C++
Gyakorlat jegyzet
10. óra

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

1. Iterátor kategóriák

Korábban már elhangzott, hogy az általunk implementált `List`-hez tartozó `Iterator` és `ConstIterator` un. forward iterátorok. A forward iterator egy iterátor kategória – ebből több is van, és elengedhetetlenek az STL algoritmusok megértéséhez. 4 iterátor típust különböztetünk meg: input iterator, forward iterator, bidirectional iterator és random access iterator.

- Az **input iterator**-okon legalább egyszer végig lehet menni egy irányba, továbbá rendelkeznek `++`, `*`, `==` és `!=` operátorral.
- A **forward iterator**-okon többször is végig lehet menni, de csak egy irányba. Továbbá rendelkeznek `++`, `*`, `==` és `!=` operátorral.
- A **bidirectional iterator**-okon többször is végig lehet menni, mindkét irányba. Továbbá rendelkeznek `++`, `--`, `*`, `==` és `!=` operátorral.
- A **random access iterator**-okon többször is végig lehet menni, mindkét irányba, ezen felül a két iterátor között bármelyik elemre azonnal lehet hivatkozni. Továbbá rendelkeznek `++`, `--`, `*`, `==`, `!=`, `-` (két iterátor távolságát adja meg), összeadás `int`-el (paraméterként kapott mennyiségű elemmel előrelépés), kivonás `int`-el (hátralépés) operátorral/függvénnyel.

E listán látható, hogy pl. egy random access iterator sokkal flexibilisebb, és gyorsabb is, hisz bármikor bármelyik elemhez hozzáférhetünk, míg a mi listánk forward iteratora sokkal limitáltabb.

Az STL konténerek iterátorai is többek között különböző iterator kategóriákban vannak:

Konténer	Iteratorának kategóriája

`std::vector`
`std::deque`
Tömb
	`random access iterator`

`std::list`
`std::set`
`std::multiset`
`std::map`
`std::multimap`
	`bidirectional iterator`

`List` (általunk írt) `std::forward_list` (C++11)	`forward iterator`
`std::istream_iterator` `std::ostream_iterator`	`input iterator`

Világos, hogy egy flexibilisebb iterátorral több mindent meg tudunk tenni, vagy ugyanazt a funkciót hatékonyabban is meg tudjuk valósítani. Emiatt minden STL algoritmusnak (mint pl. az `std::find`) tudnia kell a template paraméterként kapott iterátor kategóriáját.

Azonban ezt hogyan tudjuk elérni? Kézenfekvő megoldás lenne pl. ez:

```
struct input_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_access_iterator_tag {};

class ExampleIterator
{
```

```
public:
    typedef iterator_category forward_iterator_tag;
    //...
};
```

Ezzel a trükkel könnyedén tudunk hivatkozni egy iterátor típusára, elég csupán annyit írnunk hogy `ExampleIterator::iterator_category`. Sőt, ha tartjuk ahhoz a konvencióhoz magunkat, hogy minden iterátor-nak legyen egy `iterator_category` típusa, mely a fentebb felsoroltak egyike, akkor nagyon egyszerűen tudunk olyan algoritmusokat megalkotni, melyek bármilyen iterátorral tudnak dolgozni (erre nemsokára egy példát is látunk).

Természetesen számos egyéb információra is szükségünk lehet egy iterátorral kapcsolatban. A dereferáló perátor visszatérési értéke, erre a típusra mutató pointer típusa, stb. Ezt a legegyszerűbben úgy tudjuk megadni, ha származunk az `std::iterator` típusból.

1.0.1. Megjegyzés. Az öröklődés később lesz alaposabban boncolgatva, egyenlőre mondjuk azt, hogy az öröklődés következtében mindent átpakolunk az `std::iterator` osztályból a mi iterátorunkba.

Link: `http://en.cppreference.com/w/cpp/iterator/iterator`

Látható hogy ennek osztálynak két kötelező template paramétere van, egy iterátor típus tag, és dereferáló operátor visszatérési értéke.

```
template <class T>
class Iterator : public std::iterator<std::forward_iterator_tag, T>
{
    //...
};

template <class T>
class ConstIterator : public std::iterator<std::forward_iterator_tag, T>
{
    //...
};
```

A cppreference-en megfigyelhető, hogy az `std::iterator` több típust is tartalmaz, mellyel az iterátorunk jellemezhető. Minden STL konténer iterátora rendelkezik ezekkel a típusokkal, és számunkra is kifizetődő használni.

Ennek hála, most már tudjuk az STL algoritmusokat használni a láncolt listánkkal!

A fenti táblázatban megfigyelhettük, hogy a tömbnek is van iterátora – természetesen ez az a pointer, amivel végig tudunk lépegetni a tömb elemein (pl. egy `int[]` tömb típus esetén ez `int*` lenne). Ez valóban iterátor, hisz dereferálható, léptethető, stb. Azonban egy `int*` típusnak nincs `iterator_category`-ja! Ezt a problémát megoldhatjuk úgy, hogy ha egy tetszőleges `It` iterátor esetén nem `It::iterator_category`-val kérdezzük le az iterátor kategóriáját, hanem az `std::iterator_traits` osztállyal:

```
std::iterator_traits<std::vector<int>::iterator> >::iterator_category cat1;
std::iterator_traits<int*>::iterator_category cat2;
std::iterator_traits<List::Iterator>::iterator_category cat3;
```

2. STL algoritmusok

Az STL algoritmusok az `<algorithm>` könyvtárban találhatóak, és számos jól ismert és fontos függvényt foglalnak magukba, mint pl. adott tulajdonságú elem keresése, partícionálás, szimmetrikus differencia meghatározása, stb.

Az egyik legnagyobb erősségük, hogy nagyon jól olvasható kódot eredményez a használatuk – túl azon hogy valószínűleg évtizedek óta fejlesztett könyvtár hatékonyabban implementációkat tartalmaz mint amiket mi tudnánk rövid idő alatt írni, sokkal beszédesebb, kifejezőbb kódot tudunk alkotni velük, ami ezáltal könnyebben debugolható és értelmezhető.

Ahogy az az iterátor kategóriáknál is említve volt, nem minden iterátorral működik minden algoritmus – ezt bármikor könnyedén le tudjuk ellenőrizni a `cppreference.com` oldalról.

2.0.1. Megjegyzés. Természetesen nagyon sok algoritmussal rendelkezik a szabvány – e jegyzetnek nem célja az összeset felsorolni, inkább ezeknek a gondolatmenetét próbálja átadni.

2.1. Bevezető az algoritmusokhoz

Az STL algoritmusok alap gondolatmenetét jól demonstrálja az alábbi példa: Írjunk egy függvény mely egy `std::vector<int>` konténernek a második adott értékű elemére hivatkozó iterátort ad vissza!

```
#include <vector>
#include <iostream>

typedef std::vector<int>::iterator VectIt;

VectIt findSecond(VectIt begin, VectIt end, const int &v)
{
    while(begin != end && *begin != v)
    {
        ++begin;
    }
    if (begin == end)
        return end;
    ++begin;
    while(begin != end && *begin != v)
    {
        ++begin;
    }
    return begin;
}

int main()
{
    std::vector<int> v;
    for (int i = 0; i<10; i++)
        v.push_back(i);
    v.push_back(5);
    std::vector<int>::iterator it = findSecond(v.begin(), v.end(), 5);
    if (it != v.end())
        std::cout << *it << std::endl; // 5
}
```

Amennyiben `findSecond` nem talál két v-vel ekvivalens elemet, egy past-the-end iterátort ad vissza.

Azonban könnyű látni, hogy egyáltalán nem fontos információ az algoritmus szempontjából, hogy a `vector` `int`-eket tárol.

```
template<class T>
typename std::vector<T>::iterator
    findSecond(typename std::vector<T>::iterator begin,
               typename std::vector<T>::iterator end,
               const T &v)
{
    while(begin != end && *begin != v)
    {
        ++begin;
    }
    if (begin == end)
        return end;
    ++begin;
    while(begin != end && *begin != v)
    {
```

```
        ++begin;
    }
    return begin;
}
```

Sajnos sikerült elég kilométer függvény fejlécet sikerült írnunk, de a célt elértük.

Megállapítható, hogy itt még azt se kell tudnunk, minek az iterátorával dolgozunk, elegendő annyit tudnunk, hogy a ++ iterátorral lehet léptetni, és össze tudjuk hasonlítani őket az != operátorral, azaz teljesítik egy input iterator feltételeit. Első körben vegyük az összehasonlítandó elem típusát külön template paraméterként.

```
template <class InputIt, class Val>
InputIt findSecond(InputIt begin, InputIt end, const Val &v)
{
    //...
}
```

Az STL algoritmusok iterátorokkal dolgoznak, azonban az, hogy minek az iterátorát használják, egyáltalán nem fontos tudniuk: elegendő annyi, hogy rendelkeznek azokkal az operátorokkal, melyeket fent felsoroltunk. Így például a mi listánk iterátorát ugyanúgy megadhatnánk a fenti függvények, mint egy `std::vector<int>`-ét.

A korábban elhangzottak alapján implementáljunk az `advance` függvényt, mely egy iterátort léptet előre: Amennyiben a paraméterként kapott iterátor kategóriája bidirectional iterator, egyesével lépegessünk a kívánt helyre, random access iterator esetén egyből ugorjunk oda. Ehhez használjuk ki azt, amit fentebb láttunk: az `std::iterator`-ból való öröklés hatására már le tudjuk kérdezni az iterátorunk kategóriáját!

```
template<typename BDIT>
BDIT algorithm(BDIT it, int pos, std::bidirectional_iterator_tag)
{
    for(int i = 0; i<pos; i++)
        ++it;
    std::cout << "Slow" << ' ';
    return it;
}

template<typename RAIT>
RAIT algorithm(RAIT it, int pos, std::random_access_iterator_tag)
{
    std::cout << "Fast" << ' ';
    return it + pos;
}

template<typename IT>
IT advance(IT it, int pos)
{
    typedef typename std::iterator_traits<IT>::iterator_category cat;
    return algorithm(it, pos, cat());
}

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

```
    //advance iterátorral tér vissza, dereferálni kell a kiíratáshoz
    std::cout << *advance(v.begin(), 3) << std::endl; // Fast 3
    std::cout << *advance(l.begin(), 3) << std::endl; // Slow 6
}
```

2.1.1. Megjegyzés. Az `std::vector` és `std::list`-nél felmerült iterátor lépegetésre találtunk megoldást: a fentihez hasonló működésű `std::advance` alkalmazható e célra.

Így tudjuk elérni azt, hogy különböző iterátor kategóriára különböző algoritmust használjunk. Természetesen, ez csak egy demonstráció volt, elképzelhető hogy máshogy (és nyilván hatékonyabban) néz ki ennek az implementációja.

Ezek ismeretében vágjunk bele az STL algoritmusokba!

2.1.2. Megjegyzés. Cppreference-en megfigyelhető, hogy irtózatosan sok algoritmus van – csak úgy mint a konténereknél, a fontosabb algoritmusokról lesz szó, míg a többivel való ismerkedés gyakorlás végett az Olvasóra bízom.

2.2. find, find_if

Az `std::find` segítségével az első adott értékű elemre hivatkozó iterátort kaphatunk. Keressük meg a 4-el ekvivalens elemet!

```
int main()
{
    std::vector<int> v;
    for(int i = 0; i<10; i++)
        s.insert(i);
    std::vector<int>::iterator result = find(v.begin(), v.end(), 4);
    if (result != v.end())
        std::cout << *result << std::endl; // 4
    else
        std::cout << "4 nem eleme" << std::endl;
}
```

2.2.1. Megjegyzés. Figyeljük meg, hogy nem `std::find`-ot írtunk, a névtér rezolúciót elhagytuk. Ezt az ADL miatt tehetjük meg.

Ha `std::vector` helyett `std::set` konténerben keresnénk egy elemet, akkor lévén az `std::set` egy bináris fa a keresést logaritmikus idő alatt is meg tudjuk tenni: ezért szokás az ilyen speciális konténereknek egyedi find függvényt írni, ami az `std::set` esetében egy tagfüggvény.

```
std::set<int>::iterator it = v.find(4);
if (i != v.end())
    //...
```

Azonban megállapítandó, hogy míg az `std::find` az `==` operátorral végzi az összehasonlítást, addig az `std::set` a template paraméterként kapott rendezéssel! (ami alapértelemzetten a `<` operátortól függ.)

```
struct Circle
{
    int x, y, r;
    Circle(int _x, int _y, int _r) : x(_x), y(_y), r(_r) {}
};

bool operator<(const Circle &lhs, const Circle &rhs)
{
    return lhs.r < lhs.r;
}

bool operator==(const Circle &lhs, const Circle &rhs)
```

```
{
    return lhs.r == rhs.r && lhs.x == rhs.y && lhs.y == rhs.y;
}

int main()
{
    std::set<Circle> s;
    for(int i = 0; i<10; i++)
        s.insert(Circle(i, i + 1, i + 4));
    for(int i = 0; i<10; i++)
        s.insert(Circle(i, i + 1, i + 2));

    std::set<Circle>::iterator it1 = s.find(Circle(2, 3, 4));
    std::set<Circle>::iterator it2 = find(s.begin(), s.end(),
                                                    Circle(2, 3, 4));

    if(it1 != s.end())
        std::cout << it1->x << ' ' << it1->y << ' ' << it1->r; // 0 1 4
    if(it2 != s.end())
        std::cout << it2->x << ' ' << it2->y << ' ' << it2->r; // 1 9 10
}
```

Ezzel semmi komolyabb gond nincs, de nem szabad meglepődni, amikor az ember ettől függően más megoldást kap mint amire számít.

Néha nem egy adott értéket keresünk, hanem egy adott tulajdonságú elemet. Ezt megtehetjük az `std::find_if` függvénnyel, mely nem egy adott értéket, hanem egy *unary predicate*-et vár (azaz, ami egy adott elemhez egy logikai értéket rendel).

```
struct BigNumber
{
    bool operator()(const int param) const
    {
        return param > 100;
    }
};

int main()
{
    std::vector<int> v;
    v.push_back(1); v.push_back(7); v.push_back(110);
    std::vector<int>::iterator it = find_if(v.begin(), v.end(),
                                                        BigNumber());
    if(it != v.end())
        std::cout << *it << std::endl; // 110
}
```

2.3. sort és stable_sort

A következő példában próbáljuk rendezni egy `vector` elemeit!

```
std::vector<int> v {6,3,7,4,1,3};
std::set<int> s {6,3,7,4,1,3};
std::set<int> s1(v.begin(), v.end());
v.assign(s1.begin(), s1.end());
for(std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
{
    std::cout << *it << std::endl; // 1 3 4 6 7
}
```

2.3.1. Megjegyzés. Az fenti inicializálások a c++11-es újítás részei, sok magyarázatra gondolom nem szorulnak.

Trükkösen kihasználtuk, hogy az `std::set` alapértelmezetten rendezett: átpakoltuk az elemeket abba, majd visszaillesztettük az eredeti konténerbe. No persze, ez minden, csak nem hatékony. Ráadásul az egyik 3-as kiesett: az `std::set` megszűrte az elemet, mert ekvivalenseket nem tárol.

2.3.2. Megjegyzés. Vajon milyen konténert kellett volna használni az `std::set` helyett, mely nem szűri ki a duplikált elemeket?

Az ilyen jellegű barkácsolások sose fogadhatóak el hatékonyság szempontjából. Ismerkedjünk meg az ennél sokkal hatékonyabb `std::sort`-al!

```
std::vector<int> v {6,3,7,4,1,3};
std::sort(v.begin(), v.end());
for(std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    std::cout << *it << ' '; // 1 3 3 4 6 7
```

Az `std::sort`, ha külön paramétert nem adunk neki, az `<` operátor szerint rendez.

Rendezzük a fenti elemeket úgy, hogy a `>` operátor szerint legyenek sorban!

```
struct Greater
{
    bool operator()(int lhs, int rhs) const
    {
        return lhs > rhs;
    }
};

int main()
{
    std::vector<int> v {6,3,7,4,1,3};
    sort(v.begin(), v.end(), Greater());
    for(std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << ' '; // 7 6 4 3 3 1
}
```

2.3.3. Megjegyzés. Ez jól demonstrálja, hogy az `std::sort`, csak úgy mint a legtöbb STL algoritmus, számos overload-al rendelkezik.

Az `std::sort` az ekvivalens elemek sorrendjét nem (feltétlenül) tartja meg: rendezés után azoknak a sorrendje nem definiált.

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

int main()
{
    std::vector<std::string> v;
    v.push_back("ADA");
    v.push_back("Java");
    v.push_back("1234567");
    v.push_back("Maci");
    v.push_back("C++");
    v.push_back("Haskell");
```

```
    sort(v.begin(), v.end(), StringLength());

    for(std::vector<std::string>::iterator it = v.begin(); it != v.end();
                                                                ++it)
        std::cout << *it << ' ';
}
```

Lehetséges output: `C++, ADA, Java, Maci, Haskell, 1234567`.

Lehetséges output: `ADA, C++, Maci, Java, Haskell, 1234567`.

Amennyiben fontos, hogy az ekvivalens elemek relatív sorrendje megmaradjon, használjunk `std::sort` helyett `std::stable_sort`-ot, mely pontosan ezt csinálja.

```
std::stable_sort(v.begin(), v.end(), StringLength());
```

Kiíratás esetén az output: `ADA, C++, Java, Maci, Haskell, 1234567`

2.3.4. Megjegyzés. Ennek nyilván ára is, van, általában az `std::stable_sort` kevésbé hatékony.

Link: `http://en.cppreference.com/w/cpp/algorithm/sort`.

2.4. partition, stable_partition

Osszunk szét egy vektort úgy, hogy a vektor egyik felében páros, a másik felében páratlan számok legyenek!

```
struct Even
{
    bool operator()(const int param)
    {
        return param % 2 == 0;
    }
};

std::vector<int> v;
for(int i = 0; i<10; i++)
    v.push_back(i);
std::partition(v.begin(), v.end(), Even());
for(int i = 0; i<v.size(); i++)
    std::cout << v[i] << std::endl;
```

Lehetséges kimenet: `1 5 7 9 3 2 6 4 8`

Rögtön láthatjuk, hogy a célt sikerült elérnünk, azonban pont mint a sort esetében, az `std::partition` nem feltétlenül tartja meg az ekvivalens elemek (amelyek mind igazat adnak, vagy hasonló módon melyek mind hamisat adnak) relatív sorrendjét. Ez nem is mindig cél, azonban ha fontos hogy megőrződjön a sorrend, használható e célra az `std::stable_partition`, melynek a működése az `std::stable_sort` megértése után triviális.

2.5. remove, remove_if

Az `std::remove` algoritmus a konténerben lévő elemek törlését segíti elő. Töröljük ki egy vectorból az összes 3-al ekvivalens elemet!

```
std::vector<int> v{1,2,3,3,4,5,6};
std::cout << v.size() << std::endl; // 7
remove(v.begin(), v.end(), 3);
std::cout << v.size() << std::endl; // 7
```

Legnagyobb meglepetésünkre, ez semmit se fog törölni: az `std::remove` átrendezi a konténert, úgy hogy a konténer elején legyenek a nem törlendő elemeket, és utána a törlendőek, és az első törlendő elemre visszaad egy iterátort. Ennek segítségével tudjuk, hogy ettől az iterátortól a past-the-end iteratorig minden törlendő.

```
std::vector<int> v{1,2,3,3,4,5,6};
std::cout << v.size() << std::endl; // 7
auto it = remove(v.begin(), v.end(), 3);
v.erase(it, v.end());
std::cout << v.size() << std::endl; // 5
```

2.5.1. Megjegyzés. Ebben a kódban van egy c++11-es újítás is: az `auto` kulcsszó megadható konkrét típus helyett. Ilyenkor az egyenlőségjel bal oldalán lévő objektum típusára helyettesítődik a kulcsszó. Példaképp, fent a fordító meg tudja határozni, hogy az `std::remove`-nak a visszatérési értéke `std::vector<int>::iterator` lesz, így a kód azzal ekvivalens, mintha ezt írtuk volna:

```
std::vector<int>::iterator it = remove(v.begin(), v.end(), 3);
```

Bár az `auto` kulcsszót sokáig lehetne még boncolgatni, legyen annyi elég egyenlőre, hogy a template paraméter dedukciós szabályok szerint működik (c++11et nem szükséges tudni a vizsgához).

Az `std::remove_if` működése az `std::find_if` megértése után triviális.

Természetesen felmerülhet a kérdés, mégis mi értelme ennek az algoritmusnak, ha az `std::partition` ugyanezt csinálja? A válasz az, hogy nem ugyanezt csinálja: bár az `std::partition` is felosztja a konténert, minden egyes elem mely megtalálható volt a konténerben, az algoritmus lefutása után is ott lesz. Mivel azonban az `std::remove` számol azzal, hogy lefutása után a konténer hátsó része törölve lesz, ezért nem végez fölösleges műveleteket azzal, hogy oda másolja a törlésre szánt elemet.

Példaképp ha egy tömb elemei rendre `1,2,3,4,5`, és szeretnénk minden páros elemet törölni, elképzelhető, hogy a tömb az `std::remove` után így fog kinézni: `1,3,5,4,5` (és értelemszerűen nullától indexelve a harmadik elemre adna vissza egy iterátort).

„Általában az emberek nem szeretik, hogyha kicsesznek velük, és ha kicseszel a kollégáiddal, morcosak lesznek”
/Horváth Gábor/