



# Introducción a la STL

(**contenedores**, **iteradores**, y **algoritmos**)

Documento original: Bjarne Stroustrup

Adaptación y redacción: J. Fdez-Valdivia



# Índice

- Tareas comunes
- Programación genérica
- Contenedores, algoritmos, e iteradores
- Algunos algoritmos simples y su parametrización (Functores)
- Contenedores secuenciales
  - Vectores, listas...
- Contenedores asociativos
  - map, set...
- Algunos algoritmos estándar
  - copy, sort, ...
  - Input/output iterators
- Lista de utilidades
  - Cabeceras, algoritmos, contenedores, objetos función...



# Tareas comunes

- **Introducir datos** en contenedores
- **Organizar datos**
  - Para imprimir
  - Para tener accesos rápidos
- **Recuperar datos**
  - **Por índice** (p.ej. recuperar el n-ésimo elemento)
  - **Por valor** (p.ej. Recuperar el primer elemento con valor "**covid**")
  - **Por propiedades** (p.ej. Recuperar el primer elemento que cumpla "peso<60")
- **Insertar nuevos datos**
- **Eliminar datos**
- **Ordenar y buscar datos**
- **Operaciones numéricas simples con los datos**



# Ideales (I)

Queremos escribir programas que sean independientes del tipo de dato subyacente:

Usar un **int** no debería ser diferente a usar un **double**

Usar un **vector<int>** no debería ser diferente a usar un **vector<string>**

Deberíamos poder implementar determinadas tareas de alguna forma que no nos condujera a tener que reescribir el código simplemente porque hayamos encontrado una forma más eficiente de almacenar los datos o una interpretación ligeramente diferente de los mismos.

P.ej:

- Buscar un elemento en un **vector** no debería ser diferente a buscar un elemento en una **lista**
- Ordenar datos no debería ser diferente a si los ordenamos en un **vector** ó en una **lista**
- Copiar un **fichero** no debería ser diferente a copiar un **vector**



# Ideales (II)

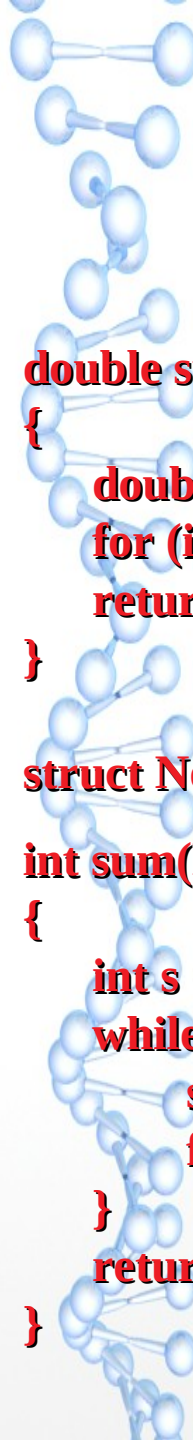
- Queremos código que:
  - Sea fácil de leer y/o modificar
  - Sea de un tamaño razonable y rápido
- Queremos un acceso uniforme a los datos
  - Independientemente de cómo estén almacenados y de su tipo
- Queremos formas simples de recorrer los datos y de almacenarlos de forma compacta que nos permita:
  - Insertarlos, borrarlos o modificarlos de forma rápida y simple
- Queremos versiones estándar de los algoritmos más usuales:
  - Copia, búsqueda, ordenación, suma....



# Programación genérica

- **Generalizar algoritmos**
  - Independizarlos del tipo de dato subyacente
- Los **objetivos** (para el usuario final) son:
  - **Mayor corrección**
    - A través de una mejor especificación
  - **Mayor amplitud de uso**
    - Posibilidades de reutilización
  - **Mayor eficiencia**
    - A través del uso de potentes bibliotecas
    - Se eliminará el código innecesariamente lento
- **Ir de lo concreto a lo abstracto**

# Ejemplo (algoritmos concretos)



```
double sum(double v[], int n) // un algoritmo concreto (en un vector de doubles)  
{  
    //para sumar los elementos del vector  
    double s = 0;  
    for (int i = 0; i < n; ++i ) s = s + v[i];  
    return s;  
}
```

```
struct Node { Nodo* siguiente; int data; };
```

```
int sum(Nodo* first) //otro algoritmo concreto (en una lista de enteros)  
{  
    int s = 0;  
    while (first) { // termina cuando una expresión es false ó cero  
        s += first->data;  
        first = first->siguiente;  
    }  
    return s;  
}
```



# Ejemplo (algoritmo abstracto )

*// pseudo-código para una versión más general de ambos algoritmos*

```
int sum(data)    // Parametrizar de alguna forma la estructura de datos  
{  
    int s = 0;        // inicializar  
    while (no se alcance el final) {    // bucle recorriendo los elementos  
        s = s + valor recuperado;    // calcular la suma  
        Ir al siguiente elemento;  
    }  
    return s;        // devolver el resultado  
}
```

- Necesitamos cuatro operaciones (sobre la estructura de datos):
  - Inicializar el recorrido
  - Saber cuando se alcanza el final
  - Recuperar valor
  - Ir al siguiente elemento



# Ejemplo (versión STL)

// Código estilo STL para una versión más general de ambos algoritmos

```
template<class Iter, class T>    // Iter debería ser un Input_iterator
                                // T debería ser algo sobre lo que podemos hacer + e =
T sum(Iter first, Iter last, T s) // T es el “tipo acumulador”
{
    while (first!=last) {
        s = s + *first;           // Podría ser válido para vectores, listas...
        ++first;
    }
    return s;
}
```

■ Dejamos que el usuario inicialice el acumulador

```
float a[] = { 1,2,3,4,5,6,7,8 };
double d = 0;
d = sum(a,a+sizeof(a)/sizeof(*a),d);
```



# La STL

Diseñada por **Alexander Stepanov y Meng Lee**

- **Objetivo general:** Representar de la forma más general, eficiente y flexible estructuras de datos y algoritmos
  - Representar conceptos separados de forma separada en el código
  - Combinar conceptos libremente cuando tenga sentido combinarlos
- Hacer programas de forma abstracta, “como en matemáticas”
  - “La buena programación es matemática”
  - **Estructuras de datos y algoritmos independientes del tipo de dato subyacente**

**Es estándar ISO C++ (desde 1994) y tiene unos 10 contenedores y alrededor de 60 algorithms conectados por iteradores**

Otras organizaciones proporcionan contenedores y algoritmos siguiendo las pautas de la STL: **Boost.org, Microsoft, SGI, ...**

**Ahora mismo, es probablemente la biblioteca de herramientas de programación genérica más conocida y más usada**



# La STL

- Si conoces los conceptos básicos y unos pocos ejemplos, puedes usar facilmente el resto.
- **Documentación**
  - SGI
    - <http://www.sgi.com/tech/stl/> (recomendado por su claridad)
  - Dinkumware
    - <http://www.dinkumware.com/refxcpp.html> (tened cuidado con las varias versiones que hay de la biblioteca)
  - Rogue Wave
    - <http://www.roguewave.com/support/docs/sourcepro/stdlibug/index.html>
- Hay muchos sitios con documentación accesible aunque sea menos completa



# ¿Por qué usar la STL?

- ❖ La STL ofrece un buen surtido de contenedores de los que informa de su complejidad en tiempo y espacio.
  - ❖ Los contenedores de la STL crecen y decrecen en tamaño de forma automática
  - ❖ La STL proporciona algoritmos integrados para procesar los contenedores
  - ❖ La STL proporciona iteradores que hacen que los contenedores y algoritmos sean flexibles y eficientes
- La STL es **extensible**, lo que significa que los usuarios pueden agregar nuevos contenedores y nuevos algoritmos de manera que:
- Los algoritmos de la STL pueden procesar no solo sus contenedores propios, sino los definidos por el usuario
  - Los algoritmos definidos por el usuario pueden procesar tanto los contenedores de la STL como los suyos propios



# Componentes de la STL

❖ La STL tiene 3 componentes básicos:

- **Contenedores**

Clases templates genéricas para almacenar colecciones de datos de cualquier tipo P.ej. vectores ó listas

- **Algoritmos**

Funciones template genéricas que se pueden aplicar sobre los contenedores para procesar sus datos. P.ej., buscar un elemento en un vector u ordenar una lista.

- **Iteradores**

Generalización del concepto de puntero. Apuntan a elementos de un contenedor. P.ej., se puede incrementar un iterador para apuntar al elemento siguiente en un vector ó una lista. Proporcionan un interface de conexión necesario para que los algoritmos puedan operar sobre los contenedores.

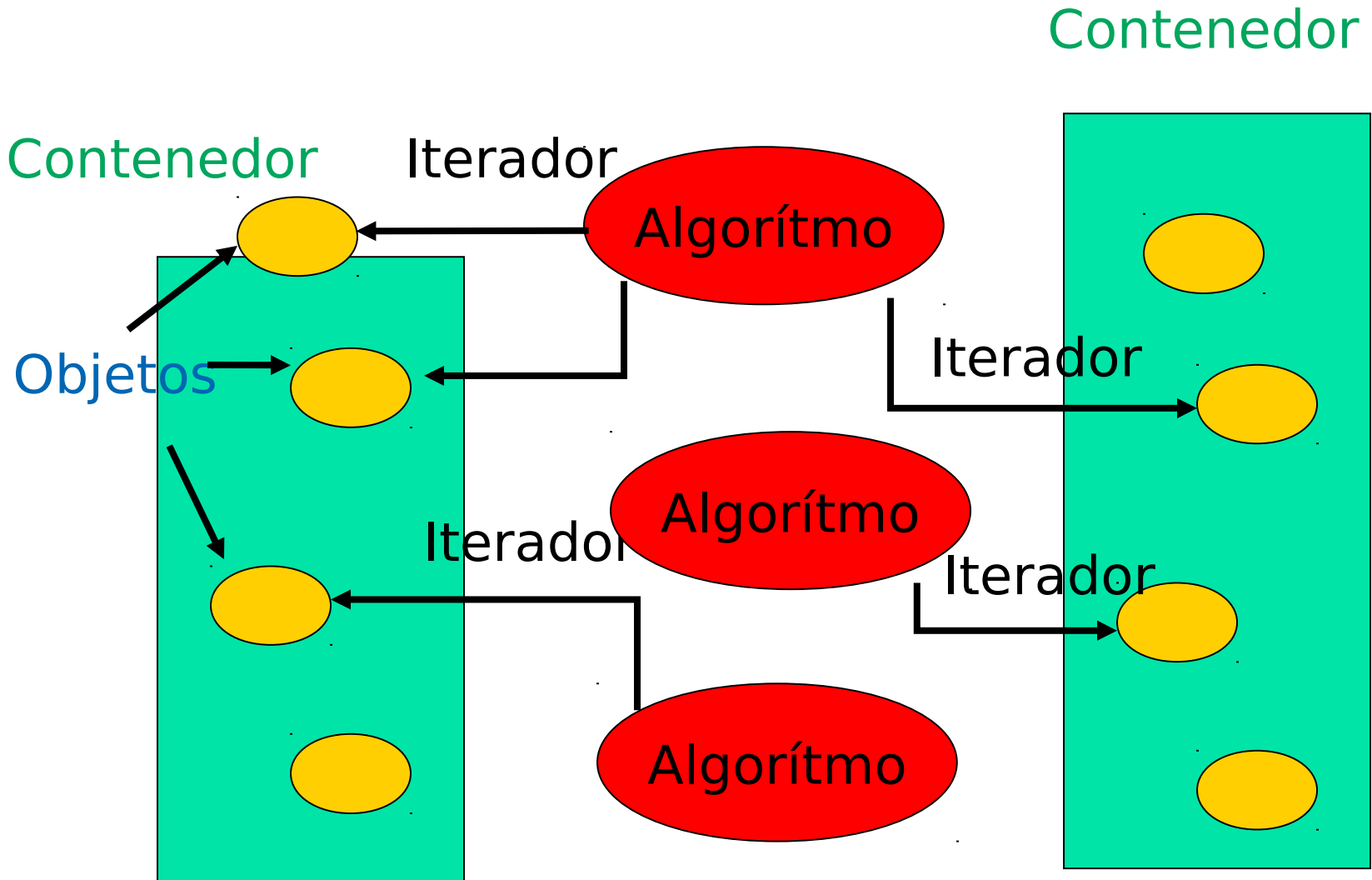
❖ **La abstracción sobre String fue añadida durante el proceso de estandarización.**

# Cabeceras en la STL

<b>#include &lt;iostream&gt;</b>	I/O streams, cout, cin, ...
<b>#include &lt;fstream&gt;</b>	file streams
<b>#include &lt;algorithm&gt;</b>	sort, copy, ...
<b>#include &lt;numeric&gt;</b>	accumulate, inner_product, ...
<b>#include &lt;functional&gt;</b>	objetos función
<b>#include &lt;string&gt;</b>	clase string
<b>#include &lt;vector&gt;</b>	clase vector
<b>#include &lt;map&gt;</b>	clase map y multimap
<b>#include &lt;unordered_map&gt;</b>	tabla hash
<b>#include &lt;list&gt;</b>	clase lista
<b>#include &lt;set&gt;</b>	clase set y multiset
<b>#include &lt;stack&gt;</b>	clase pila
<b>#include &lt;queue&gt;</b>	clases cola y cola con prioridad
<b>#include &lt;stl&gt;</b>	todas las clases de la STL

# Contenedores, Iteradores, Algoritmos

Los algoritmos usan iteradores para interactuar con los objetos de los contenedores





# Modelo básico

## ■ Algoritmos

**sort, find, search, copy, ...**

iteradores

## ■ Contenedores

**vector, list, set, unordered\_map, ...**

## Separación de intereses

Los algoritmos manipulan datos, pero no saben nada sobre los contenedores

Los contenedores almacenan datos, pero no saben nada sobre los algoritmos

Algoritmos y contenedores interactúan a través de los iteradores

Cada contenedor tiene sus propios tipos de iteradores



# Iteradores y su semántica

Proporcionan una **forma general de acceso** a los elementos tanto de contenedores secuenciales (vector, list) como asociativos (map, set).

Tienen la semántica de los punteros. P.ej:

Si **iter** es un iterador entonces :

**++iter** (ó **iter++**) avanza el iterador al siguiente elemento.

**\*iter** devuelve el valor del elemento direccionado por el iterador



# Begin y End

Cada contenedor proporciona funciones miembro **begin()** y **end()**.

- **begin()** devuelve un iterador que direcciona el **primer** elemento del contenedor.
- **end()** devuelve un iterador que direcciona el elemento **siguiente al último** del contenedor.

## Iteración sobre los contenedores

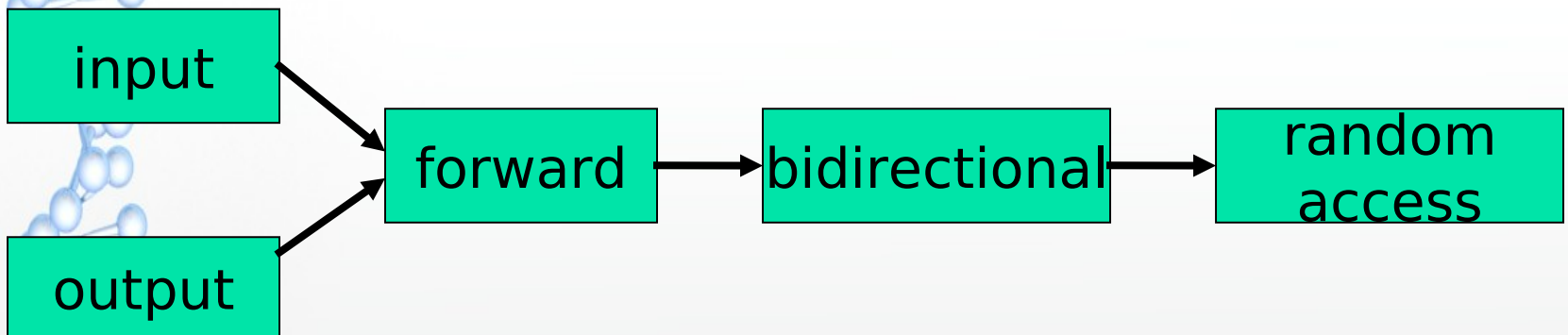
Iterar sobre los elementos de cualquier contenedor que lo permita, se hará así:

```
for ( iter = contenedor.begin();  
      iter != contenedor.end();  
      ++iter )  
{  
    // *iter hacer algo con el elemento  
}
```

# Categorías de iteradores

- No todo tipo de iterador puede usarse con cualquier contenedor. P.ej. La clase lista no proporciona el iterador de acceso aleatorio (random access iterator)
- Cada algoritmo requiere un tipo de iterador con un cierto nivel de capacidad. P.ej., para usar el operador [] se necesita un random access iterator

Los iteradores se dividen en cinco categorías en las que una categoría más alta (más específica) siempre subsume una categoría más baja (más general). P.ej. Un algoritmo que acepta un forward iterator también trabajará con bidirectional iterators y random access iterators.





# Categorías de iteradores

- **Input**
  - Leen elementos del contenedor y solo se pueden mover hacia delante
- **Output**
  - Escriben elementos en un contenedor solo hacia delante
- **Forward**
  - Combinan input y output y retienen la posición
  - “Multi-pass” (pueden pasar dos veces por la secuencia )
- **Bidirectional**
  - Como el forward, pero también se puede mover hacia atrás
- **Random access**
  - Como el bidirectional, pero puede saltar a cualquier elemento

# Operaciones con iteradores según la categoria

- Todos:

`++p, p++`

- Input iterators

`*p`

`p=p1`

`p==p1`

`p!=p1`

- Output iterators

`*p`

`p=p1`

- Forward iterators

Tienen la funcionalidad de los input y output iterators

- Bidireccional iterators

`--p`

`p--`

- Random access iterators

`p + i, p += i, p - i, p -= i, < p1, p <= p1,`

`p > p1, p >= p1`

# Operaciones con iteradores según la categoría: Resumen

	Output	Input	Forward	Bi-directional	Random
Leer		<code>x = *i</code>	<code>x = *i</code>	<code>x = *i</code>	<code>x = *i</code>
Escribir	<code>*i = x</code>		<code>*i = x</code>	<code>*i = x</code>	<code>*i = x</code>
Iteración	<code>++</code>	<code>++</code>	<code>++</code>	<code>++</code> , <code>--</code>	<code>++</code> , <code>--</code> , <code>+</code> , <code>-</code> , <code>+=</code> , <code>-=</code>
Comparación		<code>==</code> , <code>!=</code>	<code>==</code> , <code>!=</code>	<code>==</code> , <code>!=</code>	<code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&gt;</code> , <code>&lt;=</code> , <code>&gt;=</code>

- ◆ **Output:** solo puede escribir una vez
- ◆ **Input:** puede leer muchas veces
- ◆ **Forward:** soporta tanto lectura como escritura
- ◆ **Bi-directional:** soporta también el decremento
- ◆ **Random:** soporta acceso aleatorio (como un puntero C)





# Ejemplo de Input/output iterators

*// iteradores para output streams*

```
ostream_iterator<string> oo(cout);    // asignar a *oo es lo mismo  
                                         // que escribir en cout  
*oo = "Hello, ";    // es lo mismo que cout << "Hello, "  
++oo;              // "listo para la siguiente operación de salida"  
*oo = "world!\n";  // es lo mismo que cout << "world!\n"
```

*// Iteradores para input streams:*

```
istream_iterator<string> ii(cin);    // leer *ii es lo mismo que leer  
                                         // un string desde cin  
string s1 = *ii; // es lo mismo que cin>>s1  
++ii;              // "listo para la siguiente operación de entrada"  
string s2 = *ii; // es lo mismo que cin>>s2
```



# Contenedores e Iteradores que soportan

- Contenedores secuenciales
  - **vector**: random access
  - **deque**: random access
  - **list**: bidirectional
- Contenedores asociativos (en todos bidirectional)
  - **set**
  - **multiset**
  - **Map**
  - **multimap**
- Contenedores adaptados (no soportan iteradores)
  - **stack**
  - **queue**
  - **priority\_queue**
- **iostreams** Input/output/forward



# Iteradores y contenedores

Los iteradores permiten acceder/modificar elementos

```
Container C;
```

```
Container::iterator i,j;
```

- ♦ **C.insert(i,x)** // inserta **x** antes de **i**
- ♦ **C.insert(i,first,last)** // inserta elementos en **[first,last)**  
//antes de **i**
- ♦ **C.erase(i)** //borra el elemento direccionado por **i**
- ♦ **C.erase(i,j)** // borra elementos en el rango **[i,j)**



# Introducción a los contenedores

Un contenedor lo forman un conjunto de datos junto con un conjunto de operaciones asociadas

- Tres tipos de contenedores

- Contenedores secuenciales

- Estructuras de datos lineales (vectores, listas..)
    - Contenedores de “primera clase”

- Contenedores asociativos

- Son no lineales y buscan rápidamente un elemento
    - Parejas Clave/valor
    - Contenedores de “primera clase”

- Quasi contenedores (contenedores adaptados)

- Similares a los contenedores, pero con funcionalidad reducida

- Los contenedores tienen funciones asociadas comunes



# Contenedores y “quasi contenedores”

- Contenedores secuenciales
  - **vector, list, deque**
- Contenedores asociativos
  - **map, set, multimap, multiset**
- Contenedores adaptados (“quasi contenedores”)
  - **string, stack, queue, priority\_queue**
- Nuevos contenedores incluidos en el estándar C++11
  - **unordered\_map** (tabla hash), **unordered\_set**, ...
- Para cualquier duda, consultar documentación
  - Online
    - **SGL, RogueWave, Dinkumware**
  - Libros
    - **Stroustrup: The C++ Programming language 4<sup>th</sup> ed. (Capítulos 30-33, 40.6)**
    - **Austern: Generic Programming and the STL**
    - **Josuttis: The C++ Standard Library**



# Funciones miembro comunes en los contenedores

- Funciones miembro para todos los contenedores
  - **Constructor por defecto, de copias, destructor**
  - **Función `empty`**
  - **Funciones `max_size`, `size`**
  - **Operadores `=` `<` `<=` `>` `>=` `==` `!=`**
  - **Función `swap`**
- Funciones comunes para los contenedores de “primera clase”
  - **`begin`, `end`**
  - **`rbegin`, `rend`**
  - **Funciones `erase`, `clear`**



# Typedefs comunes

- **typedefs** para contenedores de “primera clase”
  - **value\_type**
  - **iterator**
  - **const\_iterator**
  - **reverse\_iterator**
  - **const\_reverse\_iterator**
  - **size\_type**





# Repaso inicial a los contenedores

■ Hagamos un repaso rápido por la definición de los contenedores más usuales

- `<vector>` : array unidimensional
- `<list>` : listas doblemente enlazadas
- `<deque>` : cola doble
- `<queue>` : cola
- `<stack>` : pila
- `<set>` : conjunto
- `<map>` : array asociativo



# Repaso inicial de contenedores

Un contenedor secuencial almacena un conjunto de elementos en una secuencia, es decir, cada elemento (excepto el primero y el último) es precedido y seguido por otro elemento (tiene un siguiente y un anterior)

`<vector>`, `<list>` y `<deque>` son contenedores secuenciales.

Un contenedor asociativo, no es secuencial, sino que usa una clave para acceder a los elementos. Usualmente las claves son numéricas o strings y permiten almacenar los elementos en un orden específico. P.ej. En un diccionario, las entradas son ordenadas alfabéticamente.

`<set>` y `<map>` son contenedores asociativos.



# Repaso inicial de contenedores

**<vector>** array unidimensional que puede crecer y decrecer de tamaño, lento para inserciones y borrados en medio de la estructura y rápido en el acceso aleatorio a los elementos

- **<list>** lista doblemente enlazada. Rápida en inserciones y borrados, lenta en el acceso aleatorio a los elementos.
- **<deque>** cola de doble entrada. En la que se pueden insertar y borrar elementos por los dos extremos. Es una especie de combinación entre una **<stack>** (last in first out) and una **<queue>** (first in first out) y constituye un compromiso entre un **<vector>** y una **<list>**



# Repaso inicial de contenedores

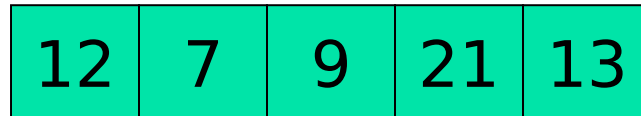
- Un **<set>** almacena elementos que contienen claves usadas para ordenarlos. P.ej. Podríamos tener un conjunto formado por elementos de una clase "persona" ordenados alfabéticamente por su nombre
- Un **<map>** almacena parejas <clave, valor> y los elementos se ordenan por la clave. Parecido a un vector, pero en lugar de acceder con índices numéricos, se puede acceder con índices de tipo arbitrario.
- <set> and <map> solo permiten una clave para cada valor. Si permitimos más de una, tenemos los contenedores **<multiset>** y **<multimap>** que permiten múltiples claves idénticas.

## Ejemplo de contenedor: Vector

```
template<class T> class vector {  
    T* elements;  
    // ...  
    using value_type = T;  
  
    iterator begin();      // apunta al primer elemento  
    const_iterator begin() const;  
    iterator end();        // apunta a la posición de trás de la última  
    const_iterator end() const;  
  
    iterator erase(iterator p);      // elimina el elemento apuntado por p y  
                                     // devuelve un iterador al elemento siguiente  
  
    iterator insert(iterator p, const T& x);    // inserta un nuevo elemento x  
                                                  // delante de p y devuelve un iterador al elemento insertado  
    .....  
};
```

# Ejemplo de contenedor: Vector

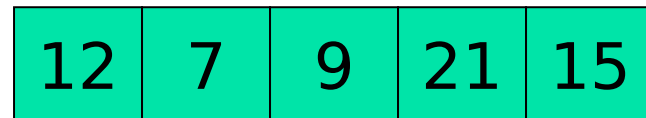
```
int array[5] = {12, 7, 9, 21, 13};  
vector<int> v(array, array+5);
```



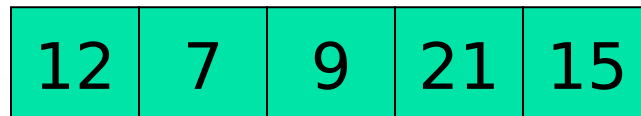
`v.pop_back();`



`v.push_back(15);`



0    1    2    3    4



`v.begin();`

`v[3]`

# Ejemplo de contenedor: Vector

```
#include <vector>
```

```
#include <iostream>
```

```
vector<int> v(3); // crea un vector de enteros de tamaño 3
```

```
v[0]=23;
```

```
v[1]=12;
```

```
v[2]=9; // vector lleno
```

```
v.push_back(17); // poner un nuevo valor al final del array
```

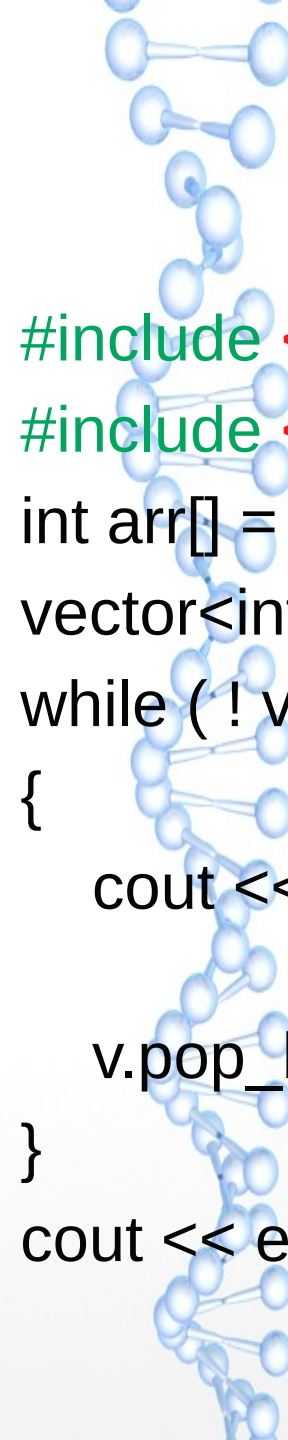
```
for (int i=0; i<v.size(); i++) // function miembro size() del vector
```

```
    cout << v[i] << " "; // acceso aleatorio al i-ésimo elemento
```

```
cout << endl;
```



# Ejemplo de contenedor: Vector



```
#include <vector>
#include <iostream>

int arr[] = { 12, 3, 17, 8 }; // array C estándar
vector<int> v(arr, arr+4); // inicializa un vector con un array C
while ( ! v.empty())      // hasta que el vector esté vacío
{
    cout << v.back() << " "; // imprime el último elemento del vector

    v.pop_back();            // borra el último elemento
}
cout << endl;
```



# Ejemplo de contenedor: Vector

- Un vector puede inicializarse especificando su tamaño y un elemento prototipo o con otro vector

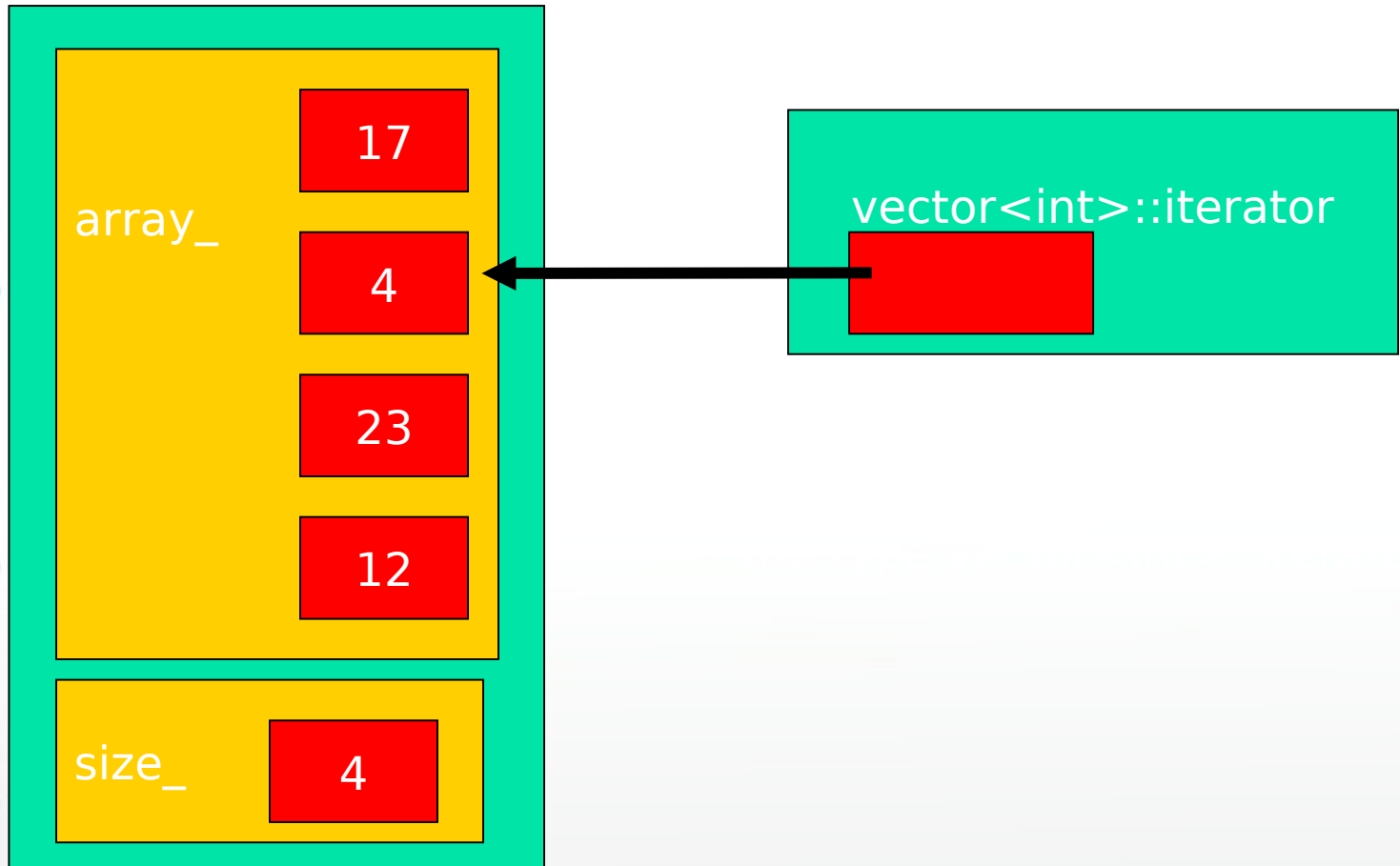
```
vector<Fecha> x(1000); //crea un vector de tamaño 1000,  
                        // requiere constructor por defecto para Fecha
```

```
vector<Fecha> dates(10,Date(17,12,1999)); // inicializa  
                                           // todos los elementos con 17.12.1999
```

```
vector<Fecha> z(x); // inicializa el vector z con el vector x
```

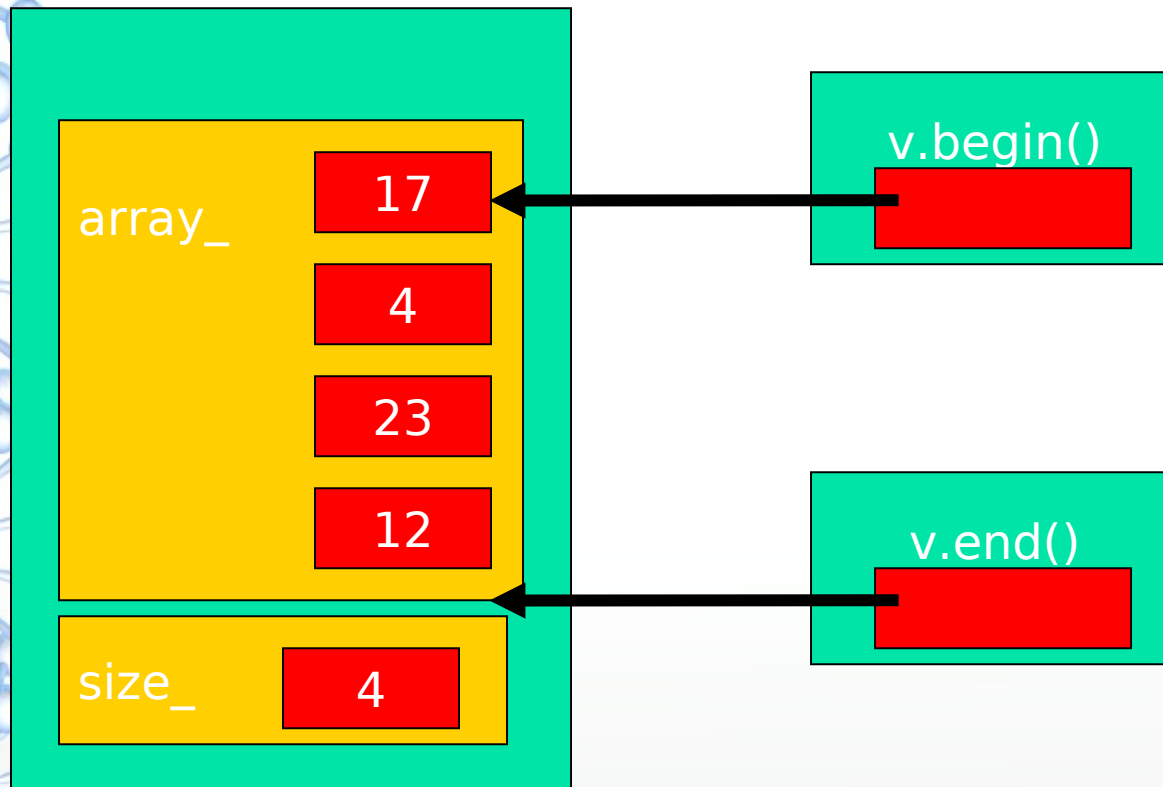
# Ejemplo de contenedor: Vector

- Se usan Iteradores para acceder a los elementos del vector e iterar sobre ellos



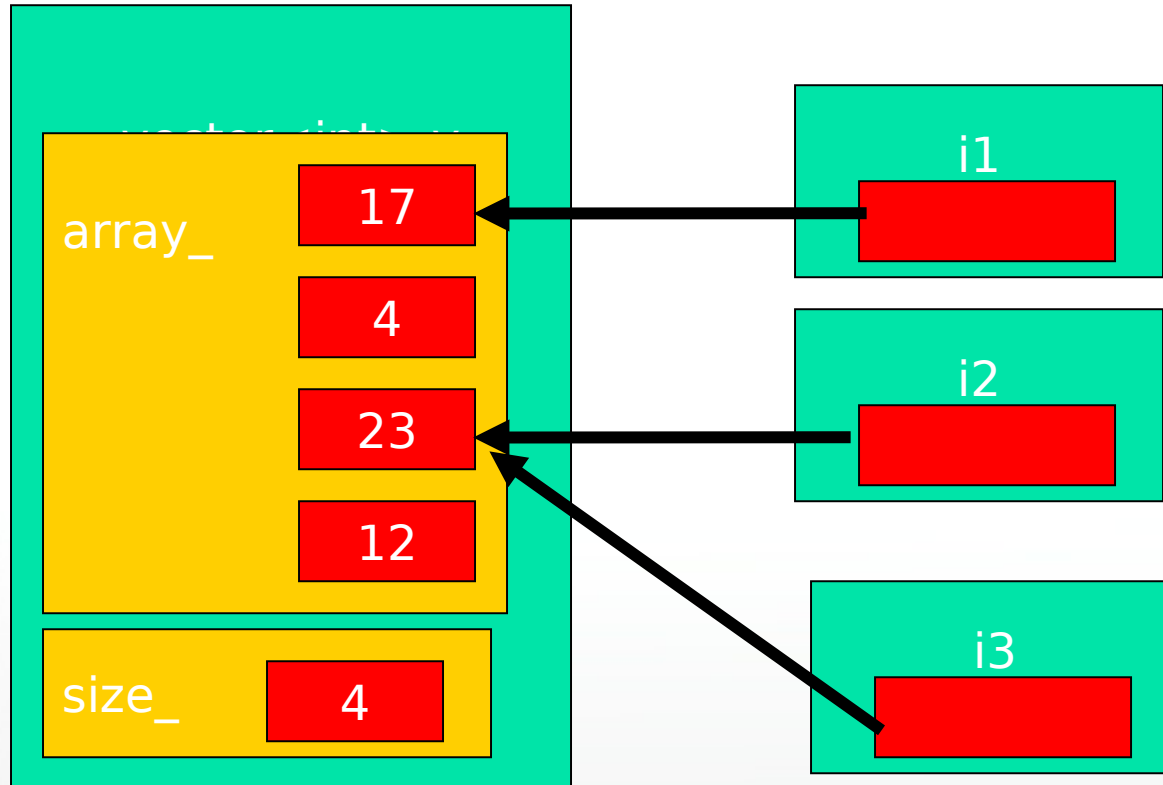
# Ejemplo de contenedor: Vector

- Las funciones miembro `begin()` y `end()` devuelven un iterador al primer elemento y al siguiente al último en el vector



# Ejemplo de contenedor: Vector

- Podemos tener múltiples iteradores apuntando a diferentes o idénticos elementos en el vector



# Ejemplo de contenedor: Vector

```
#include <vector>
#include <iostream>
```

```
int arr[] = { 12, 3, 17, 8 }; // array C estándar
vector<int> v(arr, arr+4); // inicializa vector con un array C
vector<int>::iterator iter=v.begin(); //iterador para la clase vector
// define un iterador para el vector y lo apunta al primer elemento de v
cout << "primer elemento de v=" << *iter; // deferencia iter
iter++; // mueve el iterador al siguiente elemento
iter=v.end()-1; // mueve el iterador al último elemento
```

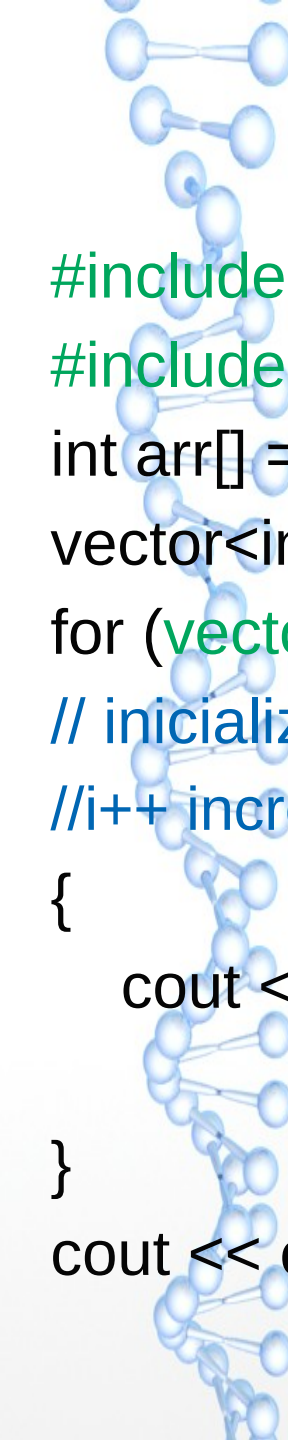
# Ejemplo de contenedor: Vector

```
int max(vector<int>::iterator inicio, vector<int>::iterator final)
{
    int m=*inicio;
    while(inicio != final)
    {
        if (*inicio > m)
            m=*inicio;
        ++inicio;
    }
    return m;
}

cout << "máximo de v = " << max(v.begin(),v.end());
```



# Ejemplo de contenedor: Vector



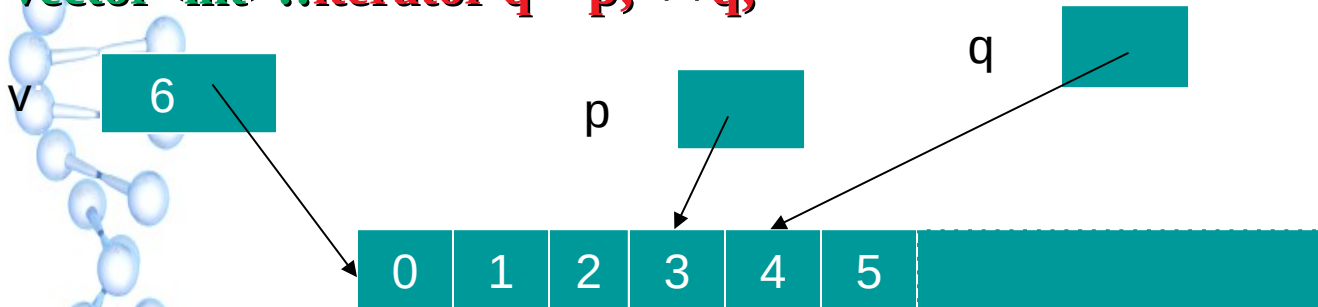
```
#include <vector>
#include <iostream>

int arr[] = { 12, 3, 17, 8 }; // array C estándar
vector<int> v(arr, arr+4); // inicializa vector con un array C
for (vector<int>::iterator p=v.begin(); p!=v.end(); p++)
// inicializa i con un puntero al primer elemento de v
//i++ incrementa el iterador, mueve el iterador al siguiente elemento
{
    cout << *p << " "; // deferencia el iterador devolviendo el valor
                        // del elemento apuntado por el iterador
}
cout << endl;
```

# Inserción en un vector

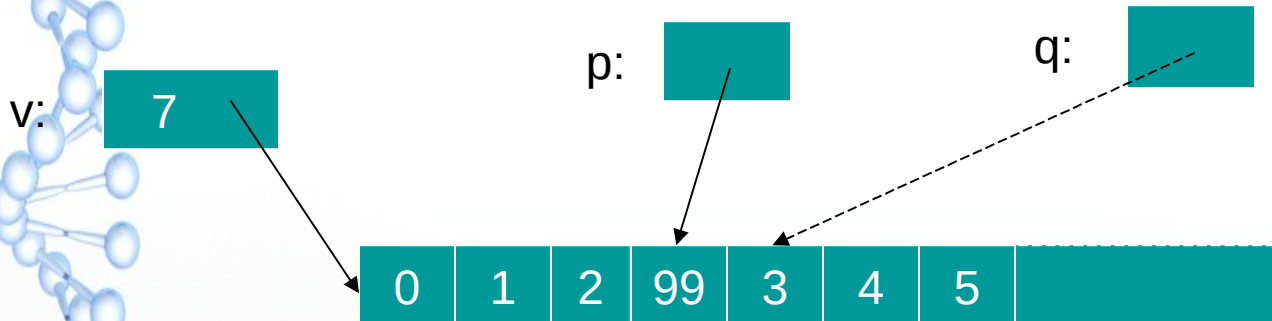
```
vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
```

```
vector<int>::iterator q = p; ++q;
```



```
p=v.insert(p,99);
```

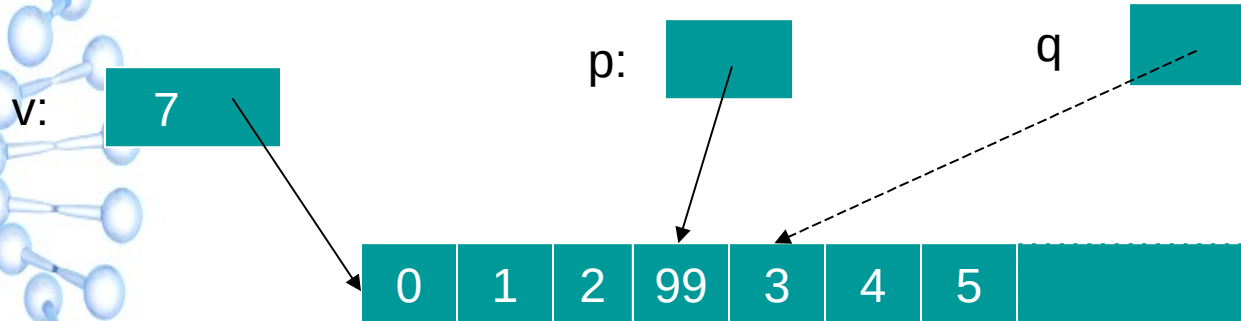
*// deja `p` apuntando al elemento insertado*



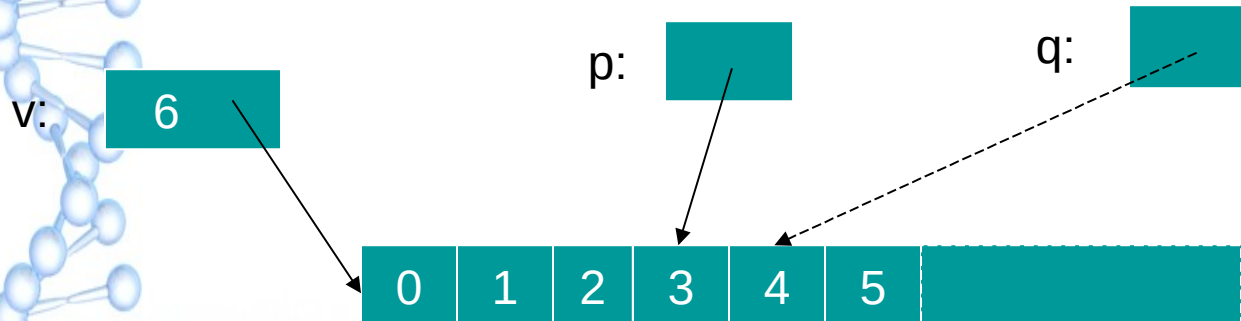
Nota: el iterador `q` es inválido tras aplicar **insert()**

Nota: Algunos elementos (podrían ser todos) se mueven de posición

# Borrado en un vector



**`p = v.erase(p);`** //deja `p` apuntando al elemento detras del que se ha borrado



- Algunos (podrían ser todos) se mueven tras aplicar **`erase()`**
- Los iteradores en el vector quedan invalidados tras **`erase()`**

# Formas de recorrer un vector

```
for(int i = 0; i<v.size(); ++i)  
... // hacer algo con v[i]
```

// ¿por qué int?

```
for(vector<T>::size_type i = 0; i<v.size(); ++i)  
... // hacer algo con v[i]
```

// más larga pero correcta

```
for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p)  
... // hacer algo con *p
```

- Conocidas ambas formas (iterador e índices):
  - El “estilo índice” puede usarse en cualquier lenguaje
  - El “estilo iterador” se usa en C++ y en los algoritmos estándar de la STL
  - El “estilo índice” no funciona para las listas (ni en C++ ni en la mayoría de los lenguajes)
- Pueden usarse ambas modalidades de recorrido sobre un vector, pero:
  - Hay ventajas en usar punteros en vez de enteros (iteradores sobre índices)
  - El “estilo iterador” funciona sobre cualquier tipo de contenedor secuencial
  - Es preferible **size\_type** sobre **int** porque puede prevenir errores raros

# Formas de recorrer un vector

```
for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p)
```

```
... // hacer algo con *p
```

```
for(vector<T>::value_type x : v)
```

```
... // hacer algo con x
```

```
for(auto& x : v)
```

```
... // hacer algo con x
```

## ■ “Rango for”

- Usarlo en bucles simples

- Cada elemento desde **begin()** a **end()**

- Sobre una secuencia

- Cuando no necesitas mirar a más de un elemento a la vez

- Cuando no necesitas conocer la posición de un elemento

## Otro ejemplo de contenedor: list

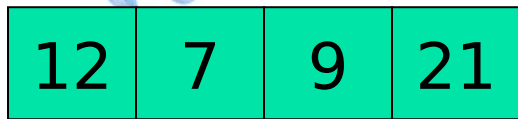
```
template<class T> class list {  
    Link* elements;  
    // ...  
    using value_type = T;  
  
    iterator begin();      // apunta al primer elemento  
    const_iterator begin() const;  
    iterator end();        // apunta a la posición detrás de la última  
    const_iterator end() const;  
  
    iterator erase(iterator p);      // elimina el elemento apuntado por p  
                                     //y devuelve un iterador al elemento siguiente  
    iterator insert(iterator p, const T& x);  // inserta un nuevo valor x  
                                               // delante de p y devuelve un iterador al elemento insertado  
};
```

# Otro ejemplo de contenedor: list

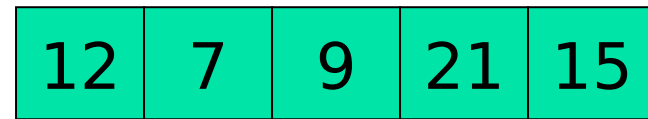
```
int array[5] = {12, 7, 9, 21, 13};  
list<int> li(array, array+5);
```



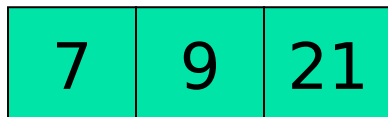
li.pop\_back();



li.push\_back(15);



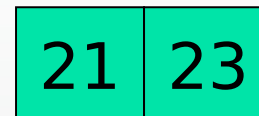
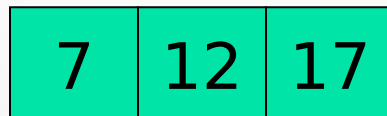
li.pop\_front();



li.push\_front(8);



li.insert(19, p)



p



# Otro ejemplo de contenedor: list

- Inicialización de listas con vectores y copia de una lista a otra

```
#include <list>
```

```
int arr1[] = { 1, 3, 5, 7, 9 };
```

```
int arr2[] = { 2, 4, 6, 8, 10 };
```

```
list<int> l1(arr1, arr1+5); // inicializa l1 con el vector arr1
```

```
list<int> l2(arr2, arr2+5); // inicializa l2 con el vector arr2
```

```
copy(l1.begin(), l1.end(), l2.begin());
```

```
// copia el contenido de l1 a l2 sobrescribiendo los elementos en l2
```

```
// queda entonces l2 = { 1, 3, 5, 7, 9 }
```

# Sort & Merge

- Sort y merge nos permiten ordenar y mezclar elementos en un contenedor

```
#include <list>
```

```
int arr1[] = { 6, 4, 9, 1, 7 };
```

```
int arr2[] = { 4, 2, 1, 3, 8 };
```

```
list<int> l1(arr1, arr1+5); //inicializa l1 con el vector arr1
```

```
list<int> l2(arr2, arr2+5); //inicializa l2 con el vector arr2
```

```
l1.sort(); // l1 = {1, 4, 6, 7, 9}
```

```
l2.sort(); // l2 = {1, 2, 3, 4, 8}
```

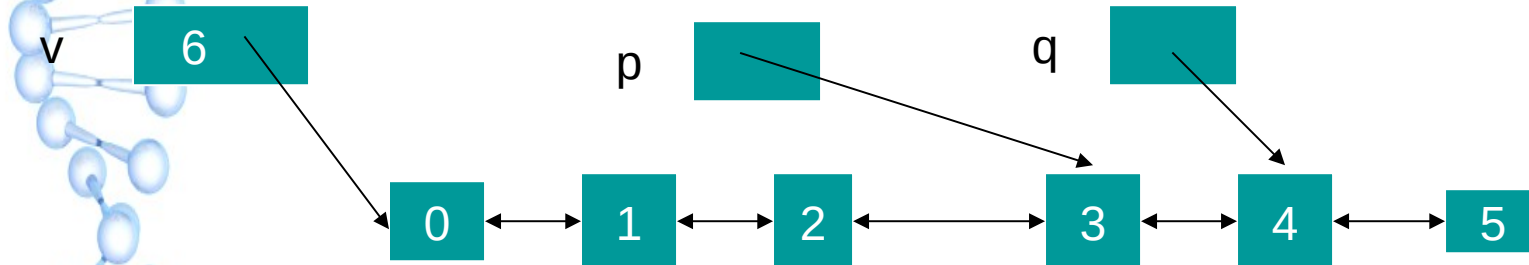
```
l1.merge(l2); // mezcla l2 en l1
```

```
// l1 = { 1, 1, 2, 3, 4, 4, 6, 7, 8, 9}, l2 = {}
```

# Inserción en una lista

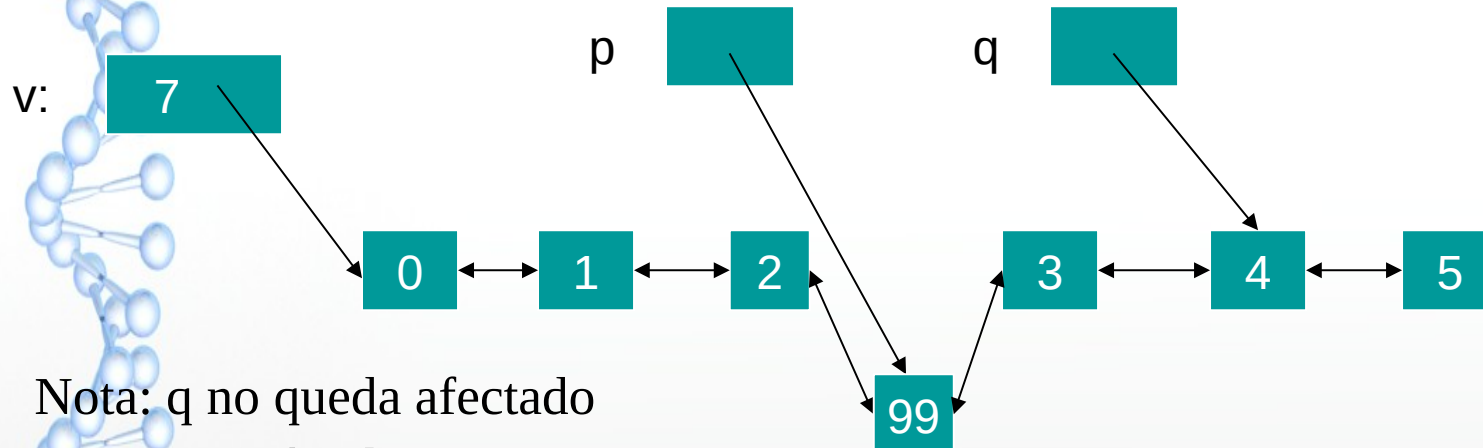
```
list<int>::iterator p = v.begin(); ++p; ++p; ++p;
```

```
list<int>::iterator q = p; ++q;
```



```
v = v.insert(p,99);
```

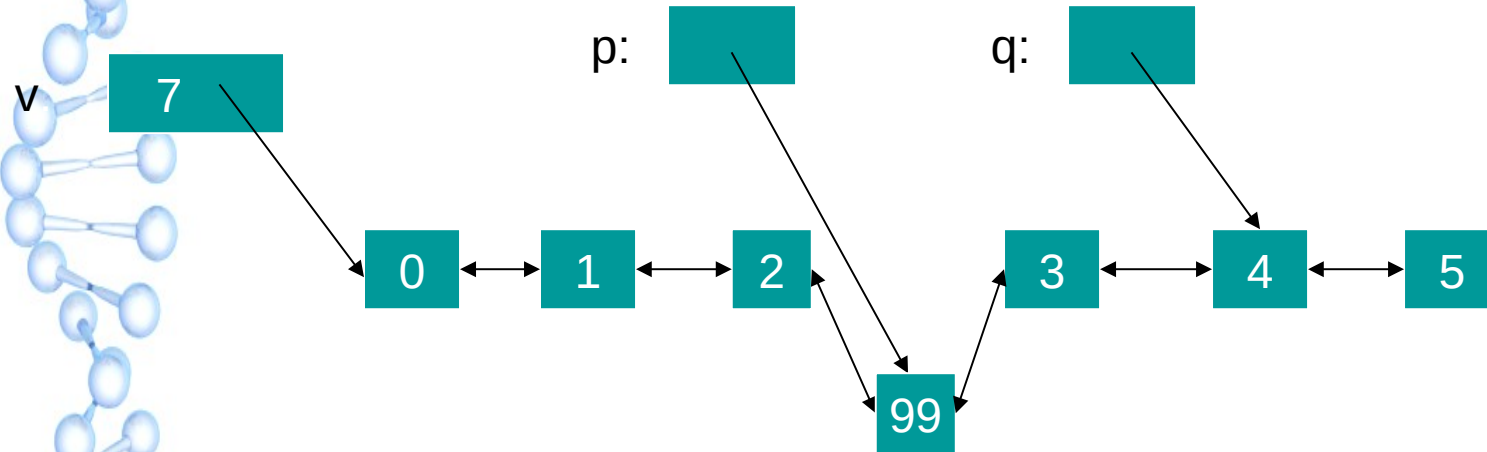
// deja p apuntando al elemento insertado



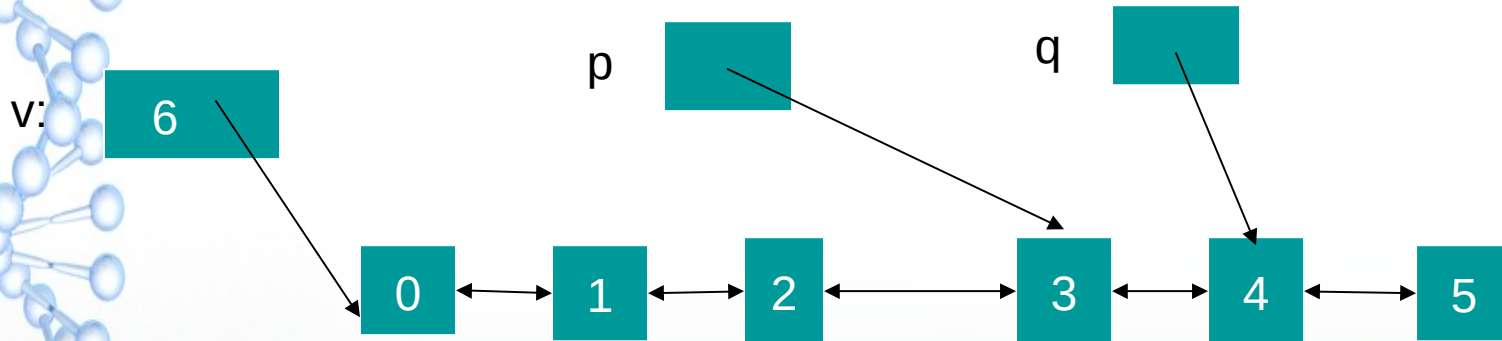
Nota: q no queda afectado

Nota: Ningún elemento se mueve

# borrado en una lista



**p = v.erase(p);** // *deja p apuntando al elemento siguiente al que se ha borrado*



Nota: los elementos de la lista no se mueven al hacer **insert()** ó **erase()**



# Vector vs. List

- Por defecto, usar un **vector**
  - Necesitas una razón para no hacerlo
  - Puedes hacer “crecer” un vector (p.ej. usando **push\_back()**)
  - Puedes insertar ( **insert()** ) y borrar ( **erase()** ) en un vector
  - Los elementos en un vector se almacenan de forma compacta y contigua
  - Para vectores pequeños con elementos “pequeños” todas las operaciones son rápidas comparadas con las listas
- Si no quieres que los elementos se muevan, usa una **list**
  - Puedes hacer “crecer” una lista (p.ej. usando **push\_back()** y **push\_front()**)
  - **Puedes insertar ( **insert()** ) y borrar ( **erase()** ) en una lista**
  - Los elementos en una lista se localizan en memoria de forma separada



# Introducción a los Algoritmos

- Un algoritmo de la STL:

- **Toma una o más secuencias**

- Normalmente como parejas de iteradores

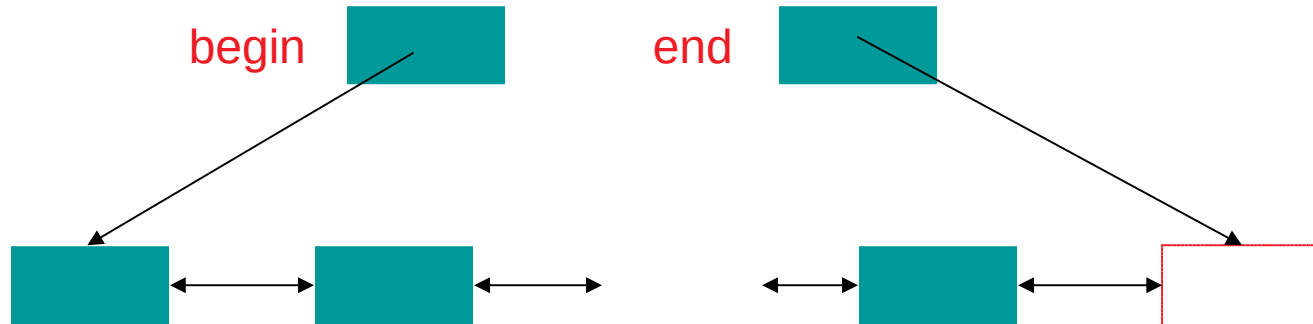
- **Toma una o más operaciones**

- Normalmente como objetos función (functores)
    - Pueden usarse también funciones ordinarias

- Normalmente **informa de un “fallo” devolviendo el `end()`** de la secuencia (posición detrás del último)

# Modelo básico

- Una pareja de iteradores define una secuencia
  - El principio (apunta al primer elemento – si existe)
  - El final (punta a la posición detrás del último)



El iterador soporta las funciones típicas de:

**++** va al siguiente elemento

**\*** devuelve el elemento apuntado por el iterador

**=!** ¿Apunta el primer iterador al mismo elemento que el segundo?

Pueden soportarse otras operaciones (p.ej.. **--**, **+**, y **[ ]**)



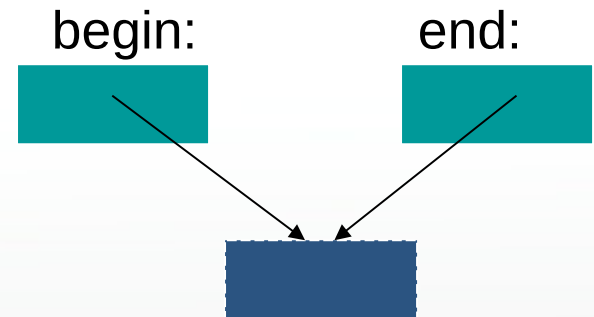
# Modelo básico

- El `end()` de la secuencia es “uno-pasado-el-último-elemento”
  - *No es* “el último elemento”
  - Es necesario para representar elegantemente secuencias vacías
  - uno-pasado-el-último-elemento no es un elemento
    - Se puede comparar un iterador que apunte a el
    - No se puede deferenciar (leer su valor)
- Devolver `end()` es lo estándar para indicar “no encontrado” ó “fracaso”

algún  
iterador:



secuencia vacia:



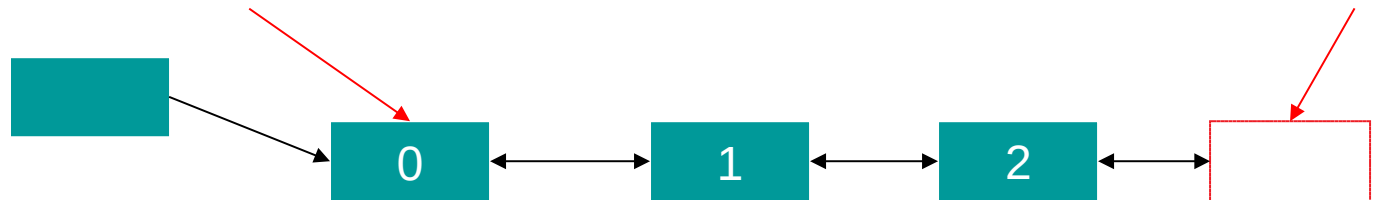
# Modelo básico

(se mantienen secuencias de diferentes formas según sea el contenedor sobre el que se actúa)

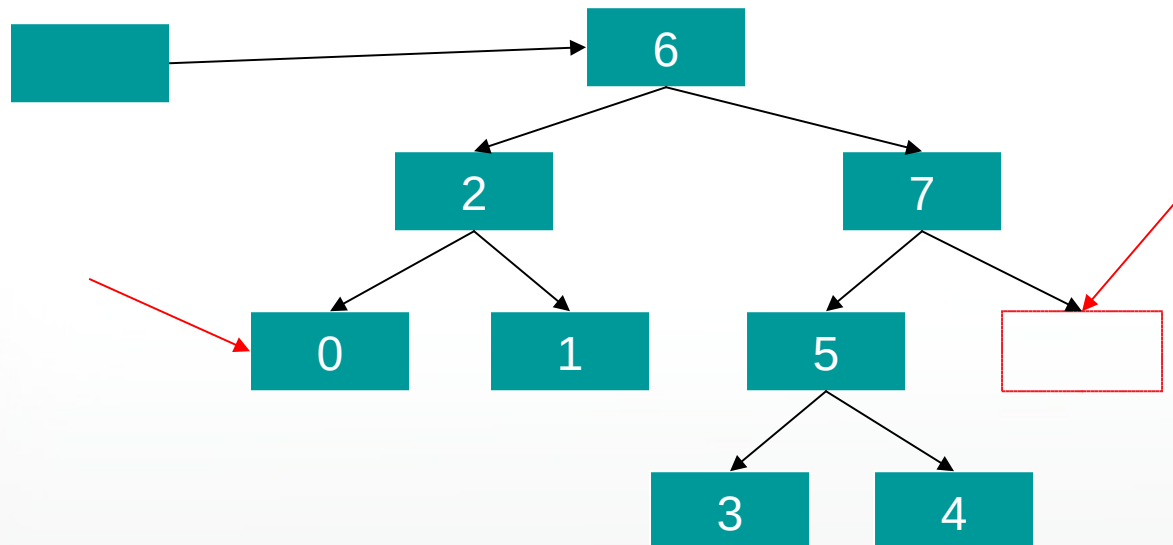
## vector



## list (doble)



## Set (un tipo de árbol )





# Visión general de los algoritmos

## ◆ Operaciones sobre secuencias

- Sin modificar: `for_each`, `find`, `count`, `search`, `mismatch`, `equal`
- Modificando: `transform`, `copy`, `swap`, `replace`, `fill`, `generate`, `remove`, `unique`, `reverse`, `rotate`, `random_shuffle`

## ◆ Operaciones sobre secuencias ordenadas

- `sort`, `lower_bound`, `upper_bound`, `equal_range`, `binary_search`, `merge`, `includes`, `set_union`, `intersection`, `set_difference`, `set_symmetric_difference`

# Algunos algoritmos estándar muy usados

- **$r = \text{find}(b, e, v)$**   $r$  apunta a la primera ocurrencia de  $v$  en  $[b, e)$
- **$r = \text{find\_if}(b, e, p)$**   $r$  apunta al primer elemento  $x$  en  $[b, e)$  para el que el predicado  $p$  es cierto ( $p(x)$  true)
- **$x = \text{count}(b, e, v)$**   $x$  es el número de ocurrencias de  $v$  in  $[b, e)$
- **$x = \text{count\_if}(b, e, p)$**   $x$  es el número de elementos en  $[b, e)$  para los cuales el predicado  $p$  es cierto ( $p(x)$  true)
- **$\text{sort}(b, e)$**  ordena  $[b, e)$  usando  $<$
- **$\text{sort}(b, e, p)$**  ordena  $[b, e)$  usando el predicado  $p$
- **$\text{copy}(b, e, b2)$**  copia  $[b, e)$  en  $[b2, b2 + (e - b))$   
debe haber suficiente espacio después de  $b2$
- **$\text{unique\_copy}(b, e, b2)$**  copia  $[b, e)$  en  $[b2, b2 + (e - b))$  pero no copia duplicados adyacentes
- **$\text{merge}(b, e, b2, e2, r)$**  mezcla dos secuencias ordenadas  $[b2, e2)$  and  $[b, e)$  en  $[r, r + (e - b) + (e2 - b2))$
- **$r = \text{equal\_range}(b, e, v)$**   $r$  es la subsecuencia de  $[b, e)$  con el valor  $v$  en todas las posiciones (hace una búsqueda binaria para encontrar  $v$ )
- **$\text{equal}(b, e, b2)$**  ¿Son iguales todos los elementos de  $[b, e)$  y  $[b2, b2 + (e - b))$  ?

# Un algoritmo simple: **find()**



*// Encuentra el primer elemento que coincide con un valor*

```
template<class In, class T>
```

```
In find(In first, In last, const T& val)
```

```
{
```

```
    while (first!=last && *first != val) ++first;
```

```
    return first;
```

```
}
```

```
void f(vector<int>& v, int x) //encuentra un entero x en un vector
```

```
{
```

```
    vector<int>::iterator p = find(v.begin(),v.end(),x);
```

```
    if (p!=v.end()) { /* hemos encontrado x */ }
```

```
    // ...
```

```
}
```

Podemos ignorar (“camino abstracto”) las diferencias entre contenedores

# find()

genérica tanto para el tipo elemento como para el contenedor

```
void f(vector<int>& v, int x) // trabaja para un vector de int
```

```
{  
    vector<int>::iterator p = find(v.begin(),v.end(),x);  
    if (p!=v.end()) { /* encontramos x */ }  
    // ...  
}
```

```
void f(list<string>& v, string x) // trabaja para una list de strings
```

```
{  
    list<string>::iterator p = find(v.begin(),v.end(),x);  
    if (p!=v.end()) { /* encontramos x */ }  
    // ...  
}
```

```
void f(set<double>& v, double x) // trabaja para un set de doubles
```

```
{  
    set<double>::iterator p = find(v.begin(),v.end(),x);  
    if (p!=v.end()) { /* encontramos x */ }  
    // ...  
}
```



# Algoritmo: `find_if()`

- Encuentra el primer elemento que cumple un criterio (predicado)
  - Aquí, un predicado toma un argumento y devuelve un **bool**

```
template<class In, class Pred>
```

```
In find_if(In first, In last, Pred pred)
```

```
{
```

```
    while (first!=last && !pred(*first)) ++first;
```

```
    return first;
```

```
}
```

```
void f(vector<int>& v)
```

```
{
```

```
    vector<int>::iterator p = find_if(v.begin(),v.end,Odd());
```

```
    if (p!=v.end()) { /* encontramos un número impar */ }
```

```
    // ...
```

```
}
```

// un predicado





# Algoritmo copy()

- El algoritmo de copia admite variantes interesantes que pueden modificar su comportamiento:

- **back\_inserter** : inserta nuevos elements al final
- **front\_inserter**: inserta nuevos elementos al principio
- **inserter** : inserta nuevos elementos en una localización específica

```
#include <list>
```

```
int arr1[] = { 1, 3, 5, 7, 9 };
```

```
int arr2[] = { 2, 4, 6, 8, 10 };
```

```
list<int> l1(arr1, arr1+5); // Inicializa l1 con el vector arr1
```

```
list<int> l2(arr2, arr2+5); // Inicializa l2 con el vector arr2
```

```
copy(l1.begin(), l1.end(), back_inserter(l2)); // uso de back_inserter
```

```
// añade elementos de l1 al final de l2 = { 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 }
```

```
copy(l1.begin(), l1.end(), front_inserter(l2)); // uso de front_inserter
```

```
// añade elementos de l1 al principio de l2 = { 9, 7, 5, 3, 1, 2, 4, 6, 8, 10 }
```

```
copy(l1.begin(), l1.end(), inserter(l2, l2.begin()));
```

```
// añade elementos de l1 en el "antiguo" principio de l2 = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 }
```

# Predicados y funtores

- Un predicado (de un argumento) es una función ó un objeto función (functor) que toma un argumento y devuelve un **bool**

- Por ejemplo

- Una función

```
bool odd(int i) { return i%2; } // % es el operador (modulo/resto)  
odd(7); // llama a odd: ¿es 7 impar?
```

- Un objeto función (functor)

```
struct Odd {  
    bool operator()(int i) const { return i%2; }  
};  
Odd odd; // define un objeto odd de tipo Odd  
odd(7); // llama a odd: ¿es 7 impar?
```



# Objetos function (functores)

- Algunos algoritmos como **sort**, **merge**, **accumulate** pueden tomar un objeto función (functor) como argumento.
- Una **function object** ó **functor** es un objeto de una clase template que tiene una sola función miembro: la sobrecarga del **operator ()**
- Los funtores pueden estar predefinidos o pueden ser definidos por el usuario
- Importantes para las técnicas de programación funcional en C++

```
#include <list>
```

```
#include <functional>
```

```
int arr1[] = { 6, 4, 9, 1, 7 };
```

```
list<int> l1(arr1, arr1+5); //inicializa l1 con el vector arr1
```

```
l1.sort(greater<int>()); // usa el functor greater<int> para
```

```
// ordenar los datos de l1 de mayor a menor l1 = { 9, 7, 6, 4, 1 }
```

# Objetos función (functores)

## ■ Otro ejemplo

```
template<class T> struct Less_than {  
    T val;    // valor para comparar  
    Less_than(T& x) :val(x) { }  
    bool operator()(const T& x) const { return x < val; }  
};
```

```
// encuentra x<43 en vector<int> :
```

```
p=find_if(v.begin(), v.end(), Less_than(43));
```

```
// encuentra x<"perfecto" en una list<string>:
```

```
q=find_if(ls.begin(), ls.end(), Less_than("perfecto"));
```



# Objetos Función predeterminados

- Estos funtores se encuentran en (**<functional>**)
  - Contienen funciones invocadas usando el operador ( )

Funtores predeterminados	Tipo
<b>divides</b> < T >	aritmético
<b>equal_to</b> < T >	relacional
<b>greater</b> < T >	relacional
<b>greater_equal</b> < T >	relacional
<b>less</b> < T >	relacional
<b>less_equal</b> < T >	relacional
<b>logical_and</b> < T >	lógico
<b>logical_not</b> < T >	lógico
<b>logical_or</b> < T >	lógico
<b>minus</b> < T >	aritmético
<b>modulus</b> < T >	aritmético
<b>negate</b> < T >	aritmético
<b>not_equal_to</b> < T >	relacional
<b>plus</b> < T >	aritmético
<b>multiplies</b> < T >	aritmético



# Objetos función (functores) definidos por el usuario

```
class squared_sum // functor definido por el usuario
{
public:
    int operator()(int n1, int n2) { return n1+n2*n2; }
};

int sq = accumulate(l1.begin(), l1.end() , 0, squared_sum() );
// calcula la suma de cuadrados
```



# Objetos función (functores) definidos por el usuario. Comparaciones

*// Diferentes comparaciones para objetos de tipo Rec (registro con varios // campos, dos de ellos son el nombre y la dirección):*

```
struct Cmp_por_nombre {  
    bool operator()(const Rec& a, const Rec& b) const  
        { return a.nombre < b.nombre; }    // mira el campo name de Rec  
};
```

```
struct Cmp_por_direccion {  
    bool operator()(const Rec& a, const Rec& b) const  
        { return 0 < strcmp(a.direccion, b.direccion, 24); }  
};
```

*// int strcmp(const char \* s1, const char \* s2, size\_t num);*

*// La función strcmp compara las cadenas hasta el carácter situado en la // posición num. Compara carácter a carácter, de forma que si las cadenas // son iguales retornara un valor 0, si la primera cadena es mayor retornará // un valor positivo y si es menor retornara un valor negativo.*



# Objetos función (functores) definidos por el usuario. Comparaciones

- Siempre que se tenga un algoritmo útil podemos parametrizarlo con el uso de los functores
  - Por ejemplo, necesitamos parameterizar **sort** por el criterio de comparación

```
struct Persona {  
    string nombre;           // string estándar  
    char direccion[24];      // otra forma (antigua) de definir un string  
    // ...  
};  
  
vector<Persona> vr;  
// ...  
  
sort(vr.begin(), vr.end(), Cmp_por_nombre()); // ordena por nombre  
sort(vr.begin(), vr.end(), Cmp_por_direccion()); // ordena por direccion
```

# Funciones lambda

- Las funciones lambda que se pueden pasar inline a otras funciones. Se usan corchetes [] para definirlas.
  - Por ejemplo, `auto func = [] () { cout << "hello world"; };` y se llama como: `func();`
  - En el ejemplo anterior, podríamos usar como funciones lambda:

```
vector<Persona> vr;
```

```
// ...
```

```
sort(vr.begin(), vr.end(),
```

```
    [] (const Rec& a, const Rec& b)
```

```
        { return a.nombre < b.nombre; }    // ordenar por nombre
```

```
);
```

```
sort(vr.begin(), vr.end(),
```

```
    [] (const Rec& a, const Rec& b)
```

```
        { return 0 < strncmp(a.direccion, b.direccion, 24); }
```

```
        // ordenar por direccion
```



# Functores vs funciones lambda

- Usa un functor como argumento
  - Si quieres hacer algo complicado
  - Si quieres hacer lo mismo en varios sitios
- Usa una función lambda como argumento
  - Si lo que quieres hacer es corto y obvio
- Elige en cualquier caso basándote en el criterio de la claridad del código porque:
  - No hay diferencias esenciales en cuanto a eficiencia entre funciones objeto y funciones lambda
  - Las funciones objeto (y las lambdas) tienden a ser más rápidas que los argumentos de funciones predicado