3.5 Llistes i iteradors, *list*s i *iterator*s

En C++ direm *contenidor* a una estructura de dades on emmagatzemarem objectes d'una forma determinada. En general, els conenidors son classes genèriques que s'instancien de la manera que ja hem vist. Un exemple de contenidor que ara presentarem és la llista.

Una *llista* és una estructura de dades lineal que permet accedir a qualsevol element *sense treure els anteriors*. Això es fa mitjançant iteradors.

Un *iterador* ens permet desplaçar-nos per un contenidor i fer referència als seus elements. Depenent de les característiques del contenidor es podran fer servir diferents tipus d'iteradors. Pel cas de llistes d'enters, declarem una llista i un iterador així

```
list<int> l;
list<int>::iterator it;
```

Noteu que l'iterador no s'associa a la llista *l* sinó a la classe. Això comporta que es podria reutilitzar el mateix iterador amb diferent llistes.

Les llistes ofereixen un mètode `begin()` que retorna un iterador que referencia el primer element del contenidor, si és que existeix. Per exemple, podem fer que l'iterador anterior referenciï el primer element de la llista de la següent forma:

```
it = l.begin();
```

També hi ha iteradors constants que impedeixen modificar l'objecte referenciat per l'iterador. És obligatori fer servir iteradors constants si són per explorar una llista que sigui un paràmetre `const`. Si, per exemple, necessitem una llista d'`Estudiant` amb un iterador d'aquesta mena, cal fer:

```
list<Estudiant> l_est;
list<Estudiant>::const_iterator it2=l_est.begin();
```

Amb l'iterador `it2` podem moure'ns per la llista `l_est` i consultar el valor dels seus elements, però no modificar-los.

Podem fer que un iterador es mogui per una llista amb la notació `++it` i `--it`, que ens permet anar al següent element i al anterior respectivament. També existeixen aquests operadors en forma postfixa.

Podem desreferenciar un iterador, obtenint l'objecte que referencia, amb la notació `*`. Per exemple, si l'iterador `it2` definit anteriorment ja s'ha mogut per `l_est`, l'element referenciat per `it2` és `*it2`, i per consultar el seu dni, podrem fer `(*it2).consultar_DNI()`. No podrem, però, modificar `*it2`, ja que `it2` ha estat definit com a constant.

Les llistes també ofereixen un mètode `end` que retorna un iterador que referencia a un element inexistent posterior al darrer element de la llista. No és recomanable desreferenciar aquest iterador.

Es poden assignar iteradors amb `it1=it2;` i comparar-los per veure si són iguals o no (és a dir, si referencien al mateix element), amb `it1==it2` i `it1!=it2` respectivament. Un cas particularment útil per recórrer llistes és comparar l'iterador amb el que ens movem amb l'iterador

`l.begin()`	Retorna un iterador que referencia el primer element d'l
`l.end()`	Retorna un iterador que referencia un element fictici posterior al darrer d'l
`++it`	Avança al següent element
`--it`	Retrocedeix a l'anterior element
`*it`	Designa l'element referenciat per `it`
`it1=it2`	Assigna l'iterador `it2` a `it1`
`it1==it2`	Diu si els iteradors `it1` i `it2` són iguals o no
`it1!=it2`	Diu si els iteradors `it1` i `it2` són diferents o no

Figura 3.1: Resum de les operacions amb iteradors

retornat pel mètode `end` que ens indica el final de l'estructura que estem tractant, per exemple fent `(it != l.end())` on `l` és una llista amb un iterador `it`. A la figura 3.1 hi ha un resum de les operacions amb iteradors.

3.6 Especificació de la classe genèrica Llista

Com en el cas de l'estructures anteriors, només donem una part de l'especificació de la classe *list*. Hi ha altres operacions que no mencionem i algunes de les mencionades tenen altres variants que no farem servir i que per tant no apareixen en aquests apunts. Per exemple, només hi ha una versió de les operacions `insert` o `splice`. Aquesta darrera la farem servir bàsicament per concatenar llistes.

```
template <class T> class list {

// Tipus de mòdul : dades
// Descripció del tipus: Estructura lineal que conté elements de tipus T, que es
// pot començar a consultar pels extrems, on des de cada element es pot accedir
// a l'element anterior i posterior (si existeixen), i que admet afegir-hi
// i esborrar-hi elements a qualsevol punt

private:

public:

// Constructores

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

list(const list & original);
```

```
/* Pre: cert */
/* Post: El resultat és una llista còpia d'original */

// Destructora: Esborra automàticament els objectes locals en sortir d'un àmbit
// de visibilitat

~list();

// Modificadores

void clear();
/* Pre: cert */
/* Post: El paràmetre implícit és una llista buida */

void insert(iterator it, const T& x);
/* Pre: it referencia algun element existent al paràmetre implícit o
   és igual a l'end() d'aquest */
/* Post: El paràmetre implícit és com el paràmetre implícit original amb x
   davant de l'element referenciat per it al paràmetre implícit original */

iterator erase(iterator it);
/* Pre: it referencia algun element existent al paràmetre implícit,
   que no és buit */
/* Post: El paràmetre implícit és com el paràmetre implícit original sense
   l'element referenciat per l'it original; el resultat referencia l'element
   següent al que referenciava it al paràmetre implícit original */

void splice(iterator it, list& l);
/* Pre: l=L, it referencia algun element del paràmetre implícit o
   és igual a l'end() d'aquest; el p.i. i l no són el mateix objecte */
/* Post:  El paràmetre implícit té els seus elements originals i els
   d'l inserits davant de l'element referenciat per it; l està buida */

// Consultores

bool empty() const;
/* Pre: cert */
/* Post: El resultat indica si el paràmetre implícit té elements o no */

int size() const;
/* Pre: cert */
/* Post: El resultat és el nombre d'elements del paràmetre implícit */
};
```

En aquesta llista s'haurien d'afegir les operacions de iteradors. Noteu que no hi ha una operació de consulta d'elements perquè farem servir els iteradors per desreferenciar elements.

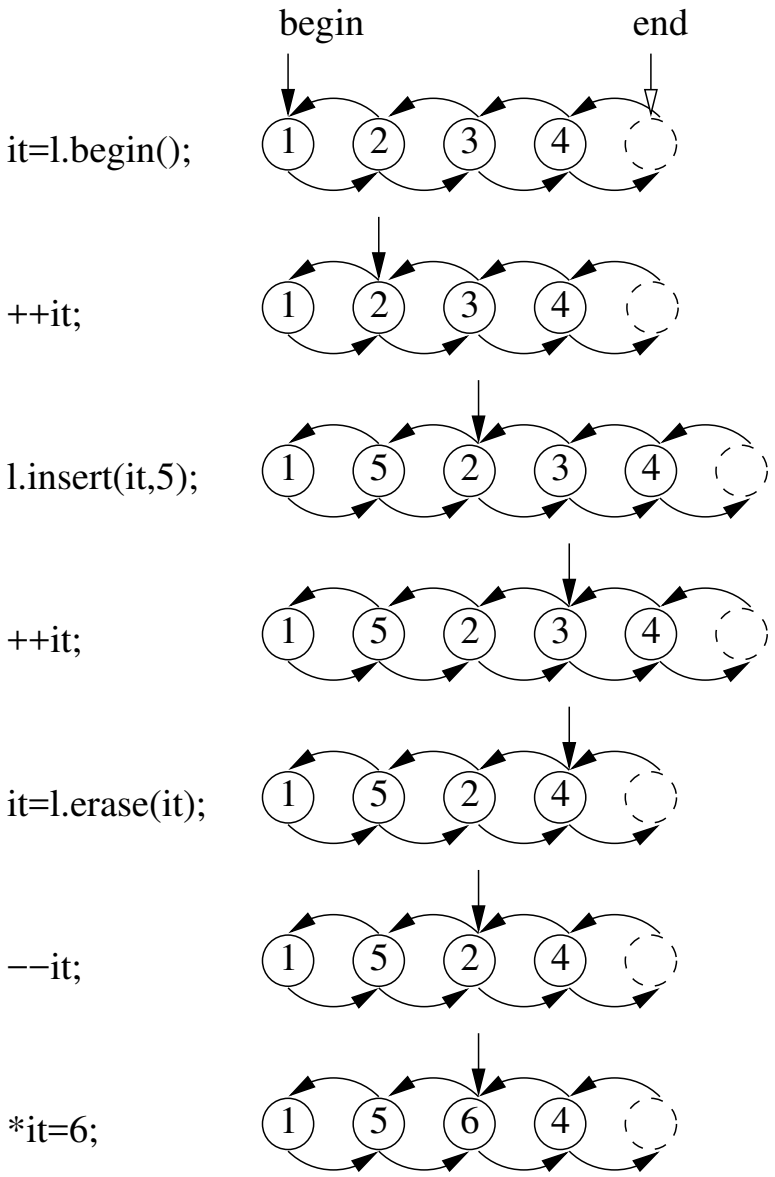

Figura 3.2: Exemple d'evolució d'una llista

També cal tenir en compte que si s'elimina el darrer element d'una llista, l'iterador referenciarà l'`end()`. A la figura 3.2 veiem un exemple de com evoluciona una llista després d'aplicar diferents operacions. Hem afegit l'operació `splice` que no farem servir gaire sovint, però que ens pot ser útil per concatenar dues llistes, per exemple `l1` i `l2`. Per aconseguir-ho haurem de escriure `l1.splice(l1.end(),l2);`. Si consultem l'especificació, veurem que així `l2` s'insereix a partir del final de `l1`.

Com passava amb les classes `stack` i `queue`, si un programa que fa servir la classe `list` executa operacions incorrectes, pot passar que el programa es continui executant, treballant de forma imprevisible. Per evitar aquesta mena de problemes cal, a l'hora de compilar, fer servir el *flag* `D_GLIBCXX_DEBUG`, és a dir:

```
> g++ -c elmeuprograma.cc -D_GLIBCXX_DEBUG -I$INCLUSIONS
```

i enllaçar normalment. Seguint aquestes instruccions, en el moment que el programa intenti accedir incorrectament a una llista, el programa s'interromprà donant un missatge d'error. D'aquesta manera ens serà més fàcil localitzar on s'ha produït l'error i corregir-lo. Entre els errors durant l'execució que ara fan interrompre el programa amb un missatge d'error explicatiu hi ha:

- Fer retrocedir un iterador a l'element `begin()` d'una llista, és a dir, fer `--it` quan per alguna llista `L` es compleix `it==L.begin()`.
- Fer avançar un iterador a l'element `end()` d'una llista, és a dir, fer `++it` quan per alguna llista `L` es compleix `it==L.end()`.
- Desreferenciar un iterador a l'element `end()` d'una llista
- Desreferenciar un iterador no assignat

3.7 Operacions de recorregut de llistes

Oferim un exemple d'algorisme d'exploració: sumar tots els elements d'una llista d'enters. Passem la llista per referència constant perquè així estalviem que es faci una còpia del paràmetre per l'operació. Contràriament al que passa amb piles i cues, es pot explorar una llista sense modificar-la. Si la llista es passa per referència constant tots els seus iteradors han de ser també iteradors constants.

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

Noteu que si una llista `l` és buida, llavors `l.begin()` i `l.end()` són iguals.

3.8 Operacions de cerca en llistes

Oferim una versió d'una cerca senzilla en una llista d'enters.

```
bool pert_llista_int(const list<int>& l, int x)
/* Pre: cert */
/* Post: El resultat indica si x hi és o no a l */
{
  bool b = false;
  list<int>::const_iterator it= l.begin();
  while ( it != l.end() and not b){
    if (*it == x) b= true;
    else ++it;
  }
  return b;
}
```

Repetim l'exemple per llistes d'`Estudiant`. Així veiem com treballar quan la llista conté objectes en comptes de tipus simples.

```
bool pert_llista_Estudiant(const list<Estudiant>& l, int x)
/* Pre: cert */
/* Post: El resultat ens indica si hi ha algun estudiant amb dni x a l o no */
{
  bool b =false;
  list<Estudiant>::const_iterator it = l.begin();
  while (it != l.end() and not b){
    if ((*it).consultar_DNI() == x) b= true;
    else ++it;
  }
  return b;
}
```

3.9 Modificació i generació d'una llista

Primer modifiquem una llista sumant un valor `k` a tots els elements.

```
void suma_llista_k(list<int>& l, int k)
/* Pre: l=L */
/* Post: El valor de cada element d'l és el valor corresponent a L més k */
{
  list<int>::iterator it= l.begin();
```

```
  while (it != l.end()){
    *it+=k;
    ++it;
  }
}
```

També seria vàlida aquesta solució, encara que no és preferible ja que "mou" més informació.

```
void suma_llista_k(list<int>& l, int k)
/* Pre: l=L */
/* Post: El valor de cada element d'l és el valor corresponent a L més k */
{
  list<int>::iterator it= l.begin();
  while (it != l.end()){
    int aux =  (*it) + k;
    it=l.erase(it);
    l.insert(it,aux);
  }
}
```

Si volem conservar la llista original i que la llista modificada sigui nova, podem obtenir una solució similar en forma de funció, on haurem de fer servir l'operació `insert` per generar la llista resultat.

```
list<int> suma_llista_k(const list<int>& l, int k)
/* Pre: cert */
/* Post: Cada element del resultat és la suma de k i l'element d'l a la seva
   mateixa posició */
{
  list<int>::const_iterator it=l.begin();
  list<int> l2;
  list<int>::iterator it2=l2.begin();

  while (it != l.end()) {
    l2.insert(it2,*it+k);
    ++it;
  }
  return l2;
}
```

Noteu que per crear o modificar una llista no podem fer servir iteradors constants. També és destacable el fet que l'iterador `it2` sempre val `l2.end()`, ja que cada nou element a `l2` s'afegeix davant seu.

Per últim, si obtenim la nova llista amb una acció, ens estalviarem l'assignació al resultat quan fem la crida corresponent

```
void suma_llista_k(const list<int>& l, list<int> &l2, int k)
/* Pre: l2 és buida */
/* Post: Cada element de l2 és la suma de k i l'element d'l a la seva
   mateixa posició */
{
  list<int>::const_iterator it=l.begin();
  list<int>::iterator it2=l2.begin();

  while (it != l.end()) {
    l2.insert(it2,*it+k);
    ++it;
  }
}
```

En resum, si no volem conservar la llista original, la versió preferible és la primera, que és una acció. La segona també és una acció, però és més ineficient perquè a cada iteració esborra i inserta elements. En cas contrari, la millor solució és la quarta.