Contenedores / Colecciones

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Motivación

La organización de una estructura de datos depende del uso que queramos hacer de ella:

- ¿ Necesita una forma rápida para buscar un elemento ?
- ¿ Necesita insertar y borrar elementos rápidamente ?
- ¿ Necesita acceso aleatorio ?
- ¿ Necesita poder crecer en tiempo de ejecución ?

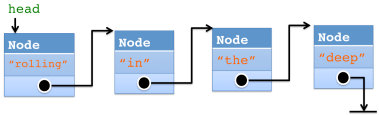
Para todas estas necesidades se han definido estructuras de datos adecuadas (vector, lista, diccionario), y para cada una de ellas existen implementaciones optimizadas para alguna de las características anteriores.

Entre distintas estructuras de datos podemos elegir la más adecuada para nuestro problema

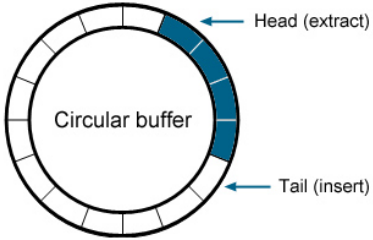
```
1  template<typename T>
2  class Queue {
3  public:
4    void add(const T& e) { ... }
5    T    remove()        { ... }
6    int  size()          { ... }
7  };
```

Podemos tener misma semántica pero diferentes implementaciones

Linear Linked List



Circular Limited Buffer



Estructuras de datos

`vector, list, map, set, bag, ...`

Puntos comunes

- Datos **homogéneos**
- Estructuras **lineales**
- Acceso **secuencial** (iterador)

Diferencias

- Implementación del recorrido (vector vs list)
- Acceso indexado (map vs vector vs list)
- Multiplicidad (set vs bag)
- Acceso directo/inverso (vector vs set)
- Estabilidad del recorrido (vector vs bag)
- ...

Pregunta

¿ Cómo podemos diseñar algoritmos que funcionen indistintamente con diferentes estructuras de datos ?

Ejemplo

```
1 void add10(??? queue) {
2   for (int i=1; i<=10; ++i)
3     queue.add(i);
4 }
```

Herencia

```
1
2 void add10(Queue& queue)
3 {
4   for (int i=1; i<=10; ++i)
5     queue.add(i);
6 }
```

Definir subclases de Queue para implementaciones concretas

Programación genérica

```
1 template<typename Q>
2 void add10(Q& queue)
3 {
4   for (int i=1; i<=10; ++i)
5     queue.add(i);
6 }
```

Asegurar mismos métodos en implementaciones.
En Java necesitaremos restricciones.

Distintas estructuras de datos pueden ofrecer interfaces similares

```
class Queue {
    virtual void add(....);
};
```

```
class LinkedList
          : public Queue {
  ...
  void add(....) override;
};
```

```
class CircularBuffer
          : public Queue {
  ...
  void add(....) override;
};
```

...para determinadas operaciones: **herencia**.

Distintas estructuras de datos pueden ofrecer interfaces similares

```
class LinkedList {
  ...
  void add(....);
};
```

```
class CircularBuffer {
  ...
  void add(....);
};
```

...para determinadas operaciones: **programación genérica**.

Sólo necesitamos saber el tipo concreto de cola a la hora de crearla.

```
CircularArray queue(100);
add10(queue);
```

```
LinkedList queue;
add10(queue);
```

Distitas estructuras de datos pueden ofrecer interfaces específicos...

```
1 class LinkedList
2 {
3   ...
4   bool empty(....);
5 }
```

```
1 class CircularBuffer
2 {
3   ...
4   int size(....);
5 }
```

...para otras operaciones.

Necesitamos saber el tipo concreto a la hora de recorrerla:

```
LinkedList lane;
```

```
CircularBuffer lane(100);
```

El uso en ciertas partes del código es distinto:

```
while (!lane.empty())
  e = lane.remove();
```

```
while (lane.size()>0)
  e = lane.remove();
```

Distitas estructuras pueden ofrecer interfaces similares o distintos:

```
1  class LinkedList {
2    void add(T e);
3    T    remove();
4    bool empty();
5  }
```

```
1  class CircularBuffer {
2    bool add(T e);
3    T    remove();
4    int  size();
5  }
```

- Una de las implementaciones es acotada (`size()` vs `empty()`)
- La forma de recorrerlas es distinta
- ¿ Y si queremos convertir `lane` en un `vector` ?

`vector, list, map, set, bag, ...`

Objetivos

- Garantizar la homogeneidad
- Establecer interfaces comunes
- Posibilitar implementaciones especializadas

Solución

Genéricos / Interfaces / Herencia

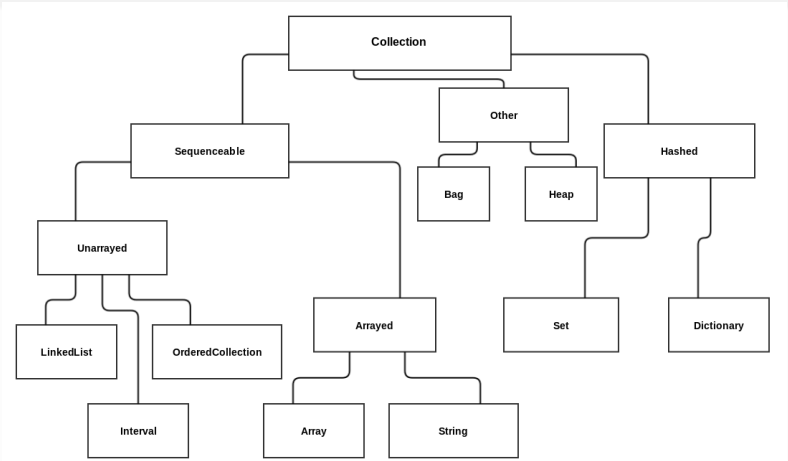
`vector, list, map, set, bag, ...`

Contenedores

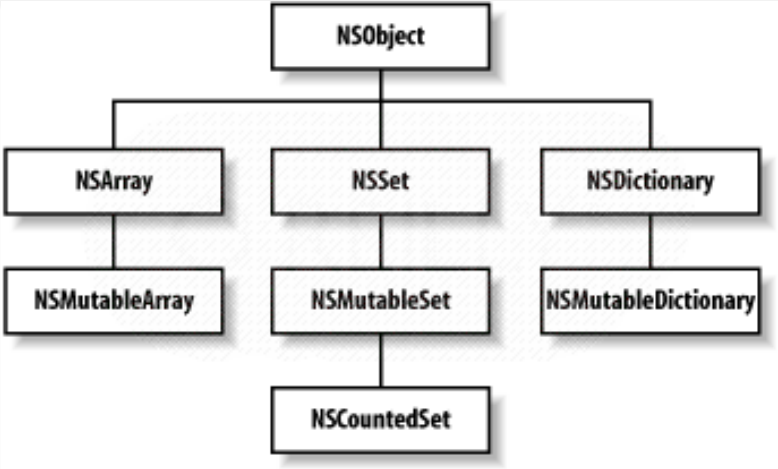
Biblioteca de clases e interfaces que se organizan en una jerarquía de herencia para garantizar la máxima reutilización de código, mediante una o ambas posibilidades del lenguaje:

- Herencia
- Genéricos

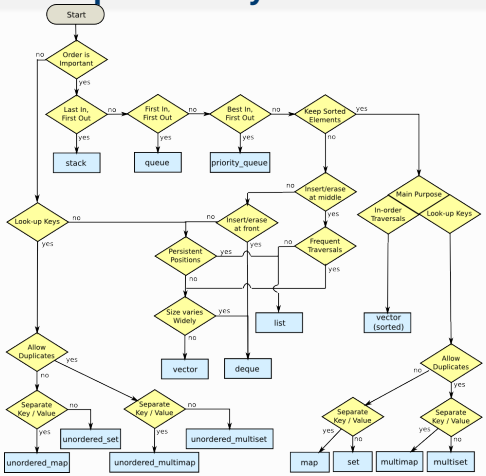
Smalltalk


Collection
Other
Sequenceable
Hashed
Bag
Heap
Unarrayed
LinkedList
OrderedCollection
Arrayed
Set
Dictionary
Interval
Array
String

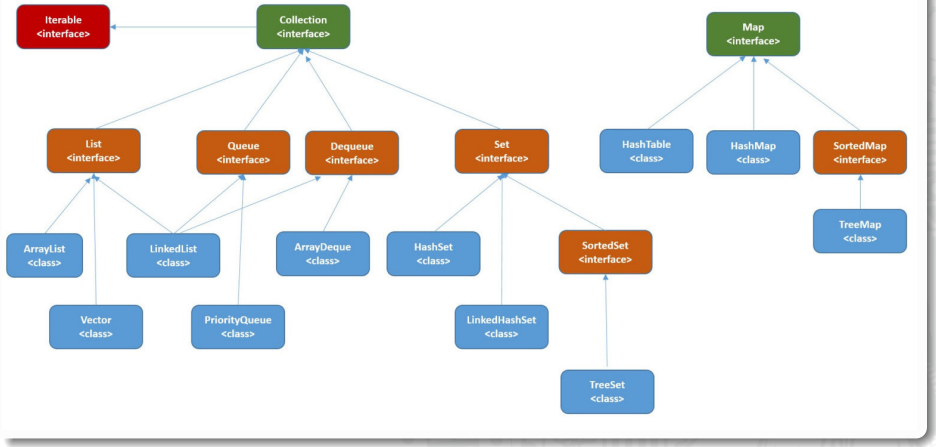
Objective-C


NSObject
NSArray
NSSet
NSDictionary
NSMutableArray
NSMutableSet
NSMutableDictionary
NSCountedSet

C++ STL - Standard Template Library



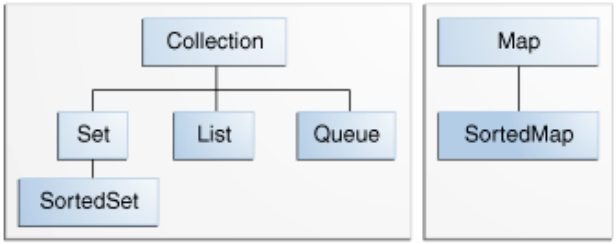
Java JCF - Java Collections Framework



Interfaces vs Conceptos

El cuerpo principal de la JCF es un conjunto de interfaces que permiten manejar distintos tipos de contenedores de una forma homogénea e independiente de la implementación.

Los interfaces se organizan en una jerarquía de herencia:



Todos son interfaces genéricos:

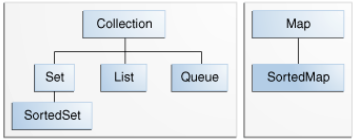
```
1 public interface Collection<E>...
```

Collection

Una colección es el tipo más general de la JCF. Se usa para pasar colecciones a funciones de la forma más general.

Todas las implementaciones tienen un constructor a partir de una Collection para implementar la conversión:

```
1 Collection<String> c;
2 List<String>       l = new ArrayList<String>(c);
```



- `Collection`: raiz de la jerarquia, con el minimo común denominador de todos los interfaces derivados.
- `Set`: sin duplicados ni orden natural
- `List`: orden externo, acceso indexado
- `Queue`: orden interno, acceso secuencial
- `Map`: sin orden, acceso por clave

Collection

Pregunta

Lluvia de ideas: ¿ Qué tienen en común todas las colecciones ?

Problema

Diseña un interfaz común para todas las colecciones.

Importante

¿Cómo recorres todos los elementos de una colección independientemente de su implementación?

```java
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int     size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();
    ...
}
```

```java
public interface Collection<E> extends Iterable<E> {
    ...
    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();
    // Array operations
    Object[] toArray();
    <T> T[]  toArray(T[] a);
}
```

Implementan `Collection`

`BlockingDeque<E>`,`BlockingQueue<E>`, `Deque<E>`,`List<E>`, `NavigableSet<E>`, `Queue<E>`, `Set<E>`, `SortedSet<E>`, `AbstractCollection`, `AbstractList`, `AbstractQueue`, `AbstractSequentialList`, `AbstractSet`, `ArrayBlockingQueue`, `ArrayDeque`, `ArrayList`, `AttributeList`, `ConcurrentLinkedQueue`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `DelayQueue`, `EnumSet`, `HashSet`, `JobStateReasons`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedHashSet`, `LinkedList`, `PriorityBlockingQueue`, `PriorityQueue`, `RoleList`, `RoleUnresolvedList`, `Stack`, `SynchronousQueue`, `TreeSet`, `Vector` ...

Esto funciona con cualquier `Collection` (de enteros):

```
1 public static int sum(Collection<Integer> list) {
2     int sum = 0;
3     for (int i: list) {
4         sum += i;
5     }
6     return sum;
7 }
```

El interfaz *List* es un sub-interfaz de *Collection* que representa una coleccion **ordenada** de elementos.
No tiene por qué ser una *lista* (ejem....).

Incluye nuevas operaciones:

- Acceso indexado
- Busquedas
- Iterador especializado
- Operaciones con rangos

```
1  public interface List<E> extends Collection<E> {
2    void add(int index, E element);
3    E    get(int index);
4    int  indexOf(Object o)
5    int  lastIndexOf(Object o)
6    E    remove(int index)
7    // Iteradores de listas. Tambien son iteradores,
8    // pero con mas operaciones (remove, por ejemplo)
9    ListIterator<E> listIterator()
10   ListIterator<E> listIterator(int index)
11   ...
12 }
```

```java
public interface Set<E> extends Collection<E> {
...
}
```

No puede contener duplicados.

No añade ningún método nuevo, solo restricciones en el comportamiemto.

En relaidad tampoco es una *cola* obligatoriamente....

```
1 public interface Queue<E> extends Collection<E> {
2   ...
3   boolean add(E e)
4   boolean offer(E e)
5   E       remove()
6   E       poll()
7   E       element()
8   E       peek()
9 }
```

Añade operaciones con posibilidad de fallo y notificación.

Deque Añade inserción y eliminación en ambos extremos.

```
1  public interface Deque<E> extends Queue<E> {
2    ...
3    boolean add(E e)
4    void    addLast(E e)
5    void    addFirst(E e)
6    E       remove()
7    E       removeFirst()
8    E       removeLast()
9    ...
10 }
```

Y así podríamos seguir...

... pero para eso está la documentación.

Nota: ojo con la elección de la implementación:

```
List<Thing> l =
  new ArrayList<Thing>();
```

```
List<Thing> l =
  new LinkedList<Thing>();
```

```
for (int i=0; i<l.size() i++)
    doSomething(l.get(i));

for (Thing t : l)
    doSomething(t);
```

La idea de la **Standard Template Library** de C++ es ligeramente diferente.

La idea de la **Standard Template Library** de C++ es ligeramente diferente.

No hace **ningún** uso de la herencia.

La idea de la **Standard Template Library** de C++ es ligeramente diferente.

No hace **ningún** uso de la herencia.

Nunca

Si no utiliza la herencia...

Pregunta

¿ Cómo hace la STL para representar comportamiento común entre diferentes colecciones?

Si no utiliza la herencia...

Pregunta

¿ Cómo hace la STL para representar comportamiento común entre diferentes colecciones?

Respuesta

Requiere que los métodos comunes tengan el mismo interfaz (nombre, parámetros,. . .).

Requiere que los métodos comunes tengan el mismo interfaz (nombre, parámetros,. . .).

```
1 template <typename T>
2 class vector {
3    //...
4    bool empty() { ... }
5    T    front() { ... }
6    T    back()  { ... }
7 };
```

```
1 template <typename T>
2 class list {
3    //...
4    bool empty() { ... }
5    T    front() { ... }
6    T    back()  { ... }
7 };
```

No requiere que todos los contenedores ofrezcan la misma funcionalidad.

```
1  template <typename T>
2  class list {
3     //...
4     bool empty() { ... }
5     T    front() { ... }
6     T    back()  { ... }
7  };
```

```
1  template <typename T>
2  class forward_list {
3     //...
4     bool empty() { ... }
5     T    front() { ... }
6
7  };
```

En lugar de implementar interfaces (usando la herencia) se implementan **conceptos**:

- **Container** - Concepto básico implementado por todos los contenedores.
  - `iterator, begin(), end()`
  - `empty(), size()`, $O(1)$ vs $O(n)$
- **SequenceContainer** - Contenedor de tamaño variable con los elementos ordenados de forma secuencial.
  - `insert(), erase()`
- **ReversibleContainer** - Se pueder recorrer en sentido inverso.
  - `reverse_iterator, rbegin(), rend()`
- **AssociativeContainer** - Contenedor de tamaño variable que permite insertar y recuperar elementos basado en una clave.

Cada concepto define una serie de tipos asociados, métodos y axiomas que debe implementar una clase:

```
1  vector<int>
2
3  vector<int>::value_type  v;  -> int   v;
4  vector<int>::iterator    it; -> (*it) es int
5
6  vector<int>::size_type vector<int>::size(); -> size_t size()
7  bool vector<int>::empty();                  -> bool empty()
```

Para versiones anteriores a C++20 se definen *Named Requirements*:
`https://en.cppreference.com/w/cpp/named_req`

Cada concepto define una serie de tipos asociados, métodos y axiomas que debe implementar una clase.

En C++20 se definen los *concepts* como parte del estándar.

Están implementados con carácter experimental desde gcc-10.

En C++20 se definen los *concepts* como parte del estándar:

```
1   template<typename T>
2   concept equality_comparable =
3     requires(const T& a, const T& b) {
4       { a == b } -> std::same_as<bool>;
5       { a != b } -> std::same_as<bool>;
6     };
7
8   template <equality_comparable T>
9   void f(const T& x, const T& y)
10  {
11    if (x!=y)
12      ...
13  }
```

Container

https://en.cppreference.com/w/cpp/named_req/Container

```
1  using Bag = vector<Thing>;  // typedef std::vector<Thing> Bag;
2  Bag::value_type   v;  // Thing  v;
3  Bag::reference    rv; // Thing& rv;
4  Bag::iterator     it; // Iterador sobre  Bag
5  Bag::size_type    sz; // Tipo que define el tamano/distancia
6
7  Bag b;
8  sz = b.size();     // Tamaño
9  sz = b.max_size(); // Tamaño maximo definido por la implementacion
10 b.empty();         // true si el contenedor esta vacio
```

SequenceContainer

https://en.cppreference.com/w/cpp/named_req/SequenceContainer

```
1  using X = std::vector<T>; // O cualquier clase que implemente Sequence
2  X::value_type   v;
3  X::reference    rv;
4  X::iterator     it;
5  X::size_type    sz;
6
7  X x;
8  x.clear();        // Vaciar de elementos
9  t = x.front();    // Primer elemento
10 t = x.back();     // Ultimo elemento
11 x.push_back(t);   // Añadir elemento
12 x.insert(pos,t);  // Insertar elemento
13 x.erase(pos);     // Borrar elemento
14 x.resize(n);      // Cambiar tamano
```

AssociativeContainer

https://en.cppreference.com/w/cpp/named_req/AssociativeContainer

```
1  using X = std::map<K,T>; // O cualquier clase que lo implemente
2  X::key_type   k;
3  X::value_type v;
4  X::iterator   p;
5  X::size_type  n;
6
7  X x;
8  n = x.count(k); // Cuenta el numero de elementos
9  p = x.find(k);  // Encuentra el elemento
10 x.clear();      // Vaciar de elementos
```

El concepto se aprovecha así:

```
1  /**
2   * Devuelve el primer elemento o un elemento por defecto
3   * S debe satisfacer 'SequenceContainer'
4   */
5  template<typename S>
6  typename S::value_type first_or(const S& seq,
7              const typename S::value_type& def) {
8    if (seq.empty()) return def;
9    else return seq.front();
10 }
```

Y así podríamos seguir...

... pero para eso está la documentación.

Iteradores

Pregunta

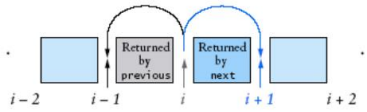
¿Cómo recorres todos los elementos de una colección independientemente de su implementación?

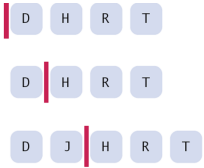
Pregunta

¿Cómo recorres todos los elementos de una colección independientemente de su implementación?

Aprovechando el concepto de *iteradores*

Modelo conceptual





- El iterador está entre dos elementos.
- El avance nos devuelve el elemento sobre el que hemos pasado.
- No se puede inspeccionar sin avanzar.
- El elemento eliminado es el que se acaba de sobrepasar.

```java
public interface Iterable<E> {
    Iterator<E> iterator();
}

public interface Iterator<E> {
    boolean  hasNext()
    E        next()
    void     remove()
}
```

Si un objeto implementa el interface `Iterable`, puede utilizarse sobre él la construcción `for-each`.

```
1 Collecion<E> coll;
```

for-each

```
1 for (E elem : coll)
2     dosomething(elem);
```

Iterator

```
1 for (Iterator<E> it = coll.iterator(); it.hasNext(); )
2     dosomething(it.next());
```

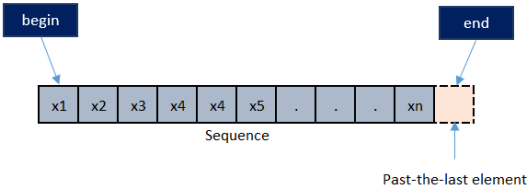
En general los bucles **for-each** son más cómodos. Se usan iteradores cuando:

- Queremos eliminar el elemento

```
1       if (!cond(it.next())) it.remove();
```

- Iteramos sobre varias colecciones a la vez

Modelo conceptual



- El iterador está en un elemento.
- El avance sólo mueve la posición del iterador.
- Se puede inspeccionar el elemento apuntado.
- El elemento eliminado es el que referencia el iterador.

Pregunta

En la STL no se utiliza la herencia.

¿ Qué hacemos para implementar iteradores ?

Pregunta

En la STL no se utiliza la herencia.

¿ Qué hacemos para implementar iteradores ?

Respuesta

Dándoles el mismo nombre: implementando conceptos.

```cpp
template < typename T >
class vector {
   //...
   class iterator  { ... };
   iterator begin(){ ... }
   iterator end()  { ... }
};
```

```cpp
template < typename T >
class list {
   //...
   class iterator  { ... };
   iterator begin(){ ... }
   iterator end()  { ... }
};
```

Uso de iteradores con contenedores:

```
1  vector<Thing> v;
2  for (vector<Thing>::iterator it=v.begin(); it!=v.end(); it++)
3  {
4    Thing& e = (*it);
5    ...
6  }
7
8  list<Thing> l;
9  for (list<Thing>::iterator it=l.begin(); it!=l.end(); it++)
10 {
11   Thing& e = (*it);
12   ...
13 }
```

Uso de iteradores con contenedores: **auto**

```
1  vector<Thing> v;
2  for (auto it=v.begin(); it!=v.end(); it++)
3  {
4    Thing& e = (*it);
5    ...
6  }
7
8  list<Thing> l;
9  for (auto it=l.begin(); it!=l.end(); it++)
10 {
11   Thing& e = (*it);
12   ...
13 }
```

```
1 std::vector<E> coll; //o tambien std::list<E> col;
```

for-each

```
1 for (E elem : coll)
2     dosomething(elem);
```

iterator

```
1 for (auto it = coll.begin(); it != coll.end(); ++it)
2     dosomething(*it);
```

Problema: vectores estandar...

```
1 Thing v[32];
2 for (auto it=v.begin(); it!=v.end(); it++)
3 {
4   Thing& e = (*it);
5   ...
6 }
```

La variable v no es un objeto de una clase:

- No existe el tipo `iterator`.
- No existen `v.begin()`, `v.end()`.

Funciones auxiliares para extenderlo a arrays (que no son clases)

```
1  template <typename V>
2  V::iterator begin(V& v)
3  { return v.begin();  }
4
5  template <typename V>
6  V::iterator end(V& v)
7  { return v.end();  }
```

```
1  template<typename T>
2  T* begin(T v[])
3  { return &(v[0]);  }
4
5  template<typename T>
6  T* end(T v[])
7  { return &(v[0])
8          + sizeof(v)/sizeof(T); }
```

```
E coll[16]; //y tambien std::list<E> coll;
```

for-each

```
for (E elem : coll)
    dosomething(elem);
```

iterator

```
for (auto it = begin(coll); it != end(coll); ++it)
    dosomething(*it);
```

Ejemplos

Vector: colección de elementos ordenados con acceso indexado.
Requerimientos de implementación:

- Los elementos deben estar consecutivos en memoria (`reserve()`, `capacity()`).
- Acceso aleatorio indexado: $\mathcal{O}(1)$.
  Implica iteración directa e inversa (bidireccional).
- Inserción al final: $\mathcal{O}(1)$.
- Inserción general: $\mathcal{O}(n)$ (distancia a `end()`).

Vector: colección de elementos ordenados con acceso indexado.

C++

```cpp
template <typename E>
class vector {
  //Concepts:
  //  Container,
  //  SequenceContainer, ContiguousContainer,
  //  ReversibleContainer.
};
```

Vector: colección de elementos ordenados con acceso indexado.

Java

```java
public class ArrayList<E>
  extends AbstractList<E>
  implements List<E>,RandomAccess,Cloneable,Serializable
```

Lista: contenedor que soporta inseción y eliminación rápida en cualquier punto.
Requerimientos de implementación:

- Inserción y eliminación $\mathcal{O}(1)$.
- Iteración directa.

Opciones:

- Iteración inversa.
- Acceso indexado.

Lista: contenedor que soporta inseción y eliminación rápida en cualquier punto.

C++

```cpp
template <typename E>
class list {
  //  Concepts: Sequence, Reversible
};
template <typename E>
class forward_list {
  //  Concepts: Sequence
};
```

Lista: contenedor que soporta inseción y eliminación rápida en cualquier punto.

Java

```java
public class LinkedList<E>
  extends AbstractSequentialList<E>
  implements List<E>, Deque<E>, Cloneable, Serializable
```

Diccionarios (**map**), conjuntos (**set**), tuplas...

... ver documentación.

```java
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
public class SortingVectorExample {
  public static void main(String[] args) {
     List<String> names = new ArrayList<>();
     names.add("Walter");
     names.add("Anna");
     names.add("Hank");
     names.add("Flynn");
     // Collection.sort() sorts the collection in ascending order
     Collections.sort(names);
     for (String s : names) System.out.println(s);
    }
}
```

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
int main( ) {
    std::vector<string> names;
    names.push_back("Walter");
    names.push_back("Anna");
    names.push_back("Hank");
    names.push_back("Flynn");
    // std::sort() sorts the collection in ascending order
    std::sort(begin(names),end(names));
    for (const auto& n : names) std::cout << n << std::endl;
}
```

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
  std::vector<int> vec { 23, 5, -10, 0, 0, 321, 1, 2, 99, 30 };
  std::sort(begin(vec), end(vec));
  for (int i = 0; i < vec.size(); ++i) {
     std::cout << vec[i] << ' ';
  }
  std::cout << std::endl;
}
```

```cpp
#include <vector>
#include <algorithm>
#include <random>
#include <iostream>

int main() {
  std::vector<int> vec(128);

  // Fill with consecutive integers
  std::iota(begin(vec),end(vec),0);

  // Mesh up randomly
  std::random_device seed;
  std::mt19937 rg(seed());
  std::shuffle(begin(vec), end(vec), rg);

  // Sort again
  std::sort(begin(vec), end(vec));
  for (auto i : vec)
    std::cout << i << ' ';
  std::cout << std::endl;
}
```

En C++20, la biblioteca ranges aún simplifica más las cosas...

```
1   #include <vector>
2   #include <algorithm>
3   #include <random>
4   #include <iostream>
5
6   int main() {
7     std::vector<int> vec(128);
8
9     // Fill with consecutive integers
10    std::ranges::iota(vec,0);
11
12    // Mesh up randomly
13    std::random_device seed;
14    std::mt19937 rg(seed());
15    std::ranges::shuffle(vec, rg);
16
17    // Sort again
18    std::ranges::sort(vec);
19    for (auto i : vec)
20      std::cout << i << ' ';
21    std::cout << std::endl;
22  }
```

Ejercicios

Ejercicios

Plantead cómo extender la JFC/STL con árboles:

- Definid interfaces comunes.
- Definid las relaciones entre esos interfaces y los de la JFC/STL.
- Dad alguna idea acerca de posibles implementaciones.

http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

http://en.cppreference.com/w/cpp/container

Contenedores / Colecciones

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20