a `set`, a `multiset`, a `map` és a `multimap` konténerek esetén fordított irányú bejárást is lehetővé tesznek az `rbegin()`, `crbegin()`, illetve az `rend()`, `crend()` tagfüggvények által visszaadott iterátorok. Ezek a függvények `reverse_iterator`, illetve `const_reverse_iterator` típusú ér-

téssel térnek vissza. A bejáráshoz ebben az esetben is előre kell léptetnünk (++) az `rbegin()/crbegin()` tagok hívásával megkapott iterátorkat.



A C++11 bevezette a `begin()` és az `end()` függvénysablonokat, amelyekkel olyan tárolókhoz is készíthetünk iterátorokat, amelyek nem rendelkeznek az azonos nevű tagfüggvényekkel, mint például a C tömbök, az inicializációs lista vagy a `valarray` értéktömb. A C++14 ezt a megoldást az összes iterátorfüggvényre kiterjesztette: `cbegin()/cend()`, `rbegin()/rend()`, `crbegin()/crend()`.

2.2.4 Az iterátor függvénysablonok használata

Az alábbi iterátor műveletek esetén azt a legkisebb „tudású” iterátortípust adjuk meg, amellyel a függvényhívás megvalósítható. Természetesen „nagyobb tudású” bejárókkal is minden további nélkül hívhatók ezek a függvények.

Az `advance(p)` függvénysablon az argumentumként megadott **input iterátort** a megadott számú lépéssel előre mozgatja, felülírva az iterátor eredeti értékét.

A `distance(p,q)` függvénysablonnal megtudhatjuk a megadott két **input iterátor** között elhelyezkedő elemek számát.

A `next(p)`, `next(p,n)` függvénysablon a megadott **előrehaladó iterátort** eggyel vagy a megadott számú lépéssel előre, míg a `prev(p)`, `prev(p,n)` a megadott **kétirányú iterátort** eggyel vagy a megadott számú lépéssel visszalépteti. Mindkét függvény visszatérési értéke tartalmazza az új iterátort.

A műveletek használatát az alábbi példaprogrammal szemléltetjük:

```
#include <iostream>
#include <iterator>
using namespace std;

int main(void) {
    int adatok[] = {1, 2, 3, 12, 23, 34};
    auto pAdatok = begin(adatok);
    advance(pAdatok, 3);
    cout << *pAdatok << endl;           // 12
    cout << distance(begin(adatok), end(adatok)) << endl; // 6
    cout << *next(begin(adatok)) << endl; // 2
    cout << *next(begin(adatok), 2) << endl; // 3
    cout << *prev(end(adatok)) << endl;   // 34
    cout << *prev(end(adatok), 2) << endl; // 23
    return 0;
}
```

2.2.5 Iterátor adapterek

Az iterátor adapter osztályok az algoritmusok valamint az adatfolyam és a tároló osztályok között egy interfészt valósítanak meg, melyekkel egyszerűbbé tehető az algoritmus hívásakor megkívánt típusú iterátor és a mögötte álló konténerobjektum összekapcsolása.

2.2.5.1 Adatfolyam-iterátor adapterek

A korábbi példákban már használtuk az `istream_iterator` és az `ostream_iterator` iterátor adaptereket. Ezeken túlmenően a puffert alkalmazó adapterváltozatokat (`istreambuf_iterator`, `ostreambuf_ite-`

tor) is megtaláljuk az STL-ben. Az *alábbi példákban karakterpufferből olvasunk karaktereket, és karakterpufferbe írunk szövegeket.*

```
#include <iostream>
#include <sstream>
#include <iterator>
using namespace std;

int main(void) {
    istringstream s("NAT");
    istreambuf_iterator<char> iter(s);
    char a = *iter++;
    char b = *iter;
    cout << "a=" << a << "\tb=" << b << endl; // a=N    b=A
    return 0;
}

#include <iostream>
#include <sstream>
#include <iterator>
using namespace std;

int main(void) {
    stringstream s;
    ostreambuf_iterator<char> iter(s);
    string m[] = {"Xenia", "Szofia", "Vaszja"};
    for (const string& elem : m) {
        copy(begin(elem), end(elem), iter);
        s << ", ";
    }
    cout << s.str() << endl; // Xenia, Szofia, Vaszja,
    return 0;
}
```

2.2.5.2 Beszúró iterátor adapterek

A beszúró adapterek használatával az algoritmusok felülírás helyett beillesztik az adatokat a konténerbe. A különböző beszúró iterátor adapterek a működésük során más és más konténer-tagfüggvényt hívnak:

<i>az iterátor adapter típusa (osztálysablon)</i>	<i>az iterátor adatt létrehozó függvénysablon</i>	<i>beszúrás az alábbi tagfüggvény hívásával</i>
<i>back_insert_iterator</i>	<i>back_inserter()</i>	<i>push_back()</i>
<i>front_insert_iterator</i>	<i>front_inserter()</i>	<i>push_front()</i>
<i>insert_iterator</i>	<i>inserter()</i>	<i>insert()</i>

2.2.5.3 Fordított (reverse) iterátor adapter

Mint láttuk a *rbegin()/rend()* tagfüggvényekkel állíthatunk elő *reverse_iterator* típusú iterátort. C++14-től fordított iterátort más iterátorból is előállíthatunk a *make_reverse_iterator()* függvénysablon segítségével.

2.2.5.4 Áthelyező (move) iterátor adapter

A C++ 11 nyelv jelentős újítása az ún. move szemantika bevezetése. A korábbi műveleteink az objektumok másolásán alapultak, most azonban választhatunk a másolás és az áthelyezés között. A legáltalább beviteli iterátort a *make_move_iterator()* függvénysablon hívásával áthelyező iterátor-adapterré (*move_iterator*) alakíthatjuk.

Az alábbi példában az elemek valóban áthelyeződtek, azonban a tömbök mérete nem változott meg, lévén azok statikus helyfoglalásúak.

```

#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main(void) {
    int v1[5] = {12, 23, 34, 45, 56};
    int v2[3];
    copy(make_move_iterator(begin(v1)),
         make_move_iterator(end(v1)),
         begin(v2) );
    for (int i=0; i<2; i++)
        cout << v1[i] << "\t"; // 45      56
    cout << endl;
    for (int i=0; i<3; i++)
        cout << v2[i] << "\t"; // 12      23      34
    cout << endl;
    return 0;
}

```

2.2.6 A konténerek bejárása

Az iterátorok és a C++ nyelv elemeinek felhasználásával különböző módon járhatjuk be a konténereinket. Mivel a konténerek részletes tárgyalását a következő fejezet tartalmazza, a példákban továbbra is C tömböket használunk. Ennek ellenére a megoldások a konténerekre is alkalmazhatók a mutatók megfelelő típusú iterátorral való helyettesítését követően.

Az első két esetben ismernünk kell a ciklusváltozóként használt iterátor típusát:

```

int adatok[5] = {12, 23, 34, 45, 56};
int * p = begin(adatok);
while (p!=end(adatok)) {
    cout << *p << endl;
    p++;
}

int adatok[5] = {12, 23, 34, 45, 56};
for (int* p=begin(adatok); p!=end(adatok); p++)
    cout << *p << endl;

```

Az **auto** kulcsszóval az iterátorok típusának felismerését a fordítóra is bízhatjuk:

```

int adatok[5] = {12, 23, 34, 45, 56};
auto p = begin(adatok);
while (p!=end(adatok)) {
    cout << *p << endl;
    p++;
}

int adatok[5] = {12, 23, 34, 45, 56};
for (auto p=begin(adatok); p!=end(adatok); p++)
    cout << *p << endl;

```

Természetesen a *p* iterátor segítségével módosíthatjuk is a tároló tartalmát:

```

int adatok[5] = {12, 23, 34, 45, 56};
auto p = begin(adatok);
while (p!=end(adatok)) {
    *p += 11;
    p++;
}

int adatok[5] = {12, 23, 34, 45, 56};
for (auto p=begin(adatok); p!=end(adatok); p++)
    *p += 11;

```

Az iterátorok kezelését is a fordítóra bízhatjuk a tartományalapú **for** ciklus alkalmazásával. Az elem típusának megválasztásakor dönthetünk arról, hogy kívánjuk-e módosítani a tároló elemeit, mivel ekkor referencia típust kell használnunk.

Az első két példában csak olvassuk az elemek értékét, míg a második kettőben meg is változtatjuk azokat:

```
int adatok[5] = {12, 23, 34, 45, 56};    int adatok[5] = {12, 23, 34, 45, 56};
for (int elem : adatok)                  for (const auto& elem : adatok)
    cout << elem << endl;                cout << elem << endl;

int adatok[5] = {12, 23, 34, 45, 56};    int adatok[5] = {12, 23, 34, 45, 56};
for (int& elem : adatok)                  for (auto& elem : adatok)
    elem += 11;                           elem += 11;
```

Végül használjuk a [*for_each\(\)*](#) algoritmust lambda-függvényekkel!

```
for_each( begin(adatok),                  for_each( begin(adatok),
        end (adatok),                      end (adatok),
        [](int& elem) { elem+=11; } );      [](int elem) { cout<<elem<<endl;} );
```

Bonyolult műveletek sokszori végrehajtásával végzett vizsgálatok kimutatták, hogy a leggyorsabb megoldást a tartományalapú for ciklus adja, melyet a [*for_each\(\)*](#) algoritmus követ kicsivel lemaradva. Ezekhez képest majdnem kétszer annyi időt igényelnek az iterátorokat használó hagyományos ciklusok.

2.3 Programozás függvényobjektumokkal

Az alábbiakban egyszerű példákon keresztül ismerkedünk meg a függvényobjektumok készítésével és használatával. A példákban C tömböket és néhány egyszerűbb STL algoritmust használunk.

2.3.1 Függvénytutatók átadása az algoritmusoknak

A `find_if()` algoritmus két bemeneti iterátorral kijelölt tartományon belül (balról zárt, jobbról nyitott), megkeresi az első olyan elemet, amelyre az egyoperandusú predikátum igaz értékkel tér vissza. Amennyiben nem talál ilyet, a második iterátort adja vissza. *A keresés menetét egy párosságot ellenőrző függvénnyel vezéreljük. Ha a tömb elemei `int` típusúak, az alábbi program a megoldás:*

```
#include <iostream>
#include <algorithm>
using namespace std;

bool Paros(int x) {
    return (x % 2) == 0;
}

int main() {
    int adatok[5] = {12, 23, 34, 45, 56};
    auto p = find_if(begin(adatok), end(adatok), Paros);
    if (p != end(adatok))
        cout << *p << endl;
    else
        cout << "nem talalt" << endl;
}
```

A függvénytutatók alkalmazásával ugyan egyszerű megoldáshoz jutunk, azonban két nagy hiányossággal találjuk szembe magunkat:

- Nem hatékony: a `find_if()` algoritmusban többször meg kell keresni a mutató által hivatkozott függvényt. A fordító nem tudja a kódot optimalizálni, hiszen a mutató tartalma elvileg meg is változtatható.
- Nem rugalmas: nem tudunk olyan predikátum függvényt készíteni, ami egy külső szinten definiált lokális változó értékét használná a függvényen belül.

Mindkét problémára megoldást jelent, ha függvényobjektumot használunk.

2.3.2 Függvényobjektumok átadása az algoritmusoknak

Függvényobjektum használatához egy olyan osztály kell készítenünk, amelyben szerepel a

típus `operator()` (paraméterek) { }

függvény. Az algoritmus hívásakor az osztály alapértelmezett (vagy más) konstruktorral előállított példányát (objektumát) adjuk át.

```
#include <iostream>
#include <algorithm>
using namespace std;

class Paros {
public:
    bool operator() (int x) const {
        return (x % 2) == 0;
    }
};
```

```
int main() {
    int adatok[5] = {12, 23, 34, 45, 56};
    auto p = find_if(begin(adatok), end(adatok), Paros());
    if (p != end(adatok))
        cout << *p << endl;
    else
        cout << "nem talalt" << endl;
}
```

Nagyon hasznos megoldás, ha a függvényobjektumhoz saját belső változót (állapotot) definiálunk, amely a konstruálás során kap értéket. Az alábbi példában megkeressük az adatok tömb első olyan elemét, amely nagyobb, mint 42.

```
#include <iostream>
#include <algorithm>
using namespace std;

class NagyobbMint {
public:
    NagyobbMint(double x = 0) : allapot(x) {}
    bool operator() (double x) const {
        return x > allapot;
    }
private:
    double allapot;
};

int main() {
    double adatok[5] = {12.01, 23.12, 34.23, 45.34, 56.45};
    auto p = find_if(begin(adatok), end(adatok), NagyobbMint(42));
    if (p != end(adatok))
        cout << *p << endl;
    else
        cout << "nem talalt" << endl;
}
```

A bevezető részben már említettük, hogy **lambda**-kifejezésekkel ún. névtelen függvényobjektumokat hozhatunk létre, amelyek még egyszerűbbé teszik a fenti két megoldást:

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int adatok[5] = {12, 23, 34, 45, 56};
    auto p = find_if(begin(adatok), end(adatok),
        [] (int x) -> bool { return (x%2)==0; } );
    if (p != end(adatok))
        cout << *p << endl;
    else
        cout << "nem talalt" << endl;
}

#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    double adatok[5] = {12.01, 23.12, 34.23, 45.34, 56.45};
    double allapot = 42;
    auto p = find_if(begin(adatok), end(adatok),
        [allapot] (double x) -> bool { return x > allapot; } );
    if (p != end(adatok))
        cout << *p << endl;
    else
        cout << "nem talalt" << endl;
}
```

A fentiekhez hasonló módon készíthetünk nem predikátum, kétparaméteres függvényobjektumokat is. A `transform()` algoritmus segítségével számítsuk ki két vektor elemeinek páronkénti szorzatát!

```
#include <iostream>
#include <algorithm>
using namespace std;

struct Szoroz {
public:
    int operator()(int a, int b) {
        return a*b;
    }
};

int main() {
    int xt[5] = { 8, 5, 3, 2, 1};
    int yt[5] = {13, 21, 34, 55, 89};
    int zt[5];
    transform(begin(xt), end(xt), begin(yt), begin(zt), Szoroz());
    for (int e : zt)
        cout << e << "\t"; // 104    105    102    110    89
    cout << endl;
}
```

A megoldás lambda-kifejezés alkalmazásával:

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int xt[5] = { 8, 5, 3, 2, 1};
    int yt[5] = {13, 21, 34, 55, 89};
    int zt[5];
    transform(begin(xt), end(xt), begin(yt), begin(zt),
        [] (int x, int y) { return x*y; } );
    for (int e : zt)
        cout << e << "\t"; // 104    105    102    110    89
    cout << endl;
}
```

2.3.3 Előre definiált függvényobjektumok

A C++ nyelv szabványos könyvtárában (`<functional>`) megtalálható, előre definiált függvényobjektumok lefedik az alapvető műveleteket. Például, a rendezésekhez használt (`<`) kisebb műveletnek a `less` függvényobjektum felel meg. Az alábbi táblázat bal oszlopában a függvényobjektumok neveit láthatjuk, míg jobb oldalon a paramétereket és a műveletet találjuk.

<i>Aritmetikai műveletek</i>	
<i>negate</i>	- param
<i>plus</i>	param1 + param2
<i>minus</i>	param1 - param2
<i>multiplies</i>	param1 * param2
<i>divides</i>	param1 / param2
<i>modulus</i>	param1 % param2
<i>Összehasonlítások</i>	
<i>equal_to</i>	param1 == param2
<i>not_equal_to</i>	param1 != param2
<i>less</i>	param1 < param2
<i>greater</i>	param1 > param2
<i>less_equal</i>	param1 <= param2
<i>greater_equal</i>	param1 >= param2

<i>Logikai műveletek (rövidzár kiértékelés nélkül!)</i>	
logical_not	<code>! param</code>
logical_and	<code>param1 && param2</code>
logical_or	<code>param1 param2</code>
<i>Bitenkénti logikai műveletek</i>	
bit_not	<code>~ param</code>
bit_and	<code>param1 & param2</code>
bit_or	<code>param1 param2</code>
bit_xor	<code>param1 ^ param2</code>

Amikor egy tömb adatait rendezzük a `sort()` algoritmussal, növekvő sorrendet kapunk. Ha csökkenő sorrendben szeretnénk látni a rendezés eredményét, a `greater` függvényobjektumot kell megadnunk a `sort()` utolsó argumentumaként:

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main() {
    int adatok[5] = { 35, 1, 3, 12, 23};
    sort(begin(adatok), end(adatok), greater<int>());
    for (int e : adatok)
        cout << e << "\t"; // 35      23      12      3      1
    cout << endl;
}
```

2.3.4 Függvényobjektumok előállítás programból

A C++11 eldobta a „rég” függvényobjektum adaptereket, és helyettük nagyon kényelmes burkolókat (*wrapper*) vezetett be a függvényekhez, a függvényobjektumokhoz és a lambda-kifejezésekhez.

A `function` osztálysablon segítségével mindenféle függvényobjektumra hivatkozhatunk. A `bind()` függvénytáblonnal argumentumokat rendelhetünk a függvényobjektumokhoz. A `mem_fn()` sablonnal pedig osztálytagok eléréséhez készíthetünk burkolókat.

Adapter hívása	Leírás
<code>g = bind(fv, argumentumok)</code>	a <code>g(argumentumok2)</code> hívás megegyezik az <code>fv(argumentumok3)</code> hívással, ahol az <code>argumentumok</code> ban található helyfoglalók (<code>_1</code> , <code>_2</code> , <code>_3</code>) az <code>argumentumok2</code> megfelelő argumentumaival helyettesítődnek,
<code>g = mem_fn(fv)</code>	a <code>g(p, argumentumok)</code> hívás jelentése <code>p->fv(argumentumok)</code> , ha <code>p</code> mutató, és <code>p.tagfv(argumentumok)</code> , ha nem az,
<code>g = not1(fv)</code>	a <code>g(x)</code> hívás jelentése <code>!fv(x)</code> ,
<code>g = not1(fv)</code>	a <code>g(x, y)</code> hívás jelentése <code>!fv(x, y)</code> ,
<code>r = ref(adat)</code>	a burkoló referenciát készít az <code>adathoz</code> ,
<code>r = cref(adat)</code>	a burkoló konstans referenciát készít az <code>adathoz</code> .

Az alábbi példában egyszerű függvények felhasználásával csokorba gyűjtöttük a fenti sablonok alkalmazásának fogásait:

```
#include <functional>
#include <iostream>
using namespace std;
using namespace std::placeholders;
```

```

int szoroz(int a, int b) {
    return a * b;
}

struct Szoroz {
    int operator()(int a, int b) const {
        return a*b;
    }
    int NegyzetOsszeg(int a, int b) const {
        return a*a + b*b;
    }
    int adat = 123;
};

int main()
{
    // normál függvény
    function<int(int,int)> fSoroz = szoroz;
    cout << fSoroz(12, 23) << endl;

    // függvényobjektum
    function<int(int,int)> fFunktor = Szoroz();
    cout << fFunktor(12, 23) << endl;

    // lambda-függvény
    function<int(int,int)> fLambda = [](int a, int b) { return a * b; };
    cout << fLambda(12, 23) << endl;

    // szorzó függvényobjektum 5*7
    auto fKot1 = bind(szoroz, 5, 7);
    cout << fKot1() << endl;

    // szorzó függvényobjektum 5*argumentum
    auto fKot2 = bind(szoroz, 5, _1);
    cout << fKot2(12) << endl;

    // tagfüggvényhívás beburkolása
    auto fKot3 = bind( &Sorozat::NegyzetOsszeg, Szoroz(), _1, _2);
    cout << fKot3(12, 23) << endl;

    // Osztálytagok elérése
    Szoroz szor;
    auto fTagFv = mem_fn(&Sorozat::NegyzetOsszeg);
    cout << fTagFv(szor, 12, 23) << endl;

    auto fAdatTag = mem_fn(&Sorozat::adat);
    cout << fAdatTag(szor) << endl;
}

```

Végül oldjuk meg az alfejezet induló feladatát kizárólag sablonok felhasználásával! A `bind()` hívásokat egymásba ágyazva összetettebb feltételeket is megfogalmazhatunk, bár a program olvashatósága kívánalmakat hagy maga után:

```

#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
using namespace std::placeholders;

int main() {
    int adatok[5] = {12, 23, 34, 45, 56};
    auto p = find_if(begin(adatok), end(adatok),
        bind(equal_to<int>(), bind(modulus<int>(), _1, 2), 0)
    );
    if (p != end(adatok))
        cout << *p << endl;
    else
        cout << "nem talalt" << endl;
}

```


2.3.5 Hasító függvények készítése

A C++ nyelv alaptípusaihoz a **hash** függvényobjektum specializált változatait a *<functional>* fejléc tartalmazza. Felhasználói típusokhoz (**struct**, **class**) a megfelelő hasító függvényt magunknak kell elkészíteni, felhasználva az alaptípusok **hash** függvényeit. Az alaptípusú adattagok hasító értékei között egyaránt végezhetünk bitenkénti és aritmetikai műveleteket.

Az alábbi példában a Pont struktúra minden adattagját felhasználtuk a PontHash hasító függvényobjektum elkészítéséhez, míg a PHash() függvényt csak a numerikus tagokra alapozva hoztuk létre.

```
#include <iostream>
#include <functional>
#include <string>
using namespace std;

struct Pont {
    string jel;
    int x,y;
};

struct PontHash {
    size_t operator()(const Pont& p) const {
        size_t h1 = hash<string>()(p.jel);
        size_t h2 = hash<int>()(p.x);
        size_t h3 = hash<int>()(p.y);
        return (h1 ^ (h2 << 1) ^ (h3<<2)) % 256;
    }
};

size_t PHash(const Pont& p) {
    size_t h1 = hash<int>()(p.x);
    size_t h2 = hash<int>()(p.y);
    return h1 ^ (h2 << 1) ;
}

int main() {
    Pont pontok[] = { {"A",1,2}, {"B",1,3}, {"B",2,3}, {"D",-1,-2}, {"A",2,1} };
    for (const auto& pont : pontok) {
        cout << PontHash()(pont) << "\t" << PHash(pont);
        cout << "\t(" << pont.jel << ", " << pont.x << ", " << pont.y << ")\n" ;
    }
}
```

243	5	(A,1,2)
191	7	(B,1,3)
185	4	(B,2,3)
155	3	(D,-1,-2)
249	0	(A,2,1)

2.4 A konténerekről programozói szemmel

Az STL-ben a konténerek is sablonok formájában vannak jelen. A fentiekben már találkoztunk osztály- és függvénysablonokkal, azonban tárolók esetén több lehetőség áll a rendelkezésünkre.

Miután kiválasztottuk a felhasználni kívánt konténertípust, a megfelelő deklarációs állományt (lásd a bevezetőt) be kell építenünk a programunkba. Ezt követően a konténer osztállysablont megfelelően paraméterezve példányosítjuk azt. Ha többször is használjuk a programban ugyanazt a tároló típust, javasolt a **typedef** alkalmazása. Bármelyik utat is választjuk, a konténerosztállyal – a szokásos módon – konténerobjektumokat kell készítenünk, gondoskodva a megfelelő konstruktor meghívásáról. A programkészítés további részeiben a tároló objektumokkal dolgozunk, hívjuk azok tagfüggvényeit, használjuk a kapcsolódó műveleteket, illetve iterátorok segítségével algoritmusokat hajtunk végre az elemeiken.

A leggyakrabban használt **vector** konténeren mutatjuk be az elmondottakat:

```
#include <iostream>
#include <vector>    // a szükséges deklarációs fájl
using namespace std; // így nem kell az std:: névtér-hivatkozást használnunk

int main() {
    vector<int> vektor(10); // tíz 0 értékű elemmel feltöltött vektor
    for(int i = 0; i < vektor.size(); i++) // tagfüggvény hívása
        cout << vektor[i] << ' ';          // az indexelés operátorának használata
    cout << endl;
    return 0;
}
```

A következőkben először megismerkedünk konténerekre vonatkozó általános tudnivalókkal, majd pedig sorra vesszük az egyes tárolókat.

2.4.1 A konténersablonok paraméterezése

Minden konténer esetén kötelező paraméter a tárolandó adat(ok) típusa (*AdatTípus*, *KulcsTípus*), míg a többi paraméter rendelkezik alapértelmezés szerinti értékkel. Minden igazi (nem adapter) konténersablon utolsó paramétere a helyfoglaló *Allocator*, melynek **allocator** kezdőértékét csak igen ritkán változtatjuk meg, ezért a későbbiekben mindig elfogadjuk az alapértelmezett beállítást. Példaként tekintsük a **vector** sablon definícióját:

```
template<typename AdatTípus,
        typename Allocator = std::allocator<AdatTípus>>
class vector {
    // ...
};
```

A **soros konténerek** mindegyikét hasonlóan kell paraméterezni, kivételt képez az **array**, amelyik egy konstans érték paraméterben várja a méretet, és amelynek nincs helyfoglaló paramétere:

```
array<AdatTípus, Méret>
vector<AdatTípus>
deque<AdatTípus>
list<AdatTípus>
forward_list<AdatTípus>
```

A **rendezett asszociatív** tárolósablonok egy további, *Compare* paraméterének egy függvényobjektumot kell átadnunk, amelyre a rendezettség fenntartása érdekében van szükség. Ez a paraméter is rendelkezik alapértelmezett értékkel az **std::less<KulcsTípus>** (kisebb, <) függvényobjektummal,

amely minden C++ és STL típus esetén megfelel számunkra. Ezt csak akkor kell lecserélnünk, ha más rendezési elvet kívánunk használni, vagy ha saját osztályt adunk meg kulcstípusként.

```
set<KulcsTípus, Compare>
multiset<KulcsTípus, Compare>
```

Az asszociatív tömbök (szótárak) esetén két adattípust is meg kell adnunk a sablon példányosítása-kor, hiszen itt `pair<const KulcsTípus, AdatTípus>` adatpárok tárolódnak a konténerben a kulcs alapján rendezve.

```
map<KulcsTípus, AdatTípus, Compare>
multimap<KulcsTípus, AdatTípus, Compare>
```

A **nem rendezett** (*unordered*) konténersablonokban a *Compare* paraméter helyett egy *Hash* és egy *KeyEqual* (függvényobjektum) paraméter jelenik meg. Mindkettőre a hasító táblák kezelése során van szükség. A *Hash* (hasító) paraméter alapértéke a már megismert `std::hash<KulcsTípus>` funktor, míg a *KeyEqual* (kulcsegyenlőség, `==`) paraméteré az `std::equal_to<KulcsTípus>`. Mivel a *hash* osztálysablonnak a C++ alaptípusok mellett csak néhány STL típushoz létezik specializált változata, gyakrabban kell saját függvényobjektummal helyettesítenünk.

```
unordered_set<KulcsTípus, Hash, KeyEqual>
unordered_multiset<KulcsTípus, Hash, KeyEqual>
unordered_map<KulcsTípus, AdatTípus, Hash, KeyEqual>
unordered_multimap<KulcsTípus, AdatTípus, Hash, KeyEqual>
```

A **konténer adapter** osztálysablonok paraméterei között nem találjuk meg a helyfoglalót, hiszen mindegyikük valamelyik alapértelmezett soros tárolóra (*Container*) épül, amit persze magunk is megadhatunk. A *Container* paraméter alapértéke `std::deque<AdatTípus>` a verem (*stack*) és a sor (*queue*) esetén, míg `std::vector<AdatTípus>` a prioritásos sor (*priority_queue*) esetén. A prioritásos sor *Compare* paraméterének alapértéke a már megismert `std::less<AdatTípus>` függvényobjektum.

```
stack<AdatTípus, Container>
queue<AdatTípus, Container>
priority_queue<AdatTípus, Container, Compare>
```

A tárolt adatok és kulcsok típusaként minden olyan típust megadhatunk, amelyek objektumai konstruktorral vagy értékadással másolhatók (*copy*) és/vagy áthelyezhetők (*move*), illetve felcserélhetők (*swap*). Amennyiben egy típus nem támogatja ezeket a műveleteket, akkor annak mutatóját (`unique_ptr<Típus>` vagy `Típus*`) tárolhatjuk a konténerekben.

2.4.2 Típusok a konténer osztályokban

A konténerosztályok mindegyike egy sor nyilvános elérésű típust definiál, amelyekből megtudhatjuk az osztály kialakítása során felhasznált típusokat. Ezeket a típusokat az osztálynév és a hatókör operátor megadásával érhetjük el, például a tárolóhoz rendelt bejáró típusa:

```
vector<double>::iterator
```

Az alábbi táblázatban összefoglaltuk a **CC**-vel jelölt konténerosztályok (*Container Class*) típusait. A konténerosztály a megfelelő osztálysablon példányosításával jön létre, mint például `vector<int>`.

Megjegyezzük, hogy az `allocator_type` típus hiányzik az `array` konténerekből, míg a fordított iterátor típusokat nem találjuk meg a `forward_list` típusú és a rendezetlen asszociatív tárolókban.

Típus	Leírás
CC::value_type	a konténerben tárolt adatok típusa,
CC::size_type	a konténer méretének típusa (általában <i>size_t</i>),
CC::reference	a tárolt adatok referenciájának típusa,
CC::const_reference	a tárolt adatok konstans referenciájának típusa,
CC::pointer	mutatótípus a tárolt adathoz,
CC::const_pointer	konstans mutatótípus a tárolt adathoz,
CC::iterator	iterátor típus,
CC::const_iterator	a konstans iterátor típusa,
CC::reverse_iterator	a fordított iterátor típusa,
CC::const_reverse_iterator	a konstans fordított iterátor típusa,
CC::difference_type	két CC::iterator típusú érték különbségének típusa (általában <i>ptrdiff_t</i>).
CC::allocator_type	a memóriakezelő típusa.

A fenti táblázatban a *const_* előtaggal ellátott típusok konstans konténerek esetén kötelező alkalmazni, míg normál tárolók esetén az előtag nélküli típusok az alapértelmezettek, azonban szükség esetén a konstans típusokat is használhatjuk.

konténer<típus> *normál*;

const konténer<típus> *konstans*;

Felhívjuk a figyelmet, hogy a konstans konténereken semmilyen módosítás sem hajtható végre, ezért elsősorban függvények bemenő paramétereiként szerepeltetjük:

fvtípus függvény(const konténer<típus>& *paraméter*) { ... }

Az asszociatív konténerek további típusokat is definiálnak:

Típus	Leírás
ASSOC::key_type	a kereső kulcs típusa,
ASSOC::mapped_type	az asszociatív tömbök (<i>maps</i>) párjaiban az adat típusa,
ASSOC::key_compare	a kulcsokat összehasonlító objektum típusa rendezett konténerek esetén,
ASSOC::value_compare	az ASSOC::value_type típusú értékek összehasonlításához használt típus (rendezett halmazok esetén),
ASSOC::hasher	a rendezetlen tárolóhoz rendelt hasító függvény típusa,
ASSOC::key_equal	a rendezetlen tárolóhoz rendelt azonosságot vizsgáló függvény típusa,
ASSOC::local_iterator	iterátor típus a bucketek (vödrök) bejárásához (rendezetlen tárolókban),
ASSOC::const_local_iterator	konstans iterátor típusa a bucketek (vödrök) bejárásához (rendezetlen tárolókban).

A konténer adapterek csupán néhány típussal rendelkeznek:

Típus	Leírás
ADAPT::container_type	annak a tárolónak a típusa, amelyre az adapter épül,
ADAPT::value_type	a konténer adapterben tárolt adatok típusa,
ADAPT::size_type	a konténer adapter méretének típusa (általában <i>size_t</i>),
ADAPT::reference	a tárolt adatok referenciájának típusa,
ADAPT::const_reference	a tárolt adatok konstans referenciájának típusa.

2.4.3 Konténerek konstruálása és értékadása

A konténerobjektumok létrehozásakor, a megfelelő konstruktor hívásával lehetőség van arra, hogy a konténert adatokkal töltsük fel. Ezeket az adatokat akár más konténerből is átmásolhatjuk vagy átmozgathatjuk. Az alábbi táblázatban **CC** a konténer osztályát jelöli, míg a **c** maga a konténerobjektum:

Konstruktorhívás	Leírás
CC c vagy CC c{}	alapértelmezett konstruktor, létrejön egy üres konténer,
CC c(n)	c inicializálása n darab value_type {} értékű elemmel (nem asszociatív konténerek esetén),
CC c(n, x)	c inicializálása n darab x értékű elemmel (nem asszociatív konténerek esetén),
CC c {init. lista}	c inicializálása a megadott inicializációs lista elemeivel,
CC c(itb, ite) vagy c{itb, ite}	c inicializálása az iterátorokkal kijelölt [itb, ite) tartomány elemeinek átmásolásával,
CC c1(c2) vagy c1{c2}	c1 inicializálása az azonos típusú elemeket tároló c2 konténer elemeinek átmásolásával ,
CC c1(move(c2)) vagy c1{move(c2)}	c1 inicializálása az azonos típusú elemeket tároló c2 konténer elemeinek átmozgatásával .

A fenti konstruktorhívások segítségével ideiglenes konténereket is készíthetünk, amennyiben a típusnevet paraméterezzük: **CC()**, **CC {init. lista}**, **CC(c)** stb. A táblázatban összegyűjtött megoldásokat ismét a **vector** típusú konténerek felhasználásával szemléltetjük:

```
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> iVektor;

int main() {
    iVektor v01, v02 {};           // üres vektorok
    iVektor v11(10);              // 10 db 0 értékű elemet tartalmazó vektor
    iVektor v21(10, 12);          // 10 db 12 értékű elemet tartalmazó vektor
    iVektor v31 {1,1,2,3,5,8,13};  // feltöltés inicializációs listából
    int a[5] {12, 23, 34, 45, 56};
    iVektor v41(&a[1], &a[3]), v42 {&a[1], &a[3]}; // 23, 34
    iVektor v51(v41), v52 {v41};   // átmásolással
    iVektor v61(move(v51)), v62 {move(v52)}; // átmozgatással
    return 0;
}
```

A konténer adapterek esetén csak a másoló és a mozgató konstruktorok használata megengedett, azonban mindkét művelet elvégezhető a konténer alapjául szolgáló soros tároló objektumának felhasználásával is. A prioritásos sor esetén a konstruktorhívás első argumentumaként az összehasonlító függvény kell megadnunk, és csak a második helyen szerepelhet a konténer.

A konténerekhez definiált **értékadás** operátorral a bal oldalon álló konténer minden elemét felülírhatjuk egy másik, azonos típusú elemeket tartalmazó konténer elemeinek átmásolásával vagy átmozgatásával, illetve egy inicializációs lista tartalmával. Az előző példa *iVektor* típusánál maradva:

```
iVektor av {1, 2, 2, 3, 3, 3}, bv, cv;
bv = av;           // elemek átmásolásával
cv = move(av);     // elemek áthelyezésével
av = {3, 5, 7, 9, 11}; // inicializációs listából
bv = vector<int> {7,1,17}; // ideiglenes objektum elemeinek átmásolásával
cv = move(iVektor(10, -1)); // ideiglenes objektum elemeinek átmozgatásával
```

Az értékadáshoz hasonlóan a konténer összes elemét felülírja az `assign()` tagfüggvény hívása.

```
iVektor av {1, 2, 2, 3, 3, 3}, bv, cv;
av.assign(10, 12);           // av feltöltése 10 darab 12 értékű elemmel
bv.assign(begin(av), end(av)); // bv feltöltése a megadott iterátor-tartományból
cv.assign({3, 5, 7, 9, 11});  // cv feltöltése inicializációs listából
```

Felhívjuk a figyelmet arra, hogy az `assign()` hívás első formája nem használható az asszociatív tárolók esetén. A konténer adapterek az értékadás operátorának csak az első két formáját (másolás, mozgás) támogatják, és nem rendelkeznek `assign()` tagfüggvénnyel.

2.4.3 A konténerek közös tagfüggvényei és operátorai

A következő néhány részben csak az elsődleges konténerekkel foglalkozunk, vagyis az elmondottak nem vonatkoznak sem a konténer adapterekre, sem pedig a tárolószerű osztálysablonokra. Valójában csak nagyon kevés közös tagfüggvénye van a tárolóknak, így a bemutatás során a kivételekről is szólnunk kell.

2.4.3.1 Iterátorok lekérdezése

Minden konténerhez tartozik egy `iterator` és egy `const_iterator` típus, amelyek segítik az adott tároló bejárását. A `forward_list` és a `rendezetlen asszociatív tárolók kivételével` az elemek fordított bejárása is lehetséges a `reverse_iterator` és a `const_reverse_iterator` típusok felhasználásával.

```
konténer<típus>::iterator iter;
konténer<típus>::const_iterator citer;
konténer<típus>::reverse_iterator riter;
konténer<típus>::const_reverse_iterator criter;
```

A fentiekben felsorolt típusú iterátor változók inicializálását tagfüggvények segítik.

Tagfüggvényhívás	Leírás
<code>iter = c.begin()</code> , <code>citer = c.begin()</code> <code>citer = c.cbegin()</code>	az első elem iterátorának lekérdezése,
<code>iter = c.end()</code> , <code>citer = c.end()</code> <code>citer = c.cend()</code>	az utolsó elem utáni pozíció iterátorának lekérdezése,
<code>riter = c.rbegin()</code> , <code>criter = c.rbegin()</code> <code>criter = c.crbegin()</code>	a fordított elemsor első eleméhez tartozó iterátor lekérdezése,
<code>riter = c.rend()</code> , <code>criter = c.rend()</code> <code>criter = c.crend()</code>	a fordított elemsor utolsó elem utáni pozícióját kijelölő iterátor lekérdezése.

Megjegyezzük, hogy üres konténer esetén a `c.begin()` és a `c.end()` hívások ugyanazt az iterátort adják.

Egyszerűbbé tehető a programunk, ha a bejáró változók típusának meghatározását a C++ fordítóra bízuk:

```
auto iter = c.begin();
auto citer = c.cend();
auto riter = c.rend();
auto criter = c.crbegin();
```

A `forward_list` esetén az `első elem előtti` pozíció iterátorát szolgáltatják a `before_begin()` és a `cbefore_begin()` tagfüggvények. Ezekre az iterátor értékekre az előrehaladó lista `insert_after()`, `emplace_after()`, `erase_after()` és `splice_after()` tagfüggvényei hívásánál, valamint az inkrementálás műveleténél lehet szükségünk.

Nézzünk néhány példát a `c` konténerobjektum bejárására! Klasszikus megoldásnak számít a **for** ciklus alábbi módon történő alkalmazása:

```
for (auto it=c.begin(); it!=c.end(); it++) {
    cout << *it << endl;
}
```

A fentivel megegyező működésű, azonban jóval olvashatóbb megoldást biztosít a C++11 nyelv:

```
for (auto& elem : c) {
    cout << elem << endl;
}
```

Az első **for** ciklusban a `c.end()` ismételt hívását kiküszöbölhetjük segédváltozó bevezetésével. A megoldást a fordított bejárással szemléltetjük:

```
auto vege = c.rend();
for (auto rit=c.rbegin(); rit!=vege; rit++) {
    cout << *rit << endl;
}
```

2.4.3.2 A tárolók elemszáma

A konténerek esetén fontos tudnunk, hogy hány elemet tárolnak, és hogy nem üresek-e. Az alábbi tagfüggvények segítenek megszerezni ezeket az információkat:

Tagfüggvényhívás	Leírás
<code>c.empty()</code>	true értékkel tér vissza, ha a <code>c</code> egyetlen elemet sem tartalmaz,
<code>n = c.size()</code>	a <code>c</code> konténerben tárolt elemek száma (<i>size_type</i> típusú adat), a <code>size()</code> a forward_list esetén nem használható ,
<code>n = c.max_size()</code>	a <code>c</code> konténerben tárolható elemek maximális száma (<i>size_type</i> típusú adat).

Az előrehaladó lista (és más konténerek) esetén a `distance()` függvénysablon segíthet az elemszám megállapításában:

```
#include <forward_list>
#include <iostream>
using namespace std;

int main()
{
    forward_list<char> betuk {'S', 'T', 'L'};
    if (!betuk.empty()) {
        size_t elemszam = distance(betuk.begin(), betuk.end());
        cout << elemszam << endl; // 3
    }
    return 0;
}
```

2.4.3.3 Az elemek közvetlen elérése

Már láttuk, hogy az iterátorok segítségével a konténerek minden elemét elérhetjük (közvetve). Bizonyos konténerek esetén néhány tagfüggvény gyors, közvetlen elérést biztosít az elemekhez. Ezek a függvények az elemek referenciájával, illetve konstans referenciájával (**const** tárolók esetén) térnek vissza.

Az `at()` tagfüggvény és az `indexelés []` operátora

Az **array**, a **vector** és a **deque** soros konténerek elemeit egy *size_type* típusú index segítségével is elérhetjük. Az érvényes indexek **0**-val kezdődnek, és a `size()-1` értékig terjednek. Az indexhatárokat a két megoldás közül csak az `at(index)` hívás ellenőrzi, és `out_of_range` kivételt generál, ha az *index* kívül esik az érvényes tartományon.

```

#include <stdexcept>
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> fibo {0, 1, 1, 2, 3, 5, 8, 13, 21, 30};
    try {
        fibo.at(9) = fibo[8] + fibo.at(7);
        fibo[12] = fibo.at(5);
        fibo.at(10) = 55;
    }
    catch(const out_of_range& ex) {
        cout << ex.what() << endl;
    }

    for (size_t i = 0; i<fibo.size(); i++) {
        cout << fibo[i] << " ";
    }
    cout << endl;
    return 0;
}

```

```

invalid vector<T> subscript
0 1 1 2 3 5 8 13 21 34

```

A legtöbb adatot lekérdező tagfüggvény az elem referenciájával (illetve konstans referenciájával) tér vissza. A programban választhatunk, hogy az elemhez egy saját nevet illesztünk, vagy az elem másolatával dolgozunk:

```

#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v{ 21, 32, 43, 54, 65 } ;
    int e = v[2];      // a 2 indexű elem másolata kerül az e1-be
    e += 11;           // a v vektor nem változik
    int& r = v[4];      // a 4 indexű elem referenciája másolódik az r-be
    r += 11;           // a v[4] elem módosul
    for (auto x: v)
        cout << x << " "; // 21 32 43 54 76
    return 0;
}

```

A **map** és az **unordered_map** konténerek esetén szintén használhatjuk az **at()** függvényt és az indexelés operátorát. Mindkét esetben argumentumként a kulcsot kell megadnunk, és a vele párban tárolt adat referenciáját kapjuk vissza. Ha kulcs nem létezik, az **at(kulcs)** hívás **out_of_range** kivételt hoz létre, míg az indexelés esetén egy új elemmel bővül az asszociatív tömb, melynek adat része az **AdatTípus {}** értékkel inicializálódik.

Az alábbi példában sorszámokkal látjuk el az angol ABC betűit:

```

#include <map>
#include <iostream>
using namespace std;

int main()
{
    map<char, int>kodtabla {{'A', 65}, {'B', 66}, {'Z', 0}, {'K', -1}};
    for (auto& par : kodtabla) {
        kodtabla[par.first] = par.first - 'A' + 1;
    }
}

```



```

    kodtabla['A'] = 1;    // Létező adat átírása
    kodtabla['M'] = 13;   // új elem hozzáadása
    for (const auto& par : kodtabla) {
        cout << par.first << " : " << par.second << endl;
    }
    return 0;
}

```

```

A : 1
B : 2
K : 11
M : 13
Z : 26

```

A `back()` és a `front()` tagfüggvények

A **soros konténerek mindegyike** esetén a `front()` hívás megadja az első tárolt elem hivatkozását. A **`forward_list` kivételével** pedig, a `back()` tagfüggvény az utolsó elem referenciájával tér vissza.

```

#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<string> C98Tarolok {"Hector", "deque", "list",
                              "set", "multiset", "map", "ulti"};
    C98Tarolok.front() = "vector";
    C98Tarolok.back() = "multimap";
    cout << C98Tarolok.front() << endl;
    cout << *C98Tarolok.begin() << endl;
    cout << C98Tarolok.back() << endl;
    cout << *(C98Tarolok.end()-1) << endl;
    return 0;
}

```

```

vector
vector
multimap
multimap

```

A `data()` tagfüggvény

Az **`array`** és a **`vector`** konténerek tartalma folytonos memóriaterületen helyezkedik el. A lefoglalt területek kezdetére mutató, **`AdatTípus*`** típusú pointerrel tér vissza a `data()` tagfüggvény. A visszatért mutató segítségével hagyományos módon is feldolgozhatjuk a tárolt adatokat:

```

#include <array>
#include <iostream>
#include <cstdint>
#include <cstring>
using namespace std;

int main() {
    const size_t meret = 12;
    array<uint8_t, meret> tomb;
    memset(tomb.data(), 123, meret);
    for (uint8_t b : tomb) {
        cout << unsigned(b) << endl;
    }
    return 0;
}

```

2.4.3.4 A konténerek módosítása

Az eddigi tagfüggvények hívásával nem változtattuk meg a konténerekben tárolt adatok elhelyezkedését a memóriában. A következő függvénycsoport elemei azonban, új elemek hozzáadásával, ele-

mek törlésével és felcserélésével érvénytelenítik az elemreferenciákat, az elemmutatókat és az iterátorokat. Így ezeket a műveletek végeztével ismételtlen le kell kérdeznünk. Felhívjuk a figyelmet arra, hogy az **array** tárolók semmilyen módosítása sem megengedett.

Műveletek a soros tárolók elején (*front*) és végén (*back*)

A soros tárolók elején kiemelt fontosságú az adatsor első és utolsó pozíciójának kezelése, hiszen ezek a műveletek igen **gyorsak**. A **vector** tárolók **nem definiálják a *_front* utótagú függvényeket**, míg a **forward_list** konténerek **nem rendelkeznek *_back* utótagú tagokkal**, a műveletek lineáris futásideje miatt.

Tagfüggvényhívás	Leírás
<code>c.push_front(x)</code> <code>c.push_front(move(x))</code> <code>c.emplace_front(argumentumok)</code>	az <i>x</i> adat bemásolása, illetve behelyezése a konténer elejére , új elem elhelyezése a <i>c</i> konténer elején, és inicializálása az argumentumoknak megfelelő konstruktor hívásával,
<code>c.pop_front()</code>	a <i>c</i> konténer első elemének eltávolítása,
<code>c.push_back(x)</code> <code>c.push_back(move(x))</code> <code>c.emplace_back(argumentumok)</code>	az <i>x</i> adat bemásolása, illetve behelyezése a konténer végére , új elem elhelyezése a <i>c</i> konténer végén, és inicializálása az argumentumoknak megfelelő konstruktor hívásával,
<code>c.pop_back()</code>	a <i>c</i> konténer utolsó elemének eltávolítása.

Az alábbi kis program a szóközzel elválasztott szavakat beolvassa, és egy vektor végéhez illeszti. (Az adatbevitelt a <Ctrl+Z> billentyűvel kell lezárni.)

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    vector<string> szavak;
    string szo;
    while (cin >> szo)
        szavak.push_back(szo);
}
```

Elem beszúrása tetszőleges pozícióra

Új elem beillesztése a konténerbe, és elem törlése a konténerből – a tároló típusától függően – igen csak időigényes is lehet. Csak emlékeztetőül, a **vector** és a **deque** tárolók esetén ezek a műveletek lineáris futásidővel rendelkeznek, a listák esetén konstans, míg az asszociatív tárolóknál logaritmikus az időigény. Az **előrehaladó listák** (*flc*) esetén az **insert()** tagfüggvény helyett a teljesen azonos paraméterezésű **insert_after()** függvényt kell használnunk.

Az asszociatív konténerek (*ac*) beszúrás művelete jelentősen különbözik a soros tárolóknál (*sc*) használttól, hiszen az előbbinél az elemek azonosítása kulccsal történik.

Tagfüggvényhívás	Leírás
<code>iter = sc.insert(itp, adat)</code> <code>iter = sc.insert(itp, move(adat))</code>	az adat beszúrása/áthelyezése az <i>itp</i> iterátorral kijelölt elem elé,
<code>iter = sc.emplace(itp, argumentumok)</code>	új elem beszúrása az <i>sc</i> konténer <i>itp</i> eleme elé, és inicializálása az argumentumoknak megfelelő konstruktor hívásával,

Tagfüggvényhívás <i>(folytatás)</i>	Leírás
<code>iter = sc.insert(itp, n, adat)</code>	<i>n</i> darab <i>adat</i> beszúrása <i>itp</i> iterátorral kijelölt elem elé,
<code>iter = sc.insert(itp, itb, ite)</code>	elemek beszúrása az <i>[itb, ite)</i> tartományból az <i>itp</i> elem elé,
<code>iter = sc.insert(itp, {init. lista})</code>	adatok beszúrása az <i>inicializációs listából</i> az <i>itp</i> elem elé,
<code>iter = fl.insert_after(itp, mint fenti!)</code> <code>iter = fl.emplace_after(itp, arg.ok)</code>	ez a két tagfüggvény a <i>forward_list</i> konténerekkel hívható; mindkét függvény az <i>itp</i> pozíció <i>után</i> szúrja be a kijelölt adatokat,
Asszociatív konténerek esetén használható hívások:	
<code>ret = ac.insert(adat)</code> <code>ret = ac.insert(move(adat))</code>	az <i>adat</i> beillesztése, illetve áthelyezése egy asszociatív konténerbe, a visszaadott érték vagy egy adatpár, vagy egy iterátor (lásd a táblázat után),
<code>ret = ac.emplace(argumentumok)</code>	új elem beszúrása a konténerbe, és inicializálása az argumentumoknak megfelelő konstruktor hívásával,
<code>iter = ac.insert(ith, adat)</code> <code>iter = ac.insert(ith, move(adat))</code> <code>iter = ac.emplace_hint(ith, arg.ok)</code>	ez a három hívás az előző három függvényhívás hatékonyabb megvalósítása, ahol egy jól megválasztott céliterátorral (<i>ith, hint</i>) segíthetünk megtalálni az új elem helyét a konténerben,
<code>ac.insert(itb, ite)</code> <code>ac.insert({init. lista})</code>	elemek beszúrása az <i>[itb, ite)</i> tartományból, adatok beszúrása az <i>inicializációs listából</i> .

Soros konténerek esetén a függvények által visszaadott *iter* iterátorok a beillesztett elemre mutatnak a tárolóban.

Az egyedi kulcsot tartalmazó asszociatív tárolók (*set, map* stb.) esetén a beszúrás csak akkor lesz sikeres, ha a kulcs még nem szerepel a konténerben. Az ilyen tárolók használatakor a *ret* visszaadott érték egy *pair<iterator, bool>* pár, ahol az iterátor a beszúrt elemre hivatkozik, vagy arra, ami megakadályozta a beillesztést, a logikai érték pedig *true*-val jelzi, ha megtörtént a művelet. A többi, azonos kulcsot is megengedő asszociatív tárolóknál (*multiset, multimap* stb.) a *ret* egy iterátor, ami kijelöli a beszúrt elemet.

A műveletek alkalmazása a lista adatstruktúra esetén:

```
#include <iostream>
#include <list>
#include <string>
using namespace std;

class Ugyfel {
    unsigned ID;
    string nev, tel;
public:
    Ugyfel(unsigned id, string n, string t) {
        ID = id, nev = n, tel = t;
    }
    Ugyfel(const Ugyfel& ugyfel) {
        *this = ugyfel;
    }
    void Kiir() const {
        cout << ID << "\t" << nev << "\t" << tel << endl;
    }
};

int main() {
    list<Ugyfel> ugyfelek { {67, "Nagy Kelemen", "-"},
                           {32, "Zold Zoltan", "06-50-234-32-12"} };
    Ugyfel ugyfel {23, "Kiss Veronika", "06-1-203-34-56"};
    // Az ugyfel beszúrása a fenti két Listaelem közé másolással:
    ugyfelek.insert(next(begin(ugyfelek)), ugyfel);
```

```
// Listaelem felépítése a lista elején:
ugyfelek.emplace(begin(ugyfelek), 17, "Okos Antal", "06-1-123-23-23");
// A lista tartalma:
for (const auto& ugyfel : ugyfelek)
    ugyfel.Kiir();
return 0;
}
```

17	Okos Antal	06-1-123-23-23
67	Nagy Kelemen	-
23	Kiss Veronika	06-1-203-34-56
32	Zold Zoltan	06-50-234-32-12

Tetszőleges elem törlése

Míg konténerek minden elemét egyetlen hívással eltávolíthatjuk, addig az egyes elemek törléséhez iterátorokat kell alkalmaznunk. Az asszociatív tárolók esetén a kulcs megadásával is azonosíthatjuk a törlendő elemet. Felhívjuk a figyelmet arra, hogy a **forward_list** konténer saját **_after** utótagú tagfüggvényeket definiál.

A táblázatban a « » jelek között részek nem minden konténertípus esetén érvényesek. A soros tárolók esetén a törlés egy iterátorral tér vissza, amely az utolsó törölt elem utáni elemre hivatkozik. Ezzel szemben az asszociatív tárolóknál az **erase()** tagfüggvény **void** típusú.

Tagfüggvényhívás	Leírás
c.clear()	a konténer összes elemének eltávolítása,
« iter = »c.erase(itp)	az itp iterátorral kijelölt pozíción álló elem törlése a c konténerből,
« iter = »c.erase(itb, ite)	az [itb, ite) iterátor tartománnyal kijelölt elemek eltávolítása a c konténerből,
iter = fl.erase_after(itp)	az itp iterátorral kijelölt pozíció után álló elem törlése az fl előrehaladó listából ,
iter = fl.erase_after(itb, ite)	az (itb, ite) nyitott iterátor tartományba eső elemek eltávolítása az fl előrehaladó listából ,
n = c.erase(kulcs)	a c asszociatív tároló adott kulcsú elemeinek törlése – a size_type típusú visszatérési érték a törölt elemek számát tartalmazza.

Soros konténerek átméretezése (resize)

A soros konténerek méretét (elemszámát) megváltoztathatjuk/beállíthatjuk a **resize()** tagfüggvény hívásával. Ez a művelet nem érhető el az **array** tárolókban, azonban a **string** típusú objektumokkal is felhasználható. Felhívjuk a figyelmet arra, hogy a **resize()** műveletet a **vector (string)** és a **deque** tárolók esetén igencsak időigényes. Az átméretezés során a megmaradnak azok a régi elemek, melyek belesznek az új méretű konténerbe.

Tagfüggvényhívás	Leírás
c.resize(n)	a c konténer mérete n -re módosul, és a hozzáadott elemek az AdatTípus alapértelmezett értékével töltődnek fel,
c.resize(n, adat)	a c konténer mérete n -re módosul, és a hozzáadott elemek az adattal töltődnek fel.

Az alábbi példában egy globális szinten definiált **vector** konténer méretét a **main()** függvényben megváltoztatjuk:

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

vector<int> szamok {121, 144};
```

```
int main() {
    int n;
    cout << "n=";
    cin >> n; cin.get();
    szamok.resize(n, 0);
    for (int i = 2; i < n; i++)
        szamok[i] = i * i;
    copy (szamok.begin(), szamok.end(), ostream_iterator<int>(cout, " "));
    return 0;
}
```

```
n=5
121, 144, 4, 9, 16,
```

Üres konténer esetén a méretet értékadással is beállíthatjuk:

```
vector<int> szamok;
szamok = move(vector<int>(n));
```

Konténerek tartalmának felcserélése (swap)

Minden konténer rendelkezik egy *swap()* tagfüggvénnyel, melynek segítségével gyorsan felcserélhetjük az aktuális konténer tartalmát egy másik ugyanilyen konténer elmeivel. Két tároló tartalmának teljes cseréjéhez használhatjuk a *swap()* algoritmus specializált változatait is.

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
using namespace std;

int main() {
    vector<int> av { 1, 2, 3, 7, 9 };
    vector<int> bv { 121, 144, 169 };
    av.swap(bv); // tagfüggvény
    copy (begin(bv), end(bv), ostream_iterator<int>(cout, " "));
    cout << endl;
    swap(av, bv); // algoritmus
    copy (begin(bv), end(bv), ostream_iterator<int>(cout, " "));
    cout << endl;
    return 0;
}
```

```
1, 2, 3, 7, 9,
121, 144, 169,
```

Konténerek tartalmának összehasonlítása

Függvénysablonok segítségével azonos típusú, soros és rendezett asszociatív konténerek tartalmát a C++ nyelv relációs műveleteivel (*==*, *!=*, *<*, *<=*, *>*, *>=*) összehasonlíthatjuk. Nem rendezett asszociatív tárolók esetén csak az azonosság (*==*) és a különbözőség (*!=*) állapítható meg.

Két konténert azonosnak (*==*) tekintünk, ha a tartalmuk megegyezik, ellenkező esetben különböznek (*!=*). A többi reláció esetén ún. *lexikografikus összehasonlítás* adja művelet eredményét. Ez azt jelenti, hogy az elemenkénti összehasonlítás során az elsőként megtalált különböző elemek összevetése határozza meg az eredményt.

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int main() {
    vector<int> av { 1, 2, 3, 7, 9 };
    vector<int> bv { 1, 5 };
    cout << (av == bv) << endl; // 0
    cout << (av != bv) << endl; // 1
    cout << (av < bv) << endl; // 1
    cout << (av <= bv) << endl; // 1
    cout << (av > bv) << endl; // 0
    cout << (av >= bv) << endl; // 0
    return 0;
}
```

2.4.4 Soros konténerek alkalmazása

A következőkben röviden áttekintjük az egyes soros tárolók használatával kapcsolatos további ismereteket.

2.4.4.1 array

Az array konténerek típusát az adattípus és a rögzített méret együtt határozza meg.

```
array<double, 100> a;
```

Ennél fogva, ellentétben a hagyományos C tömbökkel, a függvények hívása során az adattípusok egyezése mellett további feltétel a méretek azonossága is.

```
#include <array>
#include <iostream>
using namespace std;

int skalar(const array<int, 10>& a, const array<int, 10>& b) {
    int osszeg = 0;
    for (size_t i=0; i<a.size(); i++)
        osszeg += a[i]*b[i];
    return osszeg;
}

int main() {
    array<int, 10> tomb1 {10, 34, 12, 31, 14, 28, 16};
    array<int, 10> tomb2;
    tomb2 = tomb1;
    tomb1[0]--;
    array<int, 10> tomb3(tomb2);
    tomb3.fill(0);
    cout << skalar(tomb1, tomb2) << endl;
    cout << (tomb1 < tomb2) << endl;
    tomb3.swap(tomb2);
}
```

Az azonos típusú tömb tárolók esetén használhatjuk az értékadás és az összehasonlítás műveleteket, valamint a csere ([swap\(\)](#)) tagfüggvényt.

Többdimenziós tömböt is készíthetünk az alábbiak szerint, melynek elemei sorfolytonosan helyezkednek el a memóriában:

```
#include <array>
#include <iostream>
using namespace std;

int main() {
    const size_t nsor = 3, noszlop = 5;
    array <array<double, noszlop>, nsor> mat3x5 { {{ 1, 2, 3, 0, 1},
                                                    { 5, 1, 0, 0, 2},
                                                    { 3, 0, 1, 1, 0}} };
}
```

```

for (size_t s=0; s<mat3x5.size(); s++) {
    for (size_t o=0; o<mat3x5[s].size(); o++)
        cout << mat3x5[s][o] << "\t";
    cout << endl;
}
cout << endl;
for (const auto& sor : mat3x5) {
    for (double e : sor)
        cout << e << "\t";
    cout << endl;
}
return 0;
}

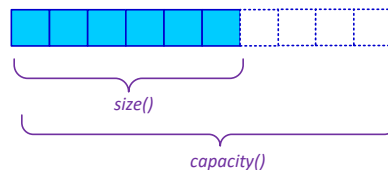
```

Amennyiben a rögzített méret korlátaitól meg szeretnénk szabadulni, a **vector** osztálysablonot kell választanunk.

2.4.4.2 vector

Mivel a C++ programozás során a **vector** osztálysablon teljesen átveheti a hagyományos C tömbök szerepét, érdemes megismerkednünk effektív használatának kérdéseivel. Mivel a vektorok a C tömbökhöz hasonlóan folytonos memóriaterületen helyezkednek el, elég időigényes művelet, ha elemeket nem a végén adunk hozzá.

Azért, hogy az elemek végén történő hozzáadása során (*push_back()*) ne kelljen folyamatosan újabb területeket lefoglalni a vektor számára, a memóriefoglalás és az elemhozzáadás elválik egymástól. Elemek hozzáadásakor a vektor számára mindig egy nagyobb terület foglalódik le a memóriában, melynek mérete a *capacity()* tagfüggvény hívásával tudható meg. A *size()* és a *capacity()* értékek viszonya: $size() \leq capacity()$.



Ha az elemek hozzáadása során (*size()*) elérjük a lefoglalt méretet (*capacity()*), új terület foglalódik le, ahova átmásolódnak a már meglévő elemek. (Az átmásolás után természetesen minden iterátor, mutató és referencia érvénytelenné válik.)

Ha előre ismerjük a vektorban tárolni kívánt elemek számát, akkor rossz programozási gyakorlatot tükröz az alábbi program, melynek futása során több mint 10-szer másolódik új területre a vektor:

```

#include <vector>
#include <iostream>
using namespace std;

int main() {
    size_t n = 1000; // ismerjük
    vector<double> v;
    for (int i=0; i < n; i++) {
        v.push_back(i);
    }
    cout << v.size() << " " << v.capacity() << endl; // 1000 1024
}

```

Egyszer sem másolódik a vektor tartalma, ha megfelelő konstruktort és értékadást használunk:

```

size_t n = 1000; // ismerjük
vector<double> v(n);

```

```
for (int i=0; i<n; i++) {
    v[i] = i;
}
```

Amennyiben egy feltöltött n -elemű vektort m elem hozzáadásával szeretnénk bővíteni, akkor a hatékony működés érdekében használnunk kell a `reserve()` tagfüggvényt, melynek segítségével megtartjuk, vagy megnövelhetjük a lefoglalt területet. Ha a `reserve()` hívás argumentuma kisebb vagy egyenlő, mint a `capacity()` érték, semmi sem történik, tehát ezen az úton nem csökkenthető a vektor számára lefoglalt terület mérete.

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
    size_t n = 1000, m=2015;
    vector<double> v(n);
    for (int i=0; i<n; i++) {
        v[i] = i;
    }
    cout << v.size() << " " << v.capacity() << endl; // 1000 1000
    v.reserve(n + m);
    cout << v.size() << " " << v.capacity() << endl; // 1000 3015
    for (int i=0; i<m; i++) {
        v.push_back(i*i);
    }
    cout << v.size() << " " << v.capacity() << endl; // 3015 3015
}
```

Amennyiben a már megismert `resize()` tagfüggvény hívásakor az új elemszám kisebb (vagy egyenlő), mint a kapacitás csak a `size()` méret változik meg, míg a `capacity()` érték változatlan marad. Ha azonban az igényelt új méret meghaladja az aktuális kapacitást, mindkét függvény az új mérettel tér vissza.

A `vector` típus méretkezelésével kapcsolatos utolsó `shrink_to_fit()` függvény a lefoglalt memóriaterület méretét (`capacity()`) az elemszámhoz (`size()`) igazítja.

A vektor, mint függvényparaméter

Mivel a `vector` osztálysablonnál a méret nem épül be a típusba, az azonos típusú adatokat tároló, de különböző méretű vektorok teljesen típuskompatibilisak egymással. Az alábbi példában ezt használjuk ki, amikor a polinomok összeadását végző függvényt (`PolinomAdd`) definiáljuk:

```
#include <iostream>
#include <vector>
using namespace std;

typedef vector<double> Polinom;

Polinom PolinomAdd(const Polinom& pola, const Polinom& polb) {
    Polinom polc;
    if (pola.size() > polb.size()) {
        polc = pola;
        for(int i=0; i<polb.size(); i++)
            polc[i] += polb[i];
    }
    else {
        polc = polb;
        for(int i=0; i<pola.size(); i++)
            polc[i] += pola[i];
    }
    return polc;
}
```



```

int main() {
    Polinom a { 3, 2, 1 };           //            $1x^2 + 2x + 3$ 
    Polinom b { 2, 5, 3, 2, 1 };    //  $1x^4 + 2x^3 + 3x^2 + 5x + 2$ 
    Polinom c;
    c = PolinomAdd(a, b);
    Polinom::const_reverse_iterator p;
    for (p = c.rbegin(); p!=c.rend(); p++)
        cout << *p << " ";        // 1 2 4 7 5
    cout << endl;
    return 0;
}

```

Mátrixok tárolása vektorokban

A **vector** osztálysablonok egymásba ágyazásával tetszőleges kiterjedésű dinamikus tömb létrehozható, melyek közül a kétdimenziós mátrixok tárolását nézzük meg részletesebben.

vector<vector<int>> *matrix* (sorok, vector<int>(oszlopok, init))

A megoldás egyetlen hátránya a kétdimenziós C tömbökkel szemben, hogy nem folytonosan helyezkednek el a memóriában a mátrix sorai. Az alábbi példában kétdimenziós mátrixok feltöltését és kiírását végző függvényeket definiálunk, különböző kiterjedésű mátrixokat (háromszög mátrixot is) használunk, végül pedig megnöveljük egy mátrix sorainak és oszlopainak számát:

```

#include <iostream>
#include <vector>
using namespace std;

typedef vector<vector<int>> Matrix;

void MatrixFeltolt(Matrix& m, int e) {
    for (int i=0; i<m.size(); i++)
        for(int j=0; j<m[i].size(); j++)
            m[i][j] = e;
}

void MatrixKiir(const Matrix& m) {
    for (int i=0; i<m.size(); i++) {
        for(int j=0; j<m[i].size(); j++)
            cout << m[i][j] << "\t";
        cout << endl;
    }
}

int main() {
    // mátrixok készítése kezdőérték-adással
    Matrix m2x2 {{1, 2},
                 {3, 4}};
    Matrix m2x3 {{1, 2, 3},
                 {4, 5, 6}};
    Matrix mhsz {{1},
                 {3, 4},
                 {5, 6, 7}};
    MatrixKiir(m2x2); cout << endl;
    MatrixKiir(m2x3); cout << endl;
    MatrixKiir(mhsz); cout << endl;

    // mátrix készítése a sorok és az oszlopok számának megadásával
    const int sorok = 3, oszlopok = 5;
    Matrix mdin = move(vector<vector<int>> (sorok, vector<int>(oszlopok, 7)));
    MatrixKiir(mdin); cout << endl;
    MatrixFeltolt(mdin, 1);
}

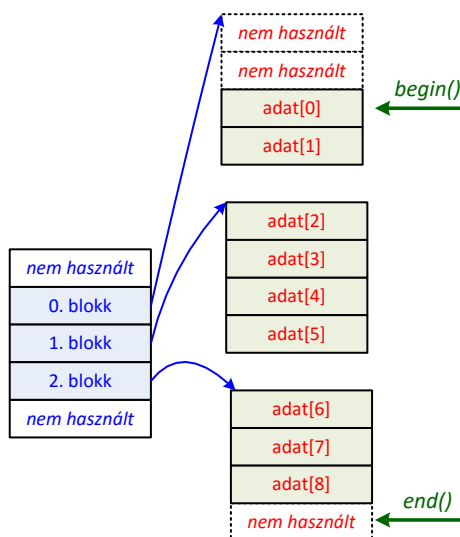
```

```
// mátrix újreméretezése
mdin.resize(2*sorok); // a sorok számának megváltoztatása
for (int i=0; i<mdin.size(); i++) // soronként az oszlopok számának módosítása
    mdin[i].resize(oszlopok+2);
MatrixKiir(mdin); cout << endl;
return 0;
}
```

1	2					
3	4					
1	2	3				
4	5	6				
1						
3	4					
5	6	7				
7	7	7	7	7		
7	7	7	7	7		
7	7	7	7	7		
1	1	1	1	1	0	0
1	1	1	1	1	0	0
1	1	1	1	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

2.4.4.3 deque

A **deque** kétvégű sor (*double ended queue*) sok szempontból emlékeztet a vektorra, ugyanúgy soros tároló, tetszőleges elérést biztosít az elemeihez, konstans idő szükséges az elemek végeken (itt kettő is van) történő hozzáadásához és levételéhez, valamint lineáris idő kell a közbenső pozíciókban a beszúráshoz és a törléshez.



A kettősvégű sor a vektorral szemben nem teszi lehetővé a lefoglalt terület méretének lekérdezését, illetve az elem-műveletek nélküli memóriafoglalást.

Egyedül a **shrink_to_fit()** tagfüggvényt használhatjuk, az elemek számára nem használt memóriaterületek törlésére. A **deque** elemei nem folytonosan helyezkednek el, leginkább a fenti matrix típus memóriafoglalására hasonlít a memóriahasználata.

A kettősvégű sor segítségével olyan verem (*stack*) és sor (*queue*) adatstruktúrát valósíthatunk meg, melyek elemei nemcsak a végeken érhető el.

Az alábbi példával bemutatjuk a **deque** használatának lépéseit és műveleteit:

```
#include <deque>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;
```

```

void Kiir(const deque<string>& deq) {
    cout << "( ";
    for (const string& s : deq)
        cout << s << " ";
    cout << ")" << endl;
}

int main() {
    deque<string> ksor { "kutya" };
    ksor.push_back("mokus");
    ksor.insert(begin(ksor),"cica");
    ksor.push_front("eger");
    cout << "Kiindulasi elemek: ";
    Kiir(ksor);

    cout << "\nBejaras indexelve: " << endl;
    for (int i=0; i<ksor.size(); i++)
        cout << ksor[i] << endl;
    cout << endl;

    cout << "Bejaras iteratorral: " << endl;
    auto iter = begin(ksor);
    while (iter != end(ksor)) {
        cout << *iter++ << endl;
    }
    cout << endl;

    cout << "Elemek elerese:" << endl;
    cout << "ksor.at(0)   = " << ksor.at(0) << endl;
    cout << "ksor.[0]      = " << ksor[0] << endl;
    cout << "ksor.front() = " << ksor.front() << endl;
    cout << "ksor.at(3)    = " << ksor.at(3) << endl;
    cout << "ksor.[3]      = " << ksor[3] << endl;
    cout << "ksor.back()   = " << ksor.back() << endl;

    cout << "\nModositasok:" << endl;
    ksor[1] = "tekno"; ksor.at(0) = "kacsa";
    Kiir(ksor);
    ksor.pop_front();
    cout << "ksor.pop_front() utan: "; Kiir(ksor);
    ksor.pop_back();
    cout << "ksor.pop_back() utan: "; Kiir(ksor);
}

```

Kiindulasi elemek: (eger cica kutya mokus)

Bejaras indexelve:

eger

cica

kutya

mokus

Bejaras iteratorral:

eger

cica

kutya

mokus

Elemek elerese:

ksor.at(0) = eger

ksor.[0] = eger

ksor.front() = eger

ksor.at(3) = mokus

ksor.[3] = mokus

ksor.back() = mokus

Modositasok:

(kacsa tekno kutya mokus)

ksor.pop_front() utan: (tekno kutya mokus)

ksor.pop_back() utan: (tekno kutya)

2.4.4.4 list

A **list** típus valójában egy kettős láncolású lista, amely nem támogatja az elemek közvetlen elérését. A kettős láncolás az adatsor előre- és hátrahaladó bejárását egyaránt biztosítja, valamint konstans idejű adatbeszúrás, illetve törlést tesz lehetővé a lista tetszőleges pozíciójában. A lista fontos jellemzője, hogy ellentétben a **vektor** és a **deque** konténerekkel, elem beszúrásakor a már lekérdezett iterátorok nem válnak érvénytelenné, törléskor pedig csak a törölt elem iterátora lesz érvénytelen.

Érdemes megjegyezni, hogy a listakezelő műveletek kivételével a lista programozói felülete ugyanazokat a tagfüggvényeket tartalmazza, mint a kettősvégű sor. Így egy listát használó program a sablontípus **deque**-ra való lecserélésével általában továbbra is futóképes marad, bár az algoritmusok hívása beleszólhat ebbe.

A **list** típus egy sor algoritmust tagfüggvényként valósít meg, melyek kihasználják a kétirányú lista speciális lehetőségeit, így hatékonyabb és biztonságosabb megoldást adnak az általános algoritmusoknál. A listakezelő tagfüggvényeket az alábbi táblázatban foglaltuk össze:

Tagfüggvényhívás	Leírás
<i>lst.merge(másLista)</i> <i>lst.merge(move(másLista))</i>	a < művelet felhasználásával összefésüli az <i>lst</i> és a <i>másLista</i> tartalmát, amennyiben azok növekvő sorrendben rendezettek ,
<i>lst.merge(másLista, Hasonlító)</i> <i>lst.merge(move(másLista), Hasonlító)</i>	a <i>Hasonlító</i> függvényobjektum segítségével összefésüli az <i>lst</i> és a <i>másLista</i> tartalmát, amennyiben azok növekvő sorrendben rendezettek ,
<i>lst.splice(iter, másLista)</i> <i>lst.splice(iter, move(másLista))</i>	a <i>másLista</i> eleminek átmásolása/átmozgatása az <i>lst</i> lista <i>iter</i> pozíciója elé,
<i>lst.splice(iter, másLista, másIter)</i> <i>lst.splice(iter, move(másLista), másIter)</i>	a <i>másLista</i> egyetlen (<i>másIter</i>) elemének átmásolása/átmozgatása az <i>lst</i> lista <i>iter</i> pozíciója elé,
<i>lst.splice(iter, másLista, itb, ite)</i> <i>lst.splice(iter, move(másLista), itb, ite)</i>	a <i>másLista</i> [<i>itb</i> , <i>ite</i>) elemeinek átmásolása/átmozgatása az <i>lst</i> lista <i>iter</i> pozíciója elé,
<i>lst.remove(adat)</i>	minden <i>adat</i> értékű elem törlése az <i>lst</i> listából,
<i>lst.remove_if(unPred)</i>	minden olyan elem törlése az <i>lst</i> listából, amelyre a megadott <i>unPred</i> függvényobjektum igaz értékkel tér vissza,
<i>lst.reverse()</i>	a lista elemsorrendjének megfordítása,
<i>lst.sort()</i>	a lista elemeinek rendezése növekvő sorrendbe a < művelet segítségével,
<i>lst.sort(Hasonlító)</i>	a lista elemeinek rendezése növekvő sorrendbe a <i>Hasonlító</i> függvényobjektum felhasználásával,
<i>lst.unique()</i>	a == művelet alkalmazásával törli a listából az egymást követő ismétlődő elemeket, és csak az első marad meg közülük,
<i>lst.unique(binPred)</i>	a <i>binPred</i> függvényobjektum hívásával törli a listából az egymást követő ismétlődő elemeket, melyek közül csak az első marad meg.

A **Hasonlító** függvényt a

bool Hasonlító(const AdatTípus& a, const AdatTípus& a)

formával kompatibilis módon kell előállítanunk. A függvény **true** értékkel tér vissza, ha $a < b$. Az **unáris predikátum** **true** értékkel mondja meg, hogy mely elemre kell a műveletet elvégezni. A rá vonatkozó formai előírás:

bool unPred(const AdatTípus& a).

A **bináris predikátum** igaz értékkel jelzi, ha az argumentumai megegyeznek. A *binPred* prototípusa:

bool binPred(const AdatTípus& a, const AdatTípus& a).

A listaműveletek használatára nézzük az alábbi példaprogramot!

```
#include <iostream>
#include <list>
#include <iterator>
#include <ctime>
#include <cstdlib>
using namespace std;

void Kiir(const string& szoveg, const list<int>& lista) {
    cout << szoveg << ":\t";
    copy(begin(lista), end(lista), ostream_iterator<int>(cout, " "));
    cout<< endl;
}

int main() {
    list<int> lista1 { 7, 2, 1, 3 };
    srand(time(nullptr));
    for(int i = 0; i < 7; i++) {
        lista1.push_back(rand()%10); // elemek hozzáadása a lista1 végéhez
    }
    Kiir("a lista1", lista1);

    lista1.sort(); // rendezés
    Kiir("a rendezett lista1", lista1);

    lista1.unique(); // az elemek kiegyelése
    Kiir("a kiegyelt lista1", lista1);

    list<int> lista2;
    for(int i = 0; i < 7; i++) {
        lista2.push_back(rand()%10);
    }
    Kiir("a lista2", lista2);

    lista2.sort();
    Kiir("a rendezett lista2", lista2);
    lista2.merge(lista1); // a listák összefésülése
    Kiir("az összefesult listak", lista2);
    return 0;
}
```

```
a lista1:      7 2 1 3 7 3 3 6 0 0
a rendezett lista1: 0 0 1 2 3 3 3 6 7 7
a kiegyelt lista1: 0 1 2 3 6 7
a lista2:      4 7 3 0 4 1 2
a rendezett lista2: 0 1 2 3 4 4 7
az összefesult listak: 0 0 1 1 2 2 3 3 4 4 6 7 7
```

2.4.4.5 forward_list

A **forward_list** egy egyszerűen, előrehaladó láncolással kialakított lista. Minden elemből csak a rákövetkező elemre van hivatkozás, ezért a műveletek egy része eltér a szokásostól, amit egy **_after** (után) utótag is jelez (*insert_after()*, *emplace_after()*, *erase_after()*, *splice_after()*).

Az előrehaladó lista rendelkezik a **list** osztálysablonnál bemutatott listaműveletekkel, és a soros konténerekre jellemző tagfüggvényének egy részével. Az alábbi példában néhány megoldást mutatunk ezek alkalmazására. Például, egy meglévő előrehaladó lista végére, csak úgy fűzhetünk új elemet (*push_back*), ha végigmegyünk az egész listán.

```
#include <forward_list>
#include <iostream>
using namespace std;
```

```

void push_back(forward_list<int> &fl, int adat) {
    auto vege_elott = fl.before_begin();
    for (const auto& e : fl)
        vege_elott++;
    fl.insert_after(vege_elott, adat);
}

int main() {
    forward_list<int> flista { 1, 3, 2, 7};
    for (int e : flista)
        cout << e << " "; cout << endl;
    flista.insert_after(flista.before_begin(), 5); // az elejére
    flista.push_front(9); // az előző elé
    push_back(flista, 11); // a végére
    for (int e : flista)
        cout << e << " "; cout << endl;

    forward_list<double> flista2; // üres
    // feltöltés az elemek hozzáfűzésével
    auto iter = flista2.before_begin();
    for(int i = 0; i < 10; ++i)
        iter = flista2.insert_after(iter, i);
    for (double e : flista2)
        cout << e << " "; cout << endl;
    return 0;
}

```

```

1 3 2 7
9 5 1 3 2 7 11
0 1 2 3 4 5 6 7 8 9

```

2.4.4.6 A soros konténerek összehasonlítása

A soros konténerek általános összevetését az előzőek folyamán már több esetben is elvégeztük. Most csupán a programozási feladatok megoldásához kívánunk további segítséget nyújtani azzal, hogy a soros konténerek időbeli viselkedését összevetjük, néhány gyakori művelet esetén.

A soros konténerek néhány jellemvonása:

- a vektor feltöltése sokkal gyorsabb, ha a helyfoglalást előzőleg elvégeztük,
- a lista nagyon lassú, amikor a teljes konténert be kell járni,
- kisméretű adatelemekkel a vektor és a kettősvégű sor mindig gyorsabb, mint a lista,
- a vektor és a kettősvégű sor lassú, amikor nagyméretű adatokat kell másolnia,
- a lista gyors nagyméretű adatelemek kezelésekor,
- a kettősvégű sor hatékonyabb a vektornál, amikor tetszőleges pozícióba kell adatokat beszúrni (hisz a *deque* elejének eléréséhez konstans idő szükséges).

Javaslatok a soros konténerek kiválasztásához:

- Beszúrás tetszőleges pozícióba, a rendezettség megtartásával: *vector*, *deque*.
- Lineáris keresés: *array*, *vector*, *deque*.
- Beszúrás tetszőleges pozícióba, törlés tetszőleges pozícióból:
 - kisméretű elemek: *vector*.
 - nagyméretű elemek: *list*, *forward_list* (ha nem kell keresni benne).
- Beszúrás a konténer elejére: *deque*, *list*, *forward_list*.
- Nem C++ alaptípusok esetén, ha nem a keresés és a sok változtatás a fő cél: *list*.
- Nem C++ alaptípusok esetén, ha a keresés és a sok változtatás a fő cél: *vector*, *deque*.

2.4.5 Programozás asszociatív konténerekkel

Az asszociatív konténerekben az adatok tárolása, elérése egy kulcs alapján történik. A kulcs felhasználási módja szerint két csoportra oszthatjuk a tárolókat.

Amikor az adatok tárolása a kulcs alapján rendezett sorrendben történik, rendezett konténerekről beszélünk (**set**, **multiset**, **map**, **multimap**). A rendezett tárolók a sorrendet egy *Compare* összehasonlító függvényobjektum felhasználásával alakítják ki, melynek alapértéke a **less<KulcsTípus>** (kisebb). A másik csoportba tartozó (rendezetlen) konténerek az adatokat a kulcs alapján képzett hasító táblában tárolják (**unordered_set**, **unordered_multiset**, **unordered_map**, **unordered_multimap**). Ehhez a tároláshoz szükséges hasító függvényt (*Hash*) és a kulcsok azonosságát vizsgáló függvényobjektumot (*KeyEqual*) szintén meg kell adnunk az osztálysablonok példányosításakor.

Egy másfajta csoportosításra ad lehetőséget a tárolt adatok vizsgálata. A halmaz tárolókban maguk a kulcsok tárolódnak egyetlen (**set**, **unordered_set**), vagy több példányban (**multiset**, **unordered_multiset**). Ezzel szemben az asszociatív tömbök (szótárak) elemeiben a kulcshoz egy adat is tartozik, adatpárokban (**pair**) összefogva. A **map** és **unordered_map** esetén a kulcsok egyetlen példányban szerepelhetnek a tárolókban, míg a **multimap** és az **unordered_multimap** konténerek megengedik a kulcsok ismétlődését.

A következőkben a második csoportosítás alapján vesszük sorra a tárolókat, végül pedig összefoglaljuk a rendezetlen tárolók hasító táblájával kapcsolatos tagfüggvényeket.

2.4.5.1 Rendezett halmaz konténerek (set, multiset)

Az asszociatív konténerek konstruktorai további lehetőségeket is tartalmaznak a 2.4.3. részben elmondottakhoz képest. Mint tudjuk az osztálysablonok példányosítása során típusokat adunk meg argumentumként, és a példányosítás eredménye egy osztály- vagy függvénytípus. Ha az argumentumként megadott típus egy hagyományos függvénymutató típusa, akkor magát a függvényt a konstruktor hívásakor kell megadnunk, teljessé téve ezzel az objektum létrehozását. Jól szemlélteti az elmondottakat az alábbi példa:

```
#include <set>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

class Auto {
public:
    Auto(const string& tip = "") : tipus(tip) {}
    bool operator<(const Auto& másik) const {
        return tipus < másik.tipus;
    }
    friend ostream& operator<<(ostream& os, const Auto& a) {
        os << a.tipus;
        return os;
    }
    string Tipus() const { return tipus; }
private:
    string tipus;
};

bool AutoHasonlit(const Auto& bal, const Auto& jobb) {
    return bal < jobb;
}
```

```
int main() {
    set<Auto, bool(*)(const Auto&, const Auto&)> ahalmaz(AutotHasonlit);
    ahalmaz.emplace("Volvo");
    ahalmaz.emplace("Scania");
    ahalmaz.emplace("Renault");
    copy(begin(ahalmaz), end(ahalmaz), ostream_iterator<Auto>(cout, "\n"));
}
```

A fenti *AutotHasonlit()* függvény típusa helyett az *Auto* típussal példányosíthatjuk a *less* osztálysablon (*<functional>*), amely aztán a megadott típushoz tartozó *operator<* műveletet alkalmazza:

```
set<Auto, less<Auto>> ahalmaz;
```

Mivel ebben az anyagban az STL használatával kapcsolatos alapismeretek közvetítjük, a példáinkban maradunk a konténersablonok minél teljesebb paraméterezésénél.

Ha az adattípusunk nem rendelkezik megfelelő relációs művelettel, akkor egy új függvényobjektum bevezetésével orvosolhatjuk a problémát:

```
#include <set>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

class Auto {
public:
    Auto(const string& tip = "") : tipus(tip) {}
    friend ostream& operator<<(ostream& os, const Auto& a) {
        os << a.tipus;
        return os;
    }
    string Tipus() const { return tipus; }
private:
    string tipus;
};

struct Kisebb {
    bool operator()(const Auto& bal, const Auto& jobb) const {
        return bal.Tipus() < jobb.Tipus();
    }
};

int main() {
    set<Auto, Kisebb> ahalmaz;
    ahalmaz.insert(string("Volvo"));
    ahalmaz.insert(string("Scania"));
    ahalmaz.insert(string("Renault"));
    copy(begin(ahalmaz), end(ahalmaz), ostream_iterator<Auto>(cout, "\n"));
}
```

A továbbiakban áttekintjük a kulcs alapján kereséseket megvalósító tagfüggvényeket. Megjegyezzük, hogy a *set* és a *multiset* konténerek interfészei teljesen megegyeznek, egyetlen kis eltéréstől, a *count()* tagfüggvény visszatérési értékétől eltekintve.

Tagfüggvényhívás	Leírás
<i>n = a.count(kulcs)</i>	a <i>size_type</i> típusú visszatérési értékben megadja, hogy az <i>a</i> rendezett konténerben hány elem rendelkezik az adott <i>kulccsal</i> (a <i>set</i> esetén ez az érték 0 vagy 1),
<i>iter = a.find(kulcs)</i>	sikeres esetben az adott <i>kulcsú</i> elem iterátorával tér vissza, míg sikertelen esetben az <i>a.end()</i> értékkel,
<i>pair<itb, ite> p = a.equal_range(kulcs)</i>	az adott <i>kulcsú</i> elemeket tartalmazó iterátor-tartomány [<i>itb</i> , <i>ite</i>) lekérdezése – sikertelen esetben mindkét visszaadott iterátor értéke <i>a.end()</i> ,

Tagfüggvényhívás <i>(folytatás)</i>	Leírás
<code>iter = a.lower_bound(kulcs)</code>	sikeres esetben egy olyan elem iterátorával tér vissza, amelyik nem kisebb , mint a megadott <i>kulcs</i> , míg sikertelen esetben az <code>a.end()</code> a függvényérték,
<code>iter = a.upper_bound(kulcs)</code>	sikeres esetben egy olyan elem iterátorával tér vissza, amelyik nagyobb , mint a megadott <i>kulcs</i> , míg sikertelen esetben az <code>a.end()</code> a függvényérték.

Megjegyezzük, hogy rendezett asszociatív konténerek esetén az `equal_range()` függvény által visszaadott iterátorokat a `lower_bound()` és az `upper_bound()` függvények is szolgáltatják. Elemismétlő halmaz esetén az alábbi példaprogram szemlélteti a kereső tagfüggvények használatát:

```
#include <set>
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main() {
    srand(unsigned(time(nullptr)));
    multiset<int> szamok {7,7};
    int n = rand()%10 + 10;
    for (int i=0; i<n; i++)
        szamok.insert(rand()%12);
    for (int e : szamok)
        cout << e << " "; cout << endl;
    if (szamok.find(7) != end(szamok)) { // tartalmazza a 7-et?
        cout << szamok.count(7) << endl;
        auto par = szamok.equal_range(7);
        cout << *par.first << "\t" << *szamok.lower_bound(7) << endl;
        cout << *par.second << "\t" << *szamok.upper_bound(7) << endl;
        for (auto p=par.first; p!=par.second; p++)
            cout << *p << "\t";
        cout << endl;
    }
    return 0;
}
```

```
0 0 0 1 2 4 5 5 5 6 7 7 7 8 9 9 9 10 11
3
7      7
8      8
7      7      7
```

2.4.5.2 Rendezett asszociatív tömb (szótár) konténerek

A `map` és a `multimap` konténerekben adatpárok tárolódnak, melyek első (*first*) tagja maga a kulcs, míg a második (*second*) tagja az adat. A szokásos konténerműveleteken túlmenően a minden változtatás nélkül alkalmazhatók a halmazoknál bemutatott keresési tagfüggvények (`find()`, `count()`, `lower_bound()` stb.). Az első példában egyedi kulcsokkal hozunk létre egy konténert, és bemutatjuk az adatpárok hozzáadásának, lekérdezésének és módosításának különböző módjait:

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

typedef map<string, string> Szotar;
```

```

void KiirMap(const Szotar& szotar) {
    cout << "- . -" << endl;
    cout << "Szotar:\n";
    for (const auto& par : szotar) {
        cout << par.first << "\t" << par.second << "\n";
    }
    cout << "- . -" << endl;
}

int main() {
    Szotar szotar { {"cat", "macska"}, {"dog", "kutya"} };
    szotar.insert(make_pair("gopher", "horcsog"));
    szotar.insert(pair<string, string>("mouse", "eger"));
    szotar["eagle"] = "sas";
    cout << szotar["rat"] << endl;
    KiirMap(szotar);
    if (szotar.find("rat") != end(szotar)) {
        cout << szotar["eagle"] << endl;
    }
    szotar["rat"] = "patkany";
    auto ret = szotar.insert(make_pair("cat", "cica"));
    if (!ret.second) {
        cout << ret.first->first << " kulcs mar szerepel a szotarban." << endl;
    }
    KiirMap(szotar);
    return 0;
}

```

```

- . -
Szotar:
cat      macska
dog      kutya
eagle    sas
gopher   horcsog
mouse    eger
rat
- . -
sas
cat kulcs mar szerepel a szotarban.
- . -
Szotar:
cat      macska
dog      kutya
eagle    sas
gopher   horcsog
mouse    eger
rat      patkany
- . -

```

Felhívjuk a figyelmet, hogy az indexelés műveletét a kulcsokkal csak a **map** (**unordered_map**) konténereknél használhatjuk. Ha az indexelés során megadott kulcs nem létezik, akkor az adott kulcsú elem mindig létrejön, még akkor is, ha lekérdezés volt a célunk. Ekkor az adat a típusának megfelelő alapértékkel inicializálódik.

Indexelés nélkül egy adatpárt úgy módosíthatunk, hogy először töröljük a bejegyzést, majd hozzáadjuk az új tartalmat a szótárhoz:

```

auto ret = szotar.insert(make_pair("cat", "cica"));
if (!ret.second) {
    szotar.erase("cat"); // a cat kulcsú elem eltávolítása
    szotar.insert(make_pair("cat", "cica"));
}

```

A következő telefonregiszter példával szemléltetjük a **multiset** használatának fogásait a fentiekben ismertetett tagfüggvények alkalmazásával:

```

#include<iostream>
#include<map>
#include<set>
#include <string>
using namespace std;

typedef multimap<string, unsigned> Telefon;
typedef pair<string, unsigned> Bejegyzes;

void MMKiiir(const Telefon& telreg) {
    cout << "Bejegyzések száma: " << telreg.size() << endl;
    Telefon::const_iterator iter = begin(telreg);
    while(iter != end(telreg)) {
        cout << "Név= " << iter->first << "\tTel = " << iter->second << endl;
        iter++;
    }
    cout << endl;
}

int main() {
    Telefon telreg { {"Szende", 1223}, {"Morgo", 1122}, {"Tudor", 1002} };

    telreg.insert(Bejegyzes("Szundi", 1234));
    telreg.insert(Bejegyzes("Hapci", 4444));
    telreg.insert(Bejegyzes("Szundi", 3690));
    telreg.insert(Bejegyzes("Kuka", 5673));
    telreg.insert(Bejegyzes("Hapci", 9999));
    telreg.insert(Bejegyzes("Vidor", 4532));
    MMKiiir(telreg);

    set<string> nevek; // a nevek összegyűjtése
    for (auto par : telreg)
        nevek.insert(par.first);

    cout<<"A nevek előfordulása:"<<endl;
    for (auto e : nevek)
        cout << e << "\ttelefonszámainak száma: " << telreg.count(e) << endl;

    cout << "\nSzundi számainak kidobása:" << endl;
    auto iterpar = telreg.equal_range("Szundi");
    for(auto iter = iterpar.first; iter != iterpar.second; ++iter)
        cout << "Név= " << iter->first << "\tTel = " << iter->second << endl;
    telreg.erase(iterpar.first, iterpar.second);
    cout << endl;
    MMKiiir(telreg);
    return 0;
}

```

```

Bejegyzések száma: 9
Nev= Hapci      Tel = 4444
Nev= Hapci      Tel = 9999
Nev= Kuka       Tel = 5673
Nev= Morgo      Tel = 1122
Nev= Szende     Tel = 1223
Nev= Szundi     Tel = 1234
Nev= Szundi     Tel = 3690
Nev= Tudor      Tel = 1002
Nev= Vidor      Tel = 4532

A nevek előfordulása:
Hapci telefonszámának száma: 2
Kuka  telefonszámának száma: 1
Morgo telefonszámának száma: 1
Szende telefonszámának száma: 1
Szundi telefonszámának száma: 2
Tudor telefonszámának száma: 1
Vidor telefonszámának száma: 1

Szundi számának kidobása:
Nev= Szundi     Tel = 1234
Nev= Szundi     Tel = 3690

Bejegyzések száma: 7
Nev= Hapci      Tel = 4444
Nev= Hapci      Tel = 9999
Nev= Kuka       Tel = 5673
Nev= Morgo      Tel = 1122
Nev= Szende     Tel = 1223
Nev= Tudor      Tel = 1002
Nev= Vidor      Tel = 4532

```

2.4.5.3 Rendezetlen halmaz és asszociatív tömb (szótár) konténerek

A 2.4.5.1. rész bevezető példájának rendezetlen halmazra való átírásával láthatjuk azokat a követelményeket, amelyek teljesítésével saját típusainkat *unordered_set*, illetve *unordered_multiset* sablonokban tárolhatjuk.

```

#include <unordered_set>
#include <string>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

class Auto {
public:
    Auto(const string& tip = "") : tipus(tip) {}
    bool operator==(const Auto& másik) const {
        return tipus == másik.tipus;
    }
    friend ostream& operator<<(ostream& os, const Auto& a) {
        os << a.tipus;
        return os;
    }
    const string& Tipus() const { return tipus; }
private:
    string tipus;
};

struct AutoHash {
    size_t operator()(const Auto& a) const {
        return hash<string>()(a.Tipus());
    }
};

```

```
int main() {
    unordered_set<Auto, AutoHash, equal_to<Auto>> aalmaz;
    aalmaz.insert(string("Volvo"));
    aalmaz.insert(string("Scania"));
    aalmaz.insert(string("Renault"));
    copy(begin(aalmaz), end(aalmaz), ostream_iterator<Auto>(cout, "\n"));
}
```

Saját típus esetén szükségünk van egy **hasító** és egy **azonosságot vizsgáló függvényobjektumra**, és ugyanez igaz a rendezetlen szótárak esetén. Felhívjuk a figyelmet, hogy a rendezetlen halmazokban és szótárakban a tárolt adatok sorrendje teljes mértékben implementációfüggő, így nem szabad a tárolási sorrendre építeni a megoldási módszereinket.

Sokkal kellemesebb helyzet, ha a programunkban C++ alaptípusú vagy a string típusú kulcsot használunk. Ekkor a rendezett asszociatív konténerekkel működő programunk kis módosítással rendezetlen halmazt vagy szótárt használó programmá alakítható. Jogos a kérdés, hogy miért tennénk ilyet?

A rendezett tárolókban a rendezettség fenntartása igen időigényes. Ez a rendezettség azonban azt is biztosítja, hogy az STL algoritmusok mindegyikét használjuk, például a halmazműveleteket is. Amennyiben ezekre nincs szükségünk, és gyorsabb programfutást szeretnénk, javasolt a rendezetlen asszociatív konténerekre való átállás. Nézzünk néhány fontos megjegyzést ezzel kapcsolatban:

- A rendezetlen tárolóknak nincs *lower_bound()* és *upper_bound()* tagfüggvényük.
- Az *equal_range()* függvény által visszaadott iterátorpár ez első és az utolsó, adott kulccsal rendelkező elemre hivatkozik.
- A rendezetlen asszociatív tárolókat csak az **==** és a **!=** műveletekkel hasonlíthatjuk össze.
- A rendezett tárolókhoz kétirányú iterátorok tartoznak, míg a rendezetlenekhez előrehaladó (egyirányú) iterátorok.
- Míg a rendezett tárolókat a *key_comp* és a *value_comp* függvényobjektumok jellemzik, addig a rendezetleneket a *hash_function* és a *key_eq* funktorok.

Ugyancsak különböznek a két konténercsoport lekérdezhető ún. megfigyelő függvényei, melyeket egy rövid programmal szemléltetünk:

```
#include <unordered_set>
#include <set>
#include <iostream>
using namespace std;

int main() {
    set<double> rset { 1, 2, 3, 4, 5 };
    unordered_set<double> uset { 1, 2, 3, 4, 5 };

    set<double>::key_compare kcmp = rset.key_comp();
    set<double>::value_compare vcmp = rset.value_comp();
    unordered_set<double>::hasher hashfv = uset.hash_function();
    unordered_set<double>::key_equal keq = uset.key_eq();

    cout << kcmp(12, 23) << endl; // 1
    cout << vcmp(12, 23) << endl; // 1
    cout << hashfv(1223) << endl; // 3750804164
    cout << keq(12, 23) << endl; // 0

    return 0;
}
```

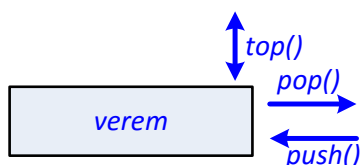
A rendezetlen tárolók további tagfüggvényei lehetőséget biztosítanak arra, hogy információkat szerezzünk a hasítóábrák szerkezetéről (*begin(n)*, *end(n)*, *bucket_count()*, *max_bucket_count()*, *bucket_size(n)*, *bucket(kulcs)*, *load_factor()*, *max_load_factor()*), de akár a tábla ismételt felépítését is kérhetjük (*rehash()*, *reserve()*).

2.4.6 Programozás konténer adapterekkel

A konténer adapter osztályoksablonok a **verem**, a **sor** és a **prioritásos sor** elméleti adatstruktúráknak megfelelő interfészt képeznek a **vector**, a **deque** vagy a **list** soros konténerek valamelyikén.

2.4.6.1 A verem adatstruktúra

A „last-in, first-out, LIFO” működésű **stack** (<stack.h>) egyaránt adaptálható **vector**, **list** és **deque** konténerekre építve. A veremre jellemző, hogy az elemeihez csak az egyik végén – a verem tetején, *top* – férhetünk hozzá.



Az **stack<típus>** osztállyal létrehozott **verem** objektum tagfüggvényhívásait és műveleteit táblázatban foglaltuk össze:

Tagfüggvényhívások	Leírás
verem.push(adat)	az <i>adat</i> bemásolása a verem tetejére,
verem.push(move(adat))	az <i>adat</i> áthelyezése (<i>move</i>) a verem tetejére,
verem.emplace(argumentumok)	új elem elhelyezése a verem tetején, és inicializálása az argumentumoknak megfelelő konstruktor hívásával,
verem.top()	megadja a verem felső elemének referenciáját vagy konstans referenciáját,
verem.pop()	a verem felső elemének levétele,
verem.empty()	true értékkel jelzi, ha a verem üres,
verem.size()	visszatér a veremben lévő elemek számával,
verem.swap(másik verem)	felcseréli a két verem tartalmát,
swap(verem1, verem2)	a <i>swap</i> algoritmus veremekre specializált változata,
==, !=, <, <=, >, >=	két verem lexikografikus összehasonlítása,
verem1 = verem2	a <i>verem2</i> tartalmának átmásolása a <i>verem1</i> -be,
verem1 = move(verem2)	a <i>verem2</i> tartalmának áthelyezése a <i>verem1</i> -be.

Az alábbi példaprogramban adott számrendszerbe (*alap*) való átváltásra használjuk a **vector** konténerre épített vermet:

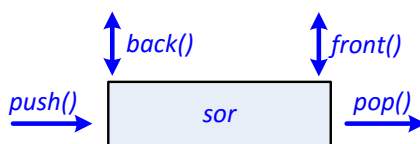
```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

int main() {
    int szam=20151002, alap=16;
    stack<int, vector<int>> verem;
    do {
        verem.push(szam % alap);
        szam /= alap;
    } while (szam>0);

    while (!verem.empty()) {
        szam = verem.top();
        verem.pop();
        cout << (szam < 10 ? char(szam + '0') : char(szam + 'A' - 10)); // 1337ADA
    }
    return 0;
}
```

2.4.6.2 A sor adatstruktúra

A „first-in, first-out, FIFO” működésű sort (**queue**) (`<queue.h>`) **list** vagy **deque** tárolókból adaptálhatjuk. A sor alapl műveletei mindegyikéhez (elem hozzáadása a sor végéhez, elem kivétele a sor elején, az első és az utolsó elem elérése) konstans ($O(1)$) idő szükséges.



Az **queue<típus>** osztállyal létrehozott **sor** objektum tagfüggvényhívásait és műveleteit az alábbi táblázat tartalmazza:

Tagfüggvényhívások	Leírás
<code>sor.push(adat)</code>	az <i>adat</i> bemásolása a sor végére,
<code>sor.push(move(adat))</code>	az <i>adat</i> áthelyezése (<i>move</i>) a sor végére,
<code>sor.emplace(argumentumok)</code>	új elem elhelyezése a sor végén, és inicializálása az argumentumoknak megfelelő konstruktor hívásával,
<code>sor.back()</code>	megadja a sor utolsó elemének referenciáját vagy konstans referenciáját,
<code>sor.front()</code>	megadja a sor első elemének referenciáját vagy konstans referenciáját,
<code>sor.pop()</code>	a sor első elemének kivétele,
<code>sor.empty()</code>	true értékkel jelzi, ha a sor üres,
<code>sor.size()</code>	visszatér a sorban tárolt elemek számával,
<code>sor.swap(másik sor)</code>	felcseréli a két sor tartalmát,
<code>swap(sor1, sor2)</code>	a <i>swap</i> algoritmus sorokra specializált változata,
<code>==, !=, <, <=, >, >=</code>	két sor lexikografikus összehasonlítása,
<code>sor1 = sor2</code>	a <i>sor2</i> tartalmának átmásolása a <i>sor1</i> -be,
<code>sor1 = move(sor2)</code>	a <i>sor2</i> tartalmának áthelyezése a <i>sor1</i> -be.

Jól használható a sor adatstruktúra a program futása során keletkező részadatok tárolására, mint ahogy ezt az egész számok törzstényezőkre való bontása során tesszük:

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    unsigned n;
    cout << "N = "; cin >> n; cin.get();
    queue<unsigned> sor;
    unsigned n0 = n;

    for (unsigned i = 2; i <= n; i++) {
        while (n % i == 0) {
            sor.push(i);
            n /= i;
        }
    }

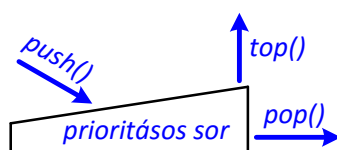
    cout << n0 << " = ";
    while (!sor.empty()) {
        cout << sor.front();
        if (sor.size() > 1) cout << "*";
        sor.pop();
    }
    cout << endl;
    return 0;
}
```

$$N = 2015$$

$$2015 = 5 \cdot 13 \cdot 31$$

2.4.6.3 A prioritásos adatstruktúra

A prioritásos sor (**`priority_queue`**) olyan tároló, melynek elemei különböző prioritással rendelkeznek. Az elsőbbséget egy rendezési előírás (összehasonlító függvényobjektum) definiálja, melyet a sablon harmadik paraméterében kell megadnunk. (Alapértelmezés szerint az **`operator<`** műveletet használja a prioritásos sor.) A sor elején mindig az egyik legnagyobb prioritású elem helyezkedik el. A prioritásos sor a **`vector`** és a **`deque`** tárolókból egyaránt adaptálható. (A prioritásos sor működését szokás a *BIFO* – *best in first out* betűkkel jellemezni, ahol a legjobb a legnagyobb prioritású elemek egyikét jelenti.)



Az **`priority_queue<tipus>`** osztállyal létrehozott *prisor* objektum tagfüggvényhívásait és műveleteit az alábbi táblázatban foglaltuk össze:

Tagfüggvényhívások	Leírás
<i>prisor</i> . <code>push</code> (adat)	az <i>adat</i> bevitele a prioritásos sorba,
<i>prisor</i> . <code>push</code> (move(adat))	az <i>adat</i> áthelyezése (<i>move</i>) a prioritásos sorba,
<i>prisor</i> . <code>emplace</code> (argumentumok)	új elem behelyezése a prioritásos sorba, és inicializálása az argumentumoknak megfelelő konstruktor hívásával,
<i>prisor</i> . <code>top</code> ()	megadja a prioritásos sor első elemének konstans referenciáját, így az elem csak kiolvasható, és nem módosítható,
<i>prisor</i> . <code>pop</code> ()	a prioritásos sorba első elemének levétele,
<i>prisor</i> . <code>empty</code> ()	<code>true</code> értékkel jelzi, ha a prioritásos sorba üres,
<i>prisor</i> . <code>size</code> ()	visszatér a prioritásos sorban lévő elemek számával,
verem. <code>swap</code> (másik <i>prisor</i>)	felcseréli a két prioritásos sor tartalmát,
<code>swap</code> (<i>prisor1</i> , <i>prisor2</i>)	a <i>swap</i> algoritmus prioritásos sorokra specializált változata,
<i>prisor1</i> = <i>prisor2</i>	a <i>prisor2</i> tartalmának átmásolása a <i>prisor1</i> -be,
<i>prisor1</i> = <code>move</code> (<i>prisor2</i>)	a <i>prisor2</i> tartalmának áthelyezése a <i>prisor1</i> -be.

Az alábbi példában a C++ szabványos **`complex`** típusát tároljuk prioritásos sorban. Az összehasonlító függvényobjektumban azt a komplex számot tekintjük nagyobbknak, melynek nagyobb az abszolút értéke:

```
#include <iostream>
#include <queue>
#include <complex>
using namespace std;

typedef complex<double> komplex;

struct KomplexKisebb {
    bool operator()(const komplex& x, const komplex& y) const {
        return abs(x) < abs(y);
    }
};

int main() {
    complex<double> cv[] = {{1,2}, {2, 3}, {5, 3}, {3, 2}};
    priority_queue <komplex, vector<komplex>, KomplexKisebb>
        prisor(begin(cv), end(cv));

    prisor.push(komplex(3,7));
```



```

while (!prisor.empty()) {
    cout << prisor.top() << endl;
    prisor.pop();
}
return 0;
}

```

```

(3,7)
(5,3)
(2,3)
(3,2)
(1,2)

```

A lexikografikus összehasonlítás (a valós rész dönt, de ha egyenlőek a képzetes rész értéke a meghatározó) megvalósításával egészen más sorrendet kapunk:

```

struct KomplexKisebb {
    bool operator()(const komplex& x, const komplex& y) const {
        if (x.real() != y.real())
            return x.real() < y.real();
        else
            return x.imag() < y.imag();
    }
};

```

```

(5,3)
(3,7)
(3,2)
(2,3)
(1,2)

```

2.5 Ismerkedés az algoritmusokkal

Az algoritmusok *<algorithm>* kiszélesítik a konténerek felhasználhatóságának lehetőségeit. Mivel függvénysablonok formájában állnak a rendelkezésünkre, típusok széles skálájához használhatjuk őket. Az algoritmusok többségét iterátorokkal kapcsoljuk a konténerekhez, és ha iterátort adnak vissza, akkor annak típusa megegyezik a bemeneti iterátor(ok) típusával. Már említettük, ha egy konténer valamely algoritmust saját tagfüggvénnyel valósít meg, akkor azt javasolt használni, mivel az hatékonyabb és biztonságosabb.

Az algoritmusok leírásánál – a jobb áttekinthetőség érdekében – az iterátorokra és a függvényobjektumokra rövidítéseket használunk, az alábbiak szerint:

input iterátorok:	<i>itb, ite, it1, it2, iter</i>	
output iterátorok:	<i>oitb, oit1, oit2, oiter</i>	
előrehaladó iterátorok:	<i>fitb, fite, fit1, fit2, fiter</i>	
kétirányú iterátorok:	<i>bitb, bite, bite2, biter</i>	
tetszőleges elérési iterátorok:	<i>ritb, rite, riter</i>	
egyoperandusos művelet:	<i>műv</i>	<i>fvtípus műv(const típus &a);</i>
kétooperandusos művelet:	<i>binműv</i>	<i>fvtípus binműv(const típus1 &a, const típus2 &b);</i>
egyoperandusos predikátum:	<i>pred</i>	<i>bool pred(const típus &a);</i>
kétooperandusos predikátum:	<i>binpred</i>	<i>bool binpred(const típus1 &a, const típus2 &b);</i>
összehasonlító függvény:	<i>compfv</i>	<i>bool compfv(const típus1& a, const típus2& b);</i>

A függvényobjektumok esetén a paramétertípusokban a **const** és **&** előírások el is hagyhatók.

2.5.1 Az algoritmusok végrehajtási ideje

A konténerműveletek időigénye mellett a felhasznált algoritmusok időigénye együtt határozza meg az adott programrész futásidejét. Az algoritmusok végrehatásához szükséges időigényt a feldolgozandó adatsor elemeinek számával (n) jellemezhetjük:

Végrehajtási idő	Algoritmusok
$O(1)$	<i>swap(), iter_swap(),</i>
$O(\log(n))$	<i>lower_bound(), upper_bound(), equal_range(), binary_search(), push_heap(), pop_heap(),</i>
$O(n \cdot \log(n))$	<i>inplace_merge()</i> (legrosszabb esetben), <i>stable_partition()</i> (legrosszabb esetben), <i>sort(), stable_sort(), partial_sort(), partial_sort_copy(), sort_heap(),</i>
$O(n^2)$	<i>find_end(), find_first_of(), search(), search_n(),</i>
$O(n)$	minden más algoritmus.

2.5.2 Nem módosító algoritmusok

A nem módosító algoritmusok csak olvassák a konténer elemeit, nem rendezik át és nem változtatják meg azokat.

2.5.2.1 Adott művelet elvégzése az elemeken – *for_each()*

A *for_each()* algoritmus a megadott *[itb, ite)* tartomány minden elemén végrehajtja a megadott műveletet. A hívás a műveletet függvényobjektumként adja vissza.

```
auto fvobj = for_each(itb, ite, műv);
```

Az alábbi példában a műveletek megadásához függvényt, lambda-kifejezést és függvényobjektumot használunk:

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void Add12(int &elem) {
    elem += 12;
}

struct Osszeg {
    Osszeg() { osszeg = 0; }
    void operator()(int n) { osszeg += n; }
    int osszeg;
};

int main() {
    vector<int> szamok{2, 3, 5, 8, 13, 21, 34};
    cout << "Hivas előtt\t";
    for (auto e : szamok)
        cout << e << " "; cout << endl;
    // minden elem növelése 12-vel - függvény segítségével
    for_each(begin(szamok), end(szamok), Add12);
    cout << "12-vel valo noveles hivasa utan\t";
    for (auto e : szamok)
        cout << e << " "; cout << endl;
    // minden elem felezése - Lambda-kifejezés alkalmazásával
    for_each(begin(szamok), end(szamok), [](int &e){ e /= 2; });
    cout << "2-vel valo osztas hivasa utan\t";
    for (auto e : szamok)
        cout << e << " "; cout << endl;
    // az Osszeg::operator() hívása minden elemre
    Osszeg s = for_each(begin(szamok), end(szamok), Osszeg());
    cout << "az oszegzes hivasa utan\t";
    for (auto e : szamok)
        cout << e << " "; cout << endl;
    cout << "Elemosszeg: " << s.osszeg << endl;
    int osszeg = 0;
    // összegzés - Lambda-kifejezés alkalmazásával
    for_each(begin(szamok), end(szamok), [&osszeg](int e){ osszeg += e; });
    cout << "Elemosszeg: " << osszeg << endl;
    return 0;
}
```

```
Hivas előtt      2 3 5 8 13 21 34
12-vel valo noveles hivasa utan 14 15 17 20 25 33 46
2-vel valo osztas hivasa utan   7 7 8 10 12 16 23
az oszegzes hivasa utan 7 7 8 10 12 16 23
Elemosszeg: 83
Elemosszeg: 83
```

2.5.2.2 Elemek vizsgálata

Algoritmus hívása	Leírás
<code>bool b = all_of(itb, ite, pred)</code>	értéke true , ha a <code>pred(elem)</code> igaz minden elemre az <code>[itb, ite)</code> tartományban,
<code>bool b = any_of(itb, ite, pred)</code>	értéke true , ha a <code>pred(elem)</code> legalább egy elemre igaz az <code>[itb, ite)</code> tartományban,
<code>bool b = none_of(itb, ite, pred)</code>	értéke true , ha a <code>pred(elem)</code> egyetlen elemre sem igaz az <code>[itb, ite)</code> tartományban.

Az alábbi példában a `v` vektor elemeinek vizsgálata során különböző módon adjuk meg a predikátum-függvényt:

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

bool Oszthato5(unsigned e) {
    return e % 5;
}

int main() {
    vector<unsigned> v {23, 17, 63, 35, 29};
    cout << "A vektor elemei: ";
    copy(v.cbegin(), v.cend(), ostream_iterator<unsigned>(cout, " "));
    cout << endl;

    // Lambda-kifejezés a predikátum
    if (all_of(v.cbegin(), v.cend(), [](unsigned e){ return e % 2 == 1; })) {
        cout << "Minden elem paratlan" << endl;
    }

    // predikátum-függvény
    if (any_of(v.cbegin(), v.cend(), Oszthato5)) {
        cout << "Legalább egy elem oszthato 5-tel" << endl;
    }

    // összeállított függvényobjektum a predikátum
    if (none_of(v.cbegin(), v.cend(), bind(greater<unsigned>(),_1, 64))) {
        cout << "Egyetlen elem sem nagyobb 64-nel" << endl;
    }

    return 0;
}
```

2.5.2.3 Elemek számlálása –count()

Algoritmus hívása	Leírás
<code>n = count(itb, ite, adat)</code>	visszatér azon elemek számával az <code>[itb, ite)</code> tartományban, melyek értéke megegyezik az <code>adattal</code> ,
<code>n = count_if(itb, ite, pred)</code>	megadja azon elemek számát az <code>[itb, ite)</code> tartományban, melyekre a <code>pred(elem)</code> hívás igaz értéket ad.

Az egyenlőség vizsgálatánál alaphelyzetben az `operator==()` függvényt használja az algoritmus, kivéve, ha predikátum is szerepel a paraméterlistán. A predikátum **true** értékkel jelzi az azonosságot. Az alábbi példában megszámloljuk, hogy a `v` vektornak hány 7-es eleme, illetve hány 10-zel osztható eleme van:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
    vector<int> v { 7, 12, 20, 30, 7, 10, 50, 23 };
    int db7 = count(begin(v), end(v), 7);
    int db10oszt = count_if(begin(v), end(v), [](int e) {return e%10==0;});
    cout << "7 értéku elemek szama: " << db7 << endl;
    cout << "10-zel oszthato elemek szama: " << db10oszt << endl;
    return 0;
}
```

7 értéku elemek szama: 2
10-zel oszthato elemek szama: 4

2.5.2.4 Elemek keresése – a find() csoport

Algoritmus hívása	Leírás
<code>iter = find(itb, ite, adat)</code>	az <code>[itb, ite)</code> tartományban megkeresi az első olyan elemet, melynek értéke megegyezik az <code>adattal</code> (<code>*iter == adat</code>),
<code>iter = find_if(itb, ite, pred)</code>	az <code>[itb, ite)</code> tartományban megkeresi az első olyan elemet, melyre a <code>pred(*iter)</code> igaz,
<code>iter = find_if_not(itb, ite, pred)</code>	az <code>[itb, ite)</code> tartományban megkeresi az első olyan elemet, melyre a <code>pred(*iter)</code> hamis,
<code>fiter = find_first_of(fitb, fite, fitb2, fite2)</code>	a <code>[fitb, fite)</code> tartományban megkeresi az első olyan elemet, amely megtalálható a <code>[fitb2, fite2)</code> tartományban is,
<code>fiter = find_first_of(fitb, fite, fitb2, fite2, binpred)</code>	a <code>[fitb, fite)</code> tartományban megkeresi az első olyan elemet, amellyel és a <code>[fitb2, fite2)</code> tartomány valamelyik elemével paraméterezve a predikátumot, az igaz értékkel tér vissza,
<code>fiter = find_end(fitb, fite, fitb2, fite2)</code>	a <code>[fitb, fite)</code> tartományban megkeresi az utolsó olyan elemet, amelytől rendre megtalálható a <code>[fitb2, fite2)</code> tartomány minden eleme,
<code>fiter = find_end(fitb, fite, fitb2, fite2, binpred)</code>	a <code>[fitb, fite)</code> tartományban megkeresi az utolsó olyan elemet, amelytől kezdődő elemekkel és a <code>[fitb2, fite2)</code> tartomány megfelelő elemeivel paraméterezve a predikátumot, az igaz értékkel tér vissza,
<code>fiter = adjacent_find(fitb, fite)</code>	megkeresi az első olyan elemet, melynek értéke megegyezik a rákövetkező elem értékével (<code>*fiter == *(fiter+1)</code>),
<code>fiter = adjacent_find(fitb, fite, binpred)</code>	megkeresi az első olyan elemet az <code>[itb, ite)</code> tartományban, melyre a <code>binpred(*fiter, *(fiter+1))</code> hívás igaz értéket ad,

A keresési algoritmusok sikertelen esetben az `ite` iterátor értékével térnek vissza. Az *első példában a v vektorban megkeressük az 5 értékű elem összes előfordulását, és megjelenítjük az elemek indexeit:*

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> v {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9};
    cout << "A vektor tartalma:\n";
    for (vector<int>::size_type i = 0; i < v.size(); i++)
        cout << v.at(i) << " ";
    cout << "\nVarakozas az Enter lenyomasara..."; cin.ignore(80, '\n');
    cout << "\nAz 5 értéku elemek indexei:";
    auto p = begin(v);
```

```

while (p != end(v)) {
    p = find(p, end(v), 5);
    if (p != end(v)) {
        cout << distance(begin(v), p) << " ";
        p++;
    }
}
return 0;
}

```

*A vektor tartalma:
 3 1 4 1 5 9 2 6 5 3 5 9
 Varakozas az Enter Lenyomasara...
 Az 5 erteku elemek indexei:4 8 10*

A következő példában a v2 vektor elemeinek utolsó előfordulását keressük a v1-ben. Az összehasonlítás alapja a számjegyek összege:

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool SzamjegyOsszeg (int a, int b) {
    int osszeg1 = 0;
    while (a != 0) {
        osszeg1 += a % 10;
        a /= 10;
    }
    int osszeg2 = 0;
    while (b != 0) {
        osszeg2 += b % 10;
        b /= 10;
    }
    return osszeg1 == osszeg2;
}

int main() {
    vector<int> v1 {17, 23, 12, 52, 7, 8, 32, 102, 25, 9, 11};
    cout << "A v1 tartalma:\n";
    for (int e : v1)
        cout << e << " "; cout << endl;
    vector<int> v2 {23, 12, 7};
    cout << "A v2 tartalma:\n";
    for (int e : v2)
        cout << e << " "; cout << endl;
    auto p = find_end(begin(v1), end(v1), begin(v2), end(v2), SzamjegyOsszeg);
    if (p != end(v1))
        cout << "A v2 utolso elofordulasanak kezdointexe a v1-ben: "
            << distance(begin(v1), p);
    return 0;
}

```

*A v1 tartalma:
 17 23 12 52 7 8 32 102 25 9 11
 A v2 tartalma:
 23 12 7
 A v2 utolso elofordulasanak kezdointexe a v1-ben: 6*

Az utolsó keresési példában az első pozitív számot, illetve a 0-ás elemet keressük meg a *v* vektorban:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<double> v {-3.1, -1.25, 0, 2.7, 4.01, 6.321};
    auto p = find_if(begin(v), end(v), [](double e) {return e > 0;});
    if (p != end(v))
        cout << "Az első pozitív szám: " << v[distance(begin(v), p)];

    p = find_if_not(begin(v), end(v), [](double e) {return e > 0 || e < 0;});
    if (p != end(v))
        cout << "\nA nulla: " << v[distance(begin(v), p)];
    return 0;
}
```

Az első pozitív szám: 2.7
A nulla: 0

2.5.2.5 Azonosság (*equal()*) és eltérés (*mismatch()*) vizsgálata

Algoritmus hívása	Leírás
bool <i>b</i> = <i>equal</i> (<i>itb</i> , <i>ite</i> , <i>itb2</i>)	igaz értékkel tér vissza, ha [<i>itb</i> , <i>ite</i>) tartomány elemei sorra megegyeznek az [<i>itb2</i> , <i>itb2</i> +(<i>ite</i> - <i>itb</i>)) tartomány elemeivel,
bool <i>b</i> = <i>equal</i> (<i>itb</i> , <i>ite</i> , <i>itb2</i> , <i>binpred</i>)	igaz értékkel tér vissza, ha [<i>itb</i> , <i>ite</i>) és az [<i>itb2</i> , <i>itb2</i> +(<i>ite</i> - <i>itb</i>)) tartomány minden elempárjára a predikátum igaz értékű,
pair < <i>it1</i> , <i>it2</i> > <i>p</i> = <i>mismatch</i> (<i>itb</i> , <i>ite</i> , <i>itb2</i>)	a visszaadott iterátorpár jelzi, hogy az [<i>itb</i> , <i>ite</i>) és [<i>itb2</i> , <i>itb2</i> +(<i>ite</i> - <i>itb</i>)) tartományok elemei hol különböznek először – ha <i>it1</i> == <i>ite</i> , nincs eltérés,
pair < <i>it1</i> , <i>it2</i> > <i>p</i> = <i>mismatch</i> (<i>itb</i> , <i>ite</i> , <i>itb2</i> , <i>binpred</i>)	a visszaadott iterátorpár jelzi, hogy az [<i>itb</i> , <i>ite</i>) és [<i>itb2</i> , <i>itb2</i> +(<i>ite</i> - <i>itb</i>)) tartományok elemei hol különböznek először a predikátum szerint – ha <i>it1</i> == <i>ite</i> , nincs eltérés.

Meg kell jegyeznünk, hogy a *mismatch()* algoritmus eredménye definiálatlan, ha a második tartomány rövidebb az elsőnél. Az alábbi példa *palindrom()* függvénye megállapítja, hogy egy szöveg visszafele megegyezik-e önmagával. A megoldásban kihasználtuk, hogy a *string* típus a *vector* típussal rokon konténerműveletekkel rendelkezik. A program további részeiben a *mismatch()* és az *equal()* algoritmusok használatának lépéseit is bemutatjuk.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

bool palindrom(const string szoveg){
    string szoveg2(begin(szoveg),
                    mismatch(begin(szoveg), end(szoveg), szoveg.rbegin()).first);
    return equal(begin(szoveg), end(szoveg), begin(szoveg2));
}
```

```

int main() {
    vector<int> v1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    vector<int> v2 = { 1, 2, 3, 5, 6, 7, 8, 9, 0, 0 };
    auto iterpar = mismatch(begin(v1), end(v1), begin(v2));
    if (iterpar.first == end(v1))
        cout << "megegyeznek" << endl;
    else
        cout << *iterpar.first << " " << *iterpar.second << endl;
    vector<int> v10 = { 1, 0, 1, 0, 1, 0, 1, 0, 1, 0 };
    bool b = equal(begin(v1), end(v1), begin(v10), [](int a, int b) {return a%2 == b;});
    if (b)
        cout << "azonosak\n";
    else
        cout << "kulonboznak\n";
    string szoveg1 = "indulagorogaludni";
    if (palindrom(szoveg1))
        cout << szoveg1 << " palindrom" << endl;
    szoveg1 = "ose az eso";
    if (!palindrom(szoveg1))
        cout << szoveg1 << " nem palindrom" << endl;
    return 0;
}

```

```

4 5
azonosak
indulagorogaludni palindrom
ose az eso nem palindrom

```

2.5.2.6 Elemsorozat keresése ([search\(\)](#))

Algoritmus hívása	Leírás
<code>fiter = search(fitb, fite, fitb2, fite2)</code>	a <code>[fitb2, fite2)</code> altartomány első előfordulásának helyével tér vissza az <code>[fitb, fite-(fite2-fitb2))</code> tartományban,
<code>fiter = search(fitb, fite, fitb2, fite2, binpred)</code>	a <code>[fitb2, fite2)</code> altartomány első előfordulásának helyével tér vissza az <code>[fitb, fite-(fite2-fitb2))</code> tartományban, az elemek egyenlőségét a predikátum dönti el,
<code>fiter = search_n(fitb, fite, n, adat)</code>	az <code>adat</code> <code>n</code> darab előfordulásának kezdetével tér vissza a <code>[fitb, fite)</code> tartományban,
<code>fiter = search_n(fitb, fite, n, adat, binpred)</code>	az <code>adat</code> <code>n</code> darab előfordulásának kezdetével tér vissza a <code>[fitb, fite)</code> tartományban, az azonosságot a predikátum jelzi.

Mind a négy esetben az `ite` értéket kapjuk vissza, ha a kívánt szekvencia nem található meg az `[fitb, fite)` tartományban. A lineáris keresés bemutatásához először egy vektorban keressük meg egy lista elemeinek első előfordulását, majd pedig meghatározzuk egy vektorban tárolt ismétlődő betűk kezdőpozícióit:

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <string>
using namespace std;

int main() {
    vector<int> vektor { 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 };
    list<int> lista { begin(vektor) + 3, begin(vektor) + 7 };
    auto iter = search(begin(vektor), end(vektor), begin(lista), end(lista));
    for (int i=0; i < lista.size(); i++)
        *(iter+i) += 123;
    for (int e : vektor)

```



```

    cout << e << " ";    cout << endl;
    string s = "ABBCCDDDEEEEEFFFFFGGGGGGHHHHHHH";
    vector<char> betuk (begin(s), end(s));
    auto p = begin(betuk);
    for (char ch='A'; ch <= 'H'; ch++) {
        p=search_n(begin(betuk), end(betuk), (ch-'A'+1), ch);
        cout << ch << "\t" << distance(begin(betuk), p) << endl;
    }
    return 0;
}

```

2	3	5	131	136	144	157	55	89	144
A							0		
B							1		
C							3		
D							6		
E							10		
F							15		
G							21		
H							28		

2.5.3 Elemek sorrendjét módosító algoritmusok

A következő algoritmusok megváltoztathatják az argumentumként átadott tartományokban az elemek sorrendjét.

2.5.3.1 Elemek átalakítása – *transform()*

Algoritmus hívása	Leírás
<code>oiter = transform(itb, ite, oitb, műv)</code>	az algoritmus az <code>[itb, ite)</code> tartomány minden elemére végrehajtja a megadott unáris műveletet, és az <code>oitb</code> output iterátorral kijelölt helyre másolja az eredményt,
<code>oiter = transform(itb, ite, itb2, oitb, binműv)</code>	az algoritmus az <code>[itb, ite)</code> és <code>[itb2, itb2+(ite-itb))</code> tartományok minden elempárjára végrehajtja a megadott bináris műveletet, és az <code>oitb</code> output iterátorral kijelölt helyre másolja az eredményt.

Az algoritmusok az utolsó átalakított elem utáni kimenő iterátorral térnek vissza. Felhívjuk a figyelmet arra, hogy a műveletfüggvények nem módosíthatják a tartományok elemeit. Az alábbi példában bemutatjuk a *transform()* algoritmus különböző hívásait. A hívásokhoz szükséges egyparaméteres függvényobjektum az átadott elemet megszorozza egy tényezővel, és eltolja egy offszettel. A kétparaméteres műveletobjektumot lambda-kifejezésként definiáljuk.

```

#include <vector>
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

// függvényobjektum egy adat megszorzására adott tényezővel,
// és eltolására adott értékkel
template <typename T> class FaktorOffszet {
private:
    T faktor, offszet;
public:
    FaktorOffszet(T f, T o) : faktor(f), offszet(o) { }
    T operator()(const T& elem) const {
        return elem * faktor + offszet;
    }
};

```

```

int main() {
    vector<int> v1 {1, 3, 5, 7, 9, 11};
    vector<double> v2 (v1.size());
    // a v1 módosítása helyben
    transform(begin(v1), end(v1), begin(v1), FaktorOffszet<int>(3,1));
    for (int e : v1)
        cout << e << "\t";    cout << endl;
    // a v1 módosításai a v2-be kerülnek
    transform(begin(v1), end(v1), begin(v2), FaktorOffszet<double>(1, 0.5));
    for (double e : v2)
        cout << e << "\t";    cout << endl;
    // a v1 és v2 elempárjaival kiszámolt adatok a v3-ba kerülnek
    FaktorOffszet<double> Modosit(2, 1.3);
    auto fv = [Modosit](double e1, double e2) {return Modosit(e1+e2);};
    vector<double> v3;
    transform(begin(v1), end(v1), begin(v2), back_inserter(v3), fv);
    for (double e : v3)
        cout << e << "\t";    cout << endl;
    return 0;
}

```

4	10	16	22	28	34
4.5	10.5	16.5	22.5	28.5	34.5
18.3	42.3	66.3	90.3	114.3	138.3

2.5.3.2 Elemek másolása, áthelyezése

Algoritmus hívása	Leírás
<code>oiter = copy(itb, ite, oitb)</code>	átmásolja az <code>[itb, ite)</code> tartomány minden elemét az <code>oitb</code> output iterátorral kijelölt pozíciótól kezdve,
<code>oiter = copy_if(itb, ite, oitb, pred)</code>	átmásolja az <code>[itb, ite)</code> tartomány azon elemeit az <code>oitb</code> kimeneti iterátorral kijelölt pozíciótól kezdve, amelyre a predikátum <code>true</code> értékkel tér vissza,
<code>oiter = copy_n(itb, n, oitb)</code>	átmásolja az <code>[itb, ite)</code> tartomány első <code>n</code> darab elemét az <code>oitb</code> output iterátorral kijelölt pozíciótól kezdve,
<code>biter = copy_backward(bitb, bite, bite2)</code>	átmásolja az <code>[bitb, bite)</code> tartomány minden elemét az <code>bite2</code> kétirányú iterátorral kijelölt pozíciótól kezdve, visszafelé haladva,
<code>oiter = move(itb, ite, oitb)</code>	áthelyezi az <code>[itb, ite)</code> tartomány minden elemét az <code>oitb</code> kimeneti iterátorral kijelölt pozíciótól kezdve,
<code>biter = move_backward(bitb, bite, bite2)</code>	áthelyezi az <code>[bitb, bite)</code> tartomány minden elemét az <code>bite2</code> kétirányú iterátorral kijelölt pozíciótól kezdve, visszafelé haladva.

Az előre haladó algoritmusok az utolsó átmásolt elem utáni pozícióval térnek vissza, míg a visszafelé működő algoritmusok megadják az utolsó átmásolt elem pozícióját. Egymást átfedő tartományok esetén a *backward* utótagú algoritmusokat kell használnunk. A *másolási műveletek alkalmazását az alábbi példaprogram szemlélteti:*

```

#include <algorithm>
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main() {
    vector<int> adatok { 2, 7, 1, 8, 2, 8, 1, 8, 2, 8, 4, 5, 9, 0 };
    cout << "az eredeti vektor:" << endl;
    copy(begin(adatok), end(adatok), ostream_iterator<int> {cout, " "});
}

```

```

cout << endl;
cout << "a vektor első fele a második felebe másolva:" << endl;
copy_backward(begin(adatok), begin(adatok)+adatok.size()/2, end(adatok));
copy(begin(adatok), end(adatok), ostream_iterator<int> {cout, " "});
cout << endl;
vector<double> dv;
cout << "a páratlan elemek átmásolása egy másik vektorba:" << endl;
copy_if(begin(adatok), end(adatok), back_inserter(dv),
        [](int e) {return e%2==1;});
copy(begin(dv), end(dv), ostream_iterator<double> {cout, " "});
cout << endl;
return 0;
}

```

az eredeti vektor:

2 7 1 8 2 8 1 8 2 8 4 5 9 0

a vektor első fele a második felebe másolva:

2 7 1 8 2 8 1 2 7 1 8 2 8 1

a páratlan elemek átmásolása egy másik vektorba:

7 1 1 7 1 1

2.5.3.3 Az ismétlődő szomszédos elemek törlése a tartományból – unique()

Algoritmus hívása	Leírás
<code>fiter = unique(fitb, fite)</code>	a <code>[fitb, fite)</code> tartományból eltávolítja a szomszédos, ismétlődő elemeket – az <code>==</code> operátorral vizsgálva az elemek azonosságát,
<code>fiter = unique(fitb, fite, binpred)</code>	a <code>[fitb, fite)</code> tartományból eltávolítja a szomszédos, ismétlődő elemeket – az elemek azonosságát a predikátum true értéke jelzi,
<code>oiter = unique_copy(itb, ite, oitb)</code>	átmásolja az <code>[itb, ite)</code> tartomány elemeit az <code>oitb</code> output iterátorral kijelölt pozíciótól kezdve, a szomszédos, ismétlődő elemek kihagyásával (<code>==</code>),
<code>oiter = unique_copy(itb, ite, oitb, binpred)</code>	átmásolja az <code>[itb, ite)</code> tartomány elemeit az <code>oitb</code> kimeneti iterátorral kijelölt pozíciótól kezdve, a szomszédos, ismétlődő elemek kihagyásával (binpred).

A `unique()` algoritmus visszatérési értéke a módosított szekvencia új végét jelzi, míg az `unique_copy()` az utolsó átmásolt elem utáni pozícióval tér vissza. Felhívjuk a figyelmet arra, hogy a `unique()` művelet az elemek balra tolásával hajtodik végre, így a konténer fizikai mérete változatlan marad. A tartomány új és régi vége közötti elemeket fizikailag a konténer `erase()` tagfüggvényével távolíthatjuk el.

Az alábbi példában 32 darab, 1 és 12 közé eső véletlen számmal töltünk fel egy vektort. Ezt követően megvizsgáljuk, hogy az 1..12 tartomány minden eleme szerepel-e a generált számok között. A program kódja után két futási eredményt is bemutatunk:

```

#include <algorithm>
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>
using namespace std;

int main() {
    srand((unsigned)time(nullptr));
    const vector<int> v0 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    vector<int> v;
    for (int i=0; i<32; i++)
        v.push_back(rand() % 12 + 1); // 1..12
}

```

```

sort(v.begin(), v.end());
auto vege = unique(v.begin(), v.end());
v.erase(vege, v.end());
for (int e : v)
    cout << e << " ";    cout << endl;
if (equal(begin(v), end(v), begin(v0)))
    cout << "sikeres szamgeneralas" << endl;
else
    cout << "sikertelen szamgeneralas" << endl;
return 0;
}

```

```

1 2 3 4 5 6 7 8 9 10 11 12
sikeres szamgeneralas

1 2 3 4 6 7 9 11 12
sikertelen szamgeneralas

```

2.5.3.4 Elemek eltávolítása a tartományból – remove()

Algoritmus hívása	Leírás
<code>fiter = remove(fitb, fite, adat)</code>	a [fitb, fite) tartományból eltávolítja az <i>adattal</i> egyező elemeket – az <code>==</code> operátorral vizsgálva az azonosságot,
<code>fiter = remove_if(fitb, fite, pred)</code>	a [fitb, fite) tartományból eltávolítja azokat az elemeket, melyekre a predikátum true értékkel tér vissza,
<code>oiter = remove_copy(itb, ite, oitb, adat)</code>	átmásolja az [itb, ite) tartomány azon elemeit az <i>oitb</i> kimeneti iterátorral kijelölt pozíciótól kezdve, amelyek nem egyeznek meg az <i>adattal</i> (<code>==</code>),
<code>oiter = remove_copy_if(itb, ite, oitb, pred)</code>	átmásolja az [itb, ite) tartomány azon elemeit az <i>oitb</i> output iterátorral kijelölt pozíciótól kezdve, amelyekre a predikátum false értéket ad.

A `remove` algoritmusok visszatérési értéke a módosított szekvencia új végét jelzik, míg az `remove_copy` *hívások* az utolsó átmásolt elem utáni pozícióval térnek vissza. A `remove()` az `unique()` algoritmushoz hasonlóan csupán átcsoportosítja az elemeket, így az elemek fizikai törléséhez most is az `erase()` tagfüggvényt kell használnunk. A `remove()` és a `replace()` algoritmusok használatát *string* típusú tárolókkal szemléltetjük, amelyek a mi szempontunkból vektorként viselkednek. A *példában* először eltávolítjuk a szövegből a tagoló (white-space) karaktereket, majd a vesszőket, végül pedig minden 'A' betűnél kisebb kódú karaktert:

```

#include <algorithm>
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main() {
    string szoveg = "A=65, B=66, C=67,\n D=68, \tE=69 , F=70";
    cout << szoveg << endl;
    auto vege = remove_if(begin(szoveg), end(szoveg), ::isspace);
    szoveg.erase(vege, end(szoveg));
    cout << szoveg << endl;
    vege = remove(begin(szoveg), end(szoveg), ',');
    szoveg.erase(vege, end(szoveg));
    cout << szoveg << endl;
    vege = remove_if(begin(szoveg), end(szoveg), [](char ch) {return ch<'A';});
    szoveg.erase(vege, end(szoveg));
    cout << szoveg << endl;
    return 0;
}

```

```

A=65, B=66, C=67,
D=68, E=69, F=70
A=65,B=66,C=67,D=68,E=69,F=70
A=65B=66C=67D=68E=69F=70
ABCDEF

```

2.5.3.5 Elemek lecserélése – replace()

Algoritmus hívása	Leírás
<code>replace(fitb, fite, radat, uadat)</code>	a <code>[fitb, fite)</code> tartomány minden (régi) <code>radat</code> tal megegyező elemét (új) <code>uadat</code> ra cseréli (<code>==</code>),
<code>replace_if(fitb, fite, pred, uadat)</code>	a <code>[fitb, fite)</code> tartományban <code>uadat</code> ra cseréli azokat az elemeket, melyekre a predikátum true értékkel tér vissza,
<code>oiter = replace_copy(itb, ite, oitb, radat, uadat)</code>	átmásolja az <code>[itb, ite)</code> tartomány elemeit az <code>oitb</code> output iterátorral kijelölt pozíciótól kezdve, és közben lecseréli az <code>radat</code> tal egyező elemeket az <code>uadat</code> ra (<code>==</code>),
<code>oiter = replace_copy_if(itb, ite, oitb, pred, uadat)</code>	átmásolja az <code>[itb, ite)</code> tartomány elemeit az <code>oitb</code> kimeneti iterátorral kijelölt pozíciótól kezdve, és közben <code>uadat</code> ra cseréli azokat az elemeket, melyekre a predikátum true értéket ad vissza.

Az alábbi példában először a szövegben megadott vesszőket szóközőkre cseréljük, majd a szöveg átmásolásakor a kisbetűket szóközzel helyettesítjük. A program végén pedig az ismétlődő szóközőket eltávolítjuk a sztringből:

```

#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
#include <cctype>
using namespace std;

int main() {
    string eredeti = "C++17,ISO/IEC,C++,szabvany,kovetkezo,valtozatanak,"
                    "nem,hivatalos,neve.";
    string s1;
    cout << eredeti << endl;
    replace(begin(eredeti), end(eredeti), ',', char(32));
    cout << eredeti << endl;
    replace_copy_if(begin(eredeti), end(eredeti),
                    back_inserter(s1), ::islower, char(32));
    auto vege = unique(begin(s1)+17, end(s1));
    s1.erase(vege, end(s1));
    cout << s1 << endl;
    return 0;
}

```

```

C++17,ISO/IEC,C++,szabvany,kovetkezo,valtozatanak,nem,hivatalos,neve.
C++17 ISO/IEC C++ szabvany kovetkezo valtozatanak nem hivatalos neve.
C++17 ISO/IEC C++ .

```

2.5.3.6 Az elemek sorrendjének módosítása

Algoritmus hívása	Leírás
<code>reverse(bitb, bite)</code> <code>oiter = reverse_copy(bitb, bite, oitb)</code>	megfordítja az <code>[bitb, bite)</code> tartomány elemeinek sorrendjét, úgy másolja át az <code>[bitb, bite)</code> tartomány elemeit az <code>oitb</code> output iterátorral kijelölt pozíciótól kezdve, hogy az új tartományban az elemek fordított sorrendben helyezkedjenek el,
<code>fiter = rotate(fitb, fitelso, fite)</code> <code>oiter = rotate_copy(fitb, fitelso, fite, oitb)</code>	a <code>[fitb, fite)</code> tartomány elemeit balra haladva körbe lépteti (<code>swap()</code> hívásokkal), egészen addig, míg az <code>fitelso</code> elem a tartomány első eleme nem lesz, úgy másolja át a <code>[fitb, fite)</code> tartomány elemeit az <code>oitb</code> kimeneti iterátorral kijelölt pozíciótól kezdve, hogy az új tartományban <code>fitelso</code> elem legyen az első helyen,
<code>random_shuffle(ritb, rite)</code>	a <code>[ritb, rite)</code> tartomány elemeit véletlen sorrendbe rendezi (keveri) egy szabványos véletlenszám-generátor segítségével (az implementációk egy részében ez a <code>rand()</code>),
<code>random_shuffle(ritb, rite, veletlenfv)</code>	a <code>[ritb, rite)</code> tartomány elemeit véletlen sorrendbe rendezi (keveri) a <code>veletlenfv</code> funktor hívásával,
<code>shuffle(itb, ite, veletlengen)</code>	az <code>[itb, ite)</code> tartomány elemeit véletlen sorrendbe rendezi (keveri) az egyenletes eloszlású véletlenszám-generátor <code>veletlengen</code> felhasználásával.

A fenti algoritmusok megfordítják (`reverse`), balra elforgatják (`rotate`) és véletlenszerűen elrendezik (keverik – `shuffle`) a megadott tartomány elemeit. Amennyiben jobbra szeretnénk forgatni az elemeket, fordított iterátorokat kell használnunk az algoritmus hívásakor. Mindhárom csoport műveleteinek végrehajtásakor az elemek cseréje a `swap()` függvény hívásával valósul meg.

A következő program az egyes műveleteket szemlélteti. Mivel a C++14 a `random_shuffle()` algoritmust elavultnak minősíti, a példában a támogatott `shuffle()` műveletet használjuk.

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>
#include <ctime>
using namespace std;

int main() {
    default_random_engine rnd;
    rnd.seed(time(nullptr)); // a generáló algoritmus inicializálása
    vector<int> v {1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105 };
    cout << "az eredeti vektor: " << endl;
    copy(begin(v), end(v), ostream_iterator<int> {cout, " " } );
    cout << endl;
    cout << "forgatas balra 2x: " << endl;
    for (int i=0; i<2; i++) {
        rotate(begin(v), begin(v)+1, end(v));
        copy(begin(v), end(v), ostream_iterator<int> {cout, " " } );
        cout << endl;
    }
    cout << "forgatas jobbra 2x: " << endl;
    for (int i=0; i<2; i++) {
        rotate(v.rbegin(), v.rbegin()+1, v.rend());
        copy(begin(v), end(v), ostream_iterator<int> {cout, " " } );
        cout << endl;
    }
}
```

```

cout << "fordított elemsorrend: " << endl;
reverse(begin(v), end(v));
copy(begin(v), end(v), ostream_iterator<int> {cout, " "} );
cout << endl;

cout << "veletlen elemsorrend 3x: " << endl;
for (int i=0; i<3; i++) {
    shuffle(begin(v), end(v), rnd);
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "} );
    cout << endl;
}
return 0;
}

```

```

az eredeti vektor:
1 3 6 10 15 21 28 36 45 55 66 78 91 105
forgatas balra 2x:
3 6 10 15 21 28 36 45 55 66 78 91 105 1
6 10 15 21 28 36 45 55 66 78 91 105 1 3
forgatas jobbra 2x:
3 6 10 15 21 28 36 45 55 66 78 91 105 1
1 3 6 10 15 21 28 36 45 55 66 78 91 105
fordított elemsorrend:
105 91 78 66 55 45 36 28 21 15 10 6 3 1
veletlen elemsorrend 3x:
6 28 15 105 66 55 78 36 21 10 3 91 45 1
6 105 28 78 21 10 66 15 55 36 45 3 91 1
78 28 105 10 66 45 1 15 6 3 36 21 55 91

```

2.5.3.7 Az elemek permutációja

Ebben az esetben permutáció alatt a megadott tartomány elemeinek következetes átrendezését értjük, a kiindulási elrendezés figyelésével. Felhívjuk a figyelmet arra, hogy az összes lehetséges permutációt csak akkor állítják elő az algoritmusok, ha a tartomány elemei között nincsenek azonos elemek, mivel az algoritmus leáll, ha a kiindulásival azonos permutáció áll elő az ismétlődő elemek miatt.

Algoritmus hívása	Leírás
bool <i>b</i> = <i>next_permutation</i> (<i>bitb</i> , <i>bite</i>)	előállítja az [<i>bitb</i> , <i>bite</i>) tartomány elemeinek következő permutációját a lexikografikus < művelet alkalmazásával,
bool <i>b</i> = <i>next_permutation</i> (<i>bitb</i> , <i>bite</i> , <i>compfv</i>)	előállítja az [<i>bitb</i> , <i>bite</i>) tartomány elemeinek következő permutációját a <i>compfv</i> függvényobjektum alkalmazásával,
bool <i>b</i> = <i>prev_permutation</i> (<i>bitb</i> , <i>bite</i>)	előállítja az [<i>bitb</i> , <i>bite</i>) tartomány elemeinek előző permutációját a lexikografikus < művelet alkalmazásával,
bool <i>b</i> = <i>prev_permutation</i> (<i>bitb</i> , <i>bite</i> , <i>compfv</i>)	előállítja az [<i>bitb</i> , <i>bite</i>) tartomány elemeinek előző permutációját a <i>compfv</i> függvényobjektum alkalmazásával,
bool <i>b</i> = <i>is_permutation</i> (<i>fitb</i> , <i>fite</i> , <i>fitb2</i>)	igaz értékkel tér vissza, ha az [<i>fitb</i> , <i>fite</i>) tartomány elemei a <i>fitb2</i> kezdetű tartomány permutációját alkotják – az ellenőrzéshez a == műveletet használja a függvény,
bool <i>b</i> = <i>is_permutation</i> (<i>fitb</i> , <i>fite</i> , <i>fitb2</i> , <i>binpred</i>)	igaz értékkel tér vissza, ha az [<i>fitb</i> , <i>fite</i>) tartomány elemei a <i>fitb2</i> kezdetű tartomány permutációját adják – az ellenőrzés során a <i>binpred</i> predikátumot alkalmazza a függvény.

A *compfv* függvényobjektum **true** értékkel tér vissza, ha az első argumentuma kisebb a másodikonál::

```
bool compfv(const típus1& a, const típus2& b);
```

A permutációt előállító algoritmusok **true** értékkel jelzik az új permutáció sikeres előállítását, míg a **false** érték azt jelenti, hogy elértük a permutáció-sorozat kezdetét (vagyis a híváskor fennálló sorrendet). A permutációk előállítását és ellenőrzését az alábbi példaprogram mutatja be:

```
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

int main() {
    string s = "ABC";
    do {
        cout << s << '\t';
    } while(next_permutation(begin(s), end(s)));
    cout << endl;

    s = "ADA";
    do {
        cout << s << '\t';
    } while(prev_permutation(begin(s), end(s)));
    cout << endl;

    vector<int> v1 { 3, 7, 10, 11, 14, 17 }, v2(v1);
    random_shuffle(begin(v2), end(v2));
    if (is_permutation(begin(v1), end(v1), begin(v2)))
        cout << "v2 v1 permutacioja" << endl;
    if (!is_permutation(begin(v1), end(v1), begin(v2)+1))
        cout << "v2+1 nem permutacioja a v1-nek" << endl;
    return 0;
}
```

```
ABC    ACB    BAC    BCA    CAB    CBA
ADA    AAD
v2 v1 permutacioja
v2+1 nem permutacioja a v1-nek
```

2.5.3.8 Az elemek felosztása (particionálása)

Algoritmus hívása	Leírás
<i>fiter</i> = <i>partition</i> (<i>fitb</i> , <i>fite</i> , <i>pred</i>)	kettéosztja a [<i>fitb</i> , <i>fite</i>) tartomány elemeit: előre kerülnek azok, melyekre a predikátum true értékkel tér vissza, a végére pedig azok, melyekre false a függvényérték; a visszaadott iterátor kijelöli a második csoport első elemét,
<i>biter</i> = <i>stable_partition</i> (<i>bitb</i> , <i>bite</i> , <i>pred</i>)	az elemek relatív helyének megőrzése mellett, a <i>partition()</i> algoritmushoz hasonlóan osztja ketté a [<i>bitb</i> , <i>bite</i>) tartomány elemeit,
<i>pair</i> < <i>oit1</i> , <i>oit2</i> > <i>p</i> = <i>partition_copy</i> (<i>itb</i> , <i>ite</i> , <i>oitb_true</i> , <i>oitb_false</i> , <i>pred</i>)	átmásolja az [<i>itb</i> , <i>ite</i>) tartomány elemeit: az <i>oitb_true</i> pozíciótól kezdve kerülnek azok az elemek, melyekre a predikátum igaz értéket ad, míg az <i>oitb_false</i> pozíciótól kezdve másolódnak a hamis predikátum-értékűek; a visszaadott pár <i>oit1</i> , <i>oit2</i> iterátorai az átmásolt tartományok végeit jelölik,
<i>fiter</i> = <i>partition_point</i> (<i>fitb</i> , <i>fite</i> , <i>pred</i>)	megvizsgálja a felosztott [<i>fitb</i> , <i>fite</i>) tartományt, és visszatér annak az elemnek a pozíciójával, melyre a predikátum először hamis értéket ad, vagy a <i>fite</i> értékkel, ha nincs ilyen elem,
<i>bool</i> <i>b</i> = <i>is_partitioned</i> (<i>itb</i> , <i>ite</i> , <i>pred</i>)	true értékkel tér vissza, ha az [<i>itb</i> , <i>ite</i>) tartomány particionált vagy üres.

Az alábbi példában először kettéosztjuk a *v* vektor elemeit a középső elem értéke alapján, majd pedig sorba rendezzük az eredeti *v* vektor elemeit a particionálást használó gyorsrendezés függvénnyel:


```

#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>
using namespace std;

template<typename RandomIter>
void GyorsRendezes(RandomIter ritb, RandomIter rite) {
    typedef typename iterator_traits<RandomIter>::value_type AdatTipus;
    if (rite - ritb < 2)
        return;
    RandomIter ritkozep = ritb + (rite - ritb)/2;
    RandomIter ritutolso = rite-1;
    iter_swap(ritkozep, ritutolso); // a pivot elem elmentése az utolsó pozícióba
    ritkozep = partition(ritb, ritutolso,
                        [ritutolso] (const AdatTipus& x) { return x < *ritutolso;});
    iter_swap(ritkozep, ritutolso); // a pivot elem visszaállítása
    GyorsRendezes(ritb, ritkozep);
    GyorsRendezes(ritkozep+1, rite);
}

int main() {
    vector<int> v {1, 12, 5, 26, 7, 14, 3, 7, 2};
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "} );
    cout << endl;
    // a v vektor partícionálása a középső elem alapján
    int kelem = v[v.size()/2];
    auto iter = partition(begin(v), end(v), [kelem] (int e) {return e <= kelem;});
    vector<int> v1(begin(v), iter);
    copy(begin(v1), end(v1), ostream_iterator<int> {cout, " "} );
    cout << endl;
    vector<int> v2(iter, end(v));
    copy(begin(v2), end(v2), ostream_iterator<int> {cout, " "} );
    cout << endl;
    GyorsRendezes(begin(v), end(v));
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "} );
    cout << endl;
    return 0;
}

```

```

1 12 5 26 7 14 3 7 2
1 2 5 7 7 3
14 26 12
1 2 3 5 7 7 12 14 26

```

2.5.3.9 Elemek inicializálása

Algoritmus hívása	Leírás
<code>fill(fitb, fite, adat)</code> <code>oiter = fill_n(oitb, n, adat)</code>	az <i>adattal</i> tölti fel a <i>[fitb, fite)</i> tartomány elemeit, az <i>adattal</i> inicializálja az <i>oitb</i> kezdetű tartomány első <i>n</i> darab elemét, és visszatér az utolsó inicializált elem utáni pozícióval,
<code>generate(fitb, fite, gen)</code> <code>oiter = generate_n(oitb, n, gen)</code>	a <i>gen</i> függvényobjektum értékeivel inicializálja a <i>[fitb, fite)</i> tartomány elemeit, a <i>gen</i> függvényobjektum értékeivel tölti fel az <i>oitb</i> kezdetű tartomány első <i>n</i> darab elemét, és visszatér az utolsó inicializált elem utáni pozícióval.

Valamely tartomány elemeit feltölthetjük egy megadott adat, illetve egy függvény által visszaadott érték **bemásolásával**. A *gen* függvényobjektumot az alábbi formában definiálhatjuk:

típus gen();

Az alábbi példában egy 12 elemű vektort a Fibonacci-számsor értékeivel töltünk fel:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

struct Fibo {
    unsigned e0{0}, e1{1};
    unsigned operator()() {
        unsigned e2 = e0 + e1;
        e0 = e1;
        e1 = e2;
        return e2;
    }
};

int main() {
    vector<unsigned> fibo(12);
    generate(begin(fibo), end(fibo), Fibo());
    copy(begin(fibo), end(fibo), ostream_iterator<unsigned>(cout, ", "));
    return 0;
}
```

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,

A *fill()* és a *generate()* algoritmusokkal már valamilyen módon inicializált elemeket új értékkel látunk el. Alacsony szintű programozás során azonban olyan területekkel is találkozhatunk, melyek elemei még nem voltak inicializálva. Az ilyen területek feltöltéséhez a *<memory>* fejláományban találunk néhány algoritmust:

Algoritmus hívása	Leírás
<i>uninitialized_fill(fitb, fite, adat)</i>	az <i>adattal</i> inicializálja a <i>[fitb, fite)</i> inicializálatlan tartomány elemeit,
<i>fiter = uninitialized_fill_n(fitb, n, adat)</i>	az <i>adattal</i> inicializálja az <i>fitb</i> kezdetű inicializálatlan tartomány első <i>n</i> darab elemét, majd visszatér az utoljára inicializált elem utáni pozícióval,
<i>fiter = uninitialized_copy(itb, ite, fitb)</i>	átmásolja az <i>[itb, ite)</i> tartomány minden elemét az <i>fitb</i> iterátorral kijelölt inicializálatlan tartományba, és visszatér az utoljára másolt elem utáni pozícióval,
<i>fiter = uninitialized_copy_n(itb, n, fitb)</i>	átmásolja az <i>itb</i> kezdetű tartomány első <i>n</i> darab elemét az <i>fitb</i> iterátorral kijelölt inicializálatlan tartományba, és visszatér az utoljára másolt elem utáni pozícióval.

Míg a *fill()*, a *generate()* és a *copy()* algoritmusok az értékadás műveletét (=) használják, addig az *uninitialized* algoritmusok a **másoló konstruktort** hívják. Az alábbi példában C könyvtári függvénnyel lefoglalt 12-elemű dinamikus tömböt inicializálunk 23 értékű elemekkel:

```
#include <memory>
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
```

```

class Egesz { // nincs alapértelmezett konstruktor
public:
    Egesz(int x) : erteke( x ) {}
    int GetErteke( ) { return erteke; }
private:
    int erteke;
};

int main() {
    const int n = 12;
    Egesz adat(23);
    Egesz* tomb = (Egesz*) malloc( n * sizeof(Egesz) ); // dinamikus helyfoglalás
    uninitialized_fill(tomb, tomb + n, adat );
    for (int i=0; i<n; i++)
        cout << tomb[i].GetErteke() << ", ";
    cout << endl;
    return 0;
}

```

2.5.3.10 A csere algoritmus – swap()

A `<utility>` fejlécfájlból deklarált `swap()` függvénysablon két azonos típusú objektum tartalmának felcserélésére készült. Általánosított formájában két változó, illetve két azonos méretű tömb eleminek cseréjére használhatjuk:

```

template<class Típus> void swap(Típus & a, Típus & b );
template<class Típus, size_t N> void swap(Típus (&a)[N], Típus (&b)[N] );

```

Ezekon kívül a legtöbb STL típushoz a csere függvény egy-egy specializált változata tartozik. Az alábbi példában háromféle típus esetén alkalmazzuk a `swap` hívásokat:

```

#include <iostream>
#include <utility>
#include <vector>
#include <cmath>
using namespace std;

int main() {
    double e=exp(1), pi=4*atan(1);
    swap(e, pi);
    cout << e << "\t" << pi << endl; // 3.14159 2.71828

    const size_t m = 5;
    int a[m] { 1, 3, 5, 7, 9 };
    int b[m] { 2, 4, 6, 8, 10 };
    swap(a, b);
    for (int e : a)
        cout << e << " "; cout << endl; // 2 4 6 8 10
    for (int e : b)
        cout << e << " "; cout << endl; // 1 3 5 7 9

    vector<int> va {a, a + m};
    vector<int> vb {b, b + m};
    swap(va, vb);
    for (int e : va)
        cout << e << " "; cout << endl; // 1 3 5 7 9
    for (int e : vb)
        cout << e << " "; cout << endl; // 2 4 6 8 10
    return 0;
}

```

A csere alábbi két változatát sorolhatjuk az algoritmusok közé:

Algoritmus hívása	Leírás
<code>fiter = swap_ranges(fitb, fite, fiteb2)</code>	a <code>[fitb, fite)</code> és a <code>fitb2</code> kezdetű tartományok megfelelő elemeit felcseréli egymással, és visszatér a második tartomány utoljára felcserélt eleme utáni pozícióval,
<code>iter_swap(fite1, fite2)</code>	felcseréli az iterátorokkal kijelölt elemeket.

A következő példában egy **int** vektor utolsó 5 elemét felcseréljük egy **double** lista első 5 elemével:

```
#include <algorithm>
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
using namespace std;

template <typename T>
void Kiir(string sz, T kontener) {
    cout << sz;
    copy(begin(kontener), end(kontener),
        ostream_iterator<typename T::value_type> {cout, " "} );
    cout << endl;
}

int main() {
    vector<int> vektor { 1, 3, 5, 7, 9, 11, 13, 15, 17 };
    list<double> lista { 2.1, 10.1, 31.2, 52.3, 73.4, 94.5 };
    Kiir("vektor: ", vektor);
    Kiir("lista: ", lista);
    cout << "-----" << endl;
    const int n = 5;
    swap_ranges(end(vektor)-n, end(vektor), begin(lista));
    Kiir("vektor: ", vektor);
    Kiir("lista: ", lista);
    return 0;
}
```

```
vektor: 1 3 5 7 9 11 13 15 17
lista:  2.1 10.1 31.2 52.3 73.4 94.5
-----
vektor: 1 3 5 7 2 10 31 52 73
lista:  9 11 13 15 17 94.5
```

2.5.4 Rendezés és keresés

A keresés és a rendezés nagy hangsúlyt kap a legtöbb programban. A rendezés elsősorban ahhoz kell, hogy gyorsabban tudjuk megtalálni a tárolt adatok között azt, amire kíváncsiak vagyunk. Természetesen a felhasználó is jobban boldogul a megjelenített adatokkal, ha azok rendezettek. Mindkét művelet során alapvető fontosságú a kisebb összehasonlítás ($a < b$), mellyel szükség esetén az azonosságot is ellenőrizhetjük: $!(a < b) \ \&\& \ !(b < a)$.

2.5.4.1 Rendezési algoritmusok

Az esetek többségében alkalmazott rendezési algoritmust a `sort()` függvénysablon testesíti meg. Első formájában a kisebb operátort használja az összehasonlítások során, míg második alakjában egy hasonlító függvényobjektum is megadható, melynek **true** értéke jelzi, ha az első argumentuma kisebb a másodikonál:

Algoritmus hívása	Leírás
<code>sort(ritb, rite)</code>	a kisebb operátor (<) felhasználásával növekvő sorrendbe rendezi az [ritb, rite) tartomány elemeit, és eközben nem garantálja, hogy az azonos elemek sorrendje megmarad,
<code>sort(ritb, rite, compfv)</code>	a compfv segítségével sorba rendezi az [ritb, rite) tartomány elemeit, és eközben az azonos elemek sorrendje megváltozhat.

A **compfv** függvényobjektum **true** értékkel tér vissza, ha az első argumentuma kisebb, mint a második. (Kisebb memóriaigényű típusok esetén a függvényfejből a **const** kulcsszó és a **&** referencijel el is hagyhatók.)

```
bool compfv(const típus1& a, const típus2& b);
```

```
bool compfv(típus1 a, típus2 b);
```

Az alábbi példában véletlen számokkal töltünk fel egy vektort, majd rendezzük az elemeit növekvő sorrendbe, majd pedig kétféleképpen csökkenő sorrendbe:

```
#include <random>
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>
#include <iterator>
#include <ctime>
using namespace std;

int main() {
    const int n = 10;
    mt19937 generator; // véletlenszám generátor
    generator.seed(time(nullptr));
    uniform_int_distribution<> elozslas(-12, 23); // [-12, 23]
    vector<int> v;
    // a vektor feltöltése
    generate_n(back_inserter(v), n, bind(elozslas, generator));
    cout << "a generalt elemek:\t";
    copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
    cout << endl;

    sort(begin(v), end(v));
    cout << "novekvo sorrend:\t";
    copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
    cout << endl;

    random_shuffle(begin(v), end(v));
    sort(begin(v), end(v), [](int e1, int e2){return e1 > e2;});
    cout << "csokkeno sorrend:\t";
    copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
    cout << endl;

    random_shuffle(begin(v), end(v));
    sort(begin(v), end(v), greater<int>());
    cout << "csokkeno sorrend:\t";
    copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
    cout << endl;
    return 0;
}
```

<i>a generalt elemek:</i>	12 1 0 5 21 -4 3 -11 -8 -1
<i>novekvo sorrend:</i>	-11 -8 -4 -1 0 1 3 5 12 21
<i>csokkeno sorrend:</i>	21 12 5 3 1 0 -1 -4 -8 -11
<i>csokkeno sorrend:</i>	21 12 5 3 1 0 -1 -4 -8 -11

A további rendezéssel kapcsolatos algoritmusokat táblázatban foglaltuk össze:

Algoritmus hívása	Leírás
<code>stable_sort(ritb, rite)</code>	a kisebb operátor (<) felhasználásával növekvő sorrendbe rendezi a <code>[ritb, rite)</code> tartomány elemeit, eközben garantálja, hogy az azonos elemek sorrendje nem változik,
<code>stable_sort(ritb, rite, compfv)</code>	a compfv segítségével sorba rendezi a <code>[ritb, rite)</code> tartomány elemeit, és eközben az azonos elemek sorrendje megmarad,
<code>partial_sort(ritb, ritm, rite)</code>	a kisebb operátor (<) felhasználásával növekvő sorrendbe rendezi a <code>[ritb, rite)</code> tartomány ritm-ritb darab legkisebb elemét, és a <code>[ritb, ritm)</code> tartományba helyezi azokat; a <code>[ritm, rite)</code> tartományban az elemsorrend definiálatlan,
<code>partial_sort(ritb, ritm, rite, compfv)</code>	a compfv segítségével sorba rendezi a <code>[ritb, rite)</code> tartomány elemeit, és az első ritm-ritb darab elemet a <code>[ritb, ritm)</code> tartományba helyezi; a <code>[ritm, rite)</code> tartományban az elemsorrend definiálatlan,
<code>riter = partial_sort_copy(itb, ite, ritb, rite)</code>	a növekvő sorrendbe rendezi az <code>[itb, ite)</code> tartomány rite-ritb darab legkisebb elemét, és a <code>[ritb, rite)</code> tartományba másolja azokat; a függvényérték a rendezett tartomány végét jelöli,
<code>riter = partial_sort_copy(itb, ite, ritb, rite, compfv)</code>	a compfv segítségével sorba rendezi az <code>[itb, ite)</code> tartomány elemeit, és az első rite-ritb darab elemet a <code>[ritb, rite)</code> tartományba másolja; a függvényérték a rendezett tartomány végét jelöli,
<code>bool b = is_sorted(fitb, fite)</code>	ellenőrzi, hogy a <code>[fitb, fite)</code> tartomány elemei növekvő sorrendbe rendezettek-e,
<code>bool b = is_sorted(fitb, fite, compfv)</code>	
<code>fiter = is_sorted_until(fitb, fite)</code>	megadja, hogy a <code>[fitb, fite)</code> elemek a tartomány elejétől meddig rendezettek növekvő sorrendben; a fiter az utolsó rendezett elem utáni pozíciót jelöli,
<code>fiter = is_sorted_until(fitb, fite, compfv)</code>	
<code>nth_element(ritb, ritn, rite)</code>	az algoritmusok úgy rendezik át a <code>[ritb, rite)</code> elemeit, hogy a közbenső ritn pozícióba az az elem kerüljön, mintha teljes rendezés történt volna; minden eleme a <code>[ritb, ritn)</code> tartománynak $\leq *itn$, és minden eleme a <code>[ritn, rite)</code> tartománynak $\geq *itn$.
<code>nth_element(ritb, ritn, rite, compfv)</code>	

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <functional>
using namespace std;

int main() {
    vector<int> v {4, 20, 8, 13, 21, 11, 23, 19, 9, 5, 12, 3, 7, 5};
    cout << "eredeti:\n";
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    partial_sort(begin(v), begin(v)+v.size()/2, end(v));
    cout << "első fele rendezett:\n";
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    vector<int> w(20);
    auto riter = partial_sort_copy(begin(v)+v.size()/2, end(v), begin(w), end(w));
    cout << "második fele egy másik vektorba rendezett:\n";
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    copy(begin(w), riter, ostream_iterator<int> {cout, " "}); cout << endl;
}
```

```

if (!is_sorted(begin(v), end(v))) {
    cout << "v vektornak csak egy resze rendezett:\n";
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
}

auto fiter = is_sorted_until(begin(v), end(v));
cout << "a rendezett resz:\n";
copy(begin(v), fiter, ostream_iterator<int> {cout, " "}); cout << endl;

stable_sort(begin(v), end(v));
cout << "a rendezett vektor:\n";
copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
random_shuffle(begin(v), end(v));
nth_element(begin(v), begin(v)+12, end(v), less<int>());
cout << "a 12. pozicioban az elem a helyen all:\n";
copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
return 0;
}

```

```

eredeti:
4 20 8 13 21 11 23 19 9 5 12 3 7 5
elso fele rendezett:
3 4 5 5 7 8 9 23 21 20 19 13 12 11
masodik fele egy masik vektorba rendezett:
3 4 5 5 7 8 9 23 21 20 19 13 12 11
11 12 13 19 20 21 23
v vektornak csak egy resze rendezett:
3 4 5 5 7 8 9 23 21 20 19 13 12 11
a rendezett resz:
3 4 5 5 7 8 9 23
a rendezett vektor:
3 4 5 5 7 8 9 11 12 13 19 20 21 23
a 12. pozicioban az elem a helyen all:
19 4 13 5 3 8 11 5 7 9 12 20 21 23

```

2.5.4.2 Bináris keresés rendezett tartományokban

A bináris rendezés algoritmusát a `binary_search()` függvénysablon valósítja meg:

Algoritmus hívása	Leírás
<code>bool b = binary_search(fitb, fite, adat)</code>	a kisebb operátor (<) felhasználásával ellenőrzi, hogy az adat megtalálható-e a megadott tartomány elemei között,
<code>bool b = binary_search(fitb, fite, adat, compfv)</code>	a compfv segítségével ellenőrzi, hogy az adat megtalálható-e a megadott tartomány elemei között.

További algoritmusok is épülnek a bináris keresésre:

Algoritmus hívása	Leírás
<code>fiter = lower_bound(fitb, fite, adat)</code>	az első olyan elem iterátorával tér vissza, amelyik nem kisebb , mint a megadott <i>adat</i> ; sikertelen esetben <i>fite</i> a függvényérték,
<code>fiter = lower_bound(fitb, fite, adat, compfv)</code>	
<code>fiter = upper_bound(fitb, fite, adat)</code>	az első olyan elem iterátorával tér vissza, amelyik nagyobb , mint a megadott <i>adat</i> ; sikertelen esetben <i>fite</i> a függvényérték,
<code>fiter = upper_bound(fitb, fite, adat, compfv)</code>	
<code>pair<fitb, fite> = equal_range(fitb, fite, adat)</code>	az <i>adattal</i> megegyező elemeket tartalmazó tartománnyal [<i>fitb</i> , <i>fite</i>) tér vissza – sikertelen esetben mindkét visszaadott iterátor értéke <i>fite</i> .
<code>pair<fitb, fite> = equal_range(fitb, fite, adat, compfv)</code>	

A fenti híváspárokból az első hívás a `<` műveletet, míg a második a *compfv* függvény használja összehasonlításra. A bináris keresés sikeres működésének előfeltétele, hogy a rendezéshez és a kereséshez ugyanazt a műveletet alkalmazzuk, mint ahogy ez az alábbi példából is látható:

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>
#include <iterator>
using namespace std;

int main() {
    vector<int> v {43, 25, 20, 03, 27, 44, 89, 85, 60, 00};
    sort(v.begin(), v.end());
    copy(begin(v), end(v), ostream_iterator<int>{cout, " "});    cout << endl;
    bool van = binary_search(begin(v), end(v), 43);
    if (van)
        cout << "43 megtalalhato az elemek kozott" << endl;
    van = binary_search(begin(v), end(v), 43, greater<int>());
    if (!van)
        cout << "43 nem talalhato meg az elemek kozott" << endl;
    sort(v.begin(), v.end(), greater<int>());
    copy(begin(v), end(v), ostream_iterator<int>{cout, " "});    cout << endl;
    van = binary_search(begin(v), end(v), 43, greater<int>());
    if (van)
        cout << "43 megtalalhato az elemek kozott" << endl;
    return 0;
}
```

```
0 3 20 25 27 43 44 60 85 89
43 megtalalhato az elemek kozott
43 nem talalhato meg az elemek kozott

89 85 60 44 43 27 25 20 3 0
43 megtalalhato az elemek kozott
```

Rendezett tartományban azonos elemek intervallumát, illetve ilyen intervallumok határait kereshetjük:

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>
#include <iterator>
using namespace std;

int main() {
    vector<int> v {1, 6, 1, 8, 3, 5, 8, 8, 7, 11};
    cout << "indexek: ";
    for (int i = 0; i < v.size(); i++)
        cout << i << " ";    cout << endl;
    sort(v.begin(), v.end());
    cout << "vektor: ";
    copy(begin(v), end(v), ostream_iterator<int>{cout, " "});    cout << endl;
    auto fiter = lower_bound(begin(v), end(v), 2);
    cout << "elso >= 2 elem pozicioja: " << distance(begin(v), fiter) << endl;
    fiter = upper_bound(begin(v), end(v), 7);
    cout << "elso > 7 elem pozicioja: " << distance(begin(v), fiter) << endl;
}
```



```

auto par = equal_range(v.begin(), v.end(), 8);
cout << "a 8 elemek indextartománya: [" << distance(begin(v), par.first) << ", ";
cout << distance(begin(v), par.second) << "]" << endl;
fiter = lower_bound(begin(v), end(v), 8);
cout << "elso >= 8 elem pozicioja: " << distance(begin(v), fiter) << endl;
fiter = upper_bound(begin(v), end(v), 8);
cout << "elso > 8 elem pozicioja: " << distance(begin(v), fiter) << endl;
return 0;
}

```

```

indexek: 0 1 2 3 4 5 6 7 8 9
vektor:  1 1 3 5 6 7 8 8 8 11
elso >= 2 elem pozicioja: 2
elso > 7 elem pozicioja: 6
a 8 elemek indextartománya: [6, 9)
elso >= 8 elem pozicioja: 6
elso > 8 elem pozicioja: 9

```

2.5.4.3 Rendezett tartományok összefésülése

A *merge* algoritmusok két rendezett tartományt egybe építenek.

Algoritmus hívása	Leírás
<pre>oiter = merge(itb1, ite1, itb2, ite2, oitb) oiter = merge(itb1, ite1, itb2, ite2, oitb, compfv)</pre>	egyetlen rendezett tartománnyá (az <i>oitb</i> -től kezdve) egyesíti az <i>[itb1, ite1)</i> és <i>[itb2, ite2)</i> rendezett tartományokat, a visszaadott iterátor kijelöli az új tartomány végét,
<pre>inplace_merge(bitb, bitm, bite)</pre>	összefésüli a szomszédos <i>[bitb, bitm)</i> és <i>[bitm, bite)</i> rendezett tartományokat egyetlen rendezett tartománnyá <i>[bitb, bite)</i> .
<pre>inplace_merge(bitb, bitm, bite, compfv)</pre>	

A fenti híváspárokból az első hívás a *<* műveletet, míg a második a *compfv* függvény használja összehasonlításra.

```

#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    vector<int> v1 {1, 3, 5, 7, 9, 11, 2, 4, 6, 8};
    vector<int> v2 {1, 1, 2, 3, 5, 8, 13, 21};
    vector<int> v3;
    cout << "vektor1: " << endl;
    copy(begin(v1), end(v1), ostream_iterator<int>(cout, " "));    cout << endl;
    inplace_merge(begin(v1), begin(v1)+6, end(v1));
    cout << "osszefesules helyben vektor1: " << endl;
    copy(begin(v1), end(v1), ostream_iterator<int>(cout, " "));    cout << endl;
    cout << "vektor2: " << endl;
    copy(begin(v2), end(v2), ostream_iterator<int>(cout, " "));    cout << endl;
    merge(begin(v1), end(v1), begin(v2), end(v2), back_inserter(v3));
    cout << "vektor1 es vektor2 osszefesulese vektor3-ba: " << endl;
    copy(begin(v3), end(v3), ostream_iterator<int>(cout, " "));    cout << endl;
    return 0;
}

```

```
vektor1:
1 3 5 7 9 11 2 4 6 8
osszefesules helyben vektor1:
1 2 3 4 5 6 7 8 9 11
vektor2:
1 1 2 3 5 8 13 21
vektor1 es vektor2 osszefesulese vektor3-ba:
1 1 1 2 2 3 3 4 5 5 6 7 8 8 9 11 13 21
```

2.5.4.4 Halmazműveletek rendezett tartományokkal

Az alábbi algoritmusok a megadott tartományokat halmazként értelmezik, és rajtuk halmazműveleteket hajtanak végre. Felhívjuk a figyelmet arra, hogy rendezett bemenő tartományokat kell megadunk, és a kimenő tartományok is rendezettek lesznek. (Ez a rendezettség nem feltétlenül a konténer sajátossága, például a `sort()` metódus hívásával is előállítható.)

Algoritmus hívása	Leírás
<pre>bool b = includes(itb1, ite1, itb2, ite2) bool b = includes(itb1, ite1, itb2, ite2, compfv)</pre>	<code>true</code> értékkel térnek vissza, ha az <code>[itb2, ite2)</code> rendezett tartomány minden eleme megtalálható az <code>[itb1, ite1)</code> rendezett tartományban, vagy ha az első tartomány üres – rész-halmaz,
<pre>oiter = set_union(itb1, ite1, itb2, ite2, oitb) oiter = set_union(itb1, ite1, itb2, ite2, oitb, compfv)</pre>	rendezett tartományba (az <code>oitb</code>-től kezdve) másolja azokat az elemeket, amelyek megtalálhatók az <code>[itb1, ite1)</code> és <code>[itb2, ite2)</code> rendezett tartományok bármelyikében; a visszaadott iterátor kijelöli az új tartomány végét – unió,
<pre>oiter = set_intersection(itb1, ite1, itb2, ite2, oitb) oiter = set_intersection(itb1, ite1, itb2, ite2, oitb, compfv)</pre>	rendezett tartományba (az <code>oitb</code>-től kezdve) másolja azokat az elemeket, amelyek megtalálhatók az <code>[itb1, ite1)</code> és <code>[itb2, ite2)</code> rendezett tartományok mindegyikében; a visszaadott iterátor kijelöli az új tartomány végét – metszet,
<pre>oiter = set_difference(itb1, ite1, itb2, ite2, oitb) oiter = set_difference(itb1, ite1, itb2, ite2, oitb, compfv)</pre>	rendezett tartományba (az <code>oitb</code>-től kezdve) másolja azokat az elemeket, amelyek szerepelnek az <code>[itb1, ite1)</code> rendezett tartományban, azonban nem találhatók meg az <code>[itb2, ite2)</code> tartományban; a visszaadott iterátor kijelöli az új tartomány végét – különbség,
<pre>oiter = set_symmetric_difference(itb1, ite1, itb2, ite2, oitb) oiter = set_symmetric_difference(itb1, ite1, itb2, ite2, oitb, compfv)</pre>	rendezett tartományba (az <code>oitb</code>-től kezdve) másolja azokat az elemeket, amelyek az <code>[itb1, ite1)</code> és <code>[itb2, ite2)</code> rendezett tartományok pontosan egyikében szerepelnek; a visszaadott iterátor kijelöli az új tartomány végét – szimmetrikus különbség.

A fenti híváspárokból az első hívás a `<` műveletet, míg a második a `compfv` függvény használja összehasonlításra.

A műveletekben üres halmazok (tartományok) is szerepelhetnek, és az üres halmaz minden halmaz részhalmaza. Az eleme viszony vizsgálatához a `set` típus `find()` tagfüggvényét, míg más konténerek esetén a `find()` algoritmust használhatjuk. Elemek hozzáadását és eltávolítását a halmazhoz a konténer típusától függő módon végezhetjük el.

A különböző halmazműveleteket – *elem és részhalmaza vizsgálat, valamint unió, metszet és különbségek képzések* – az alábbi példában szedtük csokorba:

```

#include <set>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

void HalmazKiir(const string& sz, const set<char>& halmaz) {
    cout << sz << " [ ";
    for (char ch : halmaz)
        cout << ch << " ";
    cout << "]" << endl;
}

int main() {
    set<char> a{'A', 'E', 'I', 'O', 'U', 'X', 'Y', 'L'};
    HalmazKiir("A =", a);
    set<char> b{'C', 'P', 'P', 'N', 'Y', 'E', 'L', 'V'};
    HalmazKiir("B =", b);
    set<char> c;
    HalmazKiir("C =", c);

    set_intersection( begin(a), end(a),
                     begin(b), end(b),
                     inserter(c, begin(c)));

    HalmazKiir("A intersect B =", c);
    HalmazKiir("C =", c);
    if (includes(begin(a), end(a), begin(c), end(c)))
        cout << "C reszhalmaz A-nak" << endl;
    c.clear();

    set_union( begin(a), end(a), begin(b), end(b), inserter(c, begin(c)));
    HalmazKiir("A union B      =", c);
    c.clear();

    set_difference( begin(a), end(a), begin(b), end(b), inserter(c, begin(c)));
    HalmazKiir("A diff. B      =", c);
    c.clear();

    set_symmetric_difference( begin(a), end(a), begin(b), end(b),
                              inserter(c, begin(c)));
    HalmazKiir("A sym.diff. B =", c);
    HalmazKiir("C =", c);
    if (c.find('X') != end(c))
        cout << "X eleme C-nek" << endl;
    return 0;
}

```

```

A = [ A E I L O U X Y ]
B = [ C E L N P V Y ]
C = [ ]
A intersect B = [ E L Y ]
C = [ E L Y ]
C reszhalmaz A-nak
A union B      = [ A C E I L N O P U V X Y ]
A diff. B      = [ A I O U X ]
A sym.diff. B = [ A C I N O P U V X ]
C = [ A C I N O P U V X ]
X eleme C-nek

```

2.5.4.5 Halomműveletek

A halom (*heap*) olyan adatstruktúra, amely az első helyen mindig a legnagyobb elemét tárolja. A halmot legjobban bináris faként képzelhetjük el, melynek a gyökerében található a legnagyobb elem. A következő algoritmusokkal a tetszőleges elérésű tartományokat halomként kezelhetjük.

Algoritmus hívása	Leírás
<code>make_heap(ritb, rite)</code> <code>make_heap(ritb, rite, compfv)</code>	a megadott tartomány elemeit halommá alakítja,
<code>push_heap(ritb, rite)</code> <code>push_heap(ritb, rite, compfv)</code>	a megadott tartomány (<i>rite-1</i>) pozícióján álló elemét beilleszti a [<i>ritb</i> , <i>rite-1</i>) halomba,
<code>pop_heap(ritb, rite)</code> <code>pop_heap(ritb, rite, compfv)</code>	felcseréli a megadott tartomány <i>ritb</i> és (<i>rite-1</i>) pozícióin álló elemeket, és a [<i>ritb</i> , <i>rite-1</i>) tartományt halommá alakítja,
<code>sort_heap(ritb, rite)</code> <code>sort_heap(ritb, rite, compfv)</code>	a [<i>ritb</i> , <i>rite</i>) halmot rendezett tartománnyá alakítja, ami már nem halom,
<code>bool b = is_heap(ritb, rite)</code> <code>bool b = is_heap(ritb, rite, compfv)</code>	true értékkel tér vissza, ha a megadott tartomány halom,
<code>riter = is_heap_until(ritb, rite)</code> <code>riter = is_heap_until(ritb, rite, compfv)</code>	megkeresi a megadott tartomány elejétől kezdve azt a legnagyobb tartományt, amely halom; a visszaadott iterátor a halom utolsó elemét követő pozíciót jelöli.

A fenti hívaspárokból az első hívás a `<` műveletet, míg a második a `compfv` függvény használja összehasonlításra.

A halommal kapcsolatos algoritmusok használatát az alábbi programmal szemléltetjük:

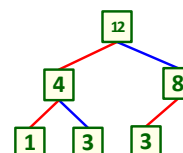
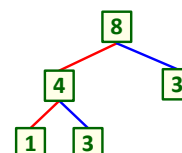
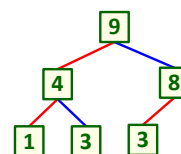
```
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main () {
    vector<int> v { 1, 9, 3, 4, 3, 8 };
    make_heap (v.begin(),v.end());
    cout << "a halom: ";
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    cout << "a legnagyobb elem: " << v.front() << '\n';

    cout << "-- pop_heap --" << endl;
    pop_heap (v.begin(),v.end());
    cout << "a kivett elem: " << v.back() << endl;
    v.pop_back();
    cout << "a halom: ";
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    cout << "a legnagyobb elem pop_heap után: " << v.front() << '\n';

    cout << "-- push_heap --" << endl;
    v.push_back(12);
    push_heap (v.begin(),v.end());
    cout << "a halom: ";
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    cout << "a legnagyobb elem push_heap után: " << v.front() << '\n';

    cout << "-- sort_heap --" << endl;
    if (is_heap(v.begin(),v.end()))
        cout << "meg halom" << endl;
    sort_heap (v.begin(),v.end());
    cout << "a rendezett halom:";
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    if (!is_heap(v.begin(),v.end()))
        cout << "mar nem halom" << endl;;
    return 0;
}
```



```

a halom: 9 4 8 1 3 3
a legnagyobb elem: 9
-- pop_heap --
a kivett elem: 9
a halom: 8 4 3 1 3
a legnagyobb elem pop_heap utan: 8
-- push_heap --
a halom: 12 4 8 1 3 3
a legnagyobb elem push_heap utan: 12
-- sort_heap --
meg halom
a rendezett halom: 1 3 3 4 8 12
mar nem halom

```

2.5.4.6 Tartományok lexicografikus összehasonlítása

Az alábbi hívások a megadott tartományokat lexicografikusan hasonlítják össze a kisebb művelet, illetve a megadott *compfv* felhasználásával. A lexicografikus összehasonlítás jellemzői:

- a két tartományt elempáronként hasonlítja,
- az első eltérő elempár viszonya meghatározza az eredményt (kisebb vagy nagyobb),
- ha az egyik tartomány megtalálható a hosszabb másik tartomány elejétől kezdve, akkor az lexicografikusan kisebb,
- egy üres tartomány lexicografikusan kisebb minden, nem üres tartománynál,
- ha két tartomány elemei páronként megegyeznek, és hosszuk is azonos, akkor lexicografikusan egyenlőek,
- két üres tartomány lexicografikusan azonos.

Algoritmus hívása	Leírás
<code>bool b = <i>lexicographical_compare</i>(itb1, ite1, itb2, ite2)</code>	true értékkel tér vissza, ha az [<i>itb1</i> , <i>ite1</i>) tartomány lexicografikusan kisebb az [<i>itb2</i> , <i>ite2</i>) tartománynál.
<code>bool b = <i>lexicographical_compare</i>(itb1, ite1, itb2, ite2, compfv)</code>	

Az alábbi példában két vektor tartalmát addig változtatjuk véletlenszerűen, míg az első lexicografikusan kisebb nem lesz a másodiknál:

```

#include <algorithm>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
using namespace std;

void Kiir(const vector<char>& v, string sz) {
    for (char ch : v)
        cout << ch << ' ';
    cout << sz;
}

int main() {
    srand(unsigned(time(nullptr)));
    vector<char> v1 {'A', 'B', 'C', 'D', 'E'};
    vector<char> v2 {'A', 'B', 'C', 'D', 'E'};
    while (!lexicographical_compare(begin(v1), end(v1), begin(v2), end(v2))) {
        Kiir(v1, ">= ");
        Kiir(v2, "\n");
        random_shuffle(begin(v1), end(v1));
        random_shuffle(begin(v2), end(v2));
    }
}

```

```

Kiir(v1, "< ");
Kiir(v2, "\n");
return 0;
}

```

```

A B C D E >= A B C D E
D E C A B >= D A B E C
C A D E B >= A D C B E
A D E C B < B E D C A

```

2.5.4.7 Legkisebb és legnagyobb kiválasztása

Az algoritmusok egy része két érték, illetve egy inicializációs lista elemei közül választja ki a legkisebb/legnagyobb adatot, illetve mindkettőt egyszerre. A választás történhet a kisebb művelet alkalmazásával, illetve egy összehasonlító függvény (*compfv*) hívásával.

Algoritmus hívása	Leírás
<i>m</i> = <i>min</i> (<i>adat1</i> , <i>adat2</i>)	két adat közül a kisebb, illetve az inicializációs listából a legkisebb elem kiválasztása,
<i>m</i> = <i>min</i> (<i>adat1</i> , <i>adat2</i> , <i>compfv</i>)	
<i>m</i> = <i>min</i> ({ <i>init.lista</i> })	
<i>m</i> = <i>min</i> ({ <i>init.lista</i> }, <i>compfv</i>)	
<i>m</i> = <i>max</i> (<i>adat1</i> , <i>adat2</i>)	két adat közül a nagyobb, illetve az inicializációs listából a legnagyobb elem kiválasztása,
<i>m</i> = <i>max</i> (<i>adat1</i> , <i>adat2</i> , <i>compfv</i>)	
<i>m</i> = <i>max</i> ({ <i>init.lista</i> })	
<i>m</i> = <i>max</i> ({ <i>init.lista</i> }, <i>compfv</i>)	
<i>pair</i> < <i>a</i> , <i>b</i> > = <i>minmax</i> (<i>adat1</i> , <i>adat2</i>)	két adat közül a kisebb (<i>a</i>), nagyobb (<i>b</i>), illetve az inicializációs listából a legkisebb (<i>a</i>), legnagyobb (<i>b</i>) elem kiválasztása.
<i>pair</i> < <i>a</i> , <i>b</i> > = <i>minmax</i> (<i>adat1</i> , <i>adat2</i> , <i>compfv</i>)	
<i>pair</i> < <i>a</i> , <i>b</i> > = <i>minmax</i> ({ <i>init.lista</i> })	
<i>pair</i> < <i>a</i> , <i>b</i> > = <i>minmax</i> ({ <i>init.lista</i> }, <i>compfv</i>)	

A műveletek más név alatt iterátorokkal kijelölt tartományokra is használhatók. Ekkor a visszatérési értékek is iterátorok, amelyek kijelöli a minimális, vagy a maximális elemet a tartományban. (Üres tartomány esetén a tartomány végét jelző iterátor a függvényérték.) A kiválasztás itt is a kisebb művelet, vagy egy összehasonlító függvény segítségével megy végbe.

Algoritmus hívása	Leírás
<i>fiter</i> = <i>min_element</i> (<i>fitb</i> , <i>fite</i>)	a [<i>fitb</i> , <i>fite</i>) tartomány legkisebb elemére hivatkozó iterátorral tér vissza,
<i>fiter</i> = <i>min_element</i> (<i>fitb</i> , <i>fite</i> , <i>compfv</i>)	
<i>fiter</i> = <i>max_element</i> (<i>fitb</i> , <i>fite</i>)	visszatér a [<i>fitb</i> , <i>fite</i>) tartomány legnagyobb elemére hivatkozó iterátorral,
<i>fiter</i> = <i>max_element</i> (<i>fitb</i> , <i>fite</i> , <i>compfv</i>)	
<i>pair</i> < <i>fit1</i> , <i>fit2</i> > = <i>minmax_element</i> (<i>fitb</i> , <i>fite</i>)	párban megadja a [<i>fitb</i> , <i>fite</i>) tartomány legkisebb (<i>fit1</i>) és legnagyobb (<i>fit2</i>) elemeire hivatkozó iterátorokat.
<i>pair</i> < <i>fit1</i> , <i>fit2</i> > = <i>minmax_element</i> (<i>fitb</i> , <i>fite</i> , <i>compfv</i>)	

```

#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

```

```
int main() {
    vector<int> v { 12, 23, 7, 29, 10, 2, 11, 30, 12, 7 };
    auto par = minmax_element(begin(v), end(v));
    cout << "a legkisebb elem erteke: " << *par.first << endl;
    cout << "a legkisebb elem indexe: " << distance(begin(v), par.first) << endl;
    cout << "a legnagyobb elem erteke: " << *par.second << endl;
    cout << "a legnagyobb elem indexe: " << distance(begin(v), par.second) << endl;
    return 0;
}
```

```
a legkisebb elem erteke: 2
a legkisebb elem indexe: 5
a legnagyobb elem erteke: 30
a legnagyobb elem indexe: 7
```

2.5.5 Numerikus algoritmusok

A `<numeric>` deklarációs állományban találunk néhány, a tartományokat feldolgozó numerikus algoritmust. Az algoritmusok egy alapértelmezett műveletet vagy a megadott *binműv* függvényobjektumot használják. Kikötés a *binműv* funktorokra, hogy egyetlen iterátort és elemet se módosítsanak a megadott tartományon belül.

Algoritmus hívása	Leírás
<code>res = accumulate(itb, ite, initadat)</code> <code>res = accumulate(itb, ite, initadat, binműv)</code>	összegzi (+) az <i>initadat</i> értékét az <i>[itb, ite)</i> tartomány elemeivel, és visszatér az összeg értékével,
<code>res = inner_product(itb, ite, itb2, initadat)</code> <code>res = inner_product(itb, ite, itb2, initadat, binműv1, binműv2)</code>	összegzi (+) az <i>initadat</i> értékét az <i>[itb, ite)</i> és az <i>itb2</i> kezdetű tartomány elempárjainak szorzatértékével (*), és visszatér a számítás eredményével,
<code>oiter = partial_sum(itb, ite, oitb)</code> <code>oiter = partial_sum(itb, ite, oitb, binműv)</code>	az <i>[itb, ite)</i> tartomány egyre növekvő hosszúságú résztartományainak összegét (+) az <i>oitb</i> iterátorral kijelölt tartományba írja, a visszaadott érték kijelöli az új tartomány végét,
<code>oiter = adjacent_difference(itb, ite, oitb)</code> <code>oiter = adjacent_difference(itb, ite, oitb, binműv)</code>	az első elem átmásolását (<i>oitb</i>) követően, az <i>[itb, ite)</i> tartomány szomszédos elemeinek különbségeit (-) az (<i>oitb+1</i>) iterátorral kijelölt tartományba írja, a visszaadott érték kijelöli az új tartomány végét,
<code>iota(fitb, fite, adat)</code>	feltölti a <i>[fitb, fite)</i> tartományt az <i>adat, adat+1, adat+2, ...</i> számsor elemeivel.

Az `accumulate()` példában először az `iota()` algoritmus hívásával feltöltünk egy hételemű vektort 1-től 7-ig, majd meghatározzuk az elemek összegét, szorzatát és négyzetösszegét:

```
#include <iostream>
#include <vector>
#include <numeric>
#include <functional>
using namespace std;

int main() {
    vector<int> v(7);
    iota(begin(v), end(v), 1); //feltöltés 1, 2, 3, ..., 7 értékekkel
    for(int e : v)
        cout << e << " "; cout << endl;
    int osszeg = accumulate(v.begin(), v.end(), 0);
    int szorzat = accumulate(v.begin(), v.end(), 1, multiplies<int>());
    cout << "elemek osszege: " << osszeg << endl;
    cout << "elemek szorzata: " << szorzat << endl;
}
```

```

function<int(int, int)> sum2=
    [](int sum, int elem)->int {return sum + elem*elem;};
int negyzetosszeg = accumulate(v.begin(), v.end(), 0, sum2);
cout << "elemek negyzetosszege: " << negyzetosszeg << endl;
return 0;
}

```

```

1 2 3 4 5 6 7
elemek osszege: 28
elemek szorzata: 5040
elemek negyzetosszege: 1407

```

Az `inner_product()` algoritmus segítségével meghatározzuk két azonos méretű vektor skaláris szorzatát, megszámloljuk azokat az elempárokat, amelyek különböző adatokat tartalmaznak, végül pedig elkészítjük az elempárösszegek szorzatát:

```

#include <iostream>
#include <vector>
#include <numeric>
#include <functional>
using namespace std;

int main() {
    vector<int> va { 7, 3, 2, 3, 5, 1, 10};
    vector<int> vb { 5, 3, 2, 7, 5, 9, 2 };
    for(int e : va)
        cout << e << " "; cout << endl;
    for(int e : vb)
        cout << e << " "; cout << endl;
    int skalarszorzat = inner_product(begin(va), end(va),
                                    begin(vb), 0);
    cout << "a ket vektor skalarszorzata: " << skalarszorzat << endl;
    int kulonbozoek = inner_product(begin(va), end(va), begin(vb), 0,
                                    plus<int>(), not_equal_to<int>());
    cout << "a ket vektor azonos helyen allo, kulonbozo elemeinek szama: "
        << kulonbozoek << endl;
    // Az elempárösszegek szorzata
    int szorszum = inner_product(begin(va), end(va), begin(vb), 1,
                                [](int x, int y) {return x*y;},
                                [](int x, int y) {return x+y;});
    cout << "az elemparok osszegeinek szorzata: " << szorszum << endl;
    return 0;
}

```

```

7 3 2 3 5 1 10
5 3 2 7 5 9 2
a ket vektor skalarszorzata: 123
a ket vektor azonos helyen allo, kulonbozo elemeinek szama: 4
az elemparok osszegeinek szorzata: 3456000

```

A `partial_sum()` példaprogramban feltöltünk egy 9-elemű vektort 1-től 9-ig számokkal, majd egy másik vektorba bemásoljuk az elemek részösszegeit. A program másik részében helyettesítjük az eredeti vektor elemeit az elemek szorzataival, így faktoriális értékeket kapunk:

```

#include <iostream>
#include <vector>
#include <numeric>
#include <functional>
#include <iterator>
#include <algorithm>
using namespace std;

```



```

int main() {
    vector<int> v(9), w;
    iota(begin(v), end(v), 1);

    copy(begin(v), end(v), ostream_iterator<int> {cout, "\\t"}); cout << endl;
    partial_sum(begin(v), end(v), back_inserter(w));
    copy(begin(w), end(w), ostream_iterator<int> {cout, "\\t"}); cout << endl;
    cout << endl;

    copy(begin(v), end(v), ostream_iterator<int> {cout, "\\t"}); cout << endl;
    partial_sum(begin(v), end(v), begin(v), multiplies<int>());
    copy(begin(v), end(v), ostream_iterator<int> {cout, "\\t"}); cout << endl;
    return 0;
}

```

1	2	3	4	5	6	7	8	9
1	3	6	10	15	21	28	36	45
1	2	3	4	5	6	7	8	9
1	2	6	24	120	720	5040	40320	362880

Az `adjacent_difference()` algoritmus segítségével először meghatározzuk az első 10 négyzetszámból álló sorozat elemeinek különbségeit, és az eredményt egy másik vektorba töltjük. Ezt követően a vektort feltöltjük 1 értékű elemekkel, majd helyben lecseréljük az elemeket a Fibonacci számsor elemeivel:

```

#include <iostream>
#include <vector>
#include <numeric>
#include <functional>
#include <iterator>
#include <algorithm>
using namespace std;

int main() {
    vector<int> v {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}, w;
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    adjacent_difference(begin(v), end(v), back_inserter(w));
    copy(begin(w), end(w), ostream_iterator<int> {cout, " "}); cout << endl;
    cout << endl;

    fill(begin(v), end(v), 1);
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    adjacent_difference(begin(v), end(v)-1, begin(v)+1, plus<int>());
    copy(begin(v), end(v), ostream_iterator<int> {cout, " "}); cout << endl;
    return 0;
}

```

1	4	9	16	25	36	49	64	81	100
1	3	5	7	9	11	13	15	17	19
1	1	1	1	1	1	1	1	1	1
1	1	2	3	5	8	13	21	34	55

2.6 Helyfoglalás allokátorokkal

Az allokátorobjektumok *<memory>* a tárolók számára lefoglalt memóriaterületet kezelik. Az allokátorok lehetővé teszik, hogy egy konkrét környezet feltételeihez igazítsuk a memóriafoglalást, méghozzá úgy, hogy megőrizzük a konténerosztály interfészének hordozhatóságát.

Az allokátor-definíciókban megtalálhatjuk a *value_type*, *reference*, *size_type*, *pointer* és a *difference_type* típusokat. A legfontosabb allokátor-tagfüggvényeket az alábbi táblázatban foglaltuk össze:

Tagfüggvény	Leírás
<i>allocator</i> ()	konstruktorok és a destruktork,
<i>allocator</i> (const <i>allocator</i> & a)	
<i>~allocator</i> ()	
pointer <i>address</i> (reference r) const	r címével tér vissza,
pointer <i>allocate</i> (size_type n)	tárfoglalás n darab objektum számára,
void <i>deallocate</i> (pointer p, size_type n)	a lefoglalt memória felszabadítása,
size_type <i>max_size</i> () const	a <i>difference_type</i> legnagyobb értékével tér vissza.

Az alábbi példában saját allokátorsablont készítünk, és alkalmazzuk azt vektor létrehozása során.

```
#include <vector>
#include <cstdint>
#include <iostream>
#include <typeinfo>
#include <limits>
using namespace std;

template <class T> class allokator {
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;

    void construct(pointer p, const_reference val) {
        new (p) T(val);
    }

    void destroy(pointer p) {
        p->~T();
    }

    pointer allocate(size_type sz, void* vp = 0) {
        pointer p = static_cast<pointer> (::operator new(sz * sizeof(value_type)));
        cout << "allocate " << dec << sz << " darab " << typeid(T).name() << " elem, a "
              << hex << (unsigned long)p << " címen" << endl;
        return p;
    }

    void deallocate(pointer p, size_type) {
        cout << "deallocate " << (unsigned long)p << endl;
        ::operator delete(p);
    }

    size_type max_size() const {
        return numeric_limits<size_type>::max() / sizeof(T);
    }

    pointer address(reference r) { return &r; }
    const_pointer address(const_reference r) { return &r; }
};
```

```
int main() {
    // alapértelmezett allokátor használata
    vector<double> v(123, 12);
    // saját allokátor alkalmazása
    vector<int, allokator<int> > vv(100, 23);

    // a helyfoglaló alkalmazás dinamikus tömbhöz:
    allokator<double> dal;
    double *pd = dal.allocate(12);
    for (int i=0; i<12; i++)
        dal.construct(pd+i, i * i);
    for (int i=0; i<12; i++) {
        cout << pd[i] << " ";
        dal.destroy(pd+i);
    }
    cout << endl;
    dal.deallocate(pd, 12);
}
```

```
allocate 100 darab i elem, a 8b33c8 címen
allocate 12 darab d elem, a 8b1c50 címen
0 1 4 9 16 25 36 49 64 81 100 121
deallocate 8b1c50
deallocate 8b33c8
```

2.7 A konténerszerű osztályok használata

Az utolsó részben szereplő osztályok több-kevesebb sikerrel felelnek meg az STL konténerekre vonatkozó előírásoknak. Legfontosabb kérdés az, hogy készíthetők-e hozzájuk iterátorok, amely szükséges feltétele az algoritmusok használatának. További kérdés, hogy miként használhatjuk az objektumokat a programozás során.

2.7.1 A C++ nyelv hagyományos tömbjei

A C++ nyelv hagyományos tömbjei esetén a tömbelemek eléréséhez használt mutatók rendelkeznek a tetszőleges elérésű bejárók jellemzőivel, így a tömböket is felhasználhatjuk az STL algoritmusok hívásakor. Ebben segítségünkre lehet, hogy C++ könyvtárban megtalálhatók a [begin\(\)/end\(\)](#), a [cbegin\(\)/cend\(\)](#), az [rbegin\(\)/rend\(\)](#) valamint a [crbegin\(\)/crend\(\)](#) – ez utóbbi két páros a C++14-től kezdve – függvénysablonok tömbökre specializált változatai.

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <functional>
#include <cstdlib>
using namespace std;

int main() {
    int a[12] {0};
    if (begin(a)==a)
        cout << "begin(a)==a" << endl;
    if (end(a)==a+12)
        cout << "end(a)==a+12" << endl;
    iota(begin(a), end(a), -distance(begin(a), end(a))/2);
    copy(begin(a), end(a), ostream_iterator<int> {cout, " "}); cout << endl;
    random_shuffle(begin(a), end(a));
    cout << "osszekeveres: " << endl;
    copy(begin(a), end(a), ostream_iterator<int> {cout, " "}); cout << endl;
    auto par = minmax_element(begin(a), end(a));
    cout << "min: " << *par.first << "\t\tmax: " << *par.second << endl;
    int abssum = accumulate(begin(a), end(a), 0,
        [](int s, int e) { return s + abs(e); });
    cout << "az elemek abszolút értékeinek összege: " << abssum << endl;
    sort(begin(a), end(a), greater<int>());
    cout << "rendezés csokkeno sorrendbe: " << endl;
    copy(begin(a), end(a), ostream_iterator<int> {cout, " "}); cout << endl;
    if (binary_search(begin(a), end(a), 2, greater<int>()))
        cout << "2 megtalalhato az elemek kozott." << endl;
    return 0;
}
```

```
begin(a)==a
end(a)==a+12
-6 -5 -4 -3 -2 -1 0 1 2 3 4 5
osszekeveres:
4 -5 3 -4 -6 5 1 -3 -2 0 2 -1
min: -6          max: 5
az elemek abszolút értékeinek összege: 36
rendezés csokkeno sorrendbe:
5 4 3 2 1 0 -1 -2 -3 -4 -5 -6
2 megtalalhato az elemek kozott.
```

A továbbiakban nemcsak azt vizsgáljuk meg, hogy az adott osztály esetén használhatunk-e algoritmusokat, hanem röviden be is mutatjuk az adott típus alkalmazási lehetőségeit.

2.7.2 Sztringek

A C++11 nyelv karaktersorozat (sztring) osztályai a tárolt karakterek típusától eltekintve azonos programozási felülettel rendelkeznek. A különböző sztringosztályok a **basic_string** osztálysablon **char**, **wchar_t**, **char16_t** és **char32_t** típusokkal elkészített példányai, melyek eléréséhez a **<string>** fejléclományt kell a programunkba beépíteni.

Sztringtípus	Definíció
string	<code>basic_string<char></code>
wstring	<code>basic_string<wchar_t></code>
u16string	<code>basic_string<char16_t></code>
u32string	<code>basic_string<char32_t></code>

A következőkben a **char** típusú elemek vektoraként megvalósított **string** típussal ismerkedünk meg részletesebben.

2.7.2.1 A string és a vector<char> típusok azonos tagfüggvényei

A **vector** konténer tagfüggvényeinek egy része sztringek esetén is elérhető. Ezekkel specifikus műveleteket végezhetünk, illetve algoritmusokat hívhatunk. A leírásokban az *str* névvel egy **string** típusú változót jelölünk, és **bordó** színnel emeljük ki azt a sztringspecifikus műveletet, amely rokon egy másik tagfüggvénnyel.

Tagfüggvények hívása	Rövid leírás
Iterátorok lekérdezése:	
<code>str.begin()</code> / <code>str.end()</code>	<i>iterator</i> ,
<code>str.cbegin()</code> / <code>str.cend()</code>	const <i>iterator</i> ,
<code>str.rbegin()</code> / <code>str.rend()</code>	<i>reverse_iterator</i> ,
<code>str.crbegin()</code> / <code>str.crend()</code>	const <i>reverse_iterator</i> ,
Karakterek elérése:	
<code>char& chn = str[n]</code>	indexelés indexhatár-ellenőrzés nélkül,
<code>char& chn = str.at(n)</code>	indexelés indexhatár-ellenőrzéssel,
<code>char& chelso = str.front()</code>	a 0 indexű karakter,
<code>char& chutolso = str.back()</code>	az utolsó karakter,
<code>const char* peleje = str.data()</code>	a tárolt karaktersorozat kezdetének mutatója – nem garantált a záró 0-ás bájtt jelenléte,
<code>const char* peleje = str.c_str()</code>	a tárolt karaktersorozat kezdetének mutatója a karaktereket 0-ás bájtt zárja,
Tárolóterület kezelése:	
<code>bool b = str.empty()</code>	üres-e a sztring, mint az <code>str==""</code> vizsgálat,
<code>size_t n = str.size()</code>	a tárolt karakterek száma,
<code>size_t n = str.length()</code>	a tárolt karakterek száma,
<code>size_t n = str.max_size()</code>	az adott környezetben tárolható leghosszabb sztring mérete,
<code>size_t n = str.capacity()</code>	a sztring számára lefoglalt tárolóterület mérete,
<code>str.reserve(n)</code>	<i>n</i> bájt méretű tároló terület foglalása a sztring számára,
<code>str.shrink_to_fit()</code>	a <code>capacity()</code> lecsökkentése a <code>size()</code> méretre,

Tagfüggvények hívása (folytatás)	Rövid leírás
Sztringek módosítása:	
<code>str.clear()</code>	a sztring karaktereinek eltávolítása – eredménye üres sztring,
<code>str.push_back(ch)</code>	karakter hozzáfűzése a sztring végéhez,
<code>str.pop_back()</code>	az utolsó karakter eltávolítása,
<code>str.resize(n)</code>	a sztring átméretezése <i>n</i> méretűre, ha <i>n</i> > <code>size()</code> , szóközzel, illetve
<code>str.resize(n, ch)</code>	<i>ch</i> karakterekkel tölti fel az új pozíciókat,
<code>str.swap(masikstr)</code>	a sztring felcserélése a <i>masikstr</i> tartalmával,
<code>swap(str1, str2)</code>	a megadott két sztring tartalmának felcserélése a <code>swap()</code> algoritmus specializált változatával.

Mivel ezek a tagfüggvények már ismertek a korábbi részekből, nem hozunk példákat a használatukra. A következő alfejezetben olyan speciális megoldásokkal foglalkozunk, amelyek a sztringek és a karakterek (**char**) mellett, a sztringet tároló karakter tömbökkel (**const char ***) is képesek műveleteket végezni.

2.7.2.2 A string típus speciális tagfüggvényei

A konténereknél látott konstruálási lehetőségek mellett, további kezdőérték-adási megoldások is rendelkezésünkre állnak. (A táblázatban csak az alapértelmezett helyfoglalóval rendelkező konstruktorok szerepelnek.)

A felsorolásban az **str string** típusú változót jelöl, míg a **cstr** C-stílusú karaktersorozat, amely egyaránt lehet **char** típusú tömbben, illetve karaktersorozat konstansban (literálban): "szöveg". A karakterpozíciót kijelölő **poz** és a karakterszámot megadó **n size_t** típusúak, a **ch** pedig **char** típusú karakter.

Konstruktorhívások	Rövid leírás
<code>string str</code> <code>string str {}</code>	üres sztringet létrehozó alapértelmezett konstruktor,
<code>string str(cstr)</code> <code>string str {cstr}</code>	sztring inicializálása a cstr tartalmával,
<code>string str(cstr, n)</code> <code>string str {cstr, n}</code>	sztring inicializálása a cstr első <i>n</i> elemével,
<code>string str(str0)</code> <code>string str {str0}</code>	sztring inicializálása az <i>str0</i> sztring tartalmának átmásolásával,
<code>string str(move(str0))</code> <code>string str {move(str0)}</code>	sztring inicializálása az <i>str0</i> sztring tartalmának áthelyezésével,
<code>string str(str0, poz)</code> <code>string str {str0, poz}</code>	sztring inicializálása az str0 karaktereinek átmásolásával az adott poz pozíciótól a sztring végéig (<code>string::npos</code>),
<code>string str(str0, poz, n)</code> <code>string str {str0, poz, n}</code>	sztring inicializálása az str0 adott poz pozíción kezdődő, <i>n</i> darab karakterének átmásolásával – ha <i>n</i> túl nagy, a sztring végéig másol,
<code>string str(n, ch)</code>	sztring inicializálása <i>n</i> darab ch karakterből álló sztringgel,
<code>string str {ch₁, ch₂, ..., ch_n}</code>	sztring inicializálása a karakterlistából előállított sztringgel,
<code>string str(itb, ite)</code> <code>string str {itb, ite}</code>	sztring inicializálása az [itb, ite) tartomány karaktereivel.

A különböző konstruktorhívásokat az alábbi példaprogramban fogtuk csokorba:

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    char cstr[] = "Abrakadabra";
    string s1 {cstr};
    cout << "s1: " << s1 << endl;
    string s2 {"C++ programozas 30 eve", 15};
    cout << "s2: " << s2 << endl;
    string s3 {move(s2)};
    cout << "s3: " << s3 << endl;
    cout << "s2: " << s2 << endl;
    string s4 {s3, 4};
    cout << "s4: " << s4 << endl;
    string s5 {s3, 4, 7};
    cout << "s5: " << s5 << endl;
    string s6 (12, 'N');
    cout << "s6: " << s6 << endl;
    string s7 {'C', '+', '+'};
    cout << "s7: " << s7 << endl;
    string s8 {begin(s3), end(s3)-4};
    cout << "s8: " << s8 << endl;
    return 0;
}
```

```
s1: Abrakadabra
s2: C++ programozas
s3: C++ programozas
s2:
s4: programozas
s5: program
s6: NNNNNNNNNNNN
s7: C++
s8: C++ program
```

A **string** típus további különlegessége, hogy a példányai közvetlenül kaphatnak értéket *input* adatfolyamokból, illetve *output* adatfolyamokba írhatjuk a tartalmukat. A műveleteket az **<iostream>** állomány deklarálja:

I/O műveletek	Rövid leírás
<code>getline(input, str)</code>	karaktereket olvas az <i>str</i> -be a '\n' (újsor – enter) karakterig, ami nem kerül be a sztringbe,
<code>getline(input, str, tagolóch)</code>	karaktereket olvas az <i>str</i> -be a tagolóch karakterig, ami nem kerül be a sztringbe,
<code>input >> str</code>	karaktereket olvas az <i>str</i> sztringbe az első tagoló karakterig; a tagoló karakter is beolvasódik, azonban nem kerül be a sztringbe; ilyen tagoló karakter a szóköz is!
<code>output << str</code>	a sztring tartalmának kiírása a megadott adatfolyamba.

Az alábbi példában egy teljes nevet olvasunk be, majd pedig szóközzel tagolt szavakat dolgozunk fel egészen a <Ctrl+Z> és <Enter> billentyűk lenyomásáig.

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main() {
    string s;
    cout << "Nev: ";
    getline(cin, s);
    cout << s << endl;
    while (cin >> s)
        cout << s << endl;
    return 0;
}
```

```
Nev: Bjarne Stroustrup↵
Bjarne Stroustrup
alfa beta gamma delta iota↵
alfa
beta
gamma
delta
iota
^Z↵
```

Az **értékadás** szokásos módjait (*operator=()* és *assign()* tagfüggvények) is használhatjuk a **string** típusú változók esetén. Az értékadás jobb oldalán sztring, move(sztring), C-stílusú sztring, karakter és karaktereket tartalmazó inicializációs lista egyaránt szerepelhet. Az *assign()* tagfüggvényt pedig a sztring típus konstruktorainak megfelelő módon paraméterezhetjük.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    char cstr[] = "Abrakadabra";
    string s1, s2, s3, s4, s5, s6, s7, s8;
    s1 = cstr;
    cout << "s1: " << s1 << endl;
    s1 = 'C';
    cout << "s1: " << s1 << endl;
    s2 = move(s1);
    cout << "s2: " << s2 << endl;
    s1 = {'C', '+', '+', 32, '3', ' ', '0', 32, 'e', 'v', 'e'};
    cout << "s1: " << s1 << endl;
    cout << endl;
    s1.assign(cstr);
    cout << "s1: " << s1 << endl;
    s2.assign("C++ programozas 30 eve", 15);
    cout << "s2: " << s2 << endl;
    s3.assign(move(s2));
    cout << "s3: " << s3 << endl;
    // C++14-től: s4.assign(s3, 4);
    // cout << "s4: " << s4 << endl;
    s5.assign(s3, 4, 7);
    cout << "s5: " << s5 << endl;
    s6.assign(12, 'N');
    cout << "s6: " << s6 << endl;
    s7.assign({'C', '+', '+'});
    cout << "s7: " << s7 << endl;
    s8.assign(begin(s3), end(s3)-4);
    cout << "s8: " << s8 << endl;
    return 0;
}
```



```

s1: Abrakadabra
s1: C
s2: C
s1: C++ 30 eve

s1: Abrakadabra
s2: C++ programozas
s3: C++ programozas
s5: program
s6: NNNNNNNNNNNN
s7: C++
s8: C++ program

```

A sztingekkel végezhető **műveletek** egy részét a **string** osztály tagfüggvényei valósítják meg, másokat azonban külső függvények hívásával érhetünk el. A tagfüggvények többsége magának a sztring objektumnak a referenciájával tér vissza – az ettől eltérő eseteket a leírásban jelezzük. A művelet elvégzésénél általában választhatunk a karaktertömbös és az iterátoros felfogás között.

Sztringműveletek	Rövid leírás
<pre> rstr = str.insert(index, n, ch) rstr = str.insert(index, cstr) rstr = str.insert(index, str0) rstr = str.insert(index, str0, poz, n) rstr = str.insert(index, str0, poz) C++14 rstr = str.insert(index, { ch₁, ch₂, ..., ch_n }) str.insert(itpoz, ch) str.insert(itpoz, n, ch) str.insert(itb, ite) str.insert(itpoz, { ch₁, ch₂, ..., ch_n }) </pre>	karakterek beszúrása az index pozíciótól kezdve – a forrás helyét különböző módon adhatjuk meg, karakter(ek) beszúrása az itpoz iterátorral kijelölt pozíciótól kezdődően,
<pre> rstr = str.erase() rstr = str.erase(index) rstr = str.erase(index, n) riter = str.erase(itpoz) riter = str.erase(itb, ite) </pre>	str karaktereinek törlése, az eredmény egy üres sztring, str karaktereinek törlése az adott indexű karaktertől a sztring végéig, az adott indexű karaktertől n darab karakter törlése, az itpoz iterátorral kijelölt karakter törlése; a visszaadott iterátor az utolsó törölt utáni karakterre hivatkozik, az [itb, ite) tartomány karaktereinek törlése; a visszaadott iterátor az utolsó törölt utáni karakterre hivatkozik,
<pre> rstr = str.append(n, ch) rstr = str.append(cstr) rstr = str.append(cstr, n) rstr = str.append(str0) rstr = str.append(str0, poz, n) rstr = str.append(str0, poz) C++14 rstr = str.append(itb, ite) rstr = str.append({ ch₁, ch₂, ..., ch_n }) </pre>	karakterek hozzáfűzése az str sztringhez – a forrás helyét különböző módon adhatjuk meg,
<pre> str += str0 str += ch str += cstr str += { ch₁, ch₂, ..., ch_n } </pre>	karakterek hozzáfűzése az str sztringhez a += operátorral – a forrás helyét különböző módon adhatjuk meg,

Sztringműveletek (folytatás)	Rövid leírás
<pre> rstr = str1 + str2 rstr = str1 + ch rstr = ch + str1 rstr = str1 + cstr rstr = cstr + str1 </pre>	a <code>+=</code> művelethez hasonló eredményre jutunk a <code>+</code> külső operátorral, amelynek bármelyik operandusa string típusú, az összekapcsolás eredménye is az lesz; megjegyezzük, hogy a <code>move(str1)</code> és/vagy a <code>move(str2)</code> is használható,
<pre> rstr = str.replace (poz, n, str0) rstr = str.replace (itb, ite, str0) rstr = str.replace (poz, n, str0, poz0, n0) rstr = str.replace (poz, n, str0, poz0) C++14 rstr = str.replace (itb, ite, itb0, ite0) rstr = str.replace (poz, n, cstr) rstr = str.replace (itb, ite, cstr) rstr = str.replace (poz, n, cstr, n0) rstr = str.replace (itb, ite, cstr, n0) rstr = str.replace (poz, n, n0, ch) rstr = str.replace (itb, ite, n0, ch) rstr = str.replace (itb, ite, {ch1, ch2, ... }) </pre>	az <code>str</code>-ben lecseréli a <code>[poz, poz+n)</code>, illetve az <code>[itb, ite)</code> karaktereket a megadott karaktársorozat karaktereivel,

A különböző sztringkezelő műveleteket az alábbi példaprogrammal szemléltetjük:

```

#include <iostream>
#include <string>
using namespace std;

int main() {
    char cszul[] = "1985. október 14.";
    string str;
    str = string("A C++ nyelv születési ideje") + '\t' + cszul;
    cout << str << endl;

    // Összerakjuk az egyre növekvő számú 'B' illetve 'C' betűkből álló
    // sztringet
    str.clear();
    for (int i=0; i<7; i++)
        str.append(i+1, {'B'+ (i % 2)});
    cout << "append: " << str << endl;

    // Pozíciószámítással minden 'B' karakterrel kezdődő sztringrész
    // elé 'A' karaktert szúrunk be
    size_t m = str.size()/4;
    for (size_t i=0, j=1; i<=m; i+=j++)
        str.insert(4*i, 1, 'A');
    cout << "insert: " << str << endl;

    // Az "AB" karaktereket "CC"-re cseréljük
    size_t poz;
    while ((poz = str.find("AB")) != string::npos)
        str.replace(poz, 2, "CC");
    cout << "replace: " << str << endl;

    // Töröljük a 'B' karaktereket tartalmazó sztringrészeket
    size_t n = 0;
    do {
        poz = str.find('B');
        if (poz != string::npos)
            str.erase(poz, n+=2);
    } while (poz != string::npos);
    cout << "erase: " << str << endl;
    return 0;
}

```

A C++ nyelv születési ideje 1985. október 14.
 append: BCCBBBCCCCBBBBCCCCCCCCBBBBBBB
 insert: ABCCABBBCCCCABBBBCCCCCABBBBBBB
 replace: CCCCCBCCCCCCCCBBBBCCCCCCCCBBBBBB
 erase: CCCCCCCCCCCCCCCCCC

Külön csoportba gyűjtöttük azokat a műveleteket, amelyek **részstringekkel** (*substring*) kapcsolatosak.

Sztringműveletek	Rövid leírás
<code>m = str.copy(chptr, n, poz)</code> <code>m = str.copy(chptr, n)</code>	kimásolja az <i>str</i> sztring [<i>poz</i> , <i>poz+n</i>), illetve [<i>0</i> , <i>n</i>), karaktereit a char* típusú <i>chptr</i> mutatóval kijelölt területre, és visszatér az átmásolt karakterek számával; amennyiben <i>poz</i> >= <i>size()</i> , out_of_range kivétel lép fel,
<code>rstr = str.substr(poz, n)</code> <code>rstr = str.substr(poz)</code> <code>rstr = str.substr()</code>	új sztringben visszaadja az <i>str</i> sztring [<i>poz</i> , <i>poz+n</i>), [<i>0</i> , <i>n</i>), illetve [<i>0</i> , <i>size()</i>) részét ; ha <i>poz</i> >= <i>size()</i> , out_of_range kivétel keletkezik,
<code>ires = str.compare(str0)</code> <code>ires = str.compare(p, n, str0)</code> <code>ires = str.compare(p1, n1, str0, p2, n2)</code> <code>ires = str.compare(p1, n1, str0, p2) C++14</code> <code>ires = str.compare(cstr)</code> <code>ires = str.compare(p, n, cstr)</code> <code>ires = str.compare(p, n1, cstr, n2)</code>	az <i>str</i> -t, illetve annak egy részét lexikografikusan összehasonlítja a zölddel kiemelt sztringgel, illetve annak egy részével; a függvényérték jelzi a sztringek viszonyát: ires < 0 , ez első sztring kisebb a másodiknál, ires == 0 , ez első sztring azonos a másodikkal, ires > 0 , ez első sztring nagyobb a másodiknál.

Felhívjuk a figyelmet arra, hogy a **copy()** tagfüggvény nem helyez nullás bájtot az átmásolt karakterek után – erről magunknak kell gondoskodnunk a visszaadott függvényérték felhasználásával. A **részstringek kezelését az alábbi példaprogramban követhetjük nyomon:**

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main() {
    string str = "VAKACIO";
    cout << str.substr() << endl;
    cout << "1." << endl;
    for (int n=str.size()-1; n>=0; n--) {
        cout << str.substr(n) << endl;
    }
    cout << "2." << endl;
    for (int n=1, poz= str.size()/2; poz>=0; n+=2, poz--) {
        cout << str.substr(poz, n) << endl;
    }
    cout << "3." << endl;
    char cs[12];
    int n = str.copy(cs, 3, 4);
    cs[n] =0; // a sztringet záró 0-ás bájtt
    cout << cs << endl;
    cout << "4." << endl;
    char *cstr = new char[str.size()+1];
    fill(cstr, cstr+str.size(), 0); // a sztringet záró 0-ás bájtt miatt
    for (size_t poz=0, cpoz=str.size()-1; poz<str.size(); poz++, cpoz--) {
        str.copy(cstr+cpoz, 1, poz);
    }
    cout << cstr << endl;
    delete[] cstr;
}
```

```

cout << "5." << endl;
cout << str.compare(str) << endl;           // "VAKACIO" == "VAKACIO"
cout << str.compare("vakacio") << endl;      // "VAKACIO" < "vakacio"
cout << str.compare(0, 3, "VAK") << endl;    // "VAK" == "VAK"
cout << str.compare(0, 4, "AKACFA", 0, 4) << endl; // "VAKA" > "AKAC"
cout << str.compare(0, 2, "VAKACIO", 0, 5) << endl; // "VA" < "VAKAC"
cout << str.compare(0, 5, "VAKACIO", 0, 2) << endl; // "VAKAV" > "VA"
return 0;
}

```

VAKACIO

1.

0

IO

CIO

ACIO

KACIO

AKACIO

VAKACIO

2.

A

KAC

AKACI

VAKACIO

3.

CIO

4.

OICAKAV

5.

0

-1

0

1

-3

3

A **string** osztály gazdag **részstringet kereső** függvénykészlettel rendelkezik. A hagyományos **find()** a karaktersorozat elejétől (**str.begin()**) keres, előre haladva, míg az **rfind()** az **str.end()**-től visszafelé halad. A kereső függvények sikertelen esetben a **string::npos** (*not a position*) értékkel térnek vissza.

Kereső műveletek	Rövid leírás
<code>rpoz = str.find(str0)</code> <code>rpoz = str.find(str0, poz)</code> <code>rpoz = str.find(cstr)</code> <code>rpoz = str.find(cstr, poz)</code> <code>rpoz = str.find(cstr, poz, n)</code> <code>rpoz = str.find(ch)</code> <code>rpoz = str.find(ch, poz)</code>	az <i>str</i> sztringben (adott <i>pozíciótól</i>) keresi a megadott <i>sztring</i> , <i>C-sztring</i> , <i>C-sztring</i> első <i>n</i> karakterének vagy egy karakter első előfordulását; a visszatérési érték a találat kezdőpozíciója,
<code>rpoz = str.rfind(str0)</code> <code>rpoz = str.rfind(str0, poz)</code> <code>rpoz = str.rfind(cstr)</code> <code>rpoz = str.rfind(cstr, poz)</code> <code>rpoz = str.rfind(cstr, poz, n)</code> <code>rpoz = str.rfind(ch)</code> <code>rpoz = str.rfind(ch, poz)</code>	az <i>str</i> sztringben (adott <i>pozíciótól</i>) visszafelé keresi a megadott <i>sztring</i> , <i>C-sztring</i> , <i>C-sztring</i> első <i>n</i> karakterének vagy egy karakter első előfordulását; a visszatérési érték a találat kezdőpozíciója.

A `find_xxx_of()` tagfüggvényekkel **karaktereket kereshetünk** a sztringben.

Kereső műveletek	Rövid leírás
<code>rpoz = str.find_first_of(str0)</code> <code>rpoz = str.find_first_of(str0, poz)</code> <code>rpoz = str.find_first_of(cstr)</code> <code>rpoz = str.find_first_of(cstr, poz)</code> <code>rpoz = str.find_first_of(cstr, poz, n)</code> <code>rpoz = str.find_first_of(ch)</code> <code>rpoz = str.find_first_of(ch, poz)</code>	az <code>str</code> sztringben (adott pozíciótól) keresi az első olyan karaktert , amely benne van a megadott sztringben , C-sztringben , C-sztring első n karakterében vagy megegyezik a megadott karakterrel ; a visszatérési érték a találat kezdőpozíciója,
<code>rpoz = str.find_first_not_of(mint fent)</code>	az <code>str</code> sztringben keresi az első olyan karaktert , amely nincs benne a fentieknek megfelelően megadott karakter-sorozatban,
<code>rpoz = str.find_last_of(mint fent)</code>	az <code>str</code> sztringben visszafelé keresi az utolsó olyan karaktert , amely benne van a fentieknek megfelelően megadott karaktersorozatban,
<code>rpoz = str.find_last_not_of(mint fent)</code>	az <code>str</code> sztringben visszafelé keresi az utolsó olyan karaktert , amely nincs benne a fentieknek megfelelően megadott karaktersorozatban.

A következő példában egy írásjelekkel tagolt mondatot az írásjelek mentén szavakra bontunk, majd pedig összerakunk:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    string mondat = "A Cplusplus 30 éves! 1985. október 14-en jelent meg";
    mondat += " a Cplusplus nyelv első kiadásának kereskedelmi"
              " fordítóprogramja.";

    // felbontjuk a mondatot szavakra írásjeleknél vágva
    vector<string> szavak;
    int poz = 0, rpoz = 0;
    char irasjelek[] = " ,.!?";
    while (rpoz < mondat.size()) {
        poz = mondat.find_first_of(irasjelek, rpoz);
        szavak.push_back(mondat.substr(rpoz, poz-rpoz));
        rpoz = poz + 1;
    }

    // összerakjuk a mondatot szócsereivel
    mondat=szavak[0];
    for (int i=1; i<szavak.size(); i++) {
        if (!szavak[i].compare("Cplusplus"))
            mondat += " C++" ;
        else if (!szavak[i].empty())
            mondat += ' ' + szavak[i];
    }
    mondat += '!';
    cout<<mondat<<endl;
    return 0;
}
```

A C++ 30 éves 1985 október 14-en jelent meg a C++ nyelv első kiadásának kereskedelmi fordítóprogramja!

2.7.2.3 Összehasonlító külső sztringműveletek

Nem tagfüggvény operátor függvénysablonok segítik két **string** típusú sztring, illetve egy C-sztring és egy **string** típusú sztring lexikografikus összehasonlítását. (Ez utóbbi esetben az operandusok sorrendje tetszőleges lehet.) A műveletek **bool** típusú értékkel jelzik a relációk igazságtartalmát.

Összehasonlító műveletek	Rövid leírás
<code>b = str1 == str2</code> <code>b = str == cstr</code> <code>b = cstr == str</code>	a lexikografikus egyenlő reláció vizsgálata,
<code>!=</code> <code><</code> <code><=</code> <code>></code> <code>>=</code>	további relációs műveletek.

A relációs műveleteket a `szavak` sztringtömb egyszerű sorbarendezés során alkalmazzuk:

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main() {
    string szavak[] {"ADA", "C++", "C#", "FORTRAN", "BASIC", "JAVA", "F#", "C"};
    int meret = distance(begin(szavak), end(szavak));
    for(int i = 0; i < meret - 1; i++)
        for (int j = i + 1; j < meret; j++)
            if (szavak[i] > szavak[j])
                swap(szavak[i], szavak[j]);
    for (const auto & szo : szavak)
        cout << szo << endl;
    cout << endl;
    return 0;
}
```

```
ADA
BASIC
C
C#
C++
F#
FORTRAN
JAVA
```

2.7.3 Bitkészletek – *bitset*

Az alacsony szintű programozás során gyakran használunk kétállapotú állapotjelzőket, a rendszer állapotainak (jó/rossz, igaz/hamis, be/ki stb.) tárolására. Az alap C++ nyelv (a C nyelvtől örökölt) bitenkénti operátorok és struktúrába ágyazott bitmezőkkel segíti az ilyen jellegű programok készítését.

A C++ STL-ben megtalálható **bitset** osztállysablon (`<bitset>` header) egységessé teszi a rövidebb, hosszabb (azonban példányosításkor rögzített méretű) bitkészletek kezelését. A `bitset<N>` osztály egy olyan tömb, ami *N* bitet tárol.

A **bitset** sokban különbözik az STL konténerektől, nincsenek iterátorai, `out_of_range` kivétellel jelzi az indexhatárok túllépését stb. A bitpozíciókat jobbról balra indexeljük `[0, N)` tartományban. A bitkészletet *N*-jegyű bináris számként képzelhetjük el legjobban.

Például, a `bitset<12> bs(2015)` bitkészlet:

11.	10.	9.	8.	7.	6.	5.	4.	3.	2.	1.	0.
0	1	1	1	1	1	0	1	1	1	1	1

2.7.3.1 Bitkészletek létrehozása

A `bitset` sokféle konstruktort biztosít számunkra. Inicializálhatjuk a bitkészletet egész számmal, sztringgel, ami nem feltétlenül 0 és 1 karaktereket tartalmaz.

Konstruktorhívások	Rövid leírás
<code>bitset<N> bs</code> <code>bitset<N> bs {}</code>	N darab 0-ás bittel feltöltött bitkészlet,
<code>bitset<N> bs(num)</code> <code>bitset<N> bs {num}</code>	az <code>unsigned long long</code> típusú <code>num</code> szám bitjeivel inicializált bitkészlet,
<code>bitset<N> bs(str)</code> <code>bitset<N> bs {str}</code>	a 0 és 1 karaktereket tartalmazó <code>sztring</code> alapján inicializált bitkészlet,
<code>bitset<N> bs(str, p)</code> <code>bitset<N> bs {str, p}</code>	a 0 és 1 karaktereket tartalmazó <code>sztring p</code> . pozíciójától a sztring végéig terjedő részsstring alapján inicializált bitkészlet,
<code>bitset<N> bs(str, p, n)</code> <code>bitset<N> bs {str, p, n}</code>	a 0 és 1 karaktereket tartalmazó <code>sztring p</code> . pozícióján kezdődő, n darab karakterből álló része alapján inicializált bitkészlet,
<code>bitset<N> bs(str, p, n, ch0)</code> <code>bitset<N> bs {str, p, n, ch0}</code>	a <code>ch0</code> és 1 karaktereket tartalmazó <code>sztring p</code> . pozícióján kezdődő, n darab karakterből álló része alapján inicializált bitkészlet,
<code>bitset<N> bs(str, p, n, ch0, ch1)</code> <code>bitset<N> bs {str, p, n, ch0, ch1}</code>	a <code>ch0</code> és <code>ch1</code> karaktereket tartalmazó <code>sztring p</code> . pozícióján kezdődő, n darab karakterből álló része alapján inicializált bitkészlet,
<code>bitset<N> bs(cstr)</code> <code>bitset<N> bs {cstr}</code>	a 0 és 1 karaktereket tartalmazó <code>C-sztring</code> alapján inicializált bitkészlet,
<code>bitset<N> bs(cstr, n)</code> <code>bitset<N> bs {cstr, n}</code>	a 0 és 1 karaktereket tartalmazó <code>C-sztring</code> elejétől az n . pozícióig terjedő részsstring alapján inicializált bitkészlet,
<code>bitset<N> bs(cstr, n, ch0)</code> <code>bitset<N> bs {cstr, n, ch0}</code>	a <code>ch0</code> és 1 karaktereket tartalmazó <code>C-sztring</code> elejétől az n . pozícióig terjedő részsstring alapján inicializált bitkészlet,
<code>bitset<N> bs(cstr, n, ch0, ch1)</code> <code>bitset<N> bs {cstr, n, ch0, ch1}</code>	a <code>ch0</code> és <code>ch1</code> karaktereket tartalmazó <code>C-sztring</code> elejétől az n . pozícióig terjedő részsstring alapján inicializált bitkészlet.

Az alábbi példaprogramban különböző konstruktorhívásokat használunk:

```
#include <bitset>
#include <string>
#include <iostream>
using namespace std;

int main() {
    // üres konstruktor
    bitset<8> bs1;
    cout << "bs1 : " << bs1 << endl;

    // unsigned long long argumentumú konstruktorok
    bitset<8> bs2(123ull);
    cout << "bs2 : " << bs2 << endl;
    bitset<16> bs3(0xffull);
    cout << "bs3 : " << bs3 << endl;

    // string argumentumú konstruktorok
    bitset<8> bs4(string("11001010"));
    cout << "bs4 : " << bs4 << endl;
    bitset<8> bs5(string("11001010"), 2);
    cout << "bs5 : " << bs5 << endl;
    bitset<8> bs6(string("11001010"), 2, 5);
    cout << "bs6 : " << bs6 << endl;
}
```

```

bitset<16> bs7(string("CXXCXXCXXCXXCCCC"), 0, 16, 'C', 'X');
cout << "bs7 : " << bs7 << endl;

// C-sztring argumentumú konstruktorok
bitset<8> bs8("01101001");
cout << "bs8 : " << bs8 << endl;
bitset<8> bs9("01101001", 5);
cout << "bs9 : " << bs9 << endl;
bitset<8> bs10("XXXYYYYY", 8, 'X', 'Y');
cout << "bs10: " << bs10 << endl;
return 0;
}

```

```

bs1 : 00000000
bs2 : 01111011
bs3 : 0000000011111111
bs4 : 11001010
bs5 : 00001010
bs6 : 00000101
bs7 : 0110111011110000
bs8 : 01101001
bs9 : 00001101
bs10: 00001111

```

Természetesen, az **azonos méretű** bitkészletek esetén az alapértelmezett **másoló konstruktor**, illetve az **értékadás** operátorával elvégzett másolás művelete is használhatók.

2.7.3.2 Bitkészlet-műveletek

A **bitset** típushoz tagfüggvények és külső operátorfüggvények egyaránt rendelkezésünkre állnak. A tagfüggvények nagy része a **bitset<N>** objektum (*bs*) referenciájával tér vissza, hogy a hívásokat egymásba építhessük.

Bitenkénti műveletek	Rövid leírás
bool <i>bn</i> = <i>bs</i> [<i>index</i>]	a <i>bs</i> <i>index</i> . bitjének lekérdezése, illetve felülírása, kivétel dobása nélkül,
<i>bs</i> [<i>index</i>] = true (1) vagy false (0)	
bool <i>bn</i> = <i>bs</i> .test(<i>index</i>)	a <i>bs</i> <i>index</i> . bitjének lekérdezése; <i>out_of_range</i> kivétel keletkezik, ha túllépjük az indexhatárokat,
<i>rbs</i> = <i>bs</i> .set()	<i>bs</i> minden bitjét 1-be állítja,
<i>rbs</i> = <i>bs</i> .set(<i>index</i> , <i>adat</i>)	<i>bs</i> [<i>index</i>] = <i>adat</i> , ahol az <i>adat</i> true (1) vagy false (0),
<i>rbs</i> = <i>bs</i> .reset()	<i>bs</i> minden bitjét 0-ba állítja,
<i>rbs</i> = <i>bs</i> .reset(<i>index</i>)	<i>bs</i> [<i>index</i>] = 0,
<i>rbs</i> = <i>bs</i> .flip()	<i>bs</i> minden bitjét ellentettjére változtatja: <i>bs</i> [<i>i</i>] = ~ <i>bs</i> [<i>i</i>],
<i>rbs</i> = <i>bs</i> .flip(<i>index</i>)	<i>bs</i> [<i>index</i>] = ~ <i>bs</i> [<i>index</i>],
<i>db</i> = <i>bs</i> .count()	visszatér a true (1) értékű bitek számával a bitkészletben,
bool <i>b</i> = <i>bs</i> .all()	igaz értékkel tér vissza, ha a bitkészletben minden bit 1 értékű,
bool <i>b</i> = <i>bs</i> .any()	igaz értékkel tér vissza, ha a bitkészletben legalább egy bit 1 értékű,
bool <i>b</i> = <i>bs</i> .none()	igaz értékkel tér vissza, ha a bitkészletben nincs 1 értékű bit,
<i>db</i> = <i>bs</i> .size()	visszatér a bitkészlet méretével.

A következőkben azokat a műveleteket vesszük sorra, melyek során az azonos méretű, alap- és egy másik **bitset** objektum megfelelő indexű bitjei között hajtódnak végre a bináris logikai műveletek,

melyek eredménye szerint változik meg az alapobjektum. Ugyancsak ebbe a csoportba soroltuk az alapobjektumra vonatkozó egyéb belső és külső műveleteket is.

<i>Bitkészlet-műveletek</i>	<i>Rövid leírás</i>
<code>bs &= masikbs</code>	bináris ÉS művelet,
<code>bs = masikbs</code>	bináris VAGY művelet,
<code>bs ^= masikbs</code>	bináris KIZÁRÓ VAGY művelet,
<code>rbs = bs~</code>	a <code>bs</code> másolatán elvégzi a bináris NEM művelet (<code>flip()</code>), és visszatér a másolattal,
<code>bs <<= n</code>	eltolja a <code>bs</code> bitjeit balra <code>n</code> pozícióval, és 0-ás bitek jönnek be jobbról,
<code>rbs = bs << n</code>	a <code>bs</code> másolatán elvégzi a biteltolás balra műveletet, és visszatér a másolattal,
<code>bs >>= n</code>	eltolja a <code>bs</code> bitjeit jobbra <code>n</code> pozícióval, és 0-ás bitek jönnek be balról,
<code>rbs = bs >> n</code>	a <code>bs</code> másolatán elvégzi a biteltolás jobbra műveletet, és visszatér a másolattal,
<code>bool b = bs == masikbs</code>	igaz értéket ad, ha az alap és a másik bitkészlet egyezők,
<code>bool b = bs != masikbs</code>	igaz értéket ad, ha az alap és a másik bitkészlet különbözők,
<code>rbs = bs1 & bs2</code>	bináris ÉS művelet,
<code>rbs = bs1 bs2</code>	bináris VAGY művelet,
<code>rbs = bs1 ^ bs2</code>	bináris KIZÁRÓ VAGY művelet,
<code>output << bs</code>	karakteres formában ('0', '1') kiírja a <code>bs</code> bitjeit az <code>output</code> adatfolyamba,
<code>input >> bs</code>	karakteres formában ('0', '1') beolvassa a <code>bs</code> bitjeit az <code>input</code> adatfolyamból.

Az alábbi példában nagy, előjel nélküli egész számok összeadására és szorzására készítünk függvény-sablonokat, melyek használatát faktoriális számítással mutatjuk be. A megoldás nagyon nagy számokra is alkalmazható, azonban a felhasznált `to_ullong()` tagfüggvény „csak” 64-bites egészeket tud megjeleníteni, amibe legfeljebb **20!** (2432902008176640000) fér bele:

```
#include <bitset>
#include <iostream>
using namespace std;

template<unsigned int N>
void bsOsszead(bitset<N>& x, const bitset<N>& y) {
    bool atvitel = false, bitszum;
    for (int i = 0; i < N; i++) {
        bitszum = (x[i] ^ y[i]) ^ atvitel;
        atvitel = (x[i] && y[i]) || (x[i] && atvitel) || (y[i] && atvitel);
        x[i] = bitszum;
    }
}

template<unsigned int N>
void bsSzoroz(bitset<N>& x, const bitset<N>& y) {
    bitset<N> tmp = x;
    x.reset();
    for (int i=0; i < N; i++) {
        if (tmp[i])
            bsOsszead(x, y << i);
    }
}
```

```
typedef bitset<128> nagyszam;

int main() {
    const nagyszam egy(1);
    nagyszam f{egy};
    nagyszam n{egy};

    for (int k=0; k < 20; k++) {
        bsSzoroz(f, n);
        bsOsszead(n, egy);
    }
    cout << f.to_ulong() << endl;
    return 0;
}
```

2.7.3.2 Konverziós műveletek

A bitkészletben tárolt biteket egész számmá és sztringgé alakítva egyaránt megkaphatjuk az alábbi konverziós tagfüggvények hívásával. A számmá alakítás során a 0 indexű bit lesz a szám legkisebb helyiértékű bitje, míg a legutolsó bit a legnagyobb helyiértékű.

Konverziós műveletek	Rövid leírás
<code>unsigned long num = bs.to_ulong()</code>	a konverzió eredménye unsigned long típusú lesz, azonban <code>overflow_error</code> lép fel, ha az átalakítás nem végezhető el,
<code>unsigned long long num = bs.to_ullong()</code>	a konverzió eredménye unsigned long long típusú lesz, azonban <code>overflow_error</code> lép fel, ha az átalakítás nem végezhető el,
<code>string str = bs.to_string()</code>	'0' és '1' karaktereket tartalmazó sztringgé alakítja a <code>bs</code> tartalmát,
<code>string str = bs.to_string(ch0)</code>	<code>ch0</code> és '1' karaktereket tartalmazó sztringgé alakítja a <code>bs</code> tartalmát,
<code>string str = bs.to_string(ch0, ch1)</code>	<code>ch0</code> és <code>ch1</code> karaktereket tartalmazó sztringgé alakítja a <code>bs</code> tartalmát.

A konverziókat egyszerű programmal mutatjuk be:

```
#include <bitset>
#include <string>
#include <iostream>
using namespace std;

int main() {
    bitset<32> bs(0x20041002ul);
    cout << bs << endl;
    cout << hex << bs.to_ulong() << endl;
    cout << hex << bs.to_ullong() << endl;
    cout << bs.to_string() << endl;
    cout << bs.to_string('o') << endl;
    cout << bs.to_string('o', '|') << endl;
    cout << endl;
    bs = ~bs;
    cout << bs << endl;
    cout << hex << bs.to_ulong() << endl;
    cout << hex << bs.to_ullong() << endl;
    cout << bs.to_string() << endl;
    cout << bs.to_string('o') << endl;
    cout << bs.to_string('o', '|') << endl;
    return 0;
}
```

```

0010000000000100000100000000010
20041002
20041002
0010000000000100000100000000010
0010000000000100000100000000010
00|000000000|00000|000000000|0

110111111111011111011111111101
dffbeffd
dffbeffd
110111111111011111011111111101
110111111111011111011111111101
||0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|

```

2.7.4 A `vector<bool>` specializáció

A `vector<bool>` a `vector` konténertípus specializációja, amely lehetővé teszi a `bool` típusú értékek tömörített (bites) tárolását. A *bitset*-hez való hasonlósága kézenfekvő, azonban sokban különbözik is attól: mérete megváltoztatható, nincsen semmilyen eszköz egészek és sztringek konvertálására, illetve lekérdezésére. Összességében a `vector<bool>` egy *bites tárolású logikai vektor*, amely azonban nem tekinthető konténernek. Ha az adatainkat konténerben szeretnénk tárolni a `deque<bool>` vagy a `vector<char>` típusokat alkalmazhatjuk helyette.

Alapvetően a már megismert `vector` konténer tagfüggvényeit használhatjuk, néhány különbséggel:

- az elemeknek sem címe, sem referenciája nem kérdezhető le, azonban iterátorokat használhatunk,
- az új `void flip()` tagfüggvény minden elemet ellentettjére állít.

Az elmondottakat szemlélteti az alábbi példaprogram:

```

#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int main () {
    vector<bool> vbool(3);
    // 3 elem beírása index operátorral
    vbool[0] = vbool[1] = true;
    vbool[2] = false;
    // további 5 elem hozzáadása
    for (int i=0; i<5; i++)
        vbool.push_back(bool(i%2));
    cout << "A bool vektor tartalma: " ;
    for (size_t i = 0; i < vbool.size(); ++i)
        cout << vbool[i] << ' ';
    cout << endl;

    // a vektorban tárolt logikai értékek tagadása
    vbool.flip();
    cout << "A bool vektor tartalma: " ;
    for (size_t i = 0; i < vbool.size(); ++i)
        cout << vbool[i] << ' ';
    cout << endl;

    transform(begin(vbool), end(vbool), begin(vbool), [](bool e) { return !e;});
    vector<bool>::iterator x = begin(vbool);
    cout << "A bool vektor tartalma: " ;
    for (; x != end(vbool); x++)
        cout << *x << ' ';
    cout << endl;
    return 0;
}

```

```
A bool vektor tartalma: 1 1 0 0 1 0 1 0
A bool vektor tartalma: 0 0 1 1 0 1 0 1
A bool vektor tartalma: 1 1 0 0 1 0 1 0
```

2.7.5 A `valarray` értéktömb

A `valarray` osztálysablon (`<valarray>`) egy általános célú numerikus értéktömb, melyet azért vezettek be, hogy a C++-ban is elérhetőek legyenek a FORTRAN, MATLAB stb. rendszerekben megszokott, tömbökön elvégezhető, hatékony numerikus műveletek.

A `valarray` önmagában olyan műveletek elvégzésére képes, melyekhez a konténerekkel algoritmusokat kell hívunk. Ennek ellenére a C++ 11-ben a `begin()`, az `end()` és a `swap()` függvénysablonok hozzáadásával az értéktömb is szerepelhet algoritmusok argumentumában.

2.7.5.1 Az értéktömb létrehozása és értékadása

A `valarray` konstruktoraival az értéktömböt numerikus C tömbök elemeivel, vagy akár különálló értékekkel is inicializálhatjuk. A `valarray` osztálysablont valamilyen numerikus típussal (*T*) kell példányosítanunk: `valarray<T>`.

Konstruktorhívások	Rövid leírás
<code>valarray<T> va</code> <code>valarray<T> va {}</code>	üres értéktömb,
<code>valarray<T> va(n)</code>	<i>n</i> -elemű értéktömb, melynek elemei <i>T</i> {} értékkel inicializálódnak,
<code>valarray<T> va(adat, n)</code>	<i>n</i> -elemű értéktömb, melynek elemei az <i>adat</i> értékkel inicializálódnak,
<code>valarray<T> va(ctomb, n)</code> <code>valarray<T> va {ctomb, n}</code>	<i>n</i> -elemű értéktömb, melynek elemei a <i>ctomb</i> első <i>n</i> elemével inicializálódnak,
<code>valarray<T> va(va2)</code> <code>valarray<T> va {va2}</code>	másoló konstruktor – a <i>va</i> a <i>va2</i> tartalmával inicializálódik,
<code>valarray<T> va(move(va2))</code> <code>valarray<T> va {move(va2)}</code>	áthelyező konstruktor – a <i>va</i> a <i>va2</i> tartalmával inicializálódik,
<code>valarray<T> va({init. lista})</code> <code>valarray<T> va {init. lista}</code>	az értéktömb a megadott inicializációs lista elemeiből jön létre,
<code>valarray<T> va(tomb)</code> <code>valarray<T> va {tomb}</code>	az értéktömb a megadott <i>tomb</i> elemeiből jön létre, amely <i>tomb</i> típusa lehet <code>slice_array<T></code> , <code>gslice_array<T></code> , <code>mask_array<T></code> vagy <code>indirect_array<T></code> .

A következő példában az értéktömbök létrehozásának különböző módjait szemléltetjük. Az utolsó konstruktorban szereplő tömbtípusokkal később ismerkedünk meg.

```
#include <iostream>
#include <valarray>
#include <string>
using namespace std;

typedef float T;

void KiirVa(const string& str, const valarray<T>& va) {
    cout << str << "\t";
    for (const auto e : va) // begin(va) és end(va) hívódik meg
        cout << e << " ";
    cout << endl;
}
```

```

int main() {
    int n = 5;
    T ctomb[6] {1.1, 2.3, 5.8, 8.13, 31.34};
    T adat = 12.23;

    valarray<T> va1(adat, n);
    KiirVa("va1", va1);

    valarray<T> va2 {ctomb, n};
    KiirVa("va2", va2);

    valarray<T> va3 {va2};
    KiirVa("va3", va3);

    valarray<T> va4 {1.2, 2.3, 4.5, 5, 6, 6.7, 7.8};
    KiirVa("va4", va4);

    valarray<T> va5 {move(va1)};
    KiirVa("va5", va5);
    KiirVa("va1", va1);
    return 0;
}

```

va1	12.23 12.23 12.23 12.23 12.23
va2	1.1 2.3 5.8 8.13 31.34
va3	1.1 2.3 5.8 8.13 31.34
va4	1.2 2.3 4.5 5 6 6.7 7.8
va5	12.23 12.23 12.23 12.23 12.23
va1	

A **valarray** saját értékadásokat definiál az = operátor túlterhelésével. Az értékadás kifejezés értéke egy referencia magára az értéktömbre:

Értékadások	Rövid leírás
<code>va = masikva</code>	a <i>masikva</i> értéktömbbel azonos lesz a <i>va</i> ,
<code>va = move(masikva)</code>	a <i>masikva</i> értéktömb átkerül a <i>va</i> -ba,
<code>va = adat</code>	a <i>va</i> értéktömb minden eleme felveszi az <i>adat</i> értéket,
<code>va = {init. lista}</code>	az <i>inicializációs listából</i> készített értéktömbbel lesz azonos a <i>va</i> ,
<code>va = tomb</code>	a <i>va</i> értéktömb méretével azonos <i>slice_array<T></i> , <i>gslice_array<T></i> , <i>mask_array<T></i> vagy <i>indirect_array<T></i> tömb elemei másolódnak át a <i>va</i> -ba,
<code>va Ⓜ= masikva</code>	összetett értékadás: <i>va[index]</i> Ⓜ= <i>masikva[index]</i> a <i>va</i> értéktömb minden elemére, ahol a Ⓜ jel a /, %, +, -, ^, &, , <<, >> műveletek egyikét jelöli,
<code>va Ⓜ= adat</code>	összetett értékadás: <i>va[index]</i> Ⓜ= <i>adat</i> a <i>va</i> értéktömb minden elemére, ahol a Ⓜ jel a /, %, +, -, ^, &, , <<, >> műveletek egyikét jelöli.

Az értékadások menete jól látható az alábbi példából:

```

#include <iostream>
#include <iomanip>
#include <valarray>
#include <string>
#include <numeric>
using namespace std;

typedef int T;

void KiirVa(const string& str, const valarray<T>& va) {
    cout << str << "\t";
    for (const auto e : va)
        cout << setw(4) << e << " ";
    cout << endl;
}

```

```

int main() {
    valarray<T> va1;
    va1 = {25, 14, 6, 0, 1, 12, 43, 108, 234, 466};
    valarray<T> va2(va1.size());
    iota(begin(va2), end(va2), 1);
    KiirVa("va1", va1);
    KiirVa("va2", va2);
    va1 -= 36;
    KiirVa("va1", va1);
    va1 /= va2;
    KiirVa("va1", va1);
    va1 += 12;
    KiirVa("va1", va1);
    return 0;
}

```

va1	25	14	6	0	1	12	43	108	234	466
va2	1	2	3	4	5	6	7	8	9	10
va1	-11	-22	-30	-36	-35	-24	7	72	198	430
va1	-11	-11	-10	-9	-7	-4	1	9	22	43
va1	1	1	2	3	5	8	13	21	34	55

2.7.5.2 Az indexelés művelete

Az értéktömbök mérete lekérdezhető, illetve szükség esetén meg is változtatható a konténereknél megszokott módon. A `[ 0, size() )` indextartományon belül az elemeket a szokásos indexelés `[]` operátorával érhetjük el (olvashatjuk vagy írhatjuk).

Értékadások	Rövid leírás
<code>m = va.size()</code>	a <code>va</code> értéktömb eleminek száma ,
<code>va.resize(n)</code>	a <code>va</code> értéktömb átméretezése <code>n</code> -eleműre az új elemek az elemtípus alapértékét veszik fel,
<code>va.resize(n, adat)</code>	a <code>va</code> értéktömb átméretezése <code>n</code> -eleműre az új elemek az <code>adat</code> értéket veszik fel,
<code>adat = va[index]</code> <code>va[index] = adat</code>	az indexelés operátora az index által kijelölt elem referenciájával tér vissza; nincs indexhatár ellenőrzés,
<code>tomb = va[kijeloles]</code> <code>va[kijeloles] = tomb</code>	a <code>va</code> értéktömb kijelölt részének olvasása/írása – bővebben a táblázat után.

Az értéktömbök indexelés operátorát a szokásos értelmezésen túlmenően különleges képességgel is ellátták. A kijelöléssel egyszerre több elemre hivatkozhatunk, az alábbi táblázatban összefoglalt eszközökkel:

Kijelölések	Rövid leírás
<code>slice (kezdo, meret, lepes)</code>	a <code>va</code> értéktömb kijelölt elemei: $kezdo+0*lepes$, $kezdo+1*lepes$, ... $kezdo + (meret-1)*lepes$,
<code>gslice(kezdo, {meret lista}, {lepes lista})</code>	a <code>va</code> értéktömb kijelölt elemei: $k_j = kezdo + \sum_j (lepes_j * meret_j)$,
<code>valarray<bool></code>	a <code>va</code> értéktömb elemeit egy <i>logikai értéktömb</i> igaz elemei jelölik ki,
<code>valarray<size_t></code>	a <code>va</code> értéktömb elemeit egy <i>indexhalmaz</i> elemi jelölik ki.

Amennyiben az indexelt tömb konstans, a kijelölt elemek másolatát `valarray<T>` típusú tömbben kapjuk meg. Ellenkező esetben, mivel a C++ közvetlenül nem engedi referenciatömbök készítését, speciális típusú értéktömbökben kapjuk meg az eredményt:

Kijelölés	Az eredménytömb típusa
<code>slice</code> (kezdő, méret, lépés)	<code>slice_array<T></code> ,
<code>gslice</code> (kezdő, {méretlista}, {lépéslista})	<code>gslice_array<T></code> ,
<code>valarray<bool></code>	<code>mask_array<T></code> ,
<code>valarray<size_t></code>	<code>indirect_array<T></code> .

A kijelölések alkalmazásának megértéséhez az alábbi példaprogram áttanulmányozásával kerülhetünk közelebb:

```
#include <iostream>
#include <valarray>
#include <string>
#include <numeric>
using namespace std;

void KiirVa(const string& str, const valarray<char>& va) {
    cout << str << "\t" << va.size() << "\t";
    for (const auto e : va)
        cout << e;
    cout << endl;
}

int main() {
    cout << "slice:" << endl;
    const valarray<char> va1 {"abcdefghijklmnopqrstuvwxy", 26};
    valarray<char> va2 {va1[slice(1, 6, 2)]};
    KiirVa("va1", va1); KiirVa("va2", va2);

    valarray<char> va3 {va1};
    const valarray<char> va4 {"ABCDEF", 6};
    va3[slice(1, 6, 2)] = va4;
    KiirVa("va3", va3); KiirVa("va4", va4);

    cout << "\ngslice:" << endl;
    const valarray<size_t> meretek {2, 3};
    const valarray<size_t> lepesek {4, 5};
    valarray<char> va5 {va1[gslice(2, meretek, lepesek)]};
    KiirVa("va1", va1); KiirVa("va5", va5);

    valarray<char> va6 {va1};
    va6[gslice(2, meretek, lepesek)] = va4;
    KiirVa("va6", va6); KiirVa("va4", va4);

    cout << "\nvalarray<bool>:" << endl;
    const valarray<bool> vb {false, true, true, false, false, true, true};
    valarray<char> vbch (vb.size());
    for (int i=0; i<vb.size(); i++)
        vbch[i] = char('0'+vb[i]);
    valarray<char> va7 {va1[vb]};
    KiirVa("va1", va1); KiirVa("vach", vbch); KiirVa("va7", va7);

    valarray<char> va8 {va1};
    va8[vb] = va4;
    KiirVa("va8", va8); KiirVa("vach", vbch); KiirVa("va4", va4);

    cout << "\nvalarray<size_t>:" << endl;
    const valarray<size_t> vidx {13, 8, 5, 1, 3, 2};
    valarray<char> vidxjel (' ', va1.size());
    for (int i=0; i<vidx.size(); i++)
        vidxjel[vidx[i]] = '0'+ i;
    valarray<char> va9 {va1[vidx]};
    KiirVa("va1", va1); KiirVa("vidxjel", vidxjel); KiirVa("va9", va9);

    valarray<char> va10 {va1};
    va10[vidx] = va4;
    KiirVa("va10", va10); KiirVa("vidxjel", vidxjel); KiirVa("va4", va4);
    return 0;
}
```

```

slice:
va1    26    abcdefghijklmnopqrstuvwxyz
va2     6    bdfhjl
va3    26    aAcBeCgDiEkFmnopqrstuvwxyz
va4     6    ABCDEF

gslice:
va1    26    abcdefghijklmnopqrstuvwxyz
va5     6    chmgllq
va6    26    abAdefdBijkECnopErstuvwxyz
va4     6    ABCDEF

valarray<bool>:
va1    26    abcdefghijklmnopqrstuvwxyz
vach    7    0110011
va7     4    bcfg
va8    26    aABdeCDhijklmnopqrstuvwxyz
vach    7    0110011
va4     6    ABCDEF

valarray<size_t>:
va1    26    abcdefghijklmnopqrstuvwxyz
vidxjel 26    354 2 1 0
va9     6    nifbdc
va10    26    aDFEeCghBjklmAopqrstuvwxyz
vidxjel 26    354 2 1 0
va4     6    ABCDEF

```

2.7.5.3 További műveletek

Az értéktömbbel további műveleteket tagfüggvények és külső függvények segítségével egyaránt végezhetünk:

Műveletek	Rövid leírás
$m = va.sum()$	a <i>va</i> értéktömb elemeinek összege (az <code>operator+=()</code> kell hozzá),
$m = va.min()$	a <i>va</i> értéktömb legkisebb elemértéke (az <code>operator<()</code> kell hozzá),
$m = va.max()$	a <i>va</i> értéktömb legnagyobb elemértéke (az <code>operator>()</code> kell hozzá),
$va1 = va.shift(n)$	egy olyan új értéktömböt ad vissza, amelyben a <i>va</i> elemei <i>n</i> pozícióval balra (a 0. elem irányába) lépnek , és a kieső elemek helyére a típus alapértéke jön be; az <i>i</i> . elem új pozíciója <i>i-n</i> ,
$va1 = va.cshift(n)$	egy olyan új értéktömböt ad vissza, amelyben a <i>va</i> elemei <i>n</i> pozícióval balra (a 0. elem irányába) lépnek , és a bal oldalon kieső elemek a jobb oldalon jönnek be – körkörös léptetés; az <i>i</i> . elem új pozíciója $(i-n) \% size()$,
$va1 = va.apply(fuggveny)$	egy olyan új értéktömböt ad vissza, amelyben a <i>va</i> elemeit a <i>fuggveny</i> funkcionál által visszatérített érték helyettesíti ; a függvény lehetséges prototípusai: <i>T fuggveny (T)</i> vagy <i>T fuggveny(const T&)</i> ,
$va1 = \oplus va$	$va1[index] = \oplus va[index]$ a <i>va</i> értéktömb minden elemére, ahol a $\oplus$ jel a +, - (előjelek), ~ és ! műveletek egyikét jelöli,
$va.swap(vamasik)$	a <i>va</i> értéktömb elemeinek felcserélése a <i>vamasik</i> elemeivel,
$va3 = va1 \oplus va2$	$va3[index] = va1[index] \oplus va2[index]$ az azonos méretű <i>va1</i> és <i>va2</i> értéktömbök minden elemére, ahol a $\oplus$ jel a +, -, *, /, %, &, , ^, <<, >>, &&, műveletek egyikét jelöli,
$va2 = va1 \oplus adat$ $va2 = adat \oplus va1$	$va2[index] = va1[index] \oplus adat$ a <i>va1</i> értéktömb minden elemére, ahol a $\oplus$ jel a +, -, *, /, %, &, , ^, <<, >>, &&, műveletek egyikét jelöli,

Műveletek (folytatás)	Rövid leírás
$vabool = va1 \oplus va2$	$vabool[index] = va1[index] \oplus va2[index]$ a $va1$ és $va2$ azonos méretű értéktömbök minden elemére, ahol a $\oplus$ jel a $==$, $!=$, $<$, $<=$, $>$, $>=$ műveletek egyikét jelöli; a $vabool$ pedig <code>valarray<bool></code> típusú,
$vabool = va1 \oplus adat$ $vabool = adat \oplus va1$ $va2 = \oplus(va1)$	$vabool[index] = va1[index] \oplus adat$ a $va1$ értéktömb minden elemére, ahol a $\oplus$ jel a $==$, $!=$, $<$, $<=$, $>$, $>=$ műveletek egyike, $va2[index] = \oplus(va1[index])$ a $va1$ értéktömb minden elemére, ahol a $\oplus$ jel az <i>abs</i> , <i>acos</i> , <i>asin</i> , <i>atan</i> , <i>cos</i> , <i>cosh</i> , <i>exp</i> , <i>log</i> , <i>log10</i> , <i>sqrt</i> , <i>sin</i> , <i>sinh</i> , <i>tan</i> , <i>tanh</i> függvények egyikének neve,
$va3 = atan2(vay, vax)$	$va3[index] = atan2(vay[index], vax[index])$ az azonos méretű vay és vax értéktömbök minden (y, x) elempárjára meghatározza az inverz tangens (y/x) értékét,
$va2 = atan2(vay, x)$ $va2 = atan2(y, vax)$	$va2[index] = atan2(vay[index], x)$, illetve $va2[index] = atan2(y, vax[index])$ a vay , illetve a vax minden elemére,
$va3 = pow(valap, vkitevo)$	$va3[index] = pow(valap[index], vkitevo[index])$ az azonos méretű $valap$ és $vkitevo$ értéktömbök minden elempárjára elvégzi a hatványszámítást,
$va2 = pow(valap, kitevo)$ $va2 = pow(alap, vkitevo)$	$va2[index] = pow(valap[index], kitevo)$, illetve $va2[index] = pow(alap, vkitevo[index])$ a $valap$, illetve a $vkitevo$ minden elemére.

Az értéktömb műveleteit adott számú másodfokú egyenlet megoldását végző programmal mutatjuk be. Az egyenletek együtthatóit véletlenszerűen töltjük fel [-10, 10] közötti számokkal, ügyelve arra, hogy a négyzetes tag együtthatója ne legyen 0.

```
#include <random>
#include <iostream>
#include <valarray>
#include <ctime>
#include <iomanip>
using namespace std;

int main() {
    mt19937 rndMotor;
    rndMotor.seed(time(nullptr));
    uniform_int_distribution<int32_t> Random(-10,10);

    const int n = 10;
    valarray<double> a(n), b(n), c(n);
    for (int i=0; i<n; i++) {
        while ((a[i] = Random(rndMotor))==0);
        b[i] = Random(rndMotor);
        c[i] = Random(rndMotor);
    };

    valarray<double> d(a.size()), x1(a.size()), x2(a.size());
    d = pow(b, 2.0) - 4.0 * a * c;
    valarray<bool> vanValosGyok = (d >= 0.0);
    x1 = (-b + sqrt(d)) / (2.0 * a);
    x2 = (-b - sqrt(d)) / (2.0 * a);

    cout << "masodfoku egyenlet" << "gyok1, gyok2" << "\n";
    for (size_t i = 0; i < a.size(); ++i) {
        if (vanValosGyok[i]) {
            cout << a[i] << "x2 + " << b[i] << "x + " << c[i] << " = 0 \t";
            cout << setw(10) << x1[i] << ", " << setw(10) << x2[i] << endl;
        }
    }
}
```

```

valarray<bool> nincsValosGyok = !vanValosGyok;
size_t nKomplexGyokok = 0;
for (bool e : nincsValosGyok)
    nKomplexGyokok += e;
cout << "\nValos gyok nélküli egyenletek száma: " << nKomplexGyokok << endl;
return 0;
}

```

<i>masodfoku egyenlet</i>	<i>gyok1,</i>	<i>gyok2</i>
$4x^2 + 1x + -3 = 0$	0.75,	-1
$6x^2 + -10x + 4 = 0$	1,	0.666667
$-5x^2 + -3x + 8 = 0$	-1.6,	1
$-10x^2 + 10x + 2 = 0$	-0.17082,	1.17082
$-1x^2 + -4x + 3 = 0$	-4.64575,	0.645751
$-4x^2 + 2x + 2 = 0$	-0.5,	1
$-4x^2 + 9x + -5 = 0$	1,	1.25

Valos gyok nélküli egyenletek száma: 3

Azoknál a műveleteknél, ahol az értéktömb mellett skalár is szerepel, fontos, hogy a skalár adattípusa megegyezzen az értéktömb elemeinek típusával. A fenti program nem fordul le, ha például a következő utasítást használjuk a diszkrimináns kiszámítására:

```
d = pow(b, 2) - 4 * a * c;
```

A javítás legyegeyszerűbb módja, ha az egész konstansokat **double** típusúakkal helyettesítjük:

```
d = pow(b, 2.0) - 4.0 * a * c;
```

Egy másik megoldáshoz jutunk típuselőírások alkalmazásával:

```
d = pow<double>(b, 2) - double(4) * a * c;
```

A példaprogrammal kapcsolatos másik kérdés, hogy mi történik a számítások során azokkal az egyenletekkel, amelyeknek nincs valós gyöke? A C++11 a valós értékekre átvette az IEEE 754 szabványból a *NAN* (nem szám) és a végtelen értelmezését, így a komplex gyökök helyett *NAN* értékek jelennek meg az *x1* és *x2* értéktömbökben.

2.7.5.4 Mátrixműveletek

A már bemutatott kijelölő megoldások közül a *slice* (szelet) osztály segítségével az értéktömb elemeit, mint egy mátrix sorait, illetve oszlopait érhetjük el. A *gslice* (általánosított *slice*) osztály segítségével pedig többdimenziós tömböt, illetve annak szeleteit kezelhetjük. Mindkét osztály objektumait az értéktömb indexeként kell megadni. Mindkét kijelölő osztállyal példákön keresztül folytatjuk az ismerkedést.

Az első példában a *slice* segítségével különválasztjuk az értéktömb páros, illetve páratlan indexű pozícióban álló elemeit:

```

#include <iostream>
#include <valarray>
using namespace std;

int main() {
    valarray<int> etomb = {0, 1, 20, 3, 40, 5, 60, 7, 80, 9};
    int kezdoindex = 0;
    int meret = etomb.size()/2; // 5
    int lepes = 2;
    valarray<int> paros = etomb[slice(kezdoindex, meret, lepes)];
    valarray<int> paratlan = etomb[slice(kezdoindex+1, meret, lepes)];
    for (int e : etomb)
        cout << e << " "; cout << endl;
    cout << "paros indexu elemek:" << endl;
    for (int e : paros)
        cout << e << " "; cout << endl;
}

```

```

cout << "paratlan indexu elemek:" << endl;
for (int e : paratlan)
    cout << e << " "; cout << endl;
return 0;
}

```

```

0 1 20 3 40 5 60 7 80 9
paros indexu elemek:
0 20 40 60 80
paratlan indexu elemek:
1 3 5 7 9

```

Az következő példában értéktömbökben tárolt négyzetes mátrixokat szorzunk össze. A művelethez az első mátrix sorainak, illetve a második mátrix oszlopainak kijelölését a [slice](#) osztály segítségével végezzük.

```

#include <cmath>
#include <iostream>
#include <iomanip>
#include <valarray>
using namespace std;

// Négyzetes mátrix alakú megjelenítés
template<class T>
void MatrixKiiras(const valarray<T>& a) {
    size_t n = size_t(sqrt(a.size()));
    for(size_t sor = 0; sor < n; sor++) {
        valarray<T> asora = a[ slice(n*sor, n, 1) ];
        for (const T& e : asora)
            cout << setw(4) << e;
        cout << endl;
    }
    cout << endl;
}

// Négyzetes mátrixok szorzása
template<class T>
valarray<T> MatrixSzozas(const valarray<T>& a, const valarray<T>& b, size_t n) {
    valarray<T> mszorzat(n * n);
    for(size_t i = 0; i < n; i++)
        for(size_t j = 0; j < n; j++) {
            valarray<T> asora = a[ slice(n*i, n, 1) ]; // az i. sor kijelölése
            valarray<T> boszlopa = b[ slice(j, n, n) ]; // a j. oszlop kijelölése
            mszorzat[i * n + j] = (asora * boszlopa).sum();
        }
    return mszorzat;
}

int main() {
    valarray<int> a = { 1, 1, 3,
                       0, -2, 0,
                       3, 0, 1 };
    MatrixKiiras(a);
    valarray<int> b = { 1, 0, -2,
                       3, 2, 1,
                       0, -1, 3 };
    valarray<int> c(MatrixSzozas(a, b, 3));
    MatrixKiiras(b);
    MatrixKiiras(c);
}

```

1	1	3
0	-2	0
3	0	1
1	0	-2
3	2	1
0	-1	3
4	-1	8
-6	-4	-2
3	-1	-3

A **gslice** használatához azonos méretű értéktömbökben kell megadni a hosszakat (*lengths*) és a lépéseket (*strides*). Az alábbi függvény szemlélteti a kijelölés menetét kételemű hosszak és lépések esetén. (Az elemszám növelésével a ciklusok számát kell növelni.)

```
void GSliceKiiras(const valarray<int> va, size_t start,
                 const valarray<size_t> lengths, const valarray<size_t> strides) {
    size_t index;
    for (int i=0; i<lengths[0]; i++) {
        for (int j=0; j<lengths[1]; j++) {
            index = start + i*strides[0] + j*strides[1];
            cout << va[index] << " ";
        }
        cout << endl;
    }
}
```

Az utolsó példában egy háromdimenziós tömb elemeit tároljuk értéktömbben. A 3x3 méretű mátrixok elemei egymás után helyezkednek el (0 és 9 kezdőpozíciókon). A **gslice** osztály segítségével kimásoljuk a mátrixok elemeit új értéktömbökbe, majd műveletek végzése után visszamásoljuk az eredeti értéktömbbe.

```
#include <iostream>
#include <iomanip>
#include <valarray>
using namespace std;

// Négyzetes mátrix alakú megjelenítés
void MatrixKiiras(const valarray<int>& a) {
    size_t n = size_t(sqrt(a.size()));
    for(size_t sor = 0; sor < n; sor++) {
        valarray<int> asora = a[ slice(n*sor, n, 1) ];
        for (int e : asora)
            cout << setw(4) << e;
        cout << endl;
    }
    cout << endl;
}

void GSliceKiiras(const valarray<int> va, size_t start,
                 const valarray<size_t> lengths, const valarray<size_t> strides) {
    size_t index;
    for (int i=0; i<lengths[0]; i++) {
        for (int j=0; j<lengths[1]; j++) {
            index = start + i*strides[0] + j*strides[1];
            cout << va[index] << " ";
        }
        cout << endl;
    }
}
```

```

int main() {
    valarray<int> va { 1, 0, 1,
                     2, 1, 2,
                     0, 3, 0,
                     1, 1, 2,
                     2, 2, 0,
                     0, 1, 3};

    valarray<size_t> hosszak{3,3}; // 3x3 mátrixok
    valarray<size_t> lepesek{3,1};
    valarray<int> velso(0,9), vmasodik(0,9); // a kijelölt elemek tárolására

    velso = va[gslice(0,hosszak,lepesek)]; // első mátrix
    vmasodik = va[gslice(9,hosszak,lepesek)]; // második mátrix
    MatrixKiiras(velso);
    MatrixKiiras(vmasodik);

    velso *= 7;
    va[gslice(0,hosszak,lepesek)] = velso;
    MatrixKiiras(velso);
    vmasodik += 12;
    va[gslice(9,hosszak,lepesek)] = vmasodik;
    MatrixKiiras(vmasodik);

    GSliceKiiras(va, 0, hosszak, lepesek);
    cout << endl;
    GSliceKiiras(va, 9, hosszak, lepesek);
    return 0;
}

```

1	0	1
2	1	2
0	3	0
1	1	2
2	2	0
0	1	3
7	0	7
14	7	14
0	21	0
13	13	14
14	14	12
12	13	15
7	0	7
14	7	14
0	21	0
13	13	14
14	14	12
12	13	15

3. Utószó helyett

A C++ nyelv más programozási nyelvekhez hasonlóan fejlődik, átalakul. A C++14 szabványtól kezdve a nyelv bizonyos összetevőit hivatalosan elavultnak nyilváníthatják, majd azokat egy későbbi változatban akár ki is hagyhatják. Ezért fontos ismételten kihangsúlyoznunk, hogy könyvünk a C++11 nyelvi szabványra épül, bár néhány helyen C++14 megjegyzések is szerepelnek benne.

Természetesen az ismeretek nagy része C++98 nyelvi környezetben is haszonnal alkalmazható. A példaprogramok többségében a C++11 nyelvi struktúrákat C++98 megoldásokra kell cserélnünk. Példaként tekintsük az alábbi egyszerű, **vector** konténert használó programot!

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main() {
    vector<int> v = { 3, 1, 2, 3, 7 };
    for (auto vi=begin(v); vi<end(v); vi++) {
        *vi *= 3;
    }
    for (int e : v) {
        cout << e << " ";
    }
    cout << endl;
}
```

A C++98-as változatban kezdőértékadást két lépésben végezzük el, az **auto** helyett magunk adjuk meg a típust, a **begin()** és **end()** sablonok helyett tagfüggvényeket használunk, míg a tartományalapú **for** ciklust lecseréljük:

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main() {
    int v0[5] = { 3, 1, 2, 3, 7 };
    vector<int> v(v0, v0+5);
    for (vector<int>::iterator vi=v.begin(); vi<v.end(); vi++) {
        *vi *= 3;
    }
    for (int i =0; i< v.size(); i++) {
        cout << v[i] << " ";
    }
    cout << endl;
}
```

A könyvben a C++11 nyelv STL elemeinek csupán egy részét alkalmaztuk. A teljesség kedvéért tekintjük át az STL további, általunk nem tárgyalt lehetőségeit is, néhány jellemző függvényt, osztályt, sablont felsorakoztatva:

Kategória	Jellemző elemek	Fejállomány
fordítás idejű racionális aritmetika	<code><ratio></code>	<code><ratio></code>
fordítás idejű típus-információk	<code>is_copy_constructible<></code> , <code>is_null_pointer<></code> , <code>add_rvalue_reference<></code> , <code>remove_volatile<></code>	<code><type_traits></code>
helyi beállítások kezelése	<code>locale</code> , <code>use_facet<></code>	<code><locale></code>

Kategória	Jellemző elemek	Fejállomány
I/O adatfolyamok	<i>istream, ostream, iostream, ifstream, ofstream, fstream, istringstream, ostringstream, stringstream,</i>	<i><iostream>, <iomanip> <fstream> <sstream></i>
időkezelés	<i>system_clock, steady_clock, high_resolution_clock, duration<>, duration_cast<>(), time_point<></i>	<i><chrono></i>
kivételek	<i>exception, rethrow_exception(), terminate()</i>	<i><exception></i>
komplex számok	<i>complex<float>, complex<double>, complex<long double></i>	<i><complex></i>
numerikus típusok határai	<i>numeric_limits<></i>	<i><limits></i>
„okos” mutatók	<i>unique_ptr<>, shared_ptr<>, weak_ptr<></i>	<i><memory></i>
párhuzamos feladat-végrehajtás	<i>thread, mutex, lock_guard<>, unique_lock<>, future, atomic<>, condition_variable</i>	<i><thread>, <mutex>, <future>, <atomic>, <condition_variable></i>
reguláris kifejezések	<i>regex, regex_match(), regex_search(), regex_replace(), match_results<>, sub_match<>, regex_iterator<></i>	<i><regex></i>
véletlen számok	<i>default_random_engine, shuffle_order_engine<>, normal_distribution<>, uniform_int_distribution<></i>	<i><random></i>

Teljesen jogosan vetődik fel a kérdés, hogy mennyire lesz hatékony a C++ programunk, ha STL konténereket és algoritmusokat használunk. Akárcsak magát a C++ nyelvet, az STL lehetőségeit is többféle módon lehet felhasználni. Amennyiben egy feladat megoldása során a legmegfelelőbb elemeket alkalmazzuk, hatékony kódhoz juthatunk.

Néhány további szempont annak eldöntéséhez, hogy az STL megoldásiból építkezünk, vagy mindent saját magunk programozunk:

- Az STL első változatát az 1980-as években fejlesztették ki, azóta több javításon, optimalizáláson és bővítésen van túl.
- Az STL elemait használva a programunk olvashatóbb lesz, és a fejlesztés során a konkrét feladatra koncentrálhatunk.
- Mivel az STL osztály- és függvénysablonokból áll, ugyanazokat a megoldásokat változtatás nélkül alkalmazhatjuk több típus esetén.
- A szabványos algoritmusok hagyományos tömbök esetén is felhasználhatók, így akár a konténerek nélkül is kezdetjük az ismerkedést az STL könyvtárral.
- ... (a folytatást az Olvasóra bízunk)