A Programozás Alapjai 2
Objektumorientált szoftverfejlesztés

Dr. Forstner Bertalan
forstner.bertalan@aut.bme.hu

BME AUT
Department of Automation and Applied Informatics

Standard Template Library

1. Tárolók

Tárolók

- Ne kelljen újból és újból megírni
- Fontosabbak:
  - vector: dinamikus tömb
  - set: halmaz
  - list: kétszeresen láncolt lista
  - map: asszociatív tömb

Alapkoncepció: iterátor

```
char str[]=”hello”;
int count = 0;

for(char* piterator=str; *piterator
!=0; ++piterator) {
  if(piterator==’h’)
    count++;
}
```

Alapkoncepció: iterátor

```
int t[] = {41, 42, 43, 44};
int* pend = t+sizeof(t);
int count = 0;

for(int* piterator=t;
piterator!=pend; ++piterator) {
  if(*piterator == 42)
    count++;
}
```

Vector

```
#include <vector>
int main() {
  vector<int> intVector;
  intVector.reserve(2);
  intVector.push_back(1);
  intVector.push_back(2);

  for(vector<int>::iterator it=intVector.begin();
      it!=intVector.end();
      ++it)
    {
      cout<<*it;
    }
}
```

Vector

- Két fontos mérőszám:
  - size (mennyi elem van most benne)
  - capacity (ált. duplázódik. Sokba kerül, ezért nem egyesével.)
- A hozzáadott elem lemásolódik, ez része az STL alapfilozófiájának
- Az STL tárolókat iterátorokkal járjuk be.
  - Ez az előző példában használt pointer általánosítása, minden tárolónál működik, ott is, ahol igazából sima pointerrel nem lehetne így megvalósítani
  - (pl. láncolt lista, vagy bináris fa van a háttérben).
- Minden egyes tárolótípusra létezik egy iterátortípus is, a fenti esetben ez a vector<int>::iterator.

Vector

- A tároló első elemére mutató iterátor a tároló begin(), az utolsó utánira mutató az end() tagfüggvényével érhető el.
- Pointereknél megszokott műveletek (+int, -int, ++, --, …) működnek
- * dereferencia is mindig, az épp mutatott elemet adja vissza
- insert tagfüggvény paraméterei: egy iterátor (ami elé szúr) és az érték.
  - intVector.insert(intVector.begin(), 0);
- Elem módosítása
  - *intVector.begin() = 2;

Vector

- erase() törli az aktuális elemet, clear() törli mindet.
  - Erase visszatér a következő elemre mutató iterátorral, ami fontos, hisz az, amelyiket épp töröljük, érvénytelenné válik…
- indexelés a hagyományos módon működik, de nem ellenőrzni a túlcímzést!

Array

- std::array
- Statikus méret, fordítási időben ismert.
- Elemei közvetlenül az osztályban tárolva
  - Tehát pl. stackre másolva felkerül az összes adat
- Gyakran hatékonyabb, mint vector (ha a fenti főbb kompromisszumok megfelelőek)

List

- std::list
- Mint vector, de nem folytonos a memóriaterület
  - Nem kell mindig újrafoglalni, mint pl. vector esetén
- Nincs előrefoglalás
- Nem random-access
- Nem érhető el az alattas tömb

További tárolók: halmaz (set), asszociatív tömb (map)

- **Halmazban a tárolás sorrendjét a tároló dönti el, „nem a programozó”, hiszen mások az elvárások:**
  - **hatékonyan kell tudni megmondani, hogy egy elem a halmazban van-e vagy sem**
- **Ahol a sorrendet a tároló felhasználója határozza meg: szekvenciális tároló, ahol pedig a berakott elem: asszociatív tároló**

Asszociatív tárolókhoz szükséges rendezési elv

szigorúan gyenge rendezés

- **ha x<y, akkor nem y<x (antiszimmetrikus)**
- **ha x<y, y<z, akkor x<z (tranzitív)**
- **x<x sosem igaz (irreflexív)**

Ezeket érdemes ellenőrizni a tárolt típus operator< függvényére

Tehát: iterátorok

- iterátorok a rajtuk végezhető műveletek alapján csoportosíthatóak:
  - beviteli iterátorok (input iterator)
  - kimeneti iterátorok (output iterator)
  - előreléptető iterátorok (forward iterator)
  - kétirányú iterátorok (bidirectional iterator)
  - véletlen hozzáférésű iterátorok (random access iterator)

Beviteli iterátorok

```
main() {
  istream_iterator<char> eofStreamIterator;
//default konstruktor = eof iterator
  for(istream_iterator<char> chariterator(cin);
      charIterator !=eofStreamIterator;
      ++charIterator)
  {
    cout<< *charIterator;
  }
}
```

Beviteli iterátorok

- Iterátor alaposztályból származik, és az egyes iterátor-műveleteket leképezi az istream osztály tagfüggvényeire ➔ iterátoradapter
- Default konstruktorral eofIterator (ld. tárolók end() függvénye)
- Nem a cin-re használjuk, hanem általában állományok esetén (ifstream)

Kimeneti iterátorok

```
ostream_iterator<char> outStream(cout, "*"); //elvalaszto szekvencia

for (istream_iterator<char> charIterator(cin);
      charIterator != eofStreamIterator;
      ++charIterator)
{
      *outStream = * charIterator;
}
```

- **Fekete lyukba** írunk: nem tudjuk visszaolvasni a kiírt értéket

Előre léptető iterátor

Előre léptető iterátor

- Egy szekvenciát képesek bejárni, csak a legelejétől kezdve, a végéig
- de ezt többször is.
- Az általuk mutatott elem lehet megváltoztatható, vagy megváltoztathatatlan
- Kb. kimeneti + bemeneti
- Ilyent adhat pl. egy egyirányban láncolt lista

További léptető iterátor

A **kétirányú**, ami visszafelé is tud „menni”

A **véletlen hozzáférésű** iterátor, ami tetszőleges méretű ugrásokat tesz lehetővé, kb. mint a C-s pointer volt a legelső példában

Iterátor adapterek

Fordító iterátor

becsomagolja az eredeti iterátort, és megváltoztatja a bejárás irányát

Beszúró iterátor (inserter)

Beszúró iterátorok

- Kimeneti iterátor kategóriába tartoznak
- Értékadásnál nem az éppen mutatott elemet írják felül, hanem beszúrják az értékül adott elemet egy meghatározott pozícióba.

  Gondoskodik róla, hogy legyen hely.
- A pozíciótól függően 3 különböző beszúró iterátor típust különböztetünk meg:
  - előre beszúró (front inserter)
  - hátra beszúró (back inserter)
  - általános beszúró (general inserter)
- Asszociatív tárolók esetén a pozíció csak ajánlás

- Példa

Beszúró iterátorok

```
vector<const char*> names;
names.push_back("Hegel");
insert_iterator<vector<const char*> >
        insertIterator(names, names.begin());
*insertIterator = "Hediegger";
*insertIterator = "Kant";
*insertIterator = "Nietzsche";
```


Intermission

Tartományok

- Két iterátorral határolt elemek balról zárt, jobbról nyílt intervalluma
- Tartomány beszúrása esetén pl. csak egyetlen elemmozgatás történik a tárolóval, nem egyesével
- Példa: vektor tartalmát listába másoljuk át

Tartományok

```
vector<const char*> namesVector;
namesVector.push_back(“Hegel”);
namesVector.push_back(“Kant”);
namesVector.push_back(“Heidegger”);

list<const char*> namesList;
namesList.insert(namesList.begin(),
    namesVector.begin(),
    namesVector.end());
```

2. Algoritmusok

Algoritmusok: bevezetés

- Iterátorok által kijelölt tartományokon végeznek műveleteket
- Az algoritmus előírja az iterátor kategóriáját
  - ami meghatározza a rajta végezhető műveleteket, de ezen kívül más megkötést nem tesznek
  - ➔ hatalmas rugalmasság
- Minden olyan adatstruktúrán képesek műveletet végezni, amely közvetlenül vagy iterátoradapteren keresztül támogatja az adott iterátort.
- Mivel csak az iterátorokon keresztül férhet hozzá az algoritmus az adatstruktúrához, nem feltétlenül a leghatékonyabb az adott adatstruktúrán, nem tudja kihasználni annak belső szerkezetét.

Az állatorvosi lovunk: numbers

```
int main() {
  vector<int> numbers;
  typedef vector<int>::iterator numbers_iterator;
  numbers_iterator position;
        //ez meg kelleni fog mindenfelere

  numbers.push_back(2);
  numbers.push_back(3);
  numbers.push_back(4);
  numbers.push_back(1);
}
```

A félév eleje: MAX, MIN

```
position =
        min_element(numbers.begin(), numbers_end());
cout << *position; //mivel iterator
```

find

- A find visszatér az első olyan tárolóbeli értékre mutató iterátorral, amely megegyezik a megadott elemmel
- vagy ha nincs eredmény, akkor a tartományt kijelölő utolsó iterátorral (pl. end())

find

```
position=find(numbers.begin(), numbers.end(), 4);
if(position!=numbers.end())
  cout <<*position;
else
  cout<<”Nincs meg.”;
```

másolás

- A **másolás** művelet igen gyakori
- A copy **paraméterei**:
  - InputIterator First,
  - InputIterator Last,
  - OutputIterator DestinationBegin
- A kimeneti iterátorunk lehet **beszúró** iterátor, hisz az gondoskodik pl. arról, hogy a céltárolóban legyen hely.

Példák

másolás

```
vector<int> numbers2;
copy(numbers.begin(), numbers.end(),
back_inserter(numbers2));
```

- numbers2 tartalmát szeretnénk cout-ra kiíratni vesszőkkel elválasztva:

```
copy(numbers2.begin(), numbers2.end(),
ostream_iterator<int>(cout, ”,”));
```

Műveletek, mint algoritmus argumentumok

- find_if algoritmus: 3 paramétere van:
- InputIterator First és Last
- Predicate Pred,
  - ami egy bool-lal visszatérő függvény, ami akkor igaz, ha az adott paraméter kielégíti a kívánt feltételt.
- Példa az első páratlan számot keressük meg a numbers tömbben

find_if

```
bool is_odd(int num)
{
  return num%2 != 0;
}

position = find_if(numbers.begin(), numbers.end(),
is_odd);
if(position!=numbers.end()) …
```

funktor

- A C++ sablonok behelyettesítő természetéből adódik, hogy a Pred predikátum helyére leírhatunk minden olyan adatstruktúrát, amely után zárójeleket írva
  - nem kapunk szintaktikai hibát, és
  - if feltételvizsgálat esetén értelmes bool eredményt ad.

- Ez lehet pl. az operator()() felüldefiniálása is egy osztálynál
- Ez esetben az osztályt függvényobjektumnak, illetve funktornak hívjuk.

find_if

```
class is_odd
{
  public:
    bool operator()(int num)
     {return num%2 != 0;}
};
is_odd isOdd;
position = find_if(numbers.begin(), numbers.end(),
isOdd);
```

funktor

- A fenti önmagában persze nem jobb, mint a sima függvény
- Ha azonban egyes elemeket, értékeket meg kell jegyeznünk, akkor jobb, mint a sima függvény + statikus változók
- Pl. a feladat a numbers vektor tagjainak négyzetösszegét kiszámítani
- Ehhez a for_each algoritmust használjuk
  - Ez a tartomány minden elemére meghívja a 3. paraméterként átadott függvényt vagy funktort, és a megadott függvénypointerrel vagy funktorral tér vissza

for_each

```
class squareSum
{
    unsigned sum;
  public:
    squareSum() :sum(0) {}
    void operator()(int num) { sum += num*num; }
    unsigned getSum() const { return sum; }
};

squareSum ss =
    for_each(numbers.begin(), numbers.end(), squareSum());
cout << "Sum is " << ss.getSum() << endl;
```

for_each

```
template<class _InputIterator, class UnaryFunction >
UnaryFunction my_for_each(_InputIterator _First,
_InputIterator _Last, UnaryFunction _Func)
{
	for (; _First != _Last; ++_First) {
		_Func(*_First);
	}
	return _Func;
}
```