

C++ Standard Template Library ¹

Gustavo Rivas Gervilla

University of Granada



**UNIVERSIDAD
DE GRANADA**

¹Basado en el Topic J del curso

Contenido

- 1 Introducción
- 2 Contenedores
- 3 Algoritmos

Standard Template Library (STL)

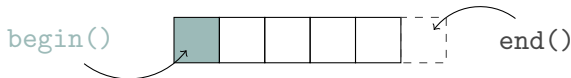
- En la STL se definen componentes basados en plantillas, reusables y de una gran versatilidad.
 - Los cuales implementan estructuras de datos y algoritmos comunes.
- Hace uso del paradigma de la programación genérica (*generic programming*) a través de plantillas.
- La podríamos dividir en tres componentes distintos:
 - **Contenedores:** estructuras de datos que almacenan objetos de cualquier tipo.
 - **Iteradores:** son las herramientas con las que se manipulan los objetos de los contenedores.
 - **Algoritmos:** distintos algoritmos de búsqueda, ordenación u otros, sobre los contenedores.

Contenedores

- Podemos distinguir tres tipos distintos de contenedores:
 - **Contenedores Secuenciales:** estructuras de datos lineales como *vectores* o *listas enlazadas*.
 - **Contenedores Asociativos:** estructuras no-lineales como las *tablas hash*.
 - **Adaptadores de Contenedores:** estructuras lineales sobre las cuales se definen ciertas restricciones como las *pilas* o las *colas*.
- Los contenedores secuenciales y asociativos suelen llamarse **contenedores de primer nivel**.

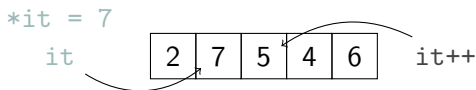
Iteradores

- Los iteradores son punteros a elementos de contenedores de primer nivel. Se pueden distinguir dos tipos:
 - `const-iterator`: define un iterador sobre un contenedor cuyos objetos no son modificables o no queremos modificar.
 - `iterator`: define un iterador sobre un contenedor sobre cuyos elementos no tenemos restricciones de escritura.
- Todos los contenedores de primer nivel proporcionan los métodos `begin()` y `end()`, las cuales devuelven, respectivamente, un iterador que apunta al primer elemento de la estructura o a un elemento después del último elemento de la estructura.



Iteradores

- Si el iterador `it` apunta a un determinado elemento, entonces
 - `it++` (o `++it`) apunta al elemento siguiente y
 - `*it` devuelve el valor del elemento apuntado por `it`



- El iterador que devuelve el método `end()` sólo se puede usar para comprobar si el iterador ha alcanzado el final de la estructura.
- En las próximas transparencias veremos cómo se usan `begin()` y `end()`.

Contenido

- 1 Introducción
- 2 Contenedores
- 3 Algoritmos

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Contenedores Secuenciales

En la STL podemos encontrar tres tipos de contenedores secuenciales:

- `vector`: basados en arrays.
- `deque` (double-ended queue): basadas en arrays.
- `list`: basadas en listas enlazadas.

Contenedores Secuenciales: vector

- La implementación de un vector está basada en arrays.
- Los vectors permiten el acceso directo a sus elementos a través de índices.
- Insertar un elemento al final del vector es normalmente eficiente, el vector simplemente crece (*a menos que haya que reservar nueva memoria debido a este crecimiento*).
- En cambio insertar o eliminar un elemento en el medio del vector resulta computacionalmente caro ya que hay que mover una porción completa del vector.

- Un vector posee un tamaño (`size`) que es el número de elementos que contiene, así como una capacidad (`capacity`) que es el número de elementos que puede llegar a alojar según la memoria que se le ha reservado.
- Cuando completamos esta capacidad entonces:
 - Se reserva memoria para un vector mayor.
 - Se copian los elementos del vector en esa nueva porción de memoria.
 - Se libera la porción de memoria anteriormente ocupada por el vector.
- Para usar los `vectors` hemos de incluir la cabecera `<vector>`.
- Aquí puedes ver los métodos que incluye esta clase.

Ejemplo de uso de la clase vector

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main(){
7      vector<int> v;
8
9      //Agregamos elementos al final del vector.
10     v.push_back(2);
11     v.push_back(3);
12     v.push_back(4);
13
14     //Mostramos el tamaño y la capacidad del vector.
15     cout << "\nEl tamaño del vector es: " << v.size()
16          << "\ny su capacidad: " << v.capacity();
17
18     //Mostramos el contenido del vector.
19     vector<int>::const_iterator it;
20
21     for (it = v.begin(); it != v.end(); it++){
22         cout << *it << endl;
23     }
24
25     return 0;
26 }
```

Contenedores Secuenciales: `list`

- Una `list` se implementa por medio de una lista doblemente enlazada.
- Las inserciones y borrados son muy eficientes en cualquier posición de la lista.
 - Sin embargo primero hay que acceder a un elemento de la lista, lo cual no es tan eficiente como recorrer un vector.
- Los iteradores bidireccionales pueden usarse para recorrer la lista en ambas direcciones.
- Para usarlas hemos de incluir la cabecera `<list>`.
- Aquí puedes ver los métodos que incluye esta clase.

Contenedores Secuenciales: deque

- deque combina los beneficios de vector y list.
- Proporciona acceso a sus elemento usando índices (algo que no es posible usando list).
- Además proporciona inserciones eficientes al comienzo (lo que no es eficiente usando vector) y al final de la estructura.

Contenedores Secuenciales: deque

- El espacio adicional para una deque se reserva usando bloques de memoria (no necesariamente contiguos).
 - Se mantiene un array de punteros a dichos bloques.
- Además de los métodos que posee un vector, deque proporciona los métodos `push_front` y `pop_front` para insertar y eliminar un elemento al comienzo de la estructura.
- Aquí puedes ver los métodos que incluye esta clase.

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Contenedores Asociativos

- Los contenedores asociativos usan claves (*keys*) para almacenar y devolver elementos.
- Hay cuatro tipo de contenedores asociativos en la STL: `multiset`, `set`, `multimap` y `map`:
 - **Todos** mantienen las claves ordenadas.
 - **Todos** son compatibles con iteradores bidireccionales.
 - `set` no permite claves repetidas.
 - `multiset` y `multimap` sí permiten que se repitan claves.
 - `multimap` y `map` permiten asociar claves a valores.

Contenedores Asociativos: `multiset`

- Los `multiset` se implementan usando un árbol binario de búsqueda rojo-negro para almacenar y devolver claves de un modo eficiente.
- Para usarlos hemos de incluir la cabecera `<set>`.
- `multiset` permite que se repitan claves.
- El orden de las claves viene determinado por la función de comparación de objetos `less<T>` de la STL.
- Por lo tanto las claves que se usen han de ser compatibles con el operador de comparación `<`.
- Aquí puedes ver los métodos que incluye esta clase.

Ejemplo de uso de la clase multiset

```
1 | #include <iostream>
2 | #include <set>
3 |
4 | using namespace std;
5 |
6 | int main(){
7 |
8 |     multiset<int, less<int>> ms;
9 |     ms.insert(10); //Insertamos 10.
10 |    ms.insert(35); //Insertamos 35.
11 |    ms.insert(10); //Insertamos 10 de nuevo (permitido).
12 |
13 |    cout << "Hay " << ms.count(10) << " elementos igual a 10." << endl;
14 |
15 |    multiset<int, less<int>>::iterator it; //Creamos un iterador.
16 |
17 |    it = ms.find(10);
18 |
19 |    if (it != ms.end()) //Si el iterador no ha llegado al final del conjunto.
20 |        cout << "Se ha encontrado un 10." << endl;
21 |
22 |    return 0;
23 | }
```

Contenedores Asociativos: set

- Un `set` es idéntico a un `multiset` salvo porque no permite el uso de claves repetidas.
- Para usarlos hemos de incluir la cabecera `<set>`.
- Aquí puedes ver los métodos que incluye esta clase.

Contenedores Asociativos: `multimap`

- Los `multimap` asocian claves a valores.
- Se implementan mediante un árbol binario de búsqueda rojo-negro para el almacenamiento y devolución eficiente de claves y valores.
- Las inserciones se realizan usando objetos de la clase `pair` (con una clave y un valor).
- Un `multimap` permite que se dupliquen claves, es decir, varios valores pueden asociarse a la misma clave.
- El orden de las claves se determina por medio de la función `less<T>` de la STL.
- Para usarlos hemos de incluir la cabecera `<map>`.
- Aquí puedes ver los métodos que incluye esta clase.

Ejemplo de uso de la clase multimap

```
1  #include <iostream>
2  #include <map>
3
4  using namespace std;
5
6  typedef multimap<int, double, std::less<int>> mp_type; //Creamos un tipo de multimap.
7
8  int main(){
9      mp_type mp;
10
11      //Se sobrecargar el metodo value_type de la clase multimap para crear objetos de la clase pair.
12      mp.insert(mp_type::value_type(10, 14.5));
13      mp.insert(mp_type::value_type(10, 18.5)); //Permitido.
14
15      cout << "Hay " << mp.count(15) << "\n";
16
17      //Usamos un iterador para recorrer mp.
18      for (mp_type::iterator it = mp.begin(); it != mp.end(); it++){
19          cout << it->first << "\t" << it->second << "\n";
20      }
21
22      return 0;
23 }
```

Contenedores Asociativos: map

- Al igual que `multimap`, se implementan por medio de un árbol binario de búsqueda rojo-negro.
- Sin embargo no permiten que se dupliquen claves.
- Para usarlos hemos de incluir la cabecera `<map>`.
- Aquí puedes ver los métodos que incluye esta clase.

Contenedores Asociativos: map

- La clase `map` sobrecarga el operador `[]` para permitir el acceso a los elementos de un modo flexible. Como se muestra en el siguiente ejemplo:

```
1 | #include <iostream>
2 | #include <map>
3 |
4 | using namespace std;
5 |
6 | int main(){
7 |     map<int, double, less<int>> map_obj;
8 |
9 |     /*Ahora asociamos el valor 125.25 a la clave 20.
10 |      Si en el map_obj no existe la clave 20 entonces
11 |      se crea un nuevo par con clave = 20 y valor = 125.25.*/
12 |
13 |     map_obj[20] = 125.25;
14 |
15 |     return 0;
16 | }
```


Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

- En la STL podemos encontrar tres tipos de adaptadores de contenedores:
 - `stack`
 - `queue`
 - `priority_queue`
- Se implementan usando los contenedores que hemos visto anteriormente, con lo cual no proporcionan una estructura de datos nueva. Sólo imponen restricciones sobre una estructura de datos existente.
- No son compatibles con el uso de iteradores.
- Los métodos `push` y `pop` son comunes a todos los adaptadores de contenedores.

Adaptadores de Contenedores: `stack`

- El acceso a los elementos de la estructura se realiza mediante una política `last-in-first-out` (LIFO, el último elemento introducido en la estructura es el primero en salir).



- Para usarlas hemos de incluir la cabecera `<stack>`.

Adaptadores de Contenedores: stack

- Pueden implementarse sobre vector, list o deque (que es la opción por defecto):

```
1 | #include <stack>
2 | #include <vector>
3 | #include <list>
4 |
5 | using namespace std;
6 |
7 | int main(){
8 |     stack<int, vector<int>> s1; //Una pila usando vector.
9 |     stack<int, list<int>> s2; //Una pila usando list.
10 |     stack<int> s3; //Una pila usando deque.
11 |
12 |     return 0;
13 | }
```

- Aquí puedes ver los métodos que incluye esta clase.

Adaptadores de Contenedores: queue

- El acceso a los elementos de la estructura se realiza mediante una política `first-in-first-out` (FIFO, el primer elemento introducido en la estructura es el primero en salir).



- Para usarlas hemos de incluir la cabecera `<queue>`.

Adaptadores de Contenedores: queue

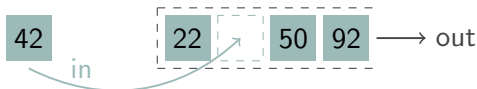
- Pueden implementarse sobre list o deque (que es la opción por defecto):

```
1 | #include <queue>
2 | #include <list>
3 |
4 | using namespace std;
5 |
6 | int main(){
7 |     queue<int, list<int>>> q1; //Una queue usando list.
8 |     queue<int> q2; //Una queue usando deque.
9 |
10 |     return 0;
11 | }
```

- Aquí puedes ver los métodos que incluye esta clase.

Adaptadores de Contenedores: `priority_queue`

- Los elementos se insertan de modo que queden ordenados por su prioridad, de mayor a menor prioridad, desde el comienzo hasta el final de la `priority_queue`.
- Siendo el elemento que está al comienzo, al igual que en una queue, el primero en salir. Con lo que el primer elemento en salir es el de mayor prioridad (por defecto se usa `less<T>` para comparar elementos).



Adaptadores de Contenedores: `priority_queue`

- Pueden implementarse sobre `vector` (que es la opción por defecto) o `deque`.
- Para usarlas hemos de incluir la cabecera `<queue>`.
- **No se debe confundir** la prioridad de un elemento con el valor de éste. En el caso por defecto (mostrado en la imagen) sí es así. Pero podríamos definir otros comparadores y dar mayor prioridad a los elementos menores, o establecer otro criterio. Prueba el código de la siguiente diapositiva.
- Aquí puedes ver los métodos que incluye esta clase.

Adaptadores de Contenedores: priority_queue

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4
5  using namespace std;
6
7  int main(){
8      float elems[] = {12.3,4.11,-6.8,20.5};
9
10     //Creamos una priority_queue que da mayor prioridad a los elementos mayores.
11     priority_queue<float, deque<float>> qdefault (elems, elems + 4);
12     /*Creamos otra especificando un comparador distinto al de por defecto.
13      En esta ocasion usamos el objeto funcional greater de la STL
14      (puedes definir los tuyos propios).*/
15     priority_queue<float, vector<float>, greater<float>> qless (elems, elems + 4);
16
17     while(!qdefault.empty()){
18         cout << qdefault.top() << "\t";
19         qdefault.pop();
20     }
21     cout << endl;
22     while(!qless.empty()){
23         cout << qless.top() << "\t";
24         qless.pop();
25     }
26     return 0;
27 }
```

Contenido

- 1 Introducción
- 2 Contenedores
- 3 Algoritmos

Algoritmos

- La STL separa contenedores y algoritmos.
 - El principal beneficio es que se evita llamadas a funciones virtuales.
 - Esto no puede hacerse por ejemplo en Java o C# ya que no tienen mecanismos flexibles para trabajar con objetos funcionales. En cambio Smalltalk sí que da esta posibilidad.
- En las siguientes diapositivas se describen los algoritmos más comunes de la STL. Puedes pulsar sobre la cabecera de cada algoritmo para acceder a una documentación más detallada.

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- **fill y generate**
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- copy y merge
- unique y reverse
- Utilidades
- Conjuntistas

Algoritmos: fill y generate

- `void fill(first, last, value)`: introduce el valor `value` en los elementos dentro del rango `[first, last)` definido por los iteradores `first` y `last`. Es decir, se incluye el primer elemento del rango, el apuntado por `first`, y no se incluye el apuntado por `last`.
- `iterator fill_n(first, n, value)`: introduce el valor `value` en `n` elementos consecutivos a partir del elemento al que apunta `first`. Devuelve un iterador apuntando al elemento siguiente al último en el que se ha introducido `value`.

Algoritmos: fill y generate

- `void generate(first, last, function)`: similar a `fill` pero en esta ocasión se invoca la función `function` para devolver el valor a introducir.
- `iterator generate_n(first, n, function)`: similar a `fill_n` pero en esta ocasión se invoca la función `function` para devolver el valor a introducir.

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: para comparar secuencias de valores

- `bool equal(first, last, result)`: comprueba si la secuencia `[first,last)`, es igual a la que comienza en `result` (y que tiene, si es posible, el mismo número de elementos). Si no hay suficientes elementos en la segunda secuencia entonces devolverá `false`. Puedes probar el código de la siguiente diapositiva para comprobarlo.
- `pair mismatch(first, last, result)`: realiza la misma comparación que `equal` pero en esta ocasión devuelve un par de iteradores apuntando a los objetos donde las secuencias dejan de ser iguales. Si las secuencias son iguales entonces un iterador será igual a `last` y el otro a un iterador que apunte a la misma posición relativa en la segunda secuencia.

Algoritmos: para comparar secuencias de valores

```
1  | #include <iostream>
2  | #include <algorithm>
3  | #include <vector>
4  |
5  | using namespace std;
6  |
7  | int main(){
8  |     int myints[] = {20,40,60,80,100};
9  |
10 |     vector<int> v1 (myints, myints + 5);
11 |     vector<int> v2 (myints, myints + 4);
12 |
13 |     if (equal(v1.begin(), v2.end(), v2.begin()))
14 |         cout << "Son iguales.";
15 |     else
16 |         cout << "No son iguales.";
17 | }
```

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: para eliminar elementos de contenedores

- `iterator remove(first, last, value):`
 - Elimina todas las ocurrencias de `value` de en el rango `[first,last)`.
 - No elimina estos elementos físicamente.
 - Lo que hace es mover el resto de elementos del contenedor hacia delante. Esto produce que el contenedor tenga el mismo número de elementos, y aquellas posiciones del vector más allá del último elemento desplazado quedarán con el mismo valor que tenían. Puedes comprobarlo con el código de la siguiente diapositiva.
 - Devuelve un iterador apuntando al nuevo último elemento de la secuencia.

Algoritmos: para eliminar elementos de contenedores

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  int main(){
7      int myints[] = {20,60,20,20,20,42,80,100};
8      vector<int> v1 (myints, myints + 8);
9
10     //Mostramos el vector original.
11     for (vector<int>::iterator i = v1.begin(); i != v1.end(); i++)
12         cout << *i << "\t";
13     cout << endl;
14
15     vector<int>::iterator newEnd = remove(v1.begin(), v1.end(), 20); //Eliminamos los elementos
        igual a 20.
16
17     //Recorremos los elementos hasta donde nos indica el puntero devuelto.
18     for (vector<int>::iterator i = v1.begin(); i != newEnd; i++)
19         cout << *i << "\t";
20     cout << endl;
21     //Vemos todos los elementos que realmente contiene el vector.
22     for (vector<int>::iterator i = v1.begin(); i != v1.end(); i++)
23         cout << *i << "\t";
24
25     return 0;
26 }
```

Algoritmos: para eliminar elementos de contenedores

- `iterator remove_copy(first, last, result, value):`
 - Copia los elementos del intervalo `[first, last)` que no son iguales a `value` a la secuencia que comienza en `result`.
 - Devuelve un iterador que apunta al final de la secuencia que comienza en `result`.

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: para reemplazar elementos de contenedores

- `void replace(first, last, value, newvalue):`
reemplaza el valor `value` por `newvalue` en los elementos en el rango `[first, last)`.
- `void replace_if(first, last, function, newvalue):`
opera igual que `replace` pero en esta ocasión para aquellos elementos sobre los cuales `function` devuelve `true`.

Algoritmos: para reemplazar elementos de contenedores

- `iterator replace_copy(first, last, result, value, newvalue)`: copia los elementos en `[first, last)` en la secuencia que empieza en `result`, cambiando el valor de los elementos que contienen `value` por `newvalue`. Devuelve un iterador apuntando al elemento siguiente al último introducido en la nueva secuencia.
- `iterator replace_copy_if(first, last, result, function, newvalue)`: hace lo análogo en base al valor booleano devuelto por `function`.

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: algoritmos de búsqueda

- `iterator find(first, last, value)`: devuelve un iterador que apunta a la primera ocurrencia de `value` en `[first, last)`. O un iterador igual a `end()` en caso de no encontrar `value`.
- `iterator find_if(first, last, function)`: hace lo análogo según el valor booleano devuelto por `function` sobre los elementos de la secuencia.

Algoritmos: algoritmos de búsqueda

- `bool binary_search(first, last, value[, criterion])`: comprueba si existe un elemento igual a `value` en una secuencia ordenada en orden creciente definida entre `first` e `last`. También podemos especificar otro criterio de comparación para la búsqueda binaria, a través de un cuarto argumento que será una función de comparación. Entonces el rango de valores tendrá que estar ordenado por ese mismo criterio.

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: algoritmos de ordenación

- `void sort(first, last[, criterion])`: ordena los elementos en `[first, last)` en orden creciente. Como se puede ver en la documentación que se enlaza, también podemos definir otro orden distinto pasando, como tercer argumento, una función de ordenación. Puedes ver un ejemplo en la siguiente diapositiva, donde definimos un criterio que dispone en primer lugar los elementos pares, y ordena los elementos de mayor a menor dentro de los elementos pares e impares.

Algoritmos: algoritmos de ordenación

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  bool criterion (int elem1, int elem2){
7      if (elem1 % 2 == 0 && elem2 % 2 != 0) //Si elem1 es par y elem2 es impar entonces elem1 va
          primero.
8          return true;
9      else if (elem1 % 2 != 0 && elem2 % 2 == 0) //A la inversa es elem2 el que va primero.
10         return false;
11     else //Si tienen la misma paridad ordenamos segun el valor de mayor a menor.
12         return elem1 > elem2;
13 }
14
15 int main(){
16     int elems[] = {3,6,2,7,10,15,1};
17     vector<int> v(elems, elems + sizeof(elems) / sizeof(int));
18
19     sort(v.begin(), v.end(), criterion);
20
21     cout << "\nOrdenado con nuestro criterio:\n";
22     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
23         cout << *it << "\t";
24
25     return 0;
26 }
```

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: copy y merge

- `iterator copy(first, last, it3)`: copia el rango de valores `[first, last)` en la secuencia que comienza en `it3`. Devuelve un iterador al final de la secuencia donde se han copiado los elementos.
- `iterator copy_backward(first, last, it3)`: en este caso se copian los elementos en orden inverso.
- `iterator merge(first1, last1, first2, last2, result[, criterion])`: los rangos `[first1, last1)` e `[first2, last2)` han de estar ordenados en orden ascendente (u otro criterio que pasemos con sexto argumento al método en forma de función de comparación). Entonces se copian los elementos en la secuencia que comienza en `result` según el criterio establecido.

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: unique y reverse

- `iterator unique(first, last[, criterion])`: elimina los elementos repetidos del intervalo `[first, last)`, devolviendo un puntero al nuevo final de la secuencia. Podemos pasar un tercer argumento que indique un criterio distinto a la igualdad clásica para determinar si dos elementos son equivalentes.
- `void reverse(first, last)`: invierte el orden de los elementos en el rango `[first, last)`.

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos

- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: utilidades

- `void random_shuffle(first, last)`: mezcla de forma aleatoria los elementos del intervalo `[first, last)`.
- `int count(first, last, value)`: devuelve el número de ocurrencias de `value` en el rango `[first, last)`. Realmente el objeto devuelto es de tipo `iterator_traits<InputIterator>::difference_type` que es un tipo de entero con signo.
- `int count_if(first, last, function)`: devuelve el número de elementos en el rango `[first, last)` sobre los cuales `function` devuelve `true`.

Algoritmos: utilidades

- `iterator min_element(first, last[, criterion])`: devuelve un iterador apuntando al menor elemento en la secuencia en el rango `[first e last)`. Como es ya de esperar, se puede pasar una función de comparación para establecer otro criterio con el que comparar elementos distinto de `<`.
- `iterator max_element(first, last[, criterion])`: hace lo análogo para el mayor elemento de la secuencia.

Algoritmos: utilidades

- `T accumulate(first, last, init[, binaryOp])`: devuelve la suma acumulada de los elementos en el rango `[first, last)`. La suma se inicializa al valor `init`. Para usarlo hay que incluir la cabecera `<numeric>` en lugar de `<algorithm>`. Además, el tipo de los datos que sumemos, `T`, tendrá que soportar el operador `+`. O bien podemos pasar como tercer argumento una función binaria que devuelva, a partir de dos elementos, un nuevo elemento con un tipo compatible con `T`.
- `Function for_each(first, last, function)`: invoca `function` sobre cada uno de los elementos de `[first, last)`.

- `iterator transform(first, last, result, function):`
invoca la función `function` sobre cada uno de los elementos del rango `[first, last)` y los copia en la secuencia que comienza en `result`. Devuelve un iterador apuntando al elemento después del último elemento insertado en la secuencia.

En la siguientes diapositivas tienes un ejemplo de uso de `accumulate`, `for_each` y `transform` de forma combinada.

Algoritmos: utilidades

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <numeric>
5
6  using namespace std;
7
8  int multiplicacion(int x, int y) {return x*y;}//devuelve el producto de dos numeros.
9
10 int mulVector (vector<int> v){//devuelve el producto de todos los elementos de un vector.
11     return accumulate(v.begin(), v.end(), 1, multiplicacion);
12 }
13
14 void print(int v){cout << v << endl;}//imprime el entero pasado como argumento.
15
16 int main(){
17     //Creamos un vector de vectores.
18     vector<vector<int>> datos;
19
20     /*En las siguiente lineas estamos usando que en C++11 podemos definir
21     vector<int> v = {1,2,3,4,5};
22     Si tu compilador no soporta esto usa la forma clasica para inicializar vectores.*/
23     datos.push_back({10,20,30,40,50});
24     datos.push_back({2,3,5,7});
25     datos.push_back({1,1,2,3,5,8});
26 }
```


Algoritmos: utilidades

```
27 | //Vector que almacena la multiplicacion de los elementos de cada vector.  
28 | vector<int> multiplicaciones;  
29 |  
30 | //Transform no reserva nueva memoria si es necesario, con lo que hemos de hacerlo a mano.  
31 | multiplicaciones.resize(datos.size());  
32 |  
33 | /*Obtenemos la multiplicacion de los elementos de cada vector y la copiamos  
34 |    a otro vector*/  
35 | transform(datos.begin(), datos.end(), multiplicaciones.begin(), mulVector);  
36 |  
37 |  
38 | //Imprimimos los resultados con for_each.  
39 | for_each(multiplicaciones.begin(), multiplicaciones.end(), print);  
40 |  
41 | return 0;  
42 | }
```

Contenido

1 Introducción

2 Contenedores

- Contenedores Secuenciales
- Contenedores Asociativos
- Adaptadores de Contenedores

3 Algoritmos




- `fill` y `generate`
- Para comparar secuencias de valores
- Para eliminar elementos
- Para reemplazar elementos
- Búsqueda
- Ordenación
- `copy` y `merge`
- `unique` y `reverse`
- Utilidades
- Conjuntistas

Algoritmos: algoritmos conjuntistas

- `bool includes(firstA, lastA, firstB, lastB[, criterion])`: comprueba si los valores de `[firstB, lastB)` están en `[firstA, lastA)`. Por defecto los elementos se comparan con `<`, pero podemos especificar otro criterio como quinto argumento. En cualquier caso ambos rangos tendrán que estar ordenados en base a ese criterio.
- `iterator set_difference(firstA, lastA, firstB, lastB, result[, criterion])`: devuelve la diferencia conjuntista `[firstA, lastA) \ [firstB, lastB)` en la secuencia que comienza en `result`. Y se aplican las mismas consideraciones para el criterio de comparación que en `includes`.



Algoritmos: algoritmos conjuntistas

- `iterator set_intersection(firstA, lastA, firstB, lastB, result[, criterion])`: opera igual que `set_difference` pero devolviendo la intersección de ambos rangos. 
- `iterator set_simmetric_difference(firstA, lastA, firstB, lastB, result[, criterion])`: opera igual que `set_difference`, en este caso devolviendo los elementos que no están en la intersección de ambos conjuntos. 
- `iterator set_union(firstA, lastA, firstB, lastB, result[, criterion])`: hace lo análogo para la unión de los conjuntos. 

Bibliografía:

- “C++, How to program”, Harvey M. Deitel, Paul J. Deitel, 4th Edition, Prentice Hall.
- “The C++ Programming Language”, Bjarne Stroustrup, 4th Edition, Addison-Wesley.