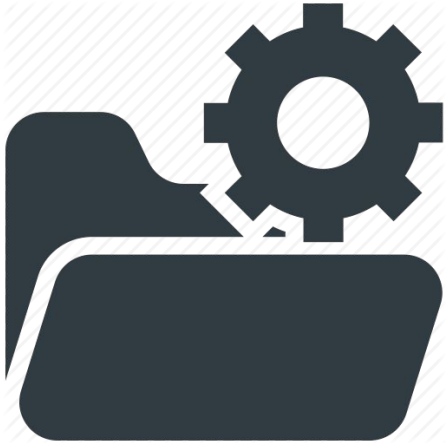
Standard Template Library (STL)

Standard Template Library

- Librería que implemente de estructuras de datos y algoritmos que forman parte del estándar de C++.
- Evita que se tenga que programar algo de uso frecuente.
- Presenta conceptos como contenedores e iteradores que permiten que las estructuras de datos sean bastantes genrales.

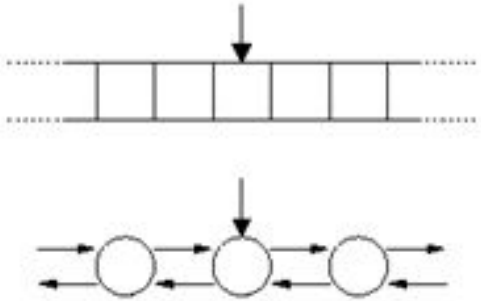


Contenedores

- Estructura que puede almacenar una colección de elementos del tipo y hacer operaciones con ellos.
- Utilizan **memoria heap** para guardar sus elementos
- Principales contenedores: ***vector, deque, set, multiset, map, stack, queue, priority queue***

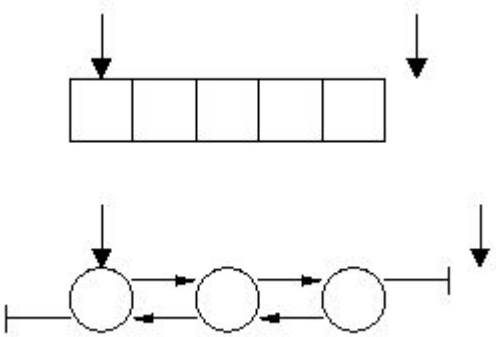
Iteradores

- Nos permiten recorrer los contenedores. Representan una posición en el contenedor.
- Normalmente los iteradores están conectados y la mayoría de veces uno puede incrementar el iterador actual o derecementarlo. En casos especiales incluso hay iteradores de tipo *random access* que nos permite acceder a cualquier posición en cualquier momento.
- Internamente los iteradores están implementados a base de un concepto de C++ llamados punteros, que permiten a las variables guardar direcciones de memoria en vez de valores.



Iteradores

- El rango que toman los iteradores de un contenedor generalmente es de tipo exclusivo. El primer iterador apunta al primer elemento, pero el último iterador apunta al elemento **ficticio** que le sigue al último.



Operadores con iteradores

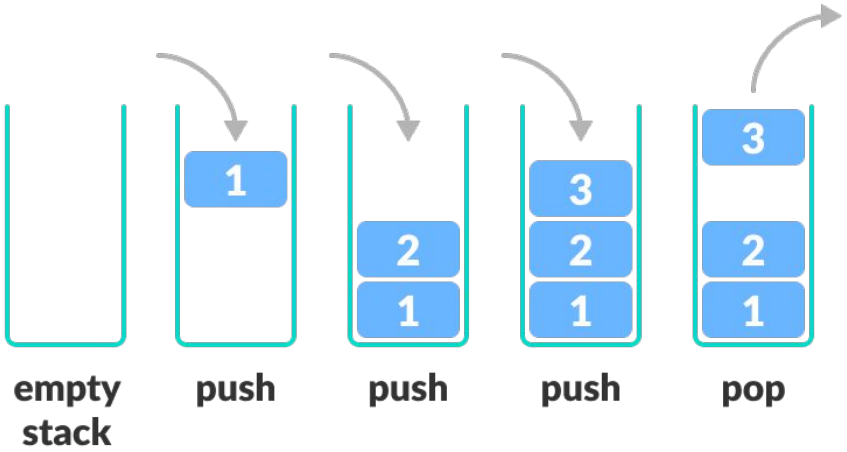
- Iterador al principio del contenedor (primer elemento) `container.begin();`
- Iterador al final del contenedor (elemento ficticio) `container.end();`
- Incrementar el iterador (ir al siguiente elemento) `iterator++;`
- Decrementar el iterador (ir al elemento anterior) `iterator--;`
- Retorna el iterador que está a n posiciones de cierto iterador `next(iterator, n);`
- Retorna el iterador que está a n posiciones anteriores de cierto iterador `prev(iterator, n);`
- Acceder el valor que apunta un iterador `*iterator;`

Operadores con contenedores

- Todos los contenedores tienen la función $size()$ que retorna el tamaño del contenedor en $O(1)$ y la función $empty()$ que retorna un booleano en True si está el contenedor vacío o en false en caso contrario, en $O(1)$

Stack (Pila)

- Guarda los elementos con una estrategia LIFO (Last In First Out)
- Inserta un elemento en la cima de la pila
- Saca elementos de la cima de la pila



Fuente: Programiz (https://www.programiz.com/dsa/stack)

Stack (Pila)

```
stack<int> s; // Stack declaration
s.push(5);    // Push element to the top of stack - O(1)
// Stack = Top {5} Bottom
s.push(6);
// Stack = Top {6, 5} Bottom
s.push(1);
// Stack = Top {1, 6, 5} Bottom
int x = s.top(); // Get the top element of stack - O(1)
// X is 1
s.pop(); // Delete the top element of stack - O(1)
// Stack = Top {6, 5} Bottom
```

Aplicación del stack – Parentización balanceada

Problema: Dado un string compuesto de ‘(‘ y ‘)’ , decir si la expresión presenta parentización balanceada. Es decir, que hay alguna forma de insertar números y operaciones matemáticas tal que la operación sea válida matemáticamente.

Formalmente, se define que una parentización balanceada de la siguiente forma:

- Una cadena vacía está balanceada
- Si s es una cadena balanceada, también lo es (s)
- Si s y t son dos cadenas balanceadas, también lo está la concatenación st

Ejemplos:

( ) Balanceada

(()()) Balanceada

(())(No balanceada

((No balanceada

Aplicación del stack – Parentización balanceada

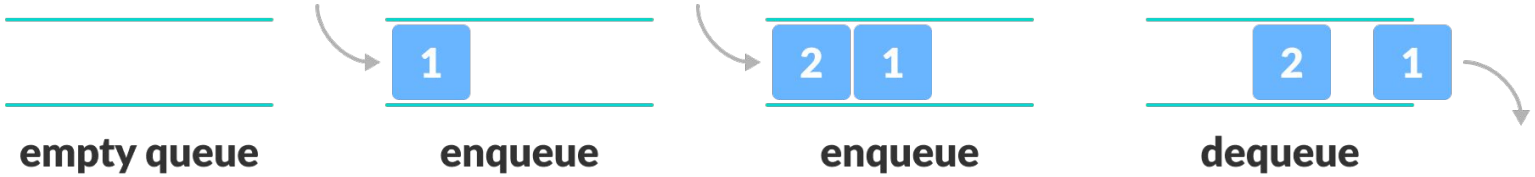
Solución: Insertar los paréntesis en una *stack*. Si el último paréntesis en el stack es *(* y el paréntesis actual es *)*, entonces tenemos un match y podemos eliminar el paréntesis *(* del *stack.* Caso contrario, insertamos el paréntesis en el stack.

Si al finalizar el proceso, el *stack* está vacío, entonces decimos que el string tiene parentización balanceada; caso contrario, no lo tiene.

Bonus: Con más observaciones se puede obtener una solución más simple sin usar un stack.

Queue (Cola)

- Guarda los elementos con una estrategia FIFO (First In First Out)
- Los elementos salen en el mismo orden que entraron



Fuente: Programiz (https://www.programiz.com/dsa/queue)

Queue (Cola)

```cpp
queue<int> q; // Queue declaration
q.push(5);    // Push element to the queue - O(1)
// Queue = Back {5} Front
q.push(6);
// Queue = Back {6, 5} Front
q.push(1);
// Queue = Back {1, 6, 5} Front
int x = q.front(); // Get the front element of the queue - O(1)
// X is 5
q.pop(); // Delete the front element of the queue - O(1)
// Queue = Back {1, 6} Front
```

Priority Queue (Cola de Prioridad)

- Mantiene todos los elementos ordenados por prioridad (de mayor a menor)
- Al sacar un elemento, saca el mayor.
- Internamente funciona con una estructura de datos llamada heap.

Priority Queue (Cola de Prioridad)

```
priority_queue<int> pq; // Priority Queue declaration
pq.push(5);             // Push element to the priority queue - O(log n)
// Priority Queue = Back {5} Front
pq.push(6);
// Priority Queue = Back {5, 6} Front
pq.push(1);
// Priority Queue = Back {1, 5, 6} Front
int x = pq.top(); // Get the greatest element of the priority queue - O(1)
pq.pop();         // Delete the greatest element of the priority queue - O(log n)
// Priority Queue = Back {1, 5} Front
```

Para cambiar el orden y sacar el menor elemento en vez del mayo, lo declararía así

```
priority_queue<int, vector<int>, greater<int>> pq;
```

Vector

- Contenedor que almacena elemento en posiciones contiguas de memoria
- Pueden cambiar de tamaño en tiempo de ejecución.
- Permite acceso aleatorio.
- Permite insertar y eliminar un elemento al final en $O(1)$.
- Permite simular una pila.

Vector

```cpp
vector<int> v = {10, 3, 7, 9, 1}; // Vector declaration
int n = v.size();                 // Access the size of the vector - O(1)
v.push_back(10);                  // Push 10 to the end of the vector - O(1)
v.pop_back();                     // Delete the last element of the vector - O(1)

int front = v.front();   // Get the first element of the vector - O(1)
int last = v.back();     // Get the last element of the vector - O(1)
int randomAccess = v[2]; // Random access position 2

vector<int>::iterator itExplicit; // Explicit declaration of an iterator variable

auto it = v.begin(); // Implicit declaration of iterator variable

int newElement = 7;
v.insert(v.begin() + 2, newElement); // Insert a new element in a position 2 - O(n)
v.erase(v.begin() + 3);              // Erase element from position 3 - O(n)

for (int x : v) { // Traverse vector - O(n)
    cout << x << "\n";
}
```

Deque

- Contenedor que almacena elemento en posiciones contiguas de memoria
- Pueden cambiar de tamaño en tiempo de ejecución.
- Permite acceso aleatorio.
- Permite insertar y eliminar un elemento **inicio y al final** en $O(1)$.
- Permite simular una pila y una cola.

Deque

```cpp
deque<int> v = {10, 3, 7, 9, 1}; // Deque declaration
int n = v.size();                // Access the size of the deque - O(1)
v.push_back(10);                 // Push 10 to the end of the deque - O(1)
v.pop_back();                    // Delete the last element of the deque - O(1)
v.push_front(7);                 // Push 7 to the start of the deque - O(1)
v.pop_front();                   // Delete the first element of the deque - O(1)

int front = v.front();   // Get the first element of the deque - O(1)
int last = v.back();     // Get the last element of the deque - O(1)
int randomAccess = v[2]; // Random access position 2

deque<int>::iterator itExplicit; // Explicit declaration of an iterator variable

auto it = v.begin(); // Implicit declaration of iterator variable

int newElement = 7;
v.insert(v.begin() + 2, newElement); // Insert a new element in a position 2 - O(n)
v.erase(v.begin() + 3);              // Erase element from position 3 - O(n)

for (int x : v) { // Traverse deque - O(n)
    cout << x << "\n";
}
```

Set

- Nos permite almacenar elementos únicos.
- Los elementos estarán ordenados de forma creciente.
- Internamente funciona con una estructura de datos compleja llamada Red Black Tree.
- Las operaciones de búsqueda, inserción y eliminación son en tiempo logarítmico respecto al tamaño del contenedor.

Set

```cpp
set<int> s = {10, 3, 7, 9, 1}; // Set declaration
int n = s.size();              // Access the size of the set - O(1)

s.insert(5); // Insert new element to the set - O(log n)
// The set will be {1, 3, 5, 7, 9, 10}

s.insert(10); // Duplicate element. The set will remain the same - O(log n)
// The set will be {1, 3, 5, 7, 9, 10}

s.erase(7); // Delete an element from the set - O(log n)
// The set will be {1, 3, 5, 9, 10}

for (int x : s) { // Traverse Set - O(n)
    cout << x << " ";
}
```

Multiset

- Tiene las mismas propiedades que un **set** pero permite almacenar elementos duplicados.

Multiset

```cpp
multiset<int> s = {10, 3, 7, 9, 1}; // Multiset declaration
int n = s.size();                   // Access the size of the multiset - O(1)

s.insert(5); // Insert new element to the multiset - O(log n)
// The multiset will be {1, 3, 5, 7, 9, 10}

s.insert(10); // Duplicate element. But will be inserted - O(log n)
// The multiset will be {1, 3, 5, 7, 9, 10, 10}

s.erase(7); // Delete an element from the multiset - O(log n)
// The multiset will be {1, 3, 5, 9, 10, 10}

s.erase(10); // Delete all the 10s from the multiset - O(log n)
// The multiset will be {1, 3, 5, 9}

for (int x : s) { // Traverse multiset - O(n)
    cout << x << " ";
}
```

Map

- También conocido como diccionario.
- Nos permite almacenar pares de la forma <*llave, valor*> , donde la llave es única.
- Los elementos son guardados en orden ascendente respecto a su llave.
- Internamente funciona con una estructura de datos compleja llamada Red Black Tree.
- Las operaciones de búsqueda, inserción y eliminación se hacen en tiempo logarítmico respecto al tamaño del contenedor.

Map

```cpp
map<string, int> age; // Map declaration
int n = age.size();   // Access the size of the map - O(1)

age["Pedro"] = 21;  // Add the key Pedro with value 21 - O(log n)
age["Sandra"] = 20; // Add the key Sandra with value 20 - O(log n)

int pedroAge = age["Pedro"]; // Access the value of certain key - O(log n)

age["Pedro"] = 7; // Update the key of a key - O(log n)

age.erase("Pedro"); // Remove the element with a certain key - O(log n)

for (auto [key, value] : age) { // Traverse map - O(n)
    cout << key << " -> " << value << "\n";
}
```

Map

```
map<string, vector<string>> friends;

friends["Pedro"] = {"Paulo", "Jorge"};
friends["Juan"] = {"Maria"};
friends["A"] = {};

friends["A"].push_back("B");
friends["Juan"].pop_back();

cout << friends["Pedro"][1] << "\n";
```

Pair y Tuple

- Nos permite almacenar un número de elementos de diferente tipo en una variable.
- Los tipos de elementos son indicados como argumentos al ser instanciados.

```cpp
pair<string, int> player_A("K. Mbappe", 25); // Pair declaration

pair<string, int> player_B; // Default declaration ("", 0)

player_B.first = "L. Yamal"; // Assign the first element
player_B.second = 17; // Assign the second element

cout << player_A.first << " vs " << player_B.first << '\n'; // Access their values
```

Pair y Tuple

```cpp
tuple<string, double, int, bool> course_A("BQU01", 9.9, 5, false); // Tuple declaration
tuple<string, double, int, bool> course_B; // Default declaration ("", 0.0, 0, false)

course_B = make_tuple("BIC01", 19, 3, true); // Assign the values
get<1>(course_B) = 20; // Assign single value

cout << get<1>(course_B) << '\n'; // Access their values
```

Funciones STL - Sorting

- Para el caso de vectores y deques, tenemos la función *sort* que ordena los elementos en $O(n \log n)$
- Lleva como parámetros dos iteradores que representa un rango **exclusivo** de los elementos que se quieren ordenar *[start, end>*
- Esta función también funciona para un arreglo estático, pero se considera al nombre de la variable (supongamos que se llama a) como el puntero de inicio. Esto se aplica para varias funciones que también funcionen en un *vector*.

```
vector<int> v;
sort(v.begin(), v.end());       // Sort all elements - O(n log n)
sort(v.begin(), v.begin() + 2); // Sort first 2 elements - O(n log n)
```

```
int a[n];
sort(a, a + n); // Sort all elements - O(n log n)
sort(a, a + 2); // Sort first 2 elements - O(n log n)
```

Funciones STL - Sorting

- Para ordenar de manera decreciente tenemos dos formas: ordenarlo y revertirlo u ordenarlo directamente de manera inversa

```
sort(a.begin(), a.end());
reverse(a.begin(), a.end());
```

```
sort(a.rbegin(), a.rend());
```

Funciones STL - Find

- Para el caso de sets, multisets, maps tenemos la función ***find*** para encontrar si un elemento está presente en el contendor en $O(\log n)$

- La función retorna el iterador donde el elemento se encuentra. En caso el elemento no esté presente en el contenedor, se retornará el iterador ficticio $\text{container}.end()$

```
set<int> s;
auto it = s.find(5); // Return the iterator where the value 5 is located - O(log n)
```

- En el caso de lo *vector* o *deque* (o arreglos estáticos) también puedo hacer una operación similar llamada *binary_search* que me retornará **true** si es que el elemento se encuentra en el contenedor. Pero es necesario manualmente ordenar el contenedor antes.

```
sort(v.begin(), v.end()); // O(n log n)
int target = 7;
bool found = binary_search(v.begin(), v.end(), target); // O (log n)
```

Funciones STL – Lower bound

- Dada un valor $target$, la función $lower\_bound(x)$ retorna el iterador que contenga al primer elemento que sea $\geq x$ en $O(\log n)$. En caso no exista, retorna el iterador ficticio.
- Es necesario que el contenedor esté ordenado (ya sea por default o manualmente).

Para *set, multiset, map*, como el contenedor siempre está ordenado, lo haría de esta forma:

```
int x = 5;
auto it = s.lower_bound(x); // O(log n)
```

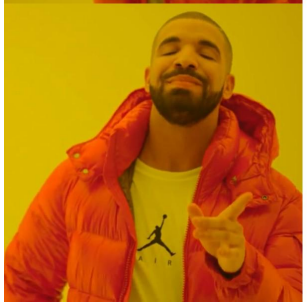
Para *vector, deque* (o arreglos estáticos) el formato es el siguiente

```
sort(v.begin(), v.end()); // O(n log n)
int x = 7;
auto it = lower_bound(v.begin(), v.end(), target); // O(log n)
```

Funciones STL – Lower bound

Mucho cuidado en el caso del set, multiset, map. Pues, si uno intenta utilizarlo con el formato del vector, esto funcionará pero será complejidad $O(n)$

```cpp
set<int> s;
int x = 5;
auto it = lower_bound(s.begin(), s.end(), x); // O(n)
```



```cpp
set<int> s;
int x = 5;
auto it = s.lower_bound(x); // O(log n)
```

Funciones STL – Upper bound

- Dada un valor $target$, la función $upper\_bound$ retorna el iterador que contenga al primer elemento que sea $> x$ en $O(\log n)$. En caso no exista, retorna el iterador ficticio.
- Es necesario que el contenedor esté ordenado (ya sea por default o manualmente).

Para *set, multiset, map*, como el contenedor siempre está ordenado, lo haría de esta forma:

```
set<int> s;
int x = 5;
auto it = s.upper_bound(x); // O(log n)
```

Para *vector, deque* (o arreglos estáticos) el formato es el siguiente

```
sort(v.begin(), v.end()); // O(n log n)
int x = 7;
auto it = upper_bound(v.begin(), v.end(), target); // O(log n)
```

¡Gracias por su atención!

Referencias

- [1] Steve Oualline. *Practical C++ Programming.* Chapter 25: Standard Template Library