Estructures lineals III

Programació 2

Facultat d'Informàtica de Barcelona, UPC

Conrado Martínez

Primavera 2019

- Apunts basats en els d'en Ricard Gavaldà
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Llistes

Llistes

Les llistes ens ofereixen operacions per a fer:

- Recorreguts seqüencials de tots els elements
- Inserció d'un element nou a qualsevol punt de la seqüència
- Eliminació d'un element qualsevol
- Concatenació

Iteradors

- El mecanisme que permet fer això amb les `list` de la STL són els iteradors
- Un *iterador* és un objecte que designa (marca, apunta, referencia) un element d'una llista o un altre contenidor
- Operacions sobre iteradors:
  - Avançar al següent element: `++it`
  - Retrocedir a l'anterior: `--it`
  - Comparar iteradors: `it1==it2`, `it1!=it2`
  - Accedir a l'objecte designat: `*it`

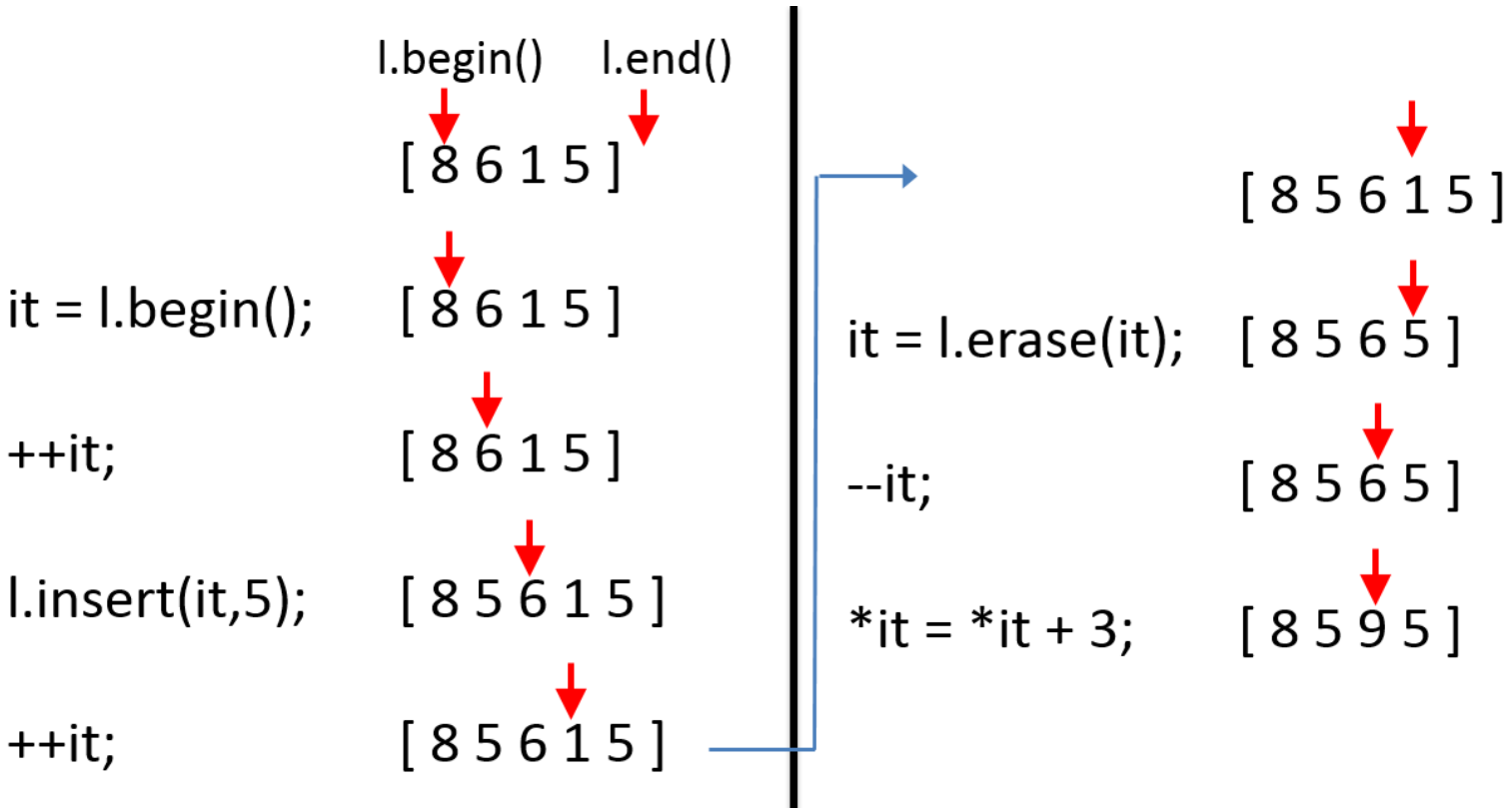
Iteradors

- Llistes i iteradors "treballen" coordinadament
- Operacions de llistes amb iteradors:
  - $L.\texttt{insert}(it, x)$: insereix a la llista L un nou element x com a predecessor de l'element apuntat per it
  - $L.\texttt{erase}(it)$: elimina de la llista L l'element apuntat per it; retorna un iterador al successor de l'element esborrat

Iteradors

- Llistes i iteradors "treballen" coordinadament
- Operacions que ens tornen iteradors:
  - L.`begin`(): torna un iterador apuntant al primer element de la llista L
  - L.`end`(): torna un iterador apuntant "fora"—a un element fictici successor de l'últim—de la llista L
  - Si L és buida, aleshores `L.begin() == L.end()`

Exemple d'evolució d'una llista



Iteradors

```
list<Estudiant> l;
list<string> lp;

list<Estudiant>::iterator it = l.begin();
list<Estudiant>::iterator it2 = l.end();
list<string>::iterator it3 = lp.begin();

it = it3; // error!! són de tipus diferents
```

Cada tipus d'iterador es defineix com a subclasse de la classe "contenidora"

Iteradors: Recorreguts

Esquema freqüent:

```
list<T> L;
list<T>::iterator it = L.begin();
while (it != L.end() and not condició sobre *it)) {
    accedir a *it
    ++it;
}
```

Iteradors constants

- Iteradors constants (`const_iterator`): prohibeixen modificar l'objecte referenciat per l'iterador
- S'han d'utilitzar per a recòrrer una llista rebuda per referència constant

```
list<Estudiant>::const_iterator it, it2;
it = it2;   // OK, it no és constant
++it;       // OK
v = *it;    // OK
*it = v+3; // error!!!
```

Iteradors constants

```
void imprimir_llista(const list<Estudiant>& L) {
  for(list<Estudiant>::const_iterator it = L.begin();
      it != L.end(); ++it)
    (*it).escriure();
}
// en comptes de (*it).escriure() podem posar
// it -> escriure();
```

Especificació de la classe genèrica `Llista`

```
template <class T> class list {
public:
// Subclasses de la classe llista
     class iterator { ... };
     class const_iterator { ... };

// Constructores

/* Pre: cert */
/* Post: El resultat es una llista sense cap element */
list();

// Destructora
~list();
```

Especificació de la classe genèrica `Llista`

```
// Modificadores
/* Pre: cert */
/* Post: La llista implícita queda buida */
void clear();

/* Pre: it referencia algun element existent a_i a la llista o
        és igual a end(), la llista és [a_1,...,a_n] */
/* Post: L'element s'ha inserit davant de l'element referenciat
         per it, la llista és ara [a_1,...,x,a_i,...] */
void insert(iterator it, const T& x);
```

Especificació de la classe genèrica `Llista`

```
/* Pre: it referencia algun element a_i existent a la
        llista [a_1,...,a_n], n > 0 */
/* Post: S'ha eliminat l'element referenciat per it, la llista
         és ara [a_1,...,a_-1,a_i+1...,a_n] i torna
         un iterador al sucessor de l'element eliminat */
iterator erase(iterator it);

/* Pre: l = [y_1,...,y_m], l i la llista implícita són
        objectes diferents, i it referencia algún element x_i de
        la llista implícita [x_1,...,x_n] */
/* Post: La llista implícita és ara
        [x_1,...,x_i-1,y_1,...,y_m,x_i,...,x_n] i l és buida */
void splice(iterator it, list& l);
```

Especificació de la classe genèrica `Llista`

```
// Consultores

/* Pre: cert */
/* Post: torna cert si i només si la llista és buida */
bool empty() const;

/* Pre: cert */
/* Post: torna el nombre d'elements de la llista*/
int size() const;
```

Especificació de la classe genèrica `Llista`

```
    ...
// tornen iteradors al primer element de la llista
 const_iterator begin() const;
 iterator begin();
// tornen iteradors a l'element fictici succesor de l'últim
// de la llista
 const_iterator end() const;
 iterator end();
 private:
    ...
```

Especificació de la classe genèrica `Llista`

La inserció d'elements nous als extrems de la llista i l'esborrat dels extrems de la llista es pot fer sense iteradors:

```
l.push_back(x); // = l.insert(l.end(), x);
l.push_front(x); // = l.insert(l.begin(), x);

l.pop_front();  // = l.erase(l.begin());
l.pop_back(); // = it = l.end(); l.erase(--it);
```

Sumar tots els elements d'una llista d'enters

```
/* Pre: cert */
/* Post: El resultat és la suma dels elements de l */
int suma(const list<int>& l) {
    int s = 0;
    for (list<int>::const_iterator it = l.begin();
                                   it != l.end();
                                   ++it) {
        s += *it;
    }
    return s;
}
```

Cerca senzilla en una llista d'enters

```
/* Pre: cert */
/* Post: El resultat indica si x és o no a l */
bool pertany(const list<int>& l, int x) {
    list<int>::const_iterator it = l.begin();
    while (it != l.end() and (*it != x))
      ++it;
    return it != l.end();
}
```

Exercici: cerca en una llista d'estudiants

```
/* Pre: cert */
/* Post: El resultat ens indica si hi ha algun estudiant
   amb dni x a l o no */
bool pertany(const list<Estudiant>& l, int x);
```

Modificar una llista sumant un valor k a tots els elements

```
/* Pre: l = [x_1, ..., x_n] */
/* Post: l = [x_1 + k, x_2 + k, ..., x_n + k] */
void suma_k(list<int>& l, int k) {
    list<int>::iterator it = l.begin();
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
```

Una alternativa

En comptes de fer

```
*it += k;
++it;
```

podriem eliminar l'element i tornar a afegir-ho

```
int aux = (*it) + k;
it = l.erase(it); // it apunta al successor
l.insert(it,aux);
```

però és molt menys eficient (implica creació+destrucció d'objectes!)

Dir si una llista és capicua

[4,8,5,8,4], [7], [4,8,8,4] són capicues

```
/* Pre: cert */
/* Post: El resultat diu si l es capicua */
bool capicua(const list<int>& l);
```

Dir si una llista és capicua

```
bool capicua(const list<int>& l) {
    list<int>::const_iterator it1 = l.begin();
    list<int>::const_iterator it2 = l.end();
    for (int i = 0; i < l.size()/2; ++i) {
        --it2;
        if (*it1 != *it2) return false;
        ++it1;
    }
    return true;
}
```

Dir si una llista és capicua

- Exercici: Penseu com fer-ho sense usar `l.size()`.
- Cada element s'ha de consultar un cop com a molt.
- Recordeu que no es pot comparar `it1 < it2`

Splice: Insert a l'engrós!

- Si `l1 = [1,2,3,4,5,6]`, `it` apunta al 4, i `l2 = [10,20,30]` llavors

  `l1.splice(it,l2)`,

  queda

Splice: Insert a l'engrós!

- Si `l1 = [1,2,3,4,5,6]`, `it` apunta al 4, i `l2 = [10,20,30]` llavors

  `l1.splice(it,l2)`,

  queda
  `l1 = [1,2,3,10,20,30,4,5,6]`, `it` apunta a 4, `l2` buida

Splice: Insert a l'engrós!

- Si `l1 = [1,2,3,4,5,6]`, `it` apunta al 4, i `l2 = [10,20,30]` llavors

  `l1.splice(it,l2)`,

  queda
  `l1 = [1,2,3,10,20,30,4,5,6]`, `it` apunta a 4, `l2` buida
- Per concatenar dues llistes farem: `l1.splice(l1.end(),l2)`
- La STL de C++ té variants més complexes de `splice` que fan altres tipus de "transferència" de continguts entre llistes
- El cost de `splice` és constant, no depèn de les longituds de les llistes

Vectors vs. llistes

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element

Vectors vs. llistes

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes

Vectors vs. llistes

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element

Vectors vs. llistes

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element
  - Constant en llistes

Vectors vs. llistes

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element
  - Constant en llistes
  - En vectors, afegir al final és constant en mitjana (`push_back()`)

Vectors vs. llistes

- Recorregut seqüencial: Temps lineal en els dos casos, constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element
  - Constant en llistes
  - En vectors, afegir al final és constant en mitjana (`push_back()`)
  - En vectors, inserir pel mig és costós

Vectors vs. llistes

- Esborrar un element: constant en llistes, costós en vectors (excepte l'últim `pop_back()`)

Vectors vs. llistes

- Esborrar un element: constant en llistes, costós en vectors (excepte l'últim `pop_back()`)
- Splice: constant en llistes, costós en vectors

Accés directe?

Accés directe per posició en llistes. Si cal ...

```
// pre: 0 <= i < l.size()
// post: retorna l'i-essim element de l
template <typename T>
T get(const list<T>& l, int i) {
    list<T>::const_iterator it = l.begin();
    for (int j = 0; j < i; ++j) ++it;
    return *it;
}
```

Accés directe

Cost lineal

```
list<double>::iterator it = l.begin();
double sum = 0;
while (it != l.end()) {
  sum += *it; ++it;
}
```

Cost quadràtic

```
double sum = 0;
for (int i = 0; i < l.size(); ++i)
   sum += get(l,i);
```