# Los algoritmos en la Biblioteca Estándar de Plantillas (STL)

En C++ son un conjunto de funciones genéricas diseñadas para trabajar con contenedores, como ***vectores, listas, conjuntos, y mapas.*** Estas funciones permiten realizar operaciones comunes como búsqueda, ordenación, transformación, y manipulación de datos de manera eficiente y con un código conciso.
A continuación, se detalla el uso de algunos de los algoritmos más importantes en la STL:

## 1. sort

Ordena los elementos de un contenedor en un rango específico.

Sintaxis:

```
std::sort(start_iterator, end_iterator);
std::sort(start_iterator, end_iterator, compare_function);
```

Ejemplo:

```
std::vector<int> vec = {5, 2, 9, 1, 5, 6};
std::sort(vec.begin(), vec.end()); // Ordena de menor a mayor
```

*Comparación personalizada*:

```
std::sort(vec.begin(), vec.end(), std::greater<int>()); // Ordena de mayor a menor
```

## 2. find

Busca un elemento en un rango específico. Devuelve un iterador apuntando al elemento encontrado o al final del rango si no se encuentra.

Sintaxis:

```
auto it = std::find(start_iterator, end_iterator, value);
```
Ejemplo:

```
auto it = std::find(vec.begin(), vec.end(), 9);
if (it != vec.end()) {
     std::cout << "Found: " << *it << std::endl;
}
```

## 3. accumulate

Suma (o combina) los valores en un rango específico.

Sintaxis:

```
T result = std::accumulate(start_iterator, end_iterator, initial_value);
T result = std::accumulate(start_iterator, end_iterator, initial_value, binary_op);
```

Ejemplo:

```
int sum = std::accumulate(vec.begin(), vec.end(), 0); // Suma de todos los elementos
```

Combinación personalizada:

```
std::string concatenated = std::accumulate(strings.begin(), strings.end(), std::string(""));
```

## 4. for_each

Aplica una función a cada elemento en un rango.

Sintaxis:

```
std::for_each(start_iterator, end_iterator, function);
```

Ejemplo:

```
std::for_each(vec.begin(), vec.end(), [](int &n){ n++; }); // Incrementa cada elemento
```

## 5. count y count_if

*Cuenta cuántas veces aparece un valor específico (count) o cuántos elementos cumplen con una condición (count_if).*

Sintaxis:

```
int n = std::count(start_iterator, end_iterator, value);
int n = std::count_if(start_iterator, end_iterator, condition);
```

Ejemplo:

```
int cnt = std::count(vec.begin(), vec.end(), 5); // Cuenta las veces que aparece el 5
int even_cnt = std::count_if(vec.begin(), vec.end(), [](int n){ return n % 2 == 0; }); // Cuenta los pares
```

## 6. copy y copy_if

Copia elementos de un rango a otro. copy_if permite copiar solo aquellos que cumplen una condición.

Sintaxis:

```
std::copy(start_iterator, end_iterator, destination_iterator);
std::copy_if(start_iterator, end_iterator, destination_iterator, condition);
```

Ejemplo:

```
std::vector<int> dest_vec(vec.size());
std::copy(vec.begin(), vec.end(), dest_vec.begin());
```

## 7. transform

Aplica una función a un rango de elementos y almacena el resultado en otro contenedor.

Sintaxis:

```
std::transform(start_iterator, end_iterator, destination_iterator, function);
```

Ejemplo:

```
std::vector<int> squared_vec(vec.size());
std::transform(vec.begin(), vec.end(), squared_vec.begin(), [](int n){ return n * n; });
```

## 8. remove y remove_if

Desplaza los elementos que no coinciden con un valor o una condición al inicio del rango y devuelve un iterador al final del rango "nuevo". Para eliminar realmente, se debe combinar con erase.

Sintaxis:

```
auto new_end = std::remove(start_iterator, end_iterator, value);
auto new_end = std::remove_if(start_iterator, end_iterator, condition);
```

Ejemplo:

```
vec.erase(std::remove(vec.begin(), vec.end(), 5), vec.end()); // Elimina todos los 5
```

## 9. unique

Elimina elementos consecutivos duplicados en un rango.

Sintaxis:

```
auto new_end = std::unique(start_iterator, end_iterator);
```

Ejemplo:

```
vec.erase(std::unique(vec.begin(), vec.end()), vec.end()); // Elimina duplicados consecutivos
```

## 10. binary_search

Verifica si un valor está presente en un rango ordenado.

Sintaxis:

```
bool found = std::binary_search(start_iterator, end_iterator, value);
```

Ejemplo:

```
bool found = std::binary_search(vec.begin(), vec.end(), 3);
```

## 11. lower_bound y upper_bound

Encuentran la primera posición donde un valor podría ser insertado (lower_bound) y la última posición (upper_bound) en un rango ordenado.

Sintaxis:

```
auto lb = std::lower_bound(start_iterator, end_iterator, value);
auto ub = std::upper_bound(start_iterator, end_iterator, value);
```

Ejemplo:

```
auto lb = std::lower_bound(vec.begin(), vec.end(), 3);
auto ub = std::upper_bound(vec.begin(), vec.end(), 3);
```

## 12. equal_range

Devuelve un par de iteradores representando el rango de elementos iguales a un valor en un contenedor ordenado.

Sintaxis:

```
auto range = std::equal_range(start_iterator, end_iterator, value);
```

Ejemplo:

```
auto range = std::equal_range(vec.begin(), vec.end(), 3);
```

## 13. merge

Combina dos secuencias ordenadas en una tercera secuencia ordenada.

Sintaxis:

```
std::merge(start1, end1, start2, end2, destination);
```

Ejemplo:

```
std::vector<int> vec1 = {1, 3, 5};
std::vector<int> vec2 = {2, 4, 6};
std::vector<int> result(vec1.size() + vec2.size());
std::merge(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), result.begin());
```

## 14. reverse

Invierte el orden de los elementos en un rango.

Sintaxis:

```
std::reverse(start_iterator, end_iterator);
```

Ejemplo:

```
std::reverse(vec.begin(), vec.end());
```

## 15. rotate

Rota los elementos en un rango de manera que el elemento en una posición específica se convierta en el primer elemento.

Sintaxis:

```
std::rotate(start_iterator, new_first, end_iterator);
```

Ejemplo:

```
std::rotate(vec.begin(), vec.begin() + 1, vec.end()); // Rota a la izquierda
```

## 16. partition y stable_partition

Reorganiza los elementos para que los que cumplen con una condición estén antes que los que no la cumplen.

Sintaxis:

```
auto it = std::partition(start_iterator, end_iterator, condition);
auto it = std::stable_partition(start_iterator, end_iterator, condition);
```

Ejemplo:

```
std::partition(vec.begin(), vec.end(), [](int n){ return n % 2 == 0; }); // Pares primero
```

Estos algoritmos, junto con muchos otros, hacen que la STL de C++ sea una herramienta poderosa y flexible para la manipulación de datos. La elección del algoritmo adecuado puede simplificar y optimizar tu código de manera significativa.