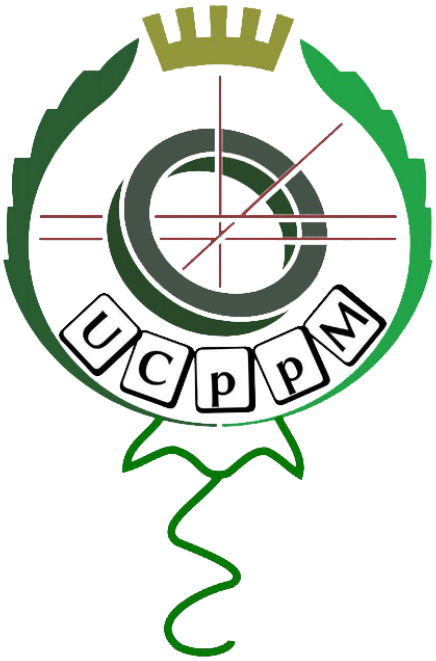
# Vectores C++ STL

Apuntes para programación competitiva




**Autor:** Jorge Hernández Palop

# 1 Introducción

## 1.1 ¿Qué es un vector?

Un **vector** es una **estructura de datos** que nos permite guardar **elementos de un mismo tipo** de manera **secuencial**. Es decir es una lista de elementos del mismo tipo.

Existen otros mecanismos que nos permiten hacer lo mismo, como los arrays. Pero la principal ventaja de los vectores es que podemos **aumentar y disminuir el tamaño de nuestro vector a lo largo de la ejecución del programa**, algo imposible con los arrays.

Para poder usar los vectores deberemos de **importalos de la librería estándar de C++** por medio del siguiente comando `#include <vector>`.

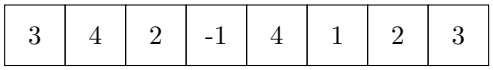



**Figura 1**: Representación gráfica de un vector `v` de tamaño 8 de tipo `int`

## 1.2 ¿Cómo acceder a los elementos de un vector?

Para acceder a cada elemento de un vector se debe usar un **indice** que corresponde con la **posición del elemento** dentro del vector. En la mayoría de lenguajes de programación el indice **comienza en 0** y C++ no es la excepción.

Para acceder a los elementos indicaremos el nombre del vector y el indice entre paréntesis, por ejemplo, `miVector[0]`. De esta manera podemos leer, modificar y operar con los elementos del vector como si se tratasen de variables comunes y corrientes.

Por medio del método `front` podemos acceder al primer elemento del vector de la siguiente manera `miVector.front()` y con `back` al último de elemento de la misma manera `miVector.back()`.

**IMPORTANTE:** Solo podemos usar indices que se encuentren en el rango $[0, \text{tamaño vector})$. Si tratamos de acceder elementos fuera del tamaño del vector lo más posible es que el programa explote.

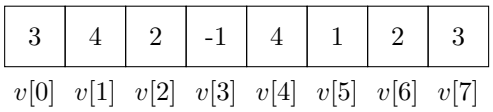



**Figura 2**: Indices del vector $v$

## 1.3 ¿Cómo se crea un vector?

Existen varias maneras de crear o inicializar un vector. Estas son algunas, la primera es usando el constructor por defecto. De esta manera creamos un vector de tamaño 0 con el tipo (`int, string, long long...`) que especifiquemos.

```
vector<tipoElementos> miVector;
```

La segunda manera es especificando el tamaño del vector antes de crearlo, de esta manera nos aseguraremos que el vector tenga un tamaño minimo antes de operar con él. Todos los elementos del vector pasarán a tener un valor por defecto que dependerá del tipo del vector, en el caso de los tipos númericos este valor será 0.

```
vector<tipoElementos> miVector(tamanoVector);
```

La tercera forma de crearlo es indicando un valor por defecto al vector además del tamaño. El resultado es el mismo que de la manera anterior, lo único que cambia es el valor por defecto con el que comienza cada elemento.

```
vector<tipoElementos> miVector(tamanoVector, valorPorDefecto);
```

La última manera que se usa en programación competitiva es mediante inicialización de listas. Le pasamos una serie de elementos entre llaves al vector. El vector resultante tendra exactamente el tamaño de la lista que le hemos introducido y los elementos en el orden en el que se encontraban en la lista.

```
vector<tipoElementos> miVector {elemento0, elemento1, ..., elementoN-1};
```

### 1.3.1 Ejemplos

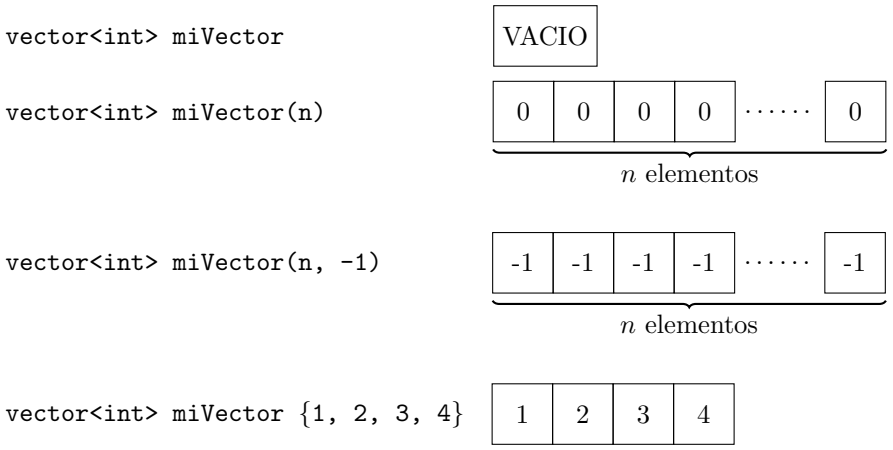



**Figura 3**: Distintos tipos de inicialización de vectores

## 1.4 Notas Extras

*En muchos problemas nos darán el tamaño del vector antes de decirnos sus elementos, en estos casos crear un vector vacio de tamaño n es nuestra mejor opción.*

*Aunque ir insertando elementos en un vector es 'rápido', esto supone un coste extra y hará que nuestro programa corra algo más lento. Por lo tanto el 75% de las veces va a ser mejor inicializar un vector con un tamaño n, antes que ir insertando elementos poco a poco.*

# 2 Operaciones

Ya hemos visto algunas operaciones básicas de los vectores, cómo crearlos y cómo acceder a sus elementos. Ahora veremos otras operaciones igual de importantes.

## 2.1 Obtener el tamaño de un vector

Para obtener el tamaño de un vector usamos el `size` que nos un numero entero sin signo. También existe el método `empty` que devuelve `true` si y solo si el vector está vacio, es decir, no tiene elementos.

**Código 1: Tamaño de un vector**

```
vector<int> v {3, 4, -1, 2};

cout << v.size() << endl; // 4
cout << v.empty() << endl; // false
```

**IMPORTANTE:** `size` devuelve un número entero sin signo asi que hay que tener cuidado con las restas. Por ejemplo:

**Código 2: Cuidado con los unsigned en `size()`**

```
vector<int> v;

// Este bucle falla 0 - 1 unsigned = 2^63 - 1
for(int i = 0; i < v.size() - 1; i++) {
    cout << i << " ";
}
cout << v[v.size() - 1] << endl;

// Este bucle no falla 0 - 1 signed = -1
for(int i = 0; i < (int) v.size() - 1; i++) {
    cout << i << " ";
}
cout << v[v.size() - 1] << endl;
```

## 2.2 Leer vectores de la entrada estándar

Hemos visto que podemos inicializar el vector con valores por defecto o por medio de una lista. Sin embargo queremos poder cambiar esos valores según la entrada del problema para ello veremos dos maneras.

La primera manera es asegurándonos de que tenemos el espacio suficiente para todos los elementos dentro del vector y por medio de un bucle ir leyendo de la entrada e ir rellenando el vector.

**Código 3: Leer un vector de la entrada estándar 1**

```
int numeroElementos;
cin >> numeroElementos;

vector<int> v(numeroElementos);
for(int i = 0; i < numeroElementos; i++) {
    // Forma 1
    int x;
    cin >> x;
    v[i] = x;

    // Forma 2
    cin >> v[i];
}
```

Otra formas menos recomendable, es inicializar un vector vacio e ir insertando elementos detrás por medio del método `push_back`. Si hacemos `v.push_back(elemento)` aumentaremos el tamaño del vector en uno e insertaremos el nuevo elemento al final.

**Código 4: Leer un vector de la entrada estándar 2**

```
    int numeroElementos;
    cin >> numeroElementos;

    vector<int> v;
    for(int i = 0; i < numeroElementos; i++) {
        int x;
        cin >> x;
        v.push_back(x);
    }
```

## 2.3 Insertar elementos

Existen dos opciones para insertar nuevos elementos dentro del vector. La primera opción es usar `push_back`, este método es muy rápido (tiene un coste amortizado de $\mathcal{O}(1)$[1]). Se usa así `miVector.push_back(elemento)` . Lo que hace es añadir el elemento al final del vector.

El segundo método, es el método `insert` que nos permite seleccionar en que posición colocar el nuevo elemento, antes de ser colocado todos los elementos a la derecha se ruedan a una posición para poder hacer hueco al nuevo elemento. Este método es mucho más lento sobretodo cuando la posición sea cercana a 0 (tiene un coste de $\mathcal{O}(n)$[1]). Además hace uso de operadores, los cuales no hemos comentado. Se usa así `miVector.insert(iteradorEnLaPosicion, elemento)`. Para determinar la posición del iterador vamos a usar `miVector.begin()`, que nos da un iterador al comienzo del vector, y le sumamos la posición donde queremos insertar el elemento de la siguiente manera `miVector.insert(miVector.begin() + posición, elemento)`.

Si no entiendes las operaciones con los iteradores no importa mucho ya que estas operaciones son muy poco frecuentes, sobretodo en vectores.

[1]No es necesario saber lo que es el costo de un algoritmo, solo que cuanto más rápido crezca la función de coste peor algoritmo será.

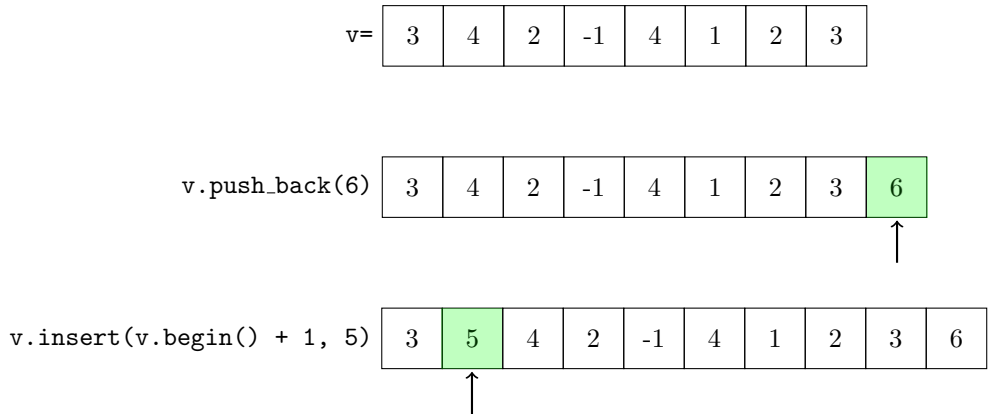



**Figura 4**: Ejemplo de inserción de elementos con `push_back` e `insert`

**Código 5: Inserción de elementos**

```
int n = 4;
vector<int> v(n);
for(int i = 0; i < n; i++) {
    v[i] = i;
}
// v = {0, 1, 2, 3};

v.push_back(6); // v = {0, 1, 2, 3, 6};

auto it = v.begin() + 1;
v.insert(it, -1); // v = {0, -1, 1, 2, 3, 6};
```

## 2.4 Eliminar elementos

La eliminación de elementos aunque no es tan común en los problemas se puede hacer a través de dos métodos nuevamente `pop_back` y `erase`. El método `pop_back` permite la eliminación del último elemento del vector, es el opuesto del método `push_back`. Este método es bastante rápido (tiene un coste amortizado de $\mathcal{O}(1)$). Se usa así `miVector.pop_back()`. El método `erase` es un método es más complejo ya que hace uso de iteradores de la misma manera que lo hacía `insert`. Se usa así `miVector.erase(iteradorEnLaPosicion)`, si tenemos la posición a eliminar podemos hacer lo siguiente `miVector.erase(miVector.begin() + posicionEliminar)`. Al igual que `insert` es más lento conforme la posición a borrar sea más cercana a 0, (tiene un coste de $\mathcal{O}(n)$).

**Código 6: Eliminación de elementos**

```
int n = 6;
vector<int> v(n);
for(int i = 0; i < 2 * n; i += 2) {
    v[i] = i;
}
// v = {0, 2, 4, 6, 8, 10};

v.pop_back(); // v = {0, 2, 4, 6, 8};

v.erase(v.begin() + 2); // v = {0, 2, 6, 8};
```

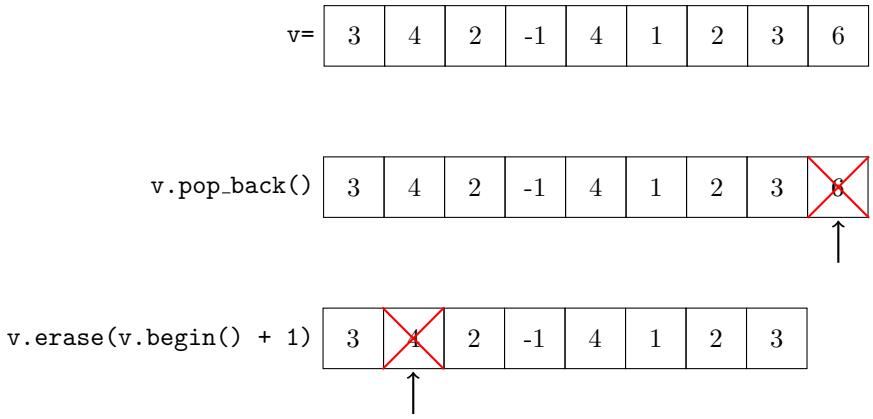



**Figura 5**: Ejemplo de inserción de elementos con `pop_back` e `erase`

## 2.5 Vectores multidimensionales

Como dijimos antes los vectores pueden contener casi cualquier tipo de elemento. Entre este los tipos que pueden se encuentran ¡otros vectores! Podemos crear vectores que contengan vectores y así sucesivamente. Lo general en los problemas es que como mucho necesitemos insertar un vector de vectores de enteros por ejemplo, el tipo de este vector sería `vector<vector<int>>` por ejemplo. Esto es común cuando estamos representado matrices o grafos (lo veremos a lo largo del grupo).

Como son vectores podemos hacer con ellos todas las operaciones que hemos mencionado antes ya sea en el vector externo como en cada uno de los vectores internos.

Si queremos acceder a los elementos de los vectores internos usaremos dos veces el operador []. Con `miVector[y]` obtendremos el un vector de tipo `vector<tipoElemento>` y con un segundo [] de la siguiente manera `miVector[y][x]` el elemento de tipo `tipoElemento` que se situa en la posición `x` del vector `y`.

Cuando usamos vectores bidimensionales es preferible iterar sobre ellos primero recorriendo el vector interno antes de pasar a recorrer el siguiente vector. Si pensamos en una matriz (no es necesario que todas las filas tengan la misma longitud) a la que accedemos de la siguiente manera `matriz[fila][columna]` es preferible recorrer todas las columnas antes de recorrer la siguiente fila

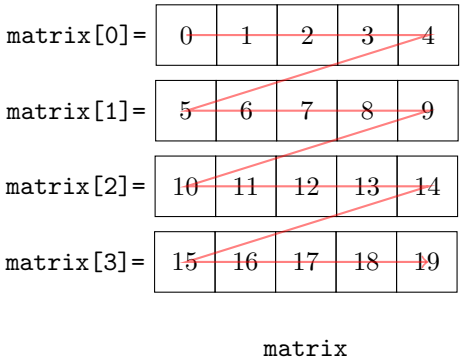



**Figura 6**: Recorrido primero por columnas y luego por filas en un `vector<vector<int>>`

**Código 7: Ejemplo de Vectores multidimensionales**

```
// Creamos un vector de vectores vacios
vector<vector<int>> matriz;
int n = 4;
for(int fila = 0; fila < n; fila++) {
    matriz.push_back(vector<int>());
    for(int columna = 0; columna < n; columna++) {
        matriz[fila].push_back(columna + fila * n);
    }
}

// Segunda forma de crear una matriz
// Ya hemos declarado la variable antes
// si no seria vector<vector<int>> matriz(n, vector<int>(n));
matriz = vector<vector<int>>(n, vector<int>(n));
for(int fila = 0; fila < n; fila++) {
    for(int columna = 0; columna < n; columna++) {matrix[1]=
        matriz[fila][columna] = columna + fila * n;
    }
}

// Imprimos por pantalla la matriz
for(int fila = 0; fila < n; fila++) {
    for(int columna = 0; columna < n; columna++) {
        cout << matriz[fila][columna] << " ";
    }
    cout << endl;
}

/*
    0 1 2 3
    4 5 6 7
    8 9 10 11
    12 13 14 15
*/
```

## 2.6 Ordenación de vectores

En muchos problemas es necesario ordenar el vector previamente como parte del algoritmo para resolverlo. En FP se ven algoritmos de ordenación, sin embargo, son muy lentos (tienen complejidad $\mathcal{O}(n^2)$), la manera más rápida y relativamente eficiente (tiene coste $\mathcal{O}(n \log(n))$) de ordenar un vector en un concurso es por medio de la función `sort`. Le deberemos pasar el rango en el que queremos que ordene el vector por medio de iteradores de la siguiente manera `sort(miVector.begin(), miVector.end())` y nos ordenará el vector de menor a mayor. Es importantes importar `algorithm` de la siguiente manera `#include <algorithm>` para usar `sort`.

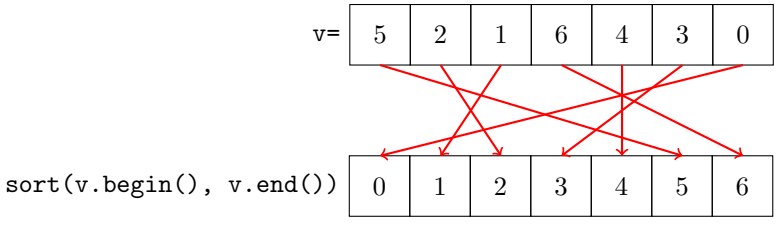



**Figura 7**: Ordenación de un vector de enteros por medio de `sort`

Muchas veces nos interesa ordenar el vector de otra manera, por ejemplo de mayor a menor, para ello existe la función `greater<tipoElemento>()` que usaremos de la siguiente manera `sort(miVector.begin(), miVector.end(), greater<tipoElemento>())`. También podemos pasarle a sort nuestras propias funciones para ordenar siguiendo otros criterios siempre que sean

de la forma `bool miFuncion(tipoElemento a, tipoElemento b)`. Donde `miFuncion` devuelve `true` si queremos que el elemento `a` se encuentre antes que `b`. Pasar una función de ordenación propia es especialmente útil cuando el vector no sea de enteros si no de pares o de algun tipo de dato que hemos creado nosotros, por ejemplo un intervalo, una esfera o lo que nos encontremos en los problemas.

**Código 8: Ejemplo de ordenación**

```cpp
// Los pares van primeros y luego ordenamos de mayor a menor
bool cmp(int a, int b) {
    return a % 2 == 0 && b % 2 == 1 || a > b;
}

int main() {
    vector<int> v {5, 2, 1, 6, 4, 3, 0};
    int n = 4;

    sort(v.begin(), v.end(), greater<int>());

    // 6 5 4 3 2 1 0
    for(int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;

    sort(v.begin(), v.end(), cmp);

    // 6 4 2 0 5 3 1
    for(int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
}
```

## 2.7 Operaciones en vectores ordenados

En comun en muchos problemas el preguntar el menor o mayor elemento que cumple cierta condición para ayudarnos a resolver esos problemas en algunos casos podemos hacer uso de las funciones `lower_bound` y `upper_bound`. Estas funciones operan sobre vectores ordenados y son muy rápidas (complejidad $\mathcal{O}(\log n)$). Necesitan de la librería `algorithm`.

`lower_bound` busca en la lista el **menor elemento que es mayor o igual a un elemento dado** y devuelve un iterador apuntando a la posición del elemento encontrado. Si ningún elemento cumple esa condición devuelve `miVector.end()`. Tambien podemos pasarle una función de comparación propia como en el caso del `sort`.

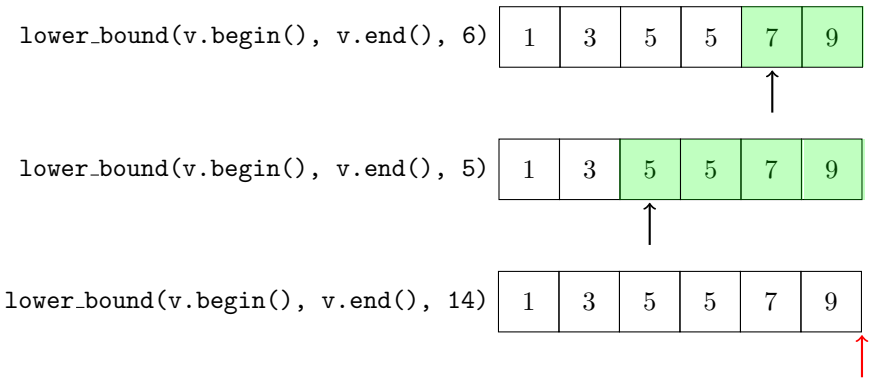



**Figura 8**: Ejemplo de `lower_bound`

`upper_bound` busca en la lista el **menor elemento que es mayor a un elemento dado** y devuelve un iterador apuntando a la posición del elemento encontrado. Si ningún elemento

cumple esa condición devuelve `miVector.end()`. Tambien podemos pasarle una función de comparación propia como en el caso del `sort`.

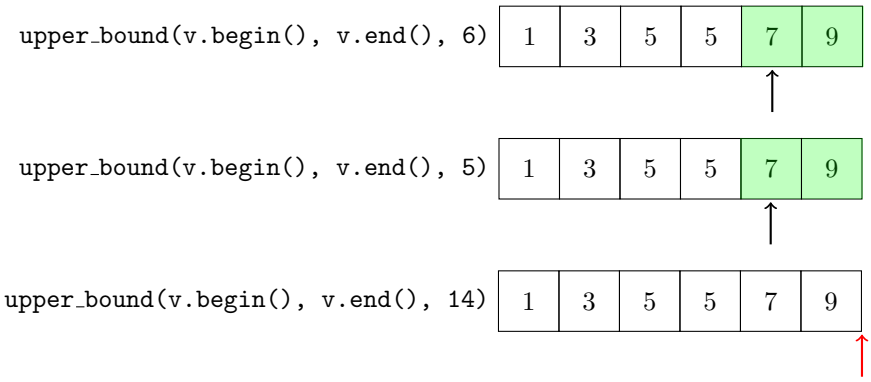



**Figura 9**: Ejemplo de `upper_bound`

**Código 9: Ejemplo de `lower_bound` y `upper_bound`**

```
vector<int> v {10, 10, 10, 20, 20, 20, 30, 30, 30};

auto lower = lower_bound (v.begin(), v.end(), 20);
auto upper = upper_bound (v.begin(), v.end(), 20);
int indiceLower = lower - v.begin();
int indiceUpper = upper - v.begin();

int elementoLower, elementoUpper;
if(lower != v.end())elementoLower = *lower;
else elementoLower = -1;
if(upper != v.end())elementoUpper = *upper;
else elementoUpper = -1;
cout << "lower: indice = " << indiceLower << ", elem = " << elementoLower
    << endl;
cout << "upper: indice = " << indiceUpper << ", elem = " << elementoUpper
    << endl;
/*10, 10, 10, 20, 20, 20, 30, 30, 30
               |           |
            lower       upper
    lower_bound: indice = 3, elemento = 20
    upper_bound: indice = 6, elemento = 30    */

lower = lower_bound (v.begin(), v.end(), 30);
upper = upper_bound (v.begin(), v.end(), 30);
indiceLower = lower - v.begin();
indiceUpper = upper - v.begin();

if(lower != v.end()) elementoLower = *lower;
else elementoLower = -1;
if(upper != v.end()) elementoUpper = *upper;
else elementoUpper = -1;
cout << "lower: indice = " << indiceLower << ", elem = " << elementoLower
    << endl;
cout << "upper: indice = " << indiceUpper << ", elem = " << elementoUpper
    << endl;
/*  10, 10, 10, 20, 20, 20, 30, 30, 30
                           |          |
                           lower      upper
    lower_bound: indice = 6, elemento = 30
    upper_bound: indice = 9, elemento = -1    */
```

## 2.8 Invertir un vector

Este operación se encuentra dentro de la librería `algorithm` y nos permite invertir el orden de un vector de la siguiente manera `reverse(miVector.begin(), miVector.end())`, de esta manera `miVector` queda invertido.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| v= | 3 | 4 | 2 | -1 | 4 | 1 | 2 | 3 |
| `reverse(v.begin(), v.end())` | 3 | 2 | 1 | 4 | -1 | 2 | 4 | 3 |

**Figura 10**: Ejemplo de operación `reverse`

## 2.9 Iterar sobre un vector

Conocemos la manera típica de iterar sobre un vector por medio del bucle `for`.

**Código 10: Iteración con `for` con indices**

```
vector<int> v {3, 4, 2, -1, 4, 1, 2, 3};
for(int i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
cout << endl; // 3 4 2 -1 4 1 2 3
```

Sin embargo, existe una sintaxis que nos permite hacerlo escribiendo menos y es la siguiente `for(tipoElemento elemento : miVector)` {}. De esta manera obtenemos directamente el elemento en vez del indice del elemento.

Esta notación nos permitirá iterar sobre otras estructuras donde no tenemos acceso directo a los elementos por medio de indices.

**Código 11: Iteración con `for` con iteradores**

```
vector<int> v {3, 4, 2, -1, 4, 1, 2, 3};
for(int i : v) {
    cout << i << " ";
}
cout << endl; // 3 4 2 -1 4 1 2 3
```

Finalmente en muchos problemas será necesario imprimir un vector como solución. El problema es que muchos jueces son muy rigurosos con la salida y no es posible insertar espacios al final de la salida como hemos hecho antes ya que nos dará Wrong Answer o Presentation Error. Por ello la forma más recomendable de imprimir un vector para dar una solución es:

**Código 12: Imprimir un vector como solución en un juez**

```
vector<int> v {3, 4, 2, -1, 4, 1, 2, 3};
for(int i = 0; i < v.size(); i++) {
    if (i > 0) cout << " ";
    cout << v[i];
}
cout << endl;
```