C++
Gyakorlat jegyzet
10. óra.

A jegyzetet Umann Kristóf készítette Brunner Tibor és Horváth Gábor gyakorlatán. (2017. június 11.)

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 különböző iterator kategóriákban vannak:

STL konténer	Iteratorának kategóriája
`std::vector` `std::deque`	`random access iterator`

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

1.0.1. Megjegyzés. `input iterator`-ra példa lehet az `std::istream_iterator` osztály, míg az általunk megírt lista iterátora egy `forward_iterator`-nak számít.

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.

Egy iterátornak úgy tudjuk a legegyszerűbben megadni a típusát, ha származunk az `std::iterator` típusból.

1.0.2. 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, mely jelöli hogy az iterátorunk milyen kategóriában van, milyen típussal tér vissza a dereferáló operátor, stb. Ennek segítségével pl. a listánk iterátorának kategóriája lekérdezhető az `List::Iterator::iterator_category`-val (erre nemsokára példát is látni fogunk).

1.0.3. Megjegyzés. Ennek alaposabb megértését az olvasóra bízom.

Így már fogjuk tudnuk majd használni az `STL` algoritmusokat is a konténereinken.

2. STL algoritmusok

2.1. Bevezető az algortmusokhoz

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 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>::const_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.

2.1.1. Megjegyzés. Lévén a konténert nem módosítjuk, bátran dolgozhatunk a nagyobb biztonságot nyújtó `const_iterator`-ral.

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

```
template<class T>
typename std::vector<T>::const_iterator
    findSecond(typename std::vector<T>::const_iterator begin,
               typename std::vector<T>::const_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 függvénydeklarációt 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 egy 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 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);
    }
    std::cout << *advance(v.begin(), 3) << std::endl; // Fast 3
    std::cout << *advance(l.begin(), 3) << std::endl; // Slow 6
}
```

2.1.2. 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 (bár nyilván ez jóval hatékonyabban van megoldva a standard könyvtárban).

Ezek ismeretében vágjunk bele az STL algoritmusokba!

2.1.3. 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

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

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

2.2.1. Megjegyzés. Lévén az `std::set` egy bináris fa, így várhatóan 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 = s.find(42);
if (i != s.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 = std::find(s.begin(), s.end(),
                                               Circle(2, 3, 4));

    std::cout << it1->x << ' ' << it1->y << ' ' << it1->r; // 0 1 4
    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.

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.

Most 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.

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};
    std::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
}
```

2.3.2. 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");

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

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

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

2.4. remove

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
std::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 = std::remove(v.begin(), v.end(), 3);
v.erase(it, v.end());
std::cout << v.size() << std::endl; // 5
```

2.4.1. Megjegyzés. Ebben a kódban van pár 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 itt `std::vector<int>::iterator` lesz, így a kód azzal ekvivalens, mintha ezt írtuk volna:

```
std::vector<int>::iterator it = std::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).

2.5. Végszó az algoritmusokhoz

Megjegyzendő, hogy nem minden algoritmus működik minden iterátor típussal (pl. az `std::sort` random access iteratort vár), és vannak egyes algoritmusok, melyeknek speciális előfeltételei is vanak (például az `std::unique` egy rendezett konténert vár).

Fontos az is, hogy ezek az algoritmusok általában nagyon hatékonyak, így gyakran nem is érdemes saját magunktól megírni.

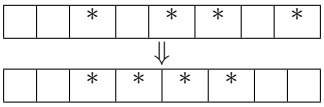
Gyakran nincs szükség arra, hogy a függvénynevek elé kiírjuk, melyik névtérből származnak.

```
//...
auto it = remove(v.begin(), v.end(), 3);
//...
```

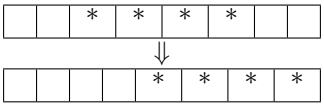
Itt a fordító a függvény paraméterekből ki tudja találni , hogy milyen névtérből van az algoritmus. Ezt *Argument Dependent Lookup*-nak nevezzük, vagy röviden ADL-nek. Mivel a `vector` ugyanúgy az `std` névtérből származik, és azt adtunk meg paraméternek, tudni fogja a fordító, hogy a `remove`-ot is abban a névtérben kell keresni.

2.6. Gyakorló feladat

Vegyünk egy konténert, és jelöljünk meg benne bár elemet, melyektől azt szeretnénk, hogy legyenek egymás mellett, de minden más elem sorrendje ne változzon.



Próbáljuk a csillaggal jelölt elemet egymás mellé tenni: ezt megtehetjük pl. két `stable_partition`-nel, mert az ekvivalens elemek sorrendjét nem változtatják.
Most csináljuk azt, hogy a kijelölt elemek a konténer végén legyenek, de a többi elem sorrendje ne változzon! Itt használhatjuk az előbbi algoritmusok után az `std::rotate`-et.



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