

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

Algoritmos

- *fill y generate*
- Comparar secuencias de valores
- Eliminar elementos
- Reemplazar elementos
- Búsqueda
- Ordenación
- *copy y merge*
- *unique y reverse*
- Utilidades
- Algoritmos 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*

<https://www.cplusplus.com/reference/algorithm/fill/>
https://www.cplusplus.com/reference/algorithm/fill_n/

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

<https://www.cplusplus.com/reference/algorithm/generate/>
https://www.cplusplus.com/reference/algorithm/generate_n/

Algoritmos: 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*
- *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

<https://www.cplusplus.com/reference/algorithm/equal/>

<https://www.cplusplus.com/reference/algorithm/mismatch/>

Algoritmos: 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(), v1.end(), v2.begin()))
14         cout << "Son iguales.";
15     else
16         cout << "No son iguales."; ←
17 }
```

<https://www.cplusplus.com/reference/algorithm/equal/>

<https://www.cplusplus.com/reference/algorithm/mismatch/>

Algoritmos: eliminar elementos

- *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
 - Devuelve un iterador apuntando al nuevo último elemento de la secuencia

<https://www.cplusplus.com/reference/algorithm/remove/>

Algoritmos: eliminar elementos

```
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";    //20 60 20 20 20 42 80 100
13     cout << endl;
14                                     // 20 20 20 20 42 80 100
15     //Eliminamos los elementos igual a 20 //20 20 20 20 42 80 100 100
16     vector<int>::iterator newEnd = remove(v1.begin(), v1.end(), 20);
17                                     // 20 20 20 20 42 80 100 100
18     //Recorremos los elementos hasta donde nos indica el puntero devuelto
19     for (vector<int>::iterator i = v1.begin(); i != newEnd; i++)
20         cout << *i << "\t";    // 60 42 80 100
21     cout << endl;
22                                     // 20 60 20 20 20 42 80
23     //Vemos todos los elementos que realmente contiene el vector.
24     for (vector<int>::iterator i = v1.begin(); i != v1.end(); i++)
25         cout << *i << "\t";    // 60 42 80 100 20 42 80 100
26     return 0;
27                                     // 20 60 20 20 20 42 80 100
```

<https://www.cplusplus.com/reference/algorithm/remove/>

Algoritmos: eliminar elementos

- *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*

https://www.cplusplus.com/reference/algorithm/remove_copy/

Algoritmos: reemplazar elementos

- *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*

<https://www.cplusplus.com/reference/algorithm/replace/>
https://www.cplusplus.com/reference/algorithm/replace_if/

Algoritmos: reemplazar elementos

- *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*

https://www.cplusplus.com/reference/algorithm/replace_copy/
https://www.cplusplus.com/reference/algorithm/remove_copy_if/

Algoritmos: 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

<https://www.cplusplus.com/reference/algorithm/find/>
[https://www.cplusplus.com/reference/algorithm/find if/](https://www.cplusplus.com/reference/algorithm/find_if/)

Algoritmos: 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* y *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 ordenados por ese mismo criterio

https://www.cplusplus.com/reference/algorithm/binary_search/

Algoritmos: 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
 - Introsort: $O(n \log n)$
 - 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

<https://www.cplusplus.com/reference/algorithm/sort/>

Algoritmos: ordenación

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  bool criterion (int elem1, int elem2){
7      //Si elem1 es par y elem2 es impar entonces elem1 va primero
8      if (elem1 % 2 == 0 && elem2 % 2 != 0)
9          return true;
10     else if (elem1 % 2 != 0 && elem2 % 2 == 0)//A la inversa es elem2 el que va primero
11         return false;
12     else //Si tienen la misma paridad ordenamos segun el valor de mayor a menor
13         return elem1 > elem2;
14 }
15
16 int main(){
17     int elems[] = {3,6,2,7,10,15,1};
18     vector<int> v(elems, elems + sizeof(elems) / sizeof(int));
19                                     //vector <int> v (elems, elems+7);
20     sort(v.begin(), v.end(), criterion);
21
22     cout << "\nOrdenado con nuestro criterio:\n";
23     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
24         cout << *it << "\t";           //10 6 2 15 7 3 1
25
26     return 0;
27 }
```

<https://www.cplusplus.com/reference/algorithm/sort/>

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

<https://www.cplusplus.com/reference/algorithm/copy/>

https://www.cplusplus.com/reference/algorithm/copy_backward/

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 $[first, last)$

<https://www.cplusplus.com/reference/algorithm/unique/>
<https://www.cplusplus.com/reference/algorithm/reverse/>

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`

https://www.cplusplus.com/reference/algorithm/random_shuffle/

<https://www.cplusplus.com/reference/algorithm/count/>

https://www.cplusplus.com/reference/algorithm/count_if/

Algoritmos: utilidades

- *iterator min_element(first, last, [criterion])*
 - Devuelve un iterador apuntando al menor elemento en la secuencia en el rango *[first y 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

https://www.cplusplus.com/reference/algorithm/min_element/
https://www.cplusplus.com/reference/algorithm/max_element/

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 cuarto argumento una función binaria que devuelva, a partir de dos elementos, un nuevo elemento con un tipo compatible con `T`

<https://www.cplusplus.com/reference/numeric/accumulate/>

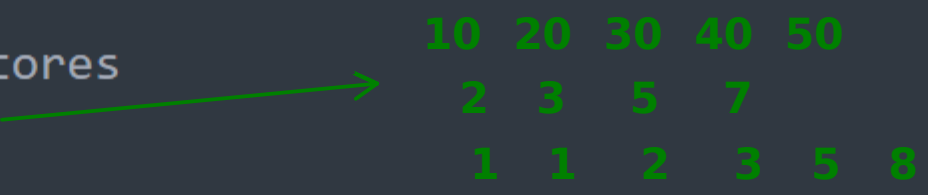
Algoritmos: utilidades

- *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

https://www.cplusplus.com/reference/algorithm/for_each/
<https://www.cplusplus.com/reference/algorithm/transform/>

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 vectores que en C++11 podemos definir como
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}); // inserta al final de datos un vector con 5 elementos
24     datos.push_back({2,3,5,7});         // inserta al final de datos un vector con 4 elementos
25     datos.push_back({1,1,2,3,5,8});     // inserta al final de datos un vector con 6 elementos
```



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()); // multiplicaciones tiene tamaño 3
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 //10 x 20 x 30 x 40 x 50
38 //Imprimimos los resultados con for_each
39 for_each(multiplicaciones.begin(), multiplicaciones.end(), print); //210 → 2 x 3 x 5 x 7
40 //240
41 return 0;
42 }
```

10 20 30 40 50
2 3 5 7
1 1 2 3 5 8

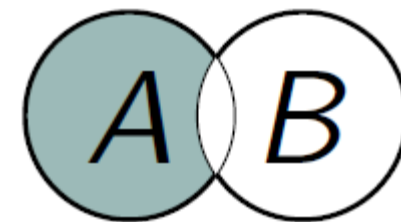
1 x 1 x 2 x 3 x 5 x 8

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 $[firstA, lastA) \setminus [firstB, lastB)$ en la secuencia que comienza en *result*
 - Y se aplican las mismas consideraciones para el criterio de comparación que en *includes*

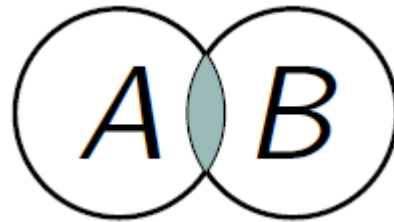
<https://www.cplusplus.com/reference/algorithm/includes/>

https://www.cplusplus.com/reference/algorithm/set_difference/

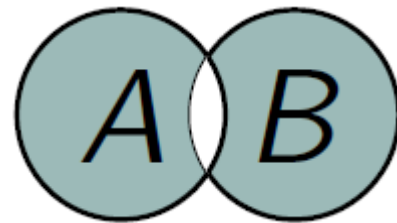


Algoritmos: conjuntistas

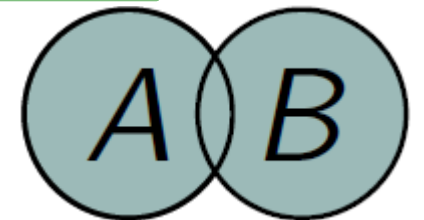
- *iterator set_intersection(firstA, lastA, firstB, lastB, result, [criterion])*
 - Opera igual que set_difference pero devolviendo la intersección de ambos rangos



- *iterator set_symmetric_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



https://www.cplusplus.com/reference/algorithm/set_intersection/
https://www.cplusplus.com/reference/algorithm/set_symmetric_difference/
https://www.cplusplus.com/reference/algorithm/set_union/

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
- <https://www.cplusplus.com/>