Programación con la STL: contenedores genéricos

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Octubre 2009

Outline

Previously en la clase

Vimos una serie de algoritmos pre-implementados de manera genérica (a través de patrones) que permiten hacer operaciones sobre elementos (dentro de un rango dado) de contenedores, accedidos por iteradores.

Outline

Containers de apuntadores

En el caso de *containers* que contienen apuntadores, una de las propiedades mas interesantes es de poder liberar los objetos que almacenamos; pero tenemos un problema : requiere usar el buen mecanismo de liberación de la memoria (el delete adaptado).

Containers de apuntadores, con patrones

```
template <class T>
class Stack {
  class Node {
    T *data;
    Node* next;
  public:
    Node(T *val, Node* nxt);
  };
  Node *head;
public:
  Stack();
  ~Stack();
  void push(T *);
  T *peek();
  T *pop();
};
```

Containers de apuntadores

Ya no tenemos ningún problema, el buen delete es llamado :

```
~Stack(){
   while(head)
      delete pop();
}
```

...sin tener que pasar por mecanismos polimorfos.

Containers : ¿ quién posee los objetos ?

Uno puede vacilar en autorizar que el container mismo pueda liberar los objetos alocados que tiene, por ejemplo porque pueden estar usados en otras partes; es útil poder especificar como parámetro si sí o no el container asume esta responsabilidad,

```
template <class T> class Stack {
  class Node {
    T *data;
    Node* next;
    Node(T *val, Node* nxt);
  };
  Node *head;
  bool ownData;
public:
  Stack(bool own = true) : ownData(own) {...}
  ...
```

Containers : ¿ quién posee los objetos ?

```
  bool getOwn() const { return ownData; }
   void setOwn(bool newownership) {
     ownData = newownership;
   }
};

template<class T> Stack<T>::~Stack() {
   if(!own) return;
   while(head)
     delete pop();
}
```

Containers de valores

Por fin, notar que los *containers* que almacenan objetos por valor son naturalmente dueños de esos datos, y que todos los procesos ligados al funcionamiento del *container* tienen que ver con los constructores por default, constructores por copia (peek, pop) y destructores de los objetos contenidos, de manera transparente.

¿Qué es un iterator ?

Un iterator es un objeto que permite recorrer los datos de un container, uno por uno, sin que se provea un acceso directo a la implementación del container. Están generalmente proveídos con el container, en particular los containers de la librería estándar.

¿Qué es un iterator ?

```
using namespace std;
list<int> l;...
list<int>::iterator lit;
for (lit=l.begin();
     lit!=l.end();
     lit++) {
int a = *lit;
...
}
```

La interfaz de std::list sólo me da acceso a copia de un elemento en particular, el arriba de la pila (por el mé todo peek). Acceder a los elementos a través de la interfaz es destructivo ! Los iteratores completan esa carencia dando acceso a todos los datos.

¿Qué es un iterator ?

Grandes similaridades con un Smart Pointer : el comportamiento mima el de un apuntador recorriendo una zona de memoria contigua (aunque en la realidad de la implementación pueda ser completamente diferente !), con mecanismos de vigilancia. Propone una abstracción práctica de los datos de un *container* dado.

- Implementa operadores $++$.
- Implementa operador de dereferenciación.
- Para no tener problemas de nombre, son en general anidados dentro de la clase sobre que trabajan :

```
std::list::iterator lit;
std::vector::iterator vit;
```

Implementación de un iterator

```
template <class T>
class Stack { ...
 class iterator {
  Stack& stackRef;
  unsigned int index;
  public:
  iterator(Stack& is) : stackRef(is), index(0) {}
  iterator(Stack& is,bool) : stackRef(is),
                index(stackRef.s) {}
  T &operator*() { return stackRef.datos[index]; }
  T &operator++() { // Prefix
     if (index >= stackRef.s-1) throw 1;
     return stackRef.datos[++index];
  }
```

Implementación de un iterator

```
T &operator++(int) { // Postfix
    if (index >= stackRef.s) throw 1;
   return stackRef.datos[index++];
 }
 // To see if you're at some point
  bool operator==(const iterator& rv) const {
    return index == rv.index;
  }
  bool operator!=(const iterator& rv) const {
    return index != rv.index;
  }
};
iterator begin() { return iterator(*this); }
// Create the "end sentinel":
iterator end() { return iterator(*this, true);}
};
```

Implementación de un iterator

Notar que se construye dos instancias particulares del iterator :

- begin(), que corresponde al "principio" de los datos en el *container*, y que corresponde a un dato valido (el iterator correspondiendo puede estar dereferenciado),
- end(), que corresponde a un limite superior, y que sirve a detectar que estamos fuera del rango admisible; actúa como el tamaño de un arreglo dinámico por ejemplo,

```
for (int *p = &a[0]; p<a+N; p++)
        *p=..;
for (Stack<int>::iterator it=s.begin();
                          it!=s.end();
                          it++)
        *it = ...;
```

Implementación de un iterator

Se le puede añadir métodos a su gusto, mientras corresponden a operaciones que usaríamos con apuntadores :

```
T &operator--(int);
T &operator--();
T &operator+=(int);
```

Interés de los iterators

Aun más adelante, se puede escribir código-patrón en términos de iteradores (ya que tenemos interfaces similares para todos : operadores de incrementación, de dereferenciación), independiente de la clase *container* que se considera.

```
class Shape {...
};
class Drawing : public list<Shape *> {...
};
class Plan : public Stack<Shape *> {...
};
class Schematic : public vector<Shape*> {...
};
```

Interés de los iterators

Super genérico : no uso la interfaz de los *containers* (que no es común), sino los iteradores que sí tienen interfaz común !

```
// A function template:
template<class Iter>
void drawShapes(Iter &start, Iter &end) {
   while(start != end) {
      (*start)->draw();
      start++;
   }
}
```

Interés de los iterators

```
Drawing d;
d.add(new Circle);
d.add(new Square);
Plan p;
p.push(new Line);
p.push(new Circle);
Schematic s;
s.push_back(new Square);
s.push_back(new Circle);
Shape* sarray[] = {  new Circle, new Line };
drawShapes(d.begin(), d.end());
drawShapes(p.begin(), p.end());
drawShapes(s.begin(), s.end());
drawShapes(sarray,
   sarray + sizeof(sarray)/sizeof(*sarray));
```

Interés de los iterators

La librería estándar tiene un montón de código según este principio.

```
template<typename _InputIterator1,
         typename _InputIterator2>
    inline bool
    equal(_InputIterator1 __first1,
          _InputIterator1 __last1,
          _InputIterator2 __first2)
 {
  .... // corté unas lineas de check
  for (; __first1 != __last1; ++__first1, ++__first2)
     if (!(*__first1 == *__first2))
          return false;
     return true;
  }
```

Contenedores

- Objetos conteniendo otros objetos.
- Extensibles en tamaño.
- Muchos de esos son definidos en la STL, difieren por sus interfaces, por la manera de manipularles.
- Hay que elegir un contenedor adaptado al problema que procesas (tanto conceptualmente que por eficiencia).
- La manera estándar de recorrer los datos almacenados es a través de iteradores.

Ejemplo: set

```
#include <set>
#include <iostream>
#include <iterator>
using namespace std;
int main() {
  set<int> iset;
  iset.insert(1);
  iset.insert(2);
  iset.insert(6);
  iset.insert(1);
  iset.insert(3);
  iset.insert(9);
  iset.insert(5);
  copy(iset.begin(),iset.end(),
       ostream_iterator<int>(cout,"\n"));
}
```

Ejemplo: set

1
2
3
5
6
9

Los datos son únicos en valor dentro del contenedor, y el recorrido por el iterador les presenta ordenados!

Ejemplo: set

```
ifstream source(fileName);
string word;
set<string> words;
while(source >> word)
  words.insert(word);
copy(words.begin(), words.end(),
     ostream_iterator<string>(cout, "\n"));
cout << "Number of unique words:"
     << words.size() << endl;
```

Un contenedor particularmente adaptado para representar un conjunto de valores, y sacar las instancias únicas en valor de objetos en una base de datos. Idea de conjunto matemático, con operaciones de unión, intersección...

Los contenedores de C++

Grandes tipos de contenedores en la librería estándar C++

- Secuencias: contenedores guardando los objetos que almacenar linealmente.
- Adaptadores: modificadores de los primeros para darles una interfaz dada.
- Contenedores asociativos: organizan el acceso a los datos por llaves.

Contenedores secuencias	vector, list, deque
Adaptores de contenedores	queue, stack, priority_queue
Contenedores asociativos	set, map, multiset, multimap

Los contenedores de C++

Un contenedor contiene copias de objetos y no se puede tener contenedor de referencias (pero de apuntadores, sí). Requerimientos mínimos para poder usar objetos dentro de un contenedor:

- existencia de un constructor por copia público
- existencia de un operador de asignación

```
class obj {
public:
  obj() {};
private:
  obj(const obj &o) {};
};
int main() {
  list<obj> l; obj o;
  l.push_back(o);
}
```

Los contenedores de C++

Para poder usarles:

```
#include <set>
#include <list>
#include <deque>
#include <vector>
```

Los contenedores secuenciales de C++

Contenedores con los datos organizados linealmente:

- vector: un poco como los arreglos, rápido para acceder a cualquier elemento dentro, pero lento en lo de insertar elementos
- list: lento para ir a recuperar elemento cualquieros en la secuencia, rápido para insertar elementos (lista doblemente ligada)
- deque: eficiente para los accesos en cualquier elemento, y mas rápido que el vector para añadir nuevos elementos en las dos extremidades

Los contenedores secuenciales de C++

Operaciones fundamentales:

- Para añadir elementos al final de la secuencia (en cualquier de los tres contenedores mencionados), usar push_back:

```
int i=1;
list<int> l; l.push_back(i);
vector<int> v; v.push_back(i);
deque<int> d; d.push_back(i);
```

- Para añadir elementos al principio de la secuencia (solo deque y list): usar push_front.
- Para acceder a un objeto: operador [] (solo deque y vector) o iteradores.

Outline

Iteradores de contenedores

Existen para la gran mayoría de los contenedores estándar, y son definidos de diferente manera:

```
Contenedor::iterator
```

y permiten recorrer "como con un apuntador" los datos, con los operadores ++, ==, != y los dos iteradores predefinidos begin() y end()

```
for (Contenedor::iterator it=c.begin();
     it!=c.end();
     it++) {
  (*it).doSomething();
  it->doSomething();
}
```

Iteradores de contenedores

Esa propiedad permite la definición de patrones genéricos manipulando contenedores

```
template<class Contenedor, class ApuntMiembro>
void apply(Contenedor& c, ApuntMiembro f) {
  typename Contenedor::iterator it = c.begin();
  while(it != c.end()) {
    ((*it).*f)(); ++it;
  }
}
```

- No olvidar en este caso la palabra llave typename (para no ser confundido con un objeto estático)
- No se puede usar

```
((it->*f)();
```

- Pensar en los algoritmos for_each() o transform()

Iteradores de contenedores

Una variación interesante del iterador clásico:

```
Contenedor::const_iterator
```

Para recorrer contenedores de tipo const, o para asegurarse de que los datos no estarán cambiados en el recorrido.

```
for (Contenedor::const_iterator it=c.begin();
     it!=c.end();
     it++) {
  (*it).doSomething();
}
```

El método doSomething tiene que estar definida con el modificador const

Iteradores de contenedores

Para los contenedores llamados reversibles, podemos manipular iteradores recorriendo en los dos sentidos de la secuencia (guardando los mismos rasgos), en particular del final hasta al principio, gracias a reverse_iterator y los métodos especiales rbegin() y rend():

```
list<int> l;
l.push_back(1); l.push_back(2); l.push_back(3);
copy(l.rbegin(),l.rend(),
     ostream_iterator<int>(cout," "));
copy(l.begin(),l.end(),
     ostream_iterator<int>(cout," "));
```

```
3 2 1 1 2 3
```

Igualmente, existe const_reverse_iterator

Iteradores de contenedores

Habíamos mencionado varias veces que los iteradores son objetos de tipo "Smart Pointers" que se comportan como apuntadores. Esta comparación rápida tiene que estar relativizada:

```
list<int> l;
l.push_back(1); l.push_back(2); l.push_back(3);
copy(l.begin(),l.end(),
     ostream_iterator<int>(cout," "));
```

Qué sentido en este caso usar un operador - - con el ostream_iterator ? Los iteradores definidos tienen rangos diferentes de propiedades.

Iteradores de contenedores

Grandes categorías:

- Iteradores de input: típicamente para fuentes de datos externos (como el istream_iterator de la clase precedente). Está limitado a simple lectura (comportamiento const) y la dereferenciación es única (ya que típicamente está ligada a la lectura en un flujo). Constructor especifico para marcar el end() (constructor por default de istream_iterator)
- Iteradores de output: dual del precedente, sólo permite escribir con un recorrido único igualmente (ostream_iterator)

Iteradores de contenedores

Ejemplo:

```
#include <iterator>
#include <fstream>
#include <iostream>
...
ifstream inf("entrada.txt");
replace_copy_if(istream_iterator<string>(inf),
                istream_iterator<string>(),
                ostream_iterator<string>(cout,"n"),
                tieneUnY, string("amigo"));
```

Iteradores de contenedores

Grandes categorías:

- Iteradores direcionales (*forward*): combinan las propiedades de los dos precedentes, y ademas de dereferenciar los objetos, se puede cambiar su valor tantas veces como queremos. Pero sentido de recorrido único (sólo operador $++$)
- Iteradores bidirectionales: iguales a los precedentes, añadiendo la posibilidad de usar el operador – para moverse en el otro sentido del recorrido. Todos los iteradores pre-implementados en el contenedor `list` son de este tipo
- Iteradores a acceso aleatorio: ese iterador tiene un comportamiento casi igual al de un apuntador, en particular se puede hacer acceso arbitrarios con el operador corchete (la única diferencia es que no hay objeto de tipo NULL). Soporta operaciones $+$,$+=$,-,$-=$, comparaciones... Los iteradores de los `vector` y `deque` proveen esos iteradores.

Iteradores de contenedores

Ejemplo de iteradores a acceso aleatorio:

```
vector<int> v;
v.push_back(1); v.push_back(2);
v.push_back(3); v.push_back(4);
vector<int>::iterator vit1 = v.begin();
vector<int>::iterator vit2 = vit1+1;
cout << vit1[2] << endl;
cout << vit2[2] << endl;
vit1+=2;
if (vit1>vit2)
        cout << "vit1 superior" << endl;
```

```
3
4
vit1 superior
```

Iteradores de contenedores: inserción

Vimos que los algoritmos bien útiles de la librería estándar están implementados como patrones con abstracciones de iteradores.
Problema: están implementados con los operadores "*" (dereferenciación), "++" (incrementación), asignación

```
template<typename Iterator>
void copy(Iterator begin,
          Iterator end,
          Iterator dest) {
  while(begin != end)
    *dest++ = *begin++;
}
```

Eso supone que los dos contenedores están recorridos simétricamente (y también el en que escribimos), dest... ¿Qué tal que queremos rellenar un contenedor vacío?

Iteradores de contenedores: inserción

La solución es definir un objeto iterador que reimplemente diferentemente el operador de asignación de tal manera a que la copia esté remplazada por una alocación de nueva memoria (por $\mathrm{push\_back}$ o $\mathrm{push\_front}$)

- Iteradores $\mathrm{back\_insert\_iterator}$ y $\mathrm{front\_insert\_iterator}$ para la inserción de datos a las dos extremidades del contenedor
- dado un contenedor, se puede obtener un iterador de esos tipos con las funciones $\mathrm{back\_inserter}()$ y $\mathrm{front\_inserter}()$
- existen para vector (incluso el front, en las versiones de compilador mas recientes), list, deque

Iteradores de contenedores: inserción

Se puede definir también un iterador para la inserción en cualquier lugar del contenedor, el iterador insert_iterator, que se apoya sobre el método insert() de los contenedores, para insertar un objeto

```
int a[5]={1,3,5,7,9};
vector<int> vv(a,a+4);
vector<int>::iterator it = vv.begin();
++it; ++it;
copy(istream_iterator<int>(cin),
     istream_iterator<int>(),
     inserter(vv, it));
copy(vv.begin(), vv.end(),
ostream_iterator<int>(cout, " "));
cout << endl;
}
```

Iteradores de contenedores: inserción

La inserción se realiza bien después del segundo elemento:

```
3
2
1
8
.
1 3 3 2 1 8 5 7
```

Iteradores sobre flujos

- No noción de $\mathrm{end}()$ para un iterador de tipo $\mathrm{ostream\_iterator}$ (nunca se necesita que especificar en funciones como copy y ademas no tiene mucho sentido)
- Pero importante para un iterador sobre input ! Vimos que el objeto construido por el constructor por default desempeña este papel
- Cuidado al intentar leer octeto por octeto: la manipulación de flujos por el operador $>>$de $\mathrm{ifstream}$ o $\mathrm{istream}$ interpretan los datos medio un formato dado (y quitan los espacios, los newlines...). En este caso, preferir $\mathrm{istreambuf\_iterator<char>}$ (que no formatea)

Iteradores sobre flujos

```
ifstream in("file.txt");
istreambuf_iterator<char> isb(in), end; // Binary
ostreambuf_iterator<char> osb(cout);
while(isb != end)
    *osb++ = *isb++; // Integral copy
cout << endl;
ifstream in2("file.txt");
stream_iterator<char> is(in2), end2; // Chars
ostream_iterator<char> os(cout);
while(is != end2)
    *os++ = *is++;
cout << endl;
```

Iteradores sobre flujos

El ejemplo nos da:

```
Este es un pequeno ejemplo de texto, con
newlines y unos espacios.

Esteesunpequenoejemplodetexto,connewlinesyunosespacios.
```

Outline

Métodos básicos sobre las secuencias

- empty(), que indica si el contenedor es vacío o no
- size(), que indica el tamaño corriente del contenedor
- max_size(), que indica el tamaño máximo que puede tener un contenedor de este tipo (diferente de capacity())
- front(), que regresa una referencia al elemento al principio del contenedor
- back(), que regresa una referencia al elemento al final del contenedor

Métodos básicos sobre las secuencias

- begin(), rbegin(), end(), rend(), regresando los iteradores de principio y fin
- assign(int nels,const obj& el): asigna al contenedor *nels* objetos duplicados de *el* (todo lo que había antes se va)
- assign(itbegin,itend): asigna al contenedor los objetos entre los iteradores *itbegin* y *itend* (todo lo que había antes se va)
- clear(): quita todos los objetos
- resize(): cambia de tamaño, inicializando los eventuales nuevos elementos con el constructor por default

Métodos básicos sobre las secuencias

- insert(): para insertar elementos dentro del contenedor

```
v.insert(it,18);// Inserción de un elemento en it
v.insert(it,4,12);// Inserción de 4 "12" en it
v.insert(it,o.begin(),o.end()); // Iteradores
```

- erase(): para suprimir elementos; regresa el iterador sobre el elemento siguiente no destruido

```
it=v.erase(it); // One element
it=v.erase(it, it2); // All between it and it2
```

  Cuidado con este en los ciclos...

- swap(): para intercambiar contenedores enteros

```
v.swap(vv);
```

Métodos básicos sobre las secuencias

Un constructor común: a partir de iteradores de otros contenedores o de arreglos, que están copiados entre un valor de inicio y un valor de fin

```
int b[]={-1,3,1,6,2,7};
vector<int> v(b,b+4);
list<double> l(v.begin(),v.begin()+3);
deque<int> d(l.begin(),l.end());
```

Contenedor vector

Se parece intrínsecamente a un arreglo, con la posibilidad de adaptar su tamaño dinámicamente según las necesidades del programa

Internamente está implementado con un arreglo de datos, lo que puede explicar todos sus comportamientos: acceso rápido, inserciones lentas, y realocación de memoria cuando llegamos a la capacidad del vector

Contenedor vector

Para manejar la cantidad de memoria:

- está alocada al principio una cantidad por default de memoria (0 por ejemplo)
- cuando se alcanza la capacidad máxima del vector, se realoca mas memoria, se copia los objetos iniciales y se destruye los primeros (costoso)
- para evitar eso, si se puede saber de cuanto tamaño necesitaremos, usar el método $\mathrm{reserve}()$, diferente del constructor con numero de elementos

Contenedor vector

¿Cuál es la mejor política (en términos de eficiencia) para la redimensionalización del vector?

Imaginemos que tenemos capacidad inicial de c, y que cada vez que se alcanza la capacidad se aumenta de una unidad. El costo de una realocación es lineal en el tamaño corriente del vector. Añadir $n > c$ elementos sería cuadrático, y el costo de un push seria lineal. Sería igual al añadir una cantidad constante de memoria.

Contenedor vector

Una mejor solución: multiplicar por dos la capacidad en cada saturación. Suponemos que tenemos $c.2^{p_0}$ elementos y que queremos añadir n elementos. Entonces, si

$$c2^p < n \leq c2^{p+1}$$

vamos a tener que redimensionar el vector $p - p_0 + 1$ veces! El costo es:

$$c(2^{p_0} + 2^{p_0+1} ... + 2^{p+1}) \leq c2^{p+2} \leq 4n$$

Entonces el cuesto de la inserción de n elementos es $O(n)$, y el cuesto promedio de un push es O(1)! (costo amortizado)

Contenedor vector

+ Acceso rápido a cualquier objeto por el operador []
+ Eficiente si se añaden/quitan objetos al final de la secuencia
- Inserciones lentas (hay que mover, en promedio un numero $\frac{n}{2}$ de datos)
- Costo (aunque amortizado) de la realocación, que puede estar evitado
  - usando reserve()
  - emparejando el vector con un deque (para la inicialización)
- Iteradores no soportan los cambios de estructura interna

Contenedor vector

```
vector<int>::iterator dit = d.begin(); dit++;
 cout << d.capacity() << endl;
 d.resize(d.capacity()*20);
 cout << d.capacity() << endl;
 cout << *dit << endl;
```

Puede provocar problemas...

Contenedor vector

Algunas complejidades:

- inserción al final?
- inserción al principio?
- inserción cualquiera?
- quitar cualquier elemento?
- acceso?

Contenedor deque

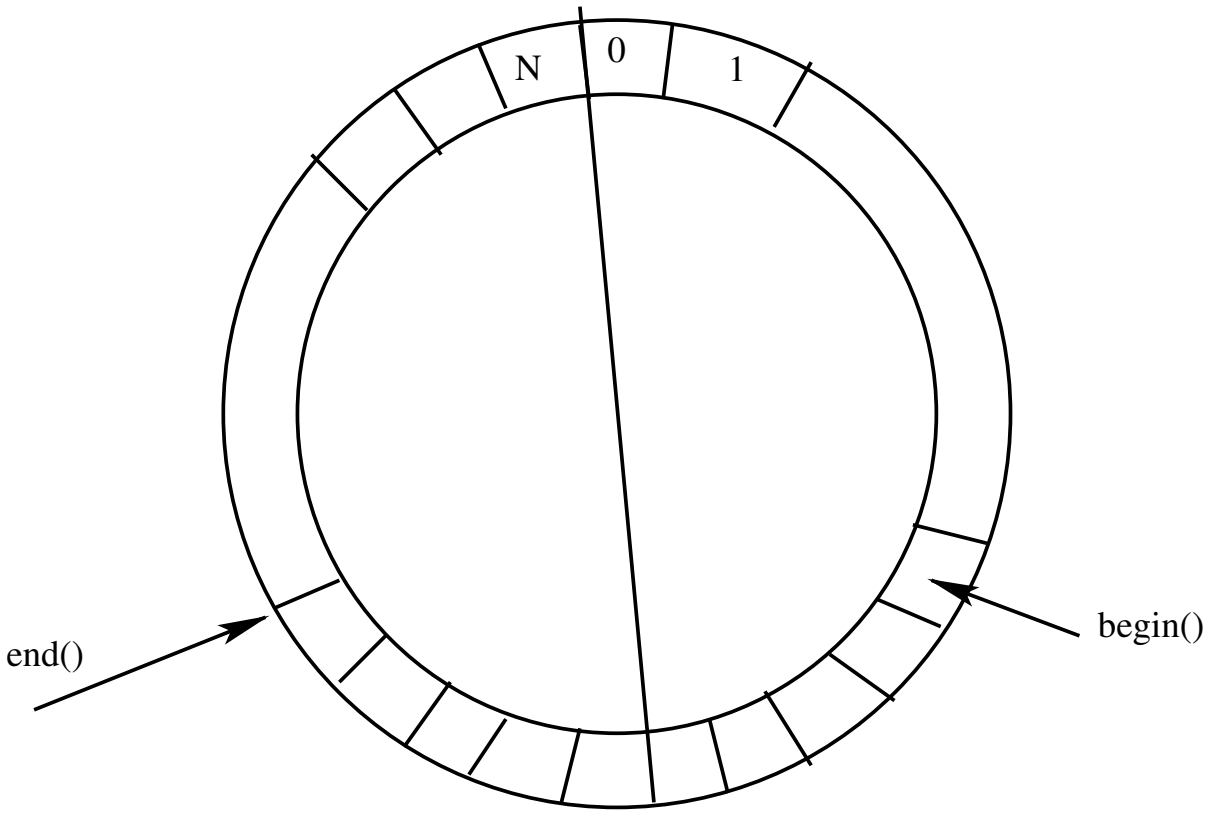
Muy similar al vector pero diferente por:

- Inserción en las dos extremidades en O(1)
- Acceso un poco más lento a los datos en cualquier lugar por []
- No costo adicional en la redimensionalización

Ideas sobre como puede ser hecha la implementación?

Contenedor deque

- Una primera idea: arreglo circular (util para los dos primeros puntos)



Pero quid de la realocación ?

Contenedor deque

Mas generalmente, la implementación de las deques es a base de varios segmentos de memoria (como arreglos) alocados, con las herramientas para manejar las direcciones en memoria:

- en caso de saturación, se aloca uno de esos segmentos (pero no se necesita liberar los previos, destruir objetos): O(1)
- mapeo entre los indices y los lugares en memoria
- este mismo mapeo autoriza también operaciones de inserción al principio (front)

Contenedor deque

Métodos de acceso (eso vale también para vector):

- operador [], al comportamiento muy similar del apuntador
- método at(), que ademas de regresar el valor deseado, verifica la validez de los indices (en caso contrario, envía una excepción)

Usar uno u otro en función de la seguridad uno tiene con los indices...

Contenedor list

Implementada como lista doblemente ligada, tiene como principal característica de poder hacer inserciones en cualquier lugar de la lista en tiempo

Pero:

- complejidad de un acceso? de hecho, aun no tiene operador []
- necesita un poco mas de memoria (por cada elemento)

Util para contenedor de objetos grandes, que se va a insertar frecuentemente

Contenedor list

- Por construcción, los objetos se quedan en su lugar inicial en la memoria (también en el caso de deque), no hay riesgos de perder iteradores al hacer operaciones de estructuras sobre list
- Por esa misma propiedad, todas operaciones de manipulación como ordenamiento, pueden hacerse sin recopiar objetos, sino solo cambiando los *links* entre ellos (mucho menos costoso)
- Consecuencia: muchos de los algoritmos genéricos (implementados como funciones globales) están reimplementados dentro de list

Contenedor list

Ejemplos:

- sort(): ordenamiento
- reverse(): poner los objetos en el orden inverso
- splice(): separar la lista entre sub-listas hacia otros contenedores de diferentes maneras (no destructivo de los objetos !)
- remove(), remove_if(): quitar los elementos de un valor dado (destructivo)
- unique(): quitar los duplicados (solo si están ordenados, destructivo)
- merge(): para fusionar dos listas (la lista argumento se vacia)

Contenedor list

Uso de splice:

```
list<int> l,l2,l3,l4;
l.assign(5,1);
l2.push_back(10);
l3.push_back(-10);
// First way
l2.splice(l2.begin(),l);
copy(l.begin(),l.end(),ostream_iterator<int>(cout,"
cout << endl;
copy(l2.begin(),l2.end(),ostream_iterator<int>(cout
cout << endl;
```

Contenedor list

Uso de splice:

```
// Second way
list<int>::iterator lit=l2.begin();
l3.splice(l3.begin(),l2,++lit);
copy(l2.begin(),l2.end(),ostream_iterator<int>(cout
cout << endl;
copy(l3.begin(),l3.end(),ostream_iterator<int>(cout
cout << endl;
// Third way
lit=l2.begin();
l4.splice(l4.begin(),l2,++lit,l2.end());
copy(l2.begin(),l2.end(),ostream_iterator<int>(cout
cout << endl;
copy(l4.begin(),l4.end(),ostream_iterator<int>(cout
cout << endl;
```

Contenedor list

Uso de splice:

```
1 1 1 1 1 10
1 1 1 1 10
1 -10
1
1 1 1 10
```

Contenedor list

- Uso de remove:

```
l4.remove(1);
copy(l4.begin(),l4.end(),ostream_iterator<int>(cou
cout << endl;
```

10

No necesita que los objetos estén ordenados...

- Uso de merge:

```
l3.merge(l4);
```

Remarca sobre swap

El método swap existe para los tres contenedores secuenciales, y se puede constatar que al usarlo, no ocurre ninguna copia o destrucción: lo que está intercambiados no son los objetos, sino los contenedores (dirección de arreglo, conjunto de bloques de memoria, apuntador al principio de la lista)

Contenedor set

La filosofía de este contenedor es de almacenar datos de valores únicos, y de poder rápidamente poder determinar una relación de membrecía de un objeto con respeto a este contenedor (noción de conjunto matemático).

Para lograr este objetivo, la implementación mas común usa arboles equilibrados y por construcción ordena los datos. Operaciones de búsqueda en que complejidad?

Contenedor set

Otra diferencia es que los objetos en este contenedor ya no son indexados por un index (como en los casos de vector, deque, list) sino por su valor

Contenedor set

Ejemplo típico: crear un index para un libro

- Leer el texto del libro
- Para cada palabra encontrada, intentar añadirla en el set
  - Si ya esta, dejarla
  - Si no está, añadirla al set, de tal manera que el conjunto quede ordenado y que el árbol subyacente sea equilibrado

Contenedor set

```
char* fname = "2donq10.txt"; // Don quijote
ifstream in(fname);
set<string> wordlist;
string line;
while(getline(in, line)) {
  transform(line.begin(), line.end(), line.begin(),
            check);
  istringstream is(line);
  string word;
  while(is >> word)
    wordlist.insert(word);
}
// Output results:
copy(wordlist.begin(), wordlist.end(),
     ostream_iterator<string>(cout, "\n"));
```

Contenedor set

```
char check(char c) {
  // Only keep lower aphas
  return (isalpha(c)) ? tolower(c) : '␣';
}
```

Ahora, las palabras del Don Quijote están ordenadas y buscar una palabra dada para saber si esta tendrá complejidad logarítmica: método find(). Notar que existe también una función-algoritmo find() pero que recorre los iteradores del contenedor (complejidad lineal)

Contenedor set

Para buscar un elemento:

```
set<string>::iterator it=wordlist.find("feliz");
 if (it!=wordlist.end())
   cout << "Found before " << *(++it) << endl;
 else
   cout << "Not found" << endl;
```

```
Found before felizmente
```