C++
Gyakorlat jegyzet
9. óra

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlata alapján. (2018. április 30.)

1. Funktorok

Mielőtt belevetnénk magunkat az STL-ben lévő algoritmusokba és konténerekbe, fontos megismerkednünk a funktorokkal, melyek a rendezéseknél lesznek majd használatosak.

A funktor egy olyan osztály, melynek túl van terhelve a gömbölyű zárójel operátora (tehát kvázi meg lehet hívni).

Egy egyszerű példa:

```
struct S
{
    int x;
    int operator()(int y)
    {
        return x + y;
    }
};
int main()
{
    S s1, s2;
    s1.x = 5;
    s2.x = 8;
    std::cout << s1(2) << std::endl; //7
    std::cout << s2(2) << std::endl; //10
}
```

Bár `x` egy objektum, függvényként is funkcionál. A funktorok segítségével nagyon könnyedén tudunk objektumokat rendezni. Tekintsünk is erre egy példát!

1.1. Predikátumok

Írjunk egy funktort, mely egész számok számokat tud összehasonlítani érték szerint.

```
template <class Compr>
bool f(int a, int b, Compr c)
{
    return c(a, b);
}

struct Less
{
    bool operator()(int a, int b) const
    {
        return a < b;
    }
};

int main()
{
    if( f(2,3, Less()) )
        std::cout << "2 kisebb mint 3! ";
}
```

Kimenet: `2 kisebb mint 3!`

A fenti funktor (`Less`) egy un. **bináris predikátum** (*binary predicate*), azaz a gömbölyű zárójel operátora két azonos típusú objektumhoz rendel egy logikai értéket (matamatikai nyelven $Less : T \times \times T \to \mathbb{L}$, ahol T ismert típus). Figyeljük meg, hogy a harmadik paraméter egy névtelen temporális változó.

1.1.1. Megjegyzés. Figyeljük meg azt is, hogy ebben az esetben érték szerint vettük át a hamradik paramétert: ez nem csak hatékonyabb, hisz `Less` mérete minimális (nincs adattagja), de mivel `Less()` egy jobbérték, csak így, vagy konstans referenciával tudnánk átvenni.

Amennyiben egy `Less` típusú objektum gömbölyű zárójel operátorát 2 `int`-el meghívjuk, és a visszatérési érték `true`, akkor az első szám a kisebb, ellenkező esetben a második. Ez alapján az egyenlőség is levezethető: ha egy a szám nem nagyobb b-nél, és b sem nagyobb a-nál, akkor egyenlőek.

$$a = b \quad \Leftrightarrow \quad \neg(a < b) \wedge \neg(b < a)$$

```
int main()
{
    int a = 2, b = 2;
    if( f(a, b, Less()) == false && f(b, a, Less()) == false )
        std::cout << "2 és 2 egyenlő!";
}
```

Kimenet: `2 és 2 egyenlő!`

Könnyű látni, hogy általánosan beszélve, `Compr` összehasonlító funktorral a és b `T` típusú objektumoknál

$$a \text{ és } b \text{ ekvivalens} \quad \Leftrightarrow \quad \neg\, Compr(a,b) \wedge \neg\, Compr(b,a).$$

Azokat a funktorokat, melyeknek az `()` operátora csak **egy** adott típusú objektumot várnak és `bool` a visszatérési értékük, *unary predicate*-nek hívjuk. Írassunk ki egy tömb páros elemeit!

```
template <class T, class Pred> //pred mint predicate
void printIf(T *start, int size, Pred pred)
{
    for (int i = 0; i<size; i++)
    {
        if( pred(start[i]) )
            std::cout << start[i] << std::endl;
    }
}

struct IsEven //unary predicate
{
    bool operator()(const int &a) const
    {
        return a % 2 == 0;
    }
};

int main()
{
    int t[] = {1,2,3,4,5,6};
    printIf(t, sizeof(t)/sizeof(t[0]), IsEven()); //2 4 6
}
```

1.1.2. Megjegyzés. Miért használunk funktorokat függvénypointerek helyett? Világos, hogyha egyedi rendezést szeretnénk, akkor muszáj ezt az információt valahogy átadni. A funktorok erre alkalmasabbak, példaképp vegyük ehhez egy olyan template függvényt, melynek egyik template paramétere egy olyan funktort vár, melynek `()` operátora egy paramétert vár és `bool`-al tér vissza, és feladata az erre igazat adó elem megkeresése.

Hogyan tudnánk mi funktorok helyett függvénypointerrel a második páros számot megkeresni vele? Vagy az ötödik 8-al oszthatót? Nos, függvényekkel igencsak nehezen (kb statikus adattagokra kényszerülnénk, vagy egy hasonlóan nem épp elegáns megoldásra), azonban egy funktorban létrehozhatunk egy számlálót, melyet tudunk növelgetni.

2. STL konténerek

Az *STL* a *Standard Template Library* rövidítése.

2.1. Bevezető a konténerekhez

C++ban az egyetlen tároló alapértelmezetten a tömb. Azonban az meglehetősen kényelmetlen: a tömb egymás mellett lévő memóriacímek összessége, nem tehetjük meg azt, hogy csak úgy hozzáveszünk 1-1 elemet (mi van ha valaki más már írt oda?). Ezért jobban járunk, ha vagy írunk egy egyedi konténert (pl. az általunk létrehozott `List`), vagy pedig válogatunk az előre megírt STL konténerek között.

Három konténertípust különböztetünk meg:

- **Szekvenciális**: Olyan sorrendben tárolja az adattagokat, ahogyan betesszük őket.
  Példa: `std::vector`, `std::deque, std::list.`
- **Asszociatív**: Az elemek rendezettek a konténerben.
  Példa: `std::set`, `std::map`, `std::multiset`, `std::multimap`.
- **Konténer adapter**: Egy meglévő konténert alakítanak át, általában szogorítanak. Példaképp ha vesszük az `std::deque` konténert, ami egy kétvégű sor, könnyen tudunk belőle egy egyvégű sort csinálni. Ezen a konténerek általában egy másik konténert tárolnak, és annak a funkcióit szigorítják.
  Példa: `std::queue`, `std::stack`.

Minden STL konténer rendelkezik konstans és nem konstans iterátorral, fordított irányú és konstans fordított irányú iterátorral melyre így hivatkozhatunk:

Iterátor típus	Ahogy hivatkozhatunk rájuk	Első elem	Past-the-end
iterátor	`/*konténer név*/::iterator`	`begin()`	`end()`
konstans iterátor	`/*konténer név*/::const_iterator`	`cbegin()`	`cend()`
fordított irányú iterátor	`/*konténer név*/::reverse_iterator`	`rbegin()`	`rend()`
fordított irányú konstans iterátor	`/*konténer név*/::reverse_const_iterator`	`crbegin()`	`crend()`

A következő leírásban nem fogunk minden létező tagfüggvénnyel foglalkozni: egyrészt olyan sok van, hogy azt teljesen irreális észben tartani, másrészt sok ilyenhez érdemi hozzáfűznivalót én nehezen tudnék tenni, így az egyes szekciók végén található link az adott konténerhez.

2.1.1. Megjegyzés. Nagyon fontos képesség az is, hogy valaki hogyan tud utánanézni valaminek, amivel nincs teljesen tisztában, így erősen javallott a `cppreference.com`-el való ismerkedés, illetve más helyeken lévő információk böngészése is. A könyvben található linkek túlnyomótöbbségben kattinthatóak, és érdemes is ellátogatnunk ezen oldalakra.

2.2. vector

A `<vector>` könyvtárban található, maga a konténer az `std::vector` névre hallgat. Előzménytárgyakból ismerős lehet ez a konténer, más néven mint dinamikus tömb hivatkoztunk rá. Bár a pontos definíciója mint megannyi STL-béli algoritmus és konténer implementációfüggő, vannak olyan tulajdonságok, melyeket elvár a szabvány: pl. rendelkezzen `push_back` függvénnyel, a `[]` operator műveletigénye legyen konstans, stb.

Tekintsünk az `std::vector` pár alkalmazását.

```cpp
#include <vector>
#inclide <iostream>

int main()
{
    std::vector<int> v;

    //elem beszúrása
    for (int i = 0; i<11; i++)
        v.push_back(i);

    //utolsó elem törlése
    v.pop_back();

    //végigiterálás a konténeren a már jól ismert módon
    for (int i = 0; i<v.size; i++)
        std::cout << v[i] << std::endl; // 0 1 2 3 4 5 6 7 8 9

    //iterátorok használata
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << std::endl; // 0 1 2 3 4 5 6 7 8 9
}
```

A `vector` leggyakrabban dinamikusan lefoglalt tömbben tárolja az adatainkat, viszont – ahogy az korábban is említve volt – egy tömb mérete nem növelhető. Ezt az `std::vector` a következőféleképpen oldja meg: ha több elemet szeretnénk beszúrni, mint amennyi az adott `vector` kapacitása, akkor lefoglal egy nagyobb memóriaterületet (leggyakrabban kétszer akkorát), és minden elemet átmásol erre az új memóriaterületre.

Így a `push_back`-nek a műveletigénye amortizált konstans: Általában konstans, de ha új memóriaterületet kell lefoglalni és a meglevő elemet átmásolni, akkor lineáris.

2.2.1. Megjegyzés. Számos okból a `vector` a leggyakoribb választás, ha konténerre van szükségünk. Flexibilitása és gyorsasága kiemelkedő, azonban megvannak a maga gyenge pontjai: a konténer közepére elemet beszúrni például csak úgy tudunk, ha egyesével valamennyi elemet odébb másolunk.

Feltűnhet, hogy ezek a műveletek, csakúgy mint számos egyéb melyet a cppreference-n olvashatunk, a konténer végére fókuszál, sőt, az elejére vonatkozó műveletek nincsenek is implementálva (nincs semmilyen `push_front` vagy `pop_front`). Ennek az az oka, hogy az `std::vector`-nál a konténer végének a módosítása a leghatékonyabb, ha a közepén/elején szeretnénk módosító műveletket végrehajtani, az gyakran a környező elemek odébb másolásával jár.

Az `std::vector` tagfüggvényei, mint a nemsoká következő STL algoritmusok, többnyire iterátorokat várnak paraméterül. A következő példában töröljük ki a 4. elemet, majd szúrjuk is vissza!

```cpp
int main()
{
    std::vector<int> v;
    for (int i = 0; i<10; i++)
        v.push_back(i);

    for (int i = 0; i<v.size(); i++)
        std::cout << v[i] << ' '; // 0 1 2 3 4 5 6 7 8 9

    std::vector<int>::iterator it = v.begin() + 3;
    v.erase(it);
    for (int i = 0; i<v.size(); i++)
        std::cout << v[i] << ' '; // 0 1 2 4 5 6 7 8 9

    it = v.begin() + 3;
    v.insert(it, 3);
```

```
    for (int i = 0; i<v.size(); i++)
        std::cout << v[i] << ' '; // 0 1 2 3 4 5 6 7 8 9
}
```

Az iterátorok kezelésének azonban komoly veszélyei is vannak, a legnagyobb gonosz itt az un. iterátor invalidáció (*iterator invalidation*). Amikor a `vector` által lefoglalt dinamikus tömb mérete túl kicsi az újabb elemek beszúrásához, és egy újabb tömbbe másolja át őket, minden iterátor, pointer és referencia ami a régebbi tömbre hivatkozott invalidálódik, lévén olyan területre hivatkoznak, melyeket már felszabadultak.

Ugyanígy a konténer közepéről történő törlés, vagy a konténer közepére történő beszúrás is iterátor invalidációval jár.

Az invalidálódott objektumokkal végzett műveletek nem definiált viselkedést eredményeznek.

```
int main()
{
    std::vector<int> v;
    v.push_back(3);
    std::vector<int>::iterator it = v.begin();

    for (int i = 0; i<1000; i++)
        v.push_back(i);

    *it = 10; // nem definiált viselkedés
}
```

Az invalidáció elkerülése végett számos tagfüggvény egy iterátorral tér vissza, erre lehet példa az `insert`, mely az új elemre, vagy az `erase`, mely az utolsó eltávolított elem rákövetkezőjére hivatkozik.

Töröljünk egy vektorból minden páratlan számot!

```
int main()
{
    std::vector<int> v;
    for (int i = 0; i<10; i++)
        v.push_back(i);
    for(std::vector<int>::iterator it = v.begin(); it != v.end(); )
    {
        if(*it % 2 == 1)
            it = v.erase(it); //iterátor invalidáció elkerülése végett
        else
            ++it;
    }
}
```

2.2.2. Megjegyzés. Oldjuk meg, hogy a fentebbi kódrészletben a 4-es értékű elem törlésénél `it`-nek ne újra `v.begin() + 3`-at adjunk értékül, hanem használjunk ki hogy az `erase` függvény visszatér egy iterátorral!

Link: `http://en.cppreference.com/w/cpp/container/vector`.

2.2.3. Megjegyzés. Az `std::deque` működése az eddiegiek alapján triviális, így annak megismerését az olvasóra bízom.

2.3. set

Az `std::set` a `<set>` könyvtárban található. Ez a konténer a matematikai halmazt valósítja meg: egyedi elemeket tárol, így ha egy olyan elemet próbálnánk beszúrni, mellyel ekvivalens már szerepel a `set`-ben, nem történne semmi (*no-op*).

A szabvány azt is megköveteli, hogy ez a konténer rendezett legyen. Ahhoz, hogy ezt meg tudja valósítani, szüksége van a konténernek egy bináris predikátum funktorra is mint template paraméter a tárolandó `T` típus mellett.

Ez utóbbi template paramétert nem kell feltétlenül megadni, alapértelmezetten ugyanis az `std::set` az `std::less<T>` alapján rendez, mely gyakorlatilag az `<` operátorral ekvivalens.

Ezt a következőféleképpen képzelhetjük el:

```
namespace std
{
    template <class T>
    struct less
    {
        bool operator()(const T &lhs, const T &rhs) const
        {
            return lhs < rhs;
        }
    };

    template <class T, class Compr = less<T> >
    class set;
}
```

2.3.1. Megjegyzés. cppreference-en megfigyelhető, hogy ennél több template paraméterrel is rendelkezik az `std::set` – mivel ezek mind rendelkeznek alapértelmezett értékkel, a továbbiakban az egyszerűség kedvéért figyelmen kívül hagyjuk őket.

2.3.2. Megjegyzés. Az `std::set` általában piros-fekete bináris faként van implementálva, a hatékonyság végett. Ez (vagy egy hasonlóan hatékony implementáció) fontos, mert a szabvány elvárja, hogy az `insert`, és sok egyéb tagfüggvény műveletigénye logaritmikus legyen.

Tekintsünk pár példát az `std::set` alkalmazására! Mivel ez a konténer rendezett, így ha sok adatot pakolunk bele, nagyon nehéz megmondani, hogy az elemek milyen sorrendben lesznek, így az `std::set` elemeihez többnyire csak iterátorokkal tudunk hozzáférni.

Lássunk példát pár speciális tagfüggvényre is!

```
#include <set>
#include <iostream>

template <class T>
void printSet(const std::set<T> &s)
{
    for(typename std::set<T>::const_iterator it = s.begin();
                                                it != s.end(); ++it)
        std::cout << *it << ' ';
}

int main()
{
    std::set<int> si;
    for(int i = 10; i>0; i--) //forditva!
        si.insert(i);
    std::cout << si.size() << std::endl; // 10
    si.insert(5); //5 már szerepel a halmazban
    std::cout << si.size() << std::endl; // 10

    printSet(si); // 1 2 3 4 5 6 7 8 9 10
    //operator< növekvően rendez, hiába fordított sorrendben illesztettük be az elmeket

    //adott értékű elem törlése
    std::set<int>::iterator it = si.find(4);
    if(it != si.end())
        si.erase(it);
```

```
    printSet(si); // 1 2 3 5 6 7 8 9 10

    //töröljük ki az [6, 8) intervallumot!
    std::set<int>::iterator begin = si.find(6), end = si.find(8);
    if(begin != si.end())
        si.erase(begin, end);
    printSet(si); // 1 2 3 5 8 9

}
```

Figyeljük meg, hogy az elemek törlésekor, leellenőriztük, valóban szerepel-e az az adott elem a halmazban. Mivel a `find` tagfüggvény egy past-the-end iterátorral tér vissza, hogyha az adott elemet nem találja, az azzal való egyenlőség segítéségvel tudjuk ezt megvizsgálni.

Ha egy past-the-end iterátort próbálunk törölni az `erase` függvénnyel, nem definiált viselkedést kapunk.

A következő példában tekintsünk egy `int`-től különböző típust tároló halmazt. Ne feledjük, mivel e konténer rendezi az elemeit, mindenképpen muszáj a struktúránk mellé még valamit megírnunk. Ez lehet az alapértelmezetten alkalmazott `operator<` függvény vagy egy új funktor.

```
struct Point
{
    int x, y;
    Point(int _x, int _y) : x(_x), y(_y) {}
};

std::ostream& operator<<(std::ostream& os, const Point &p)
{
    os << p.x << ' ' << p.y;
    return os;
}

//operator< y koordináta szerint rendez
bool operator<(const Point &lhs, const Point &rhs)
{
    return lhs.y < rhs.y;
}

//LessByX funktor x koordináta szerint
struct LessByX
{
    bool operator()(const Point &lhs, const Point &rhs) const
    {
        return lhs.x < rhs.x;
    }
};

int main()
{
    std::set<Point> spy;
    spy.insert(Point(3, 1));
    spy.insert(Point(1, 3));
    spy.insert(Point(2, 2));
    for(std::set<Point>::iterator it = spy.begin(); it != spy.end(); ++it)
        std::cout << *it << ", "; // 3 1, 2 2, 1 3,

    std::set<Point, LessByX> spx;
    spx.insert(Point(3, 1));
    spx.insert(Point(1, 3));
```

```
    spx.insert(Point(2, 2));
    for(std::set<Point, LessByX>::iterator it = spx.begin();
                                                it != spx.end(); ++it)
        std::cout << *it << ", "; // 1 3, 2 2, 3 1,

    spy.insert(Point(1, 1)); spx.insert(Point(1, 1));
    std::cout << spy.size() << ' ' << spx.size() << std::endl; // 3 3
}
```

2.3.3. Megjegyzés. A funktorok mellett az beszúrófüggvények használata is jó példa arra, mikor hasznosak a név nélküli temporális változók.

A fenti példában megfigyelhető, hogy bár 1, 1 koordinátákkal rendelkező pontot egyik halmaz sem tartalmaz, az mégis **ekvivalens** egy, a halmazokban már szereplő elemekkel.

$$\begin{aligned} \texttt{spy}\text{ esetében: } & \neg\big(\texttt{Point}(1,1) < \texttt{Point}(3,1)\big) \quad \wedge \quad \neg\big(\texttt{Point}(3,1) < \texttt{Point}(1,1)\big) \\ \texttt{spx}\text{ esetében: } & \neg\big(\texttt{Point}(1,1) < \texttt{Point}(1,3)\big) \quad \wedge \quad \neg\big(\texttt{Point}(1,3) < \texttt{Point}(1,1)\big) \end{aligned}$$

Vagy másféleképpen a funktorok neveivel:

$$\begin{aligned} \texttt{spy}\text{ esetében: } & \neg\texttt{less}\big(\texttt{Point}(1,1), \texttt{Point}(3,1)\big) \quad \wedge \quad \neg\texttt{less}\big(\texttt{Point}(3,1), \texttt{Point}(1,1)\big) \\ \texttt{spx}\text{ esetében: } & \neg\texttt{LessByX}\big(\texttt{Point}(1,1), \texttt{Point}(1,3)\big) \quad \wedge \quad \neg\texttt{LessByX}\big(\texttt{Point}(1,3), \texttt{Point}(1,1)\big) \end{aligned}$$

Egy egyedi rendezés használata veszélyekkel is járhat azonban. Térjünk át `std::string`-ekre, és rendezzünk azok hossza szerint.

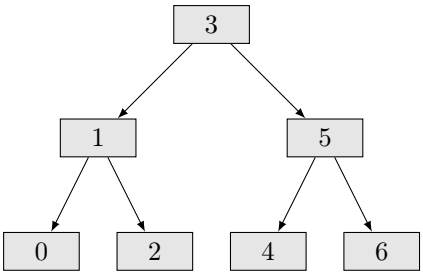
```
struct strlen
{
    bool operator()(const std::string &lhs, const std::string &rhs) const
    {
        return lhs.length() < rhs.length();
    }
};

int main()
{
    std::set<std::string, strlen> s;
    s.insert("C++");
    s.insert("Java");
    s.insert("Haskell");
    s.insert("GOD");
    std::cout << s.size() << std::endl; // 3
    std::cout << s.count("GOD") << ' ' << s.count("ADA"); // 1 1
}
```

Az eddig leírtak alapján látható, hogy az `insert` függvény nem teszi be a `GOD`-ot, lévén az a rendezés szerint a `C++`al ekvivalens. Azonban a rendezésünknek egy kellemetlen hátulütője az, hogy annak ellenére, hogy `GOD` nem került beszúrásra, ha rákérdezünk hány `GOD`-al ekvivalens elemet tartalmaz a halmaz, mégis 1-et kapunk, sőt, minden 3 hosszú `string`-re.

2.3.4. Megjegyzés. Könnyű megállapítani, hogy a fenti rendezés szimmetrikus, reflexív és tranzitív, így ekvivalenciaosztályokra osztja a `string`-ek halmazát hossz szerint.

Most figyeljük meg (emlékezzünk vissza dimatra), mi történik hogyha szigorú rendezés helyett egy gyenge rendezést definiálunk! Miért lehet ez problémás?

```
struct strlenWrong
{
    bool operator()(const std::string &lhs, const std::string &rhs) const
    {
        return lhs.length() <= rhs.length(); //ekvivalensek is lehetnek
```



1. ábra. Az `std::set` egy lehetséges ábrázolása, bináris keresőfával. Erre az irányított gráfra igaz, hogy minden gyökér a bal oldali gyerekénél nagyobb, és a jobb oldali gyerekénél kisebb.

```
    }
};

int main()
{
    std::set<std::string, strlenWrong> s;
    s.insert("C++");
    s.insert("Java");
    s.insert("Haskell");
    s.insert("GOD");
    std::cout << s.count("GOD") << std::endl; // 0
    std::cout << s.size() << std::endl; // 4
}
```

Keressünk magyarázatot erre az eredményre, vizsgáljuk meg `C++` és `GOD` ekvivalenciáját.

$$\neg\,\texttt{strlenWrong}(\texttt{"GOD"},\ \texttt{"C++"}) \quad \wedge \quad \neg\,\texttt{strlenWrong}(\texttt{"C++"},\ \texttt{"GOD"})$$

Látjuk, hogy ez a formula hamisat ad, így nem bizonyulnak majd ekvivalensnek. Ha nem szigorú részben rendezést használunk, hanem gyengét, akkor a reflexivitást is elvesztjük. Ez azt jelenti, hogy egy elem nem lehet ekvivalens önmagával!

Így `strlenWrong` nem egy jó rendezés, mivel ha ekvivalenciát vizsgálunk, sose fog igazat adni, sose tudjuk meg, egy adott benne van-e, és az elemek törlése is sok problémához vezetne.

2.3.5. Megjegyzés. Bár sokszor volt ez fentebb leírva, hangsúlyozandó hogy az `std::set` **nem** az `==` operátor segítéségével vizsgálja az ekvivalencát. Két objektum lehet egyszerre **ekvivalens** és nem **egyenlő**, függően az `==` operátor és a rendezés implementációjától.

Link: `http://en.cppreference.com/w/cpp/container/set`

2.3.6. Megjegyzés. Az `std::multiset` működése az eddigiek alapján triviális, így annak megismerését ismét az olvasóra bízom.

2.4. list

Az `std::list` konténer a `<list>` könyvtár része, és nagyon hasonló ahhoz, mint amit mi írtunk, viszont létezik fejelemmel, és kétirányú.

Az `std::vector`-ral szemben ennek a konténernek nincs `[]` operátora, hiszen mielőtt egy adott indexű elemet vissza tudna adni, el kell oda lépegetnie egyesével. Szögletes zárójel operátort csak akkor szokás írni, hogy ha az nagyon hatékony, lehetőleg konstans műveletigényű, de ez a listánál nem teljesül.

2.4.1. Megjegyzés. Bár elméletben gyorsabb egy láncolt listába beszúrni a `vector`-ral (vagy hasonló konténerrel) szemben, hisz csak pár pointert kell átállítani, ez a gyakorlatban csak kivételes esetekben teljesül. Ennek az az oka, hogy bár az elemeket egyesével odébb kell tolni egy vektorban, azok szekvenciálisan vannak a memóriában, és mivel a processzor számít arra, hogy nem csak egy adott elemet, de a környezőket is módosítani szeretnénk, minden alkalommal amikor egy adott elemet kérdezünk le, azzal

együtt a környező elemeket is visszaadja nekünk. Ennek következtében kevesebb processzorművelettel végrehajtható a tologatás.

Mivel egy láncolt listában egyesével kell haladni az egyik elemről a következőre, ez gyakran annyira költséges, hogy jobban járunk a vektor alkalmazásával akkor is, ha sokat kell a konténer közepére beszúrni.

Kivételes eset lehet, ha nagyon sok elemből álló listánk van, és azok nagy méretű objektumokat tárolnak, valamint nagyon gyakran kell két elem közé beszúrni. Ilyenkor valóban hatékonyabb tud lenni a `list`.

Mivel már a korábban megismerkedtünk a láncolt listákkal, valamint a korábbi szekciókban mutatott tagfüggvények alkalmazása gyakran teljesen megegyezik a `list`-nél használatosakkal, túl komoly példákat itt nem fogunk venni.

Azonban érdemes felhívni a figyelmet egy problémára. Kérdezzük le egy lista harmadik elemét!

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

int main()
{
    std::list<int> l;
    for (int i = 0; i < 5; i++)
        l.push_back(i);
    std::list<int>::iterator it = l.begin();
    ++it; ++it;
    std::cout << *it << std::endl; // 2
}
```

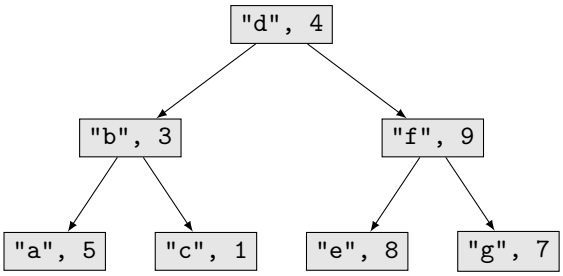
Az iterátorok léptetésére kell hogy legyen egy egyszerűbb módszer. Írhatnánk egy ciklust, azonban beszédesebb lenne egy léptető függvény írása.

Azonban nem ártana, ha nem csak tetszőleges típust tároló lista iterátorát, hanem tetszőleges konténer iterátorát is tudná ez a függvény léptetni. Mivel az `std::vector`-nál bármikor ugorhatunk egy tetszőleges elemre, hatékonyabb lenne annak konstans műveletigényű léptető függvényt írni: jó lenne egy olyan algoritmust találni, mely e kettőt egybefoglalja (erre később látunk is majd példát az STL algoritmusoknál).

Ha böngészünk cppreference-en, feltűnhet, hogy az `std::vector`-ral szemben létezik `push_front` és `pop_front` metódus: ezeket az `std::vector`-ral ellentétben konstans idő alatt el lehet végezni (elegendő a fejelemet a rákövetkező elemre állítani és kész is).

Léteznek kifejezetten a listára specifikus műveletek is. Ezek azért nagyon fontosak, mert nem igénylik azt, hogy az adott objektumot lehessen másolni (vagy mozgatni, C++11 szerint). Ez olyankor is fontos lehet, ha például a tárolt objektumok mérete nagyon nagy, és költséges lenne másolni. Példaképp, a `splice` tagfüggvény egy másik listát (is) vár paraméterül, és azt a listát fűzi be a saját elemei közé.

```
std::ostream& operator<<(std::ostream& out, const std::list<int> &l)
{
    for (std::list<int>::const_iterator it = l.begin(); it != l.end(); it++)
    out << *it << ' ';
    return out;
}

int main()
{
    std::list<int> list1, list2{};
    for(int i = 0; i<5; i++)
        list1.push_back(i);
    for(int i = 0; i<5; i++)
        list2.push_front(i);
    std::cout << list1 << std::endl; // 0 1 2 3 4
    std::cout << list2 << std::endl; // 4 3 2 1 0
```



2. ábra. Az `std::map` egy lehetséges ábrázolása, bináris keresőfával. Az ábrán egy `std::string, int` párosokat tároló `map` szerepel – figyeljük meg, hogy a a konténer a kulcs szerint rendez, az értékek a rendezés szempontjából nem számítanak.

```
    list1.splice(list1.end(), list2);
    std::cout << list1 << std::endl; // 0 1 2 3 4 4 3 2 1 0
}
```

Link: `http://en.cppreference.com/w/cpp/container/list`

2.5. map

Az `std::map` a `<map>` könyvtárban található. Ez egy egyedi kulcs-érték párokat tároló konténer, mely minden kulcshoz egy értéket rendel. Működése nagyon hasonlít az `std::set`-ére, annyi különbséggel, hogy párokat tárol (így két kötelező template paramétere van), és mindig a kulcs szerint rendez.

Van egy speciális beszúró függvénye is:

```
#include <map>
#include <iostream>

int main()
{
    std::map<std::string, int> m;
    m["Hello"] = 42;
    m["xyz"] = 8;
    m["Hello"] = 9;
    std::cout << m.size() << std::endl; // 2
    std::cout << m["Hello"] << std::endl; // 9
}
```

A `[]` operátor egy új kulcsot hoz létre, és ahhoz rendel egy új értéket. Az utolsó sornál mivel a `''Hello''`-t már tartalmazza a map, így annak az értékét felülírja.

Sajnos ez az operátor azonban okozhat pár kellemetlen meglepetést is.

```
int main()
{
    std::map<std::string, int> m;
    std::cout << m.size() << std::endl; // 0
    if(m["c++"] != 0)
        std::cout << "nem 0" << std::endl;
    std::cout << m.size() << std::endl; // 1
}
```

A `[]` operátor úgy működik, hogy ha a `map` nem tartalmazza a paraméterként kapott kulcsot, akkor létrehozza és beszúrja, ha benne van, visszaadja az értékét. Bár mi nem tettük bele azt hogy `c++` szándékosan, de pusztán azzal, hogy rákérdeztünk, akaratlanul is megtettük. (Figyeljük meg, hogy akkor tudnánk új elemet hozzátenni, ha azt írnánk pl. hogy `m["C++"] = 3`; azonban mi nem adtunk meg ehhez a kulcshoz értéket. Ilyenkor a map alapértelmezett értéket rendel a kulcshoz, szám típusoknál 0-t, így be fog szúrni egy `("c++", 0)` párt.)

```
bool contains(const std::map<std::string, int> &m)
{
    //m["c++"]; fordítási hiba: [] operator nem konstans függvény.
    return m.find("C++") != m.end();
}
```

Ez a függvény helyesen vizsgálja, hogy a `c++` benne van-e. (Ha cppreference-en rákeresünk, a `map`-nek a `find` nevű tagfüggvénye egy iterátort ad vissza a talált elemre, illetve visszaadja egy past-the-end iterátort ha nem találja meg.)

`std::map`-ban az iterálás kicsit trükkösebb, ugyanis `std::map` párokat tárol, méghozzá egy más alapból megírt struktúrát használ, az `std::pair`-t, ami kb. így néz ki:

```
template <typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};
```

Így az az adott kulcs ill. érték lekérdezése így fog kinézni:

```
for(std::map<std::string, int>::iterator i = m.begin(); i != m.end(); i++)
{
    std::cout << i->first << " " << i->second;
}
```

Amennyiben új elemeket szúrunk be, gyakran megkerülhető a problémás `[]` használata. Az `std::map` insert függvénye egy `std::pair`-t vár paraméterül, melyet könnyen tudunk alkotni `std::make_pair` függvény segítségével.

```
std::map<int, char> m;
m.insert(std::make_pair(1, 'a'));
```

Link: `http://en.cppreference.com/w/cpp/container/map`

2.5.1. Megjegyzés. Az `std::multimap` ehhez szintén hasonló, így gyakorlásként ezt ismét az olvasóra bíznám.