Tema 6 - Librería Estándar de C++

a. Introducción a contenedores, iteradores y algoritmos

b. Contenedores de secuencias: stack, list, queue

La Librería Estándar

- Define componentes altamente reutilizables que implementan las estructuras utilizadas de forma habitual.
- La reutilizabilidad está garantizada ya que todas las funciones y métodos se escribieron en formato de plantilla es decir usando template <class T, etc…
- De hecho, el nombre de la librería estándar es: Standard Template Library (STL).
- Entre las implementaciones más útiles encontramos **Contenedores, Iteradores y Algoritmos.**

Contenedores

- Son clases definidas como plantillas que permiten juntar muchos objetos y operar con ellos de forma simultánea.
- Algunos ejemplos son:

Contenedor	Descripción
array	de tamaño fijo y acceso directo a cualquier element [i]
list	Enlaza los objetos y permite inserción en cualquier posición
vector	Agiliza aún más la inserción de elementos.
set	Grupo de elementos que no permite objetos duplicados
map	‘Mapeo’ uno-a-uno sin duplicados: Función Biyectiva
stack	Apilamiento de objetos: Last-In-First-Out (LIFO)
queue	Fila de objetos: First-In-First-Out(FIFO)

Contenedores

- Todos los contenedores definen los operadores conocidos, así como funciones de construcción.
- Además, definen métodos útiles para todo contenedor como begin(), end(), clear(), erase(int i).
- Según el caso, los contenedores pueden implementar más o menos métodos.
- En esta asignatura, trabajaremos con array, queue, list y stack.

array: el contenedor fijo ordenado por posición

- C++ reimplementó el agrupamiento de distintos elementos del mismo tipo en la clase array.
- Funciona de la misma forma que al usar [ ], pero como ***array*** es una clase, ésta tiene nuevas funcionalidades que pueden resultar útiles.
- Incluir la librería <array> es necesario.
- Para inicializar un arreglo de, por ejemplo, 5 flotantes ha de utilizarse: **array<float, 5> a;**
- Entendiendo que la clase se declara con:
  template <class T, typename I> array<T, I>
- Para la asignación del elemento i se procede de igual forma: **a[i] = 5.0f;**
- La ventaja más notoria es que como es un objeto, el arreglo **puede ser el resultado de (devuelto por) una función**, debido a que podemos retornar una instancia de la clase, sin necesidad de recurrir a punteros y direcciones.

array >01

- En cualquier caso, siempre se deberá definir el tipo y la cantidad de elementos dentro de los corchetes angulares:

`array<tipo, tamano>`

- El tipo puede inclusive ser una clase cualquiera generada por el programador.
- La clase array tiene definida la función size() que permite obtener el tamaño de forma programática.

```
array<float, 5> numeros = {1, 2, 3, 4, 5};
```

Tema7 - utils.h

```
1  array<float, 5> crear_arreglo(){
2      array<float, 5> numeros = {1, 2, 3, 4, 5};
3      return numeros;
4  }
```

Tema7 - Clase7.cpp

```
1  for (int i = 0; i < numeros.size(); i++){
2      cout << numeros[i] << endl;
3  }
```

array: Características principales

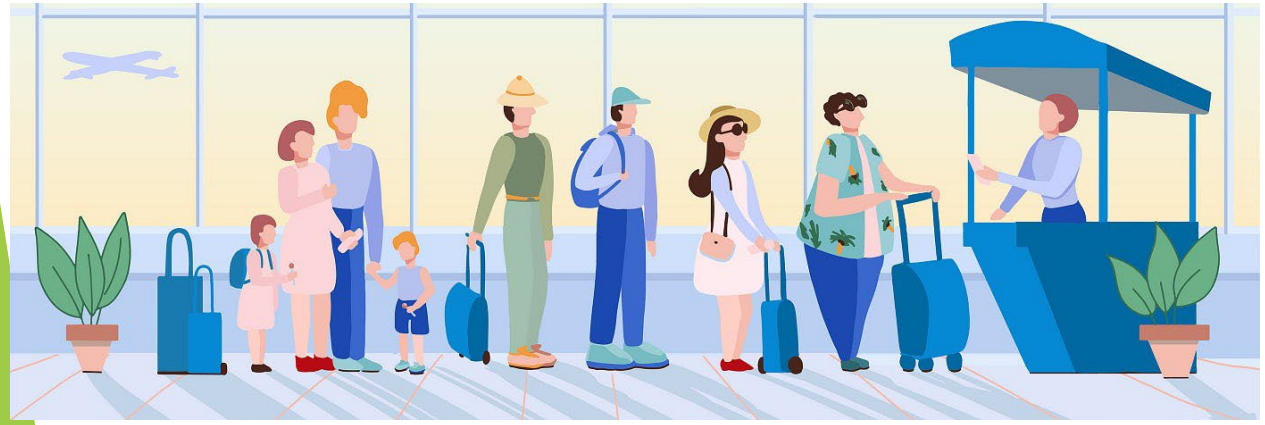
- El uso es equivalente al de utilizar arreglos definidos como en lenguaje C.
- Incluye métodos adicionales: size(), fill().
- Es de acceso aleatorio, es decir que puede permitir el acceso al elemento 3 y justo a continuación el acceso al elemento 0.
- Es de tamaño fijo, lo que lo hace extremadamente eficiente en términos de memoria, lectura y escritura.

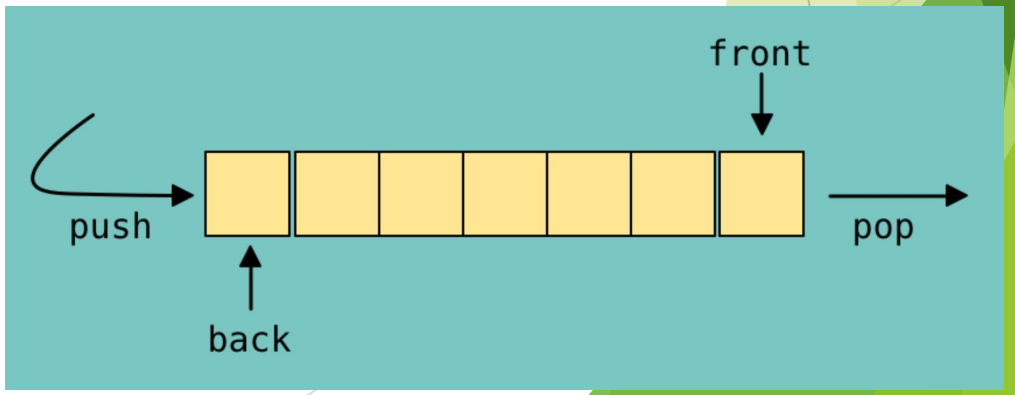
queue: contenedor dinámico tipo FIFO

- STL define al contenedor de cola (queue) de forma a que puedan almacenarse en conjunto un número indeterminado de elementos del mismo tipo, siendo el primer objeto agregado el único objeto accesible.
- Para utilizarlo habrá que incluir la librería queue y también usar el namespace std.
- Como no tiene un tamaño fijo, la construcción de un objeto es:

```
1   queue<float> numeros;
```

- FIFO significa First In-First Out, por lo que el comportamiento esperado de un objeto queue es exactamente el de una cola





queue

- Para agregar un nuevo elemento a la cola podemos usar el método .push(T elemento):

```
1  numeros.push(1.1f);
```

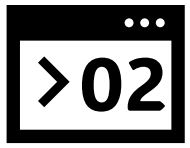
- Esto agrega un elemento a la cola al final del todo, por lo que, al agregar muchos elementos, estos automáticamente se sitúan al final de la cola.
- Podemos obtener una referencia al elemento que se encuentra al inicio de la cola mediante .front() :

```
1  numeros.front();
```

- Y para el último elemento utilizaremos .back():

```
1  numeros.back();
```

queue



- En una cola, no se puede acceder a un elemento cualquiera, por lo que no es posible utilizar corchetes [ ].
- Esto no significa que la cola no esté rellena con varios elementos, es más:

```
1   numeros.size()
```

- Indica la cantidad de elementos que posee la queue, mientras que:

```
1   numeros.empty()
```

- Indica si la cola está o no vacía.

queue: Características principales

Expande la funcionalidad de arreglos al permitir un número dinámico de elementos.

Permite agregar un nuevo elemento al final del contenedor usando **.push (elemento: T)** o eliminar el primero usando **.pop()**

Su funcionamiento se basa en el concepto FIFO

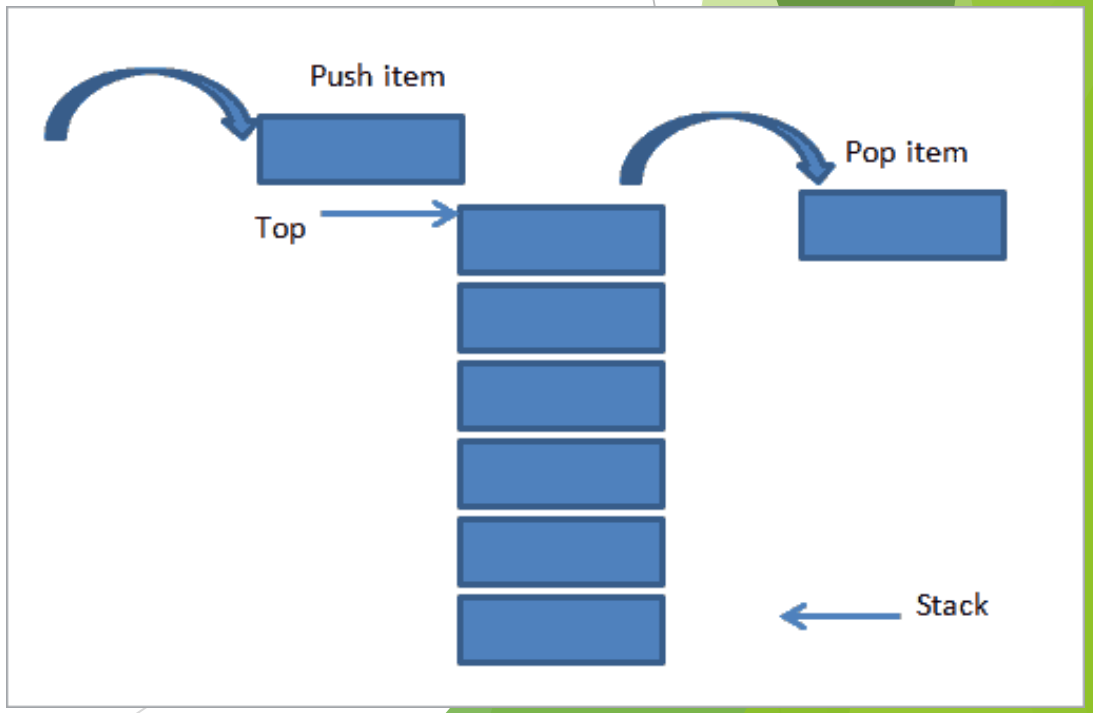
No se puede borrar ni acceder a cualquier elemento aleatorio

stack: contenedor dinámico tipo LIFO

- STL define al contenedor de pila (stack) de forma a que puedan almacenarse en conjunto un número indeterminado de elementos del mismo tipo, siendo el último objeto agregado el único objeto accesible.
- Para utilizarlo habrá que incluir la librería stack y también usar el namespace std.
- Como no tiene un tamaño fijo, la construcción de un objeto es:

```
1   stack<int> IDs;
```

- LIFO significa Last In-First Out:



stack

- Para agregar un nuevo elemento a la pila podemos usar el método .push (T elemento):

```
1   IDs.push(dato);
```

- Esto agrega un elemento al final del todo, por lo que, al agregar muchos elementos, estos automáticamente “se apilan”.
- Podemos obtener una referencia al elemento que se encuentra encima en la pila mediante .top() :

```
1   IDs.top()
```

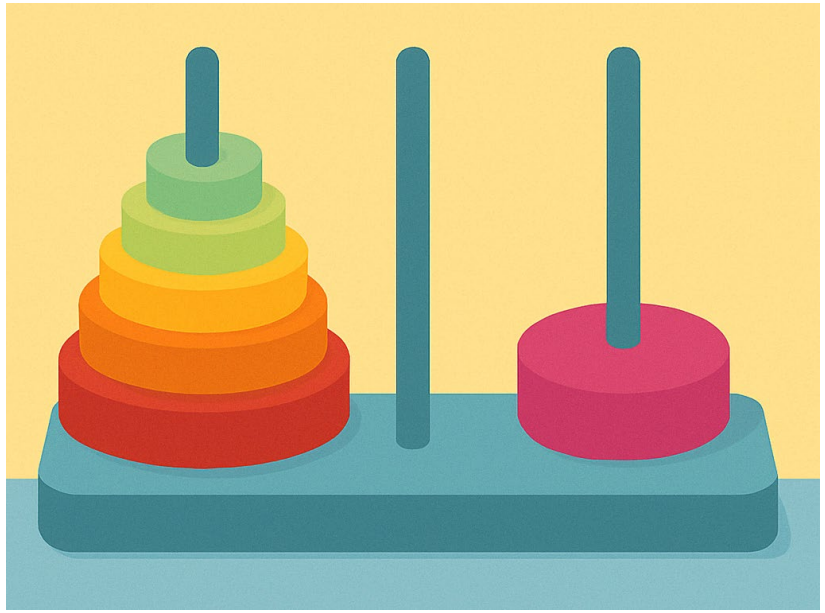
- Y para eliminarlo utilizamos:

```
1   IDs.pop();
```

stack

>03

- Así como en una cola, en un stack, no se puede acceder a un elemento cualquiera, por lo que no es posible utilizar corchetes [ ].
- También nos encontramos con los métodos de .size() y .empty() con los mismos comportamientos.
- Una stack y una cola se diferencian en la forma de acceso a los elementos. Normalmente en una cola, se tiene acceso al primer y último elemento, mientras que en la stack solamente se tiene acceso al último elemento.
- Ambos contenedores comparten el comportamiento de que los elementos se agregan siempre al final.



stack: Características principales

Simplifica el contenedor al disponer solamente un único acceso

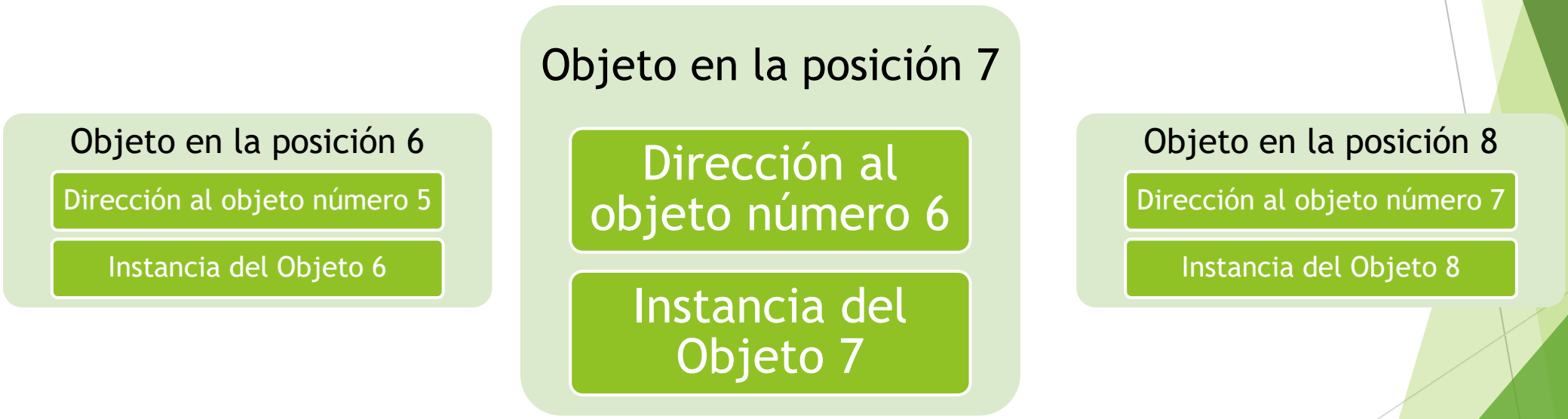
Permite agregar un nuevo elemento al final del contenedor usando **.push (elemento: T)** o eliminar el último agregado usando **.pop()**

Su funcionamiento se basa en el concepto LIFO

No se puede borrar ni acceder a cualquier elemento aleatorio

Stack y Queue en memoria

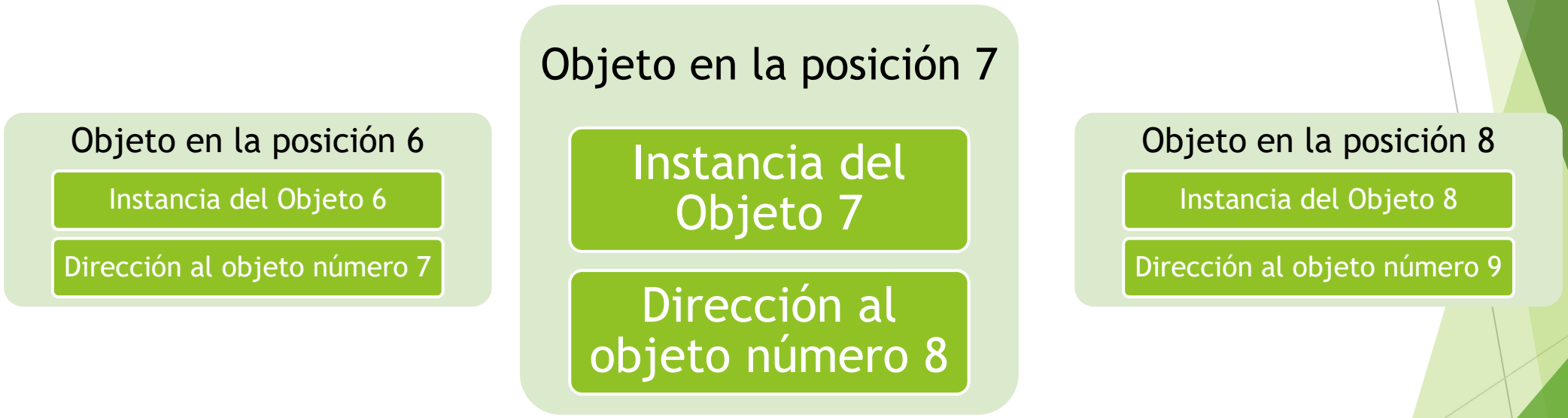
- Estos contenedores poseen una estructura de simple enlace: el elemento i además de tener información de su contenido (su valor), también tiene un puntero que apunta al elemento anterior (i-1) o siguiente (i+1).



- En este caso, si nuestro contenedor tuviera 9 elementos, para acceder al elemento 7, debemos acceder al último elemento y desde ahí movernos hacia los objetos añadidos previamente, es decir que el Ultimo ingresado debe ser el Primero en salir (stack)

Stack y Queue en memoria

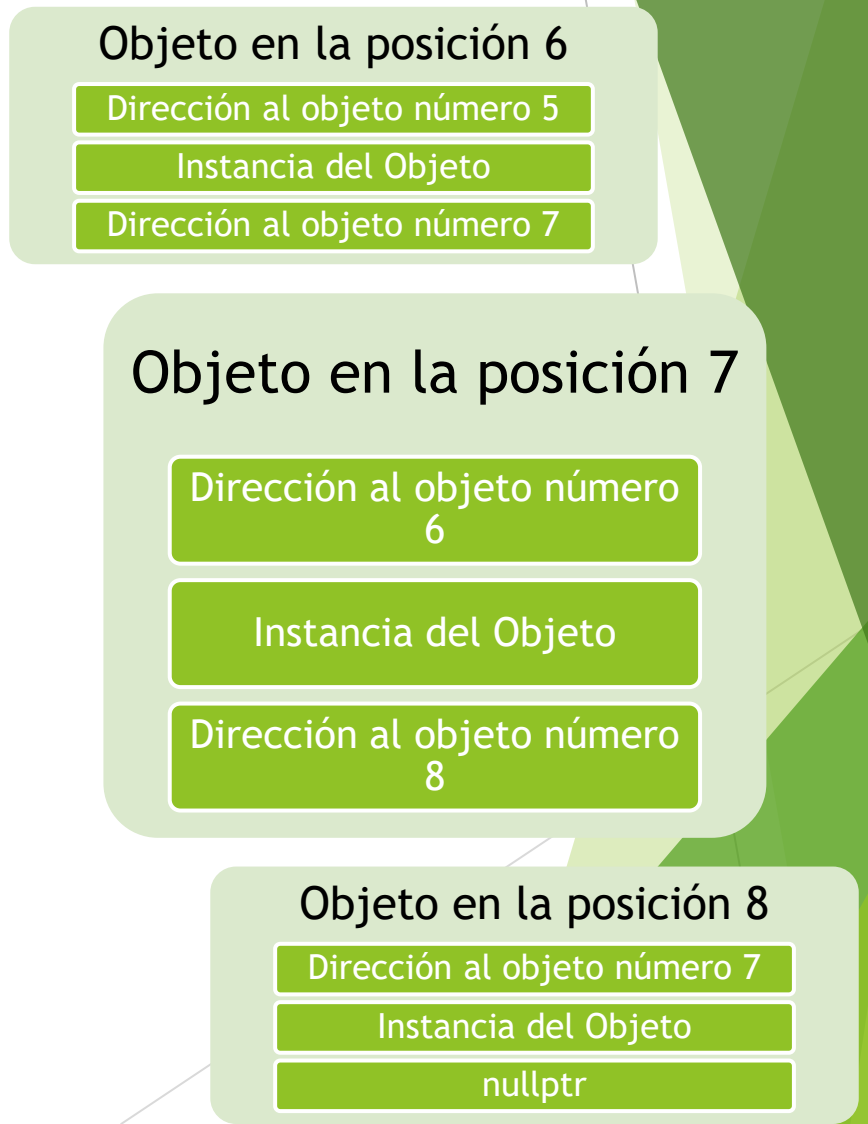
- Por otro lado, si el contenedor guarda las direcciones en el otro sentido, es decir desde el primero hasta el último, tenemos el primer contenedor estudiado:



- Ya que, si nuestro contenedor tuviera 9 elementos, para acceder al elemento 7, debemos acceder al primer elemento y desde ahí movernos hacia los objetos añadidos posteriormente, es decir que el Primer ingresado debe ser el Primero en salir (queue)

list: Contenedor de elementos enlazados en memoria

- Una lista es un contenedor con doble enlace: el elemento i además de tener información de su contenido, también tiene 2 punteros, uno apunta al elemento anterior (i-1) y otro al elemento (i+1) siguiente.
- Al tener el doble enlace, la lista tiene un posible acceso del primero al ultimo o del ultimo al primero, por lo que las funciones pop y push necesitarían información adicional.
- Se agregan funciones de: **push_front**, **pop_front**, **push_back y pop_back.**
- Como el enlace es basado en punteros, se podrían insertar y borrar elementos con facilidad mediante el uso de los métodos **insert** y **erase.**



list y los iteradores

- Para poder utilizar los métodos **insert** y **erase** debemos empezar a utilizar los iteradores.
- Al enlazar los objetos mediante punteros, la librería STL no solo trabaja internamente con objetos especiales que permiten acceder al frente o atrás, sino que también los ofrece para que podamos hacer uso de éstos cuando sea necesario.
- En cualquier caso, ahora necesitamos trabajar con un objeto que es elemento de un contenedor, por lo que no podemos trabajar con una referencia a un objeto como hacíamos anteriormente.
- Para poder trabajar con un elemento de un contenedor, en este caso necesitamos un iterador.

- La creación de una lista sigue la misma forma:

```
1   list<float> numeros = {1, 2, 3, 4, 5};
```

- Pero ya se admite opcionalmente valores de inicialización.
- Un iterador depende de un contenedor especifico, o sea que una lista de flotantes necesita un iterador de una lista de flotantes.

```
1   list<float>::iterator it
```

list >04

- Los métodos de .begin() y .end() generan iteradores que apuntan o bien al primer o al último elemento de la lista, por lo que podríamos utilizarlos para acceder a la lista.

```
1  list<float>::iterator it1 = numeros.begin();
2  list<float>::iterator it2 = numeros.end();
```

- Al tratarse de punteros, podemos acceder al valor almacenado en cada iterador mediante:

```
1  cout << "Al inicio de la lista existe: " << *it1 << endl;
2  cout << "Al final de la lista existe: " << *it2 << endl;
```

list >05

- Para acceder a todos los elementos, como no es de acceso aleatorio, habrá que recorrerse la lista, ya que está enlazada.
- Para avanzar en la lista se usa la función advance(it: iterator, cantidad: int): void

```
1  list<float> numeros = {1, 2, 3, 4, 5};
2  list<float>::iterator it = numeros.begin();
3  for (int i = 0; i < numeros.size(); i++){
4      cout << *it << endl;
5      advance(it, 1);
6  }
```

- Si no se desea modificar el iterador y que siempre apunte o al inicio o al final, podemos utilizar las funciones **next(nombre: iterator): iterator** o **prev (nombre: iterator): iterator**.

list >06

- Para utilizar **.sort()** debe existir el operador< definido.
- En tipos primitivos, eso no supone ningún problema:

```
1  list<float> numeros = {5, 1, 3, 4, 2};
2  numeros.sort();
```

- Pero en el caso de clases definidas por el usuario, habrá que asegurar la existencia del operador:

Tema7 - Vector2D.h

```
1  bool operator<(const Vector2D& otro);
```

- Por supuesto, esto significa que, a la hora de crear un contenedor, debemos especificar que el tipo es Vector2D:

```
1  list<Vector2D> vectores;
2  vectores.push_back(Vector2D(5, 5));
3  ...
4  vectores.sort();
```

list: Características principales

El almacenamiento se realiza mediante dobles enlaces de direcciones de memoria.

Comportamiento similar al de la clase vector, con la diferencia de que no es de acceso aleatorio

Gracias a la definición de enlace, borrar e insertar es más eficiente en tiempo de ejecución

Si operator< está definido, puede utilizarse .sort y .unique

Comparación de contenedores

Característica	array	queue	stack	list
Tamaño fijo/Estático	**Sí**	No	No	No
Puede inicializarse	Si	No	No	Si
Acceso aleatorio	Sí	No	No	No
Inserción	No	Final	Final	Todos
Eliminación	No	Inicio	Final	Todos
Orden	Indexado	FIFO	LIFO	Doble enlace
Posee iteradores	Sí	No	No	Sí

Ejercicios

- (Problema 1) Cree un programa que instancie 3 objetos diferentes de una clase Articulo e inserte estos elementos a un contenedor de la clase list. Imprima los objetos. A continuación, agregue al inicio un nuevo objeto de la clase Articulo y por último inserte una copia del objeto 1 en la posición 3 de la lista. Imprima los objetos. Ordene los elementos usando y vuelva a imprimir los objetos.
- En el mismo programa, cambie los atributos de precio a cada uno de los objetos de la lista. Ej.: Sumar +5.

- (Problema 2) Cree un programa que instancie 3 objetos diferentes de la clase Articulo e inserte estos elementos a un contenedor de la clase stack. Imprima los objetos. A continuación, inserte un nuevo objeto de la clase Articulo agréguelo a la stack. Imprima los objetos.
- En el mismo programa, cambie los atributos de precio a cada uno de los objetos de la lista. Ej.: Sumar +5.

Algoritmos de ordenación

- ¿Como hace una lista para ordenar los objetos de una clase?

art1:Articulo	art2:Articulo	art3:Articulo
nombre= “A1” serial_no= 123	nombre= “A2” serial_no= 456	nombre= “A3” serial_no= 789

- Ordenar un conjunto de elementos requiere de la existencia de jerarquías de organización (Condición mínima y suficiente: operator<).
- Existen varias formas de ordenar un grupo de objetos dentro de un contenedor.
- Pero, si queremos optimizar recursos, tendremos que ser lo más eficientes que sea posible.

Algoritmos de ordenación

- Actualmente existen muchos algoritmos de ordenación que son ampliamente utilizados.
- De entre ellos podemos destacar algunos que tienen un reconocimiento por alguna característica en particular.
- **Bubble Sort:** En cada iteración i comparar el elemento i con el siguiente. Si es mayor, intercambiar. Repetir hasta que no se hagan intercambios.
- **Selection Sort:** En cada iteración i (1 por elemento), buscar el elemento mínimo e intercambiarlo con el elemento en la posición i de la iteración i.
- **Insertion Sort:** En cada iteración i, reubicar al elemento i entre los elementos anteriores que sean un poco mayor y un poco menor.
- **Quick Sort:** es un algoritmo de ordenación basado en la estrategia "divide y vencerás". La idea fundamental es seleccionar un elemento, llamado "pivote", y particionar la lista de elementos en dos subconjuntos: aquellos menores que el pivote y aquellos mayores que el pivote. Luego, se aplica recursivamente el mismo proceso a cada subconjunto. **Elección del Pivote:** Selecciona un elemento de la lista como pivote. La elección del pivote puede hacerse de varias maneras; una opción común es elegir el primer elemento de la lista. **Partición:** Reordena la lista de manera que todos los elementos menores que el pivote estén a la izquierda, y todos los elementos mayores estén a la derecha. Después de la partición, el pivote se encuentra en su posición final. **Recursión:** Aplica recursivamente el mismo proceso a las sublistas generadas por la

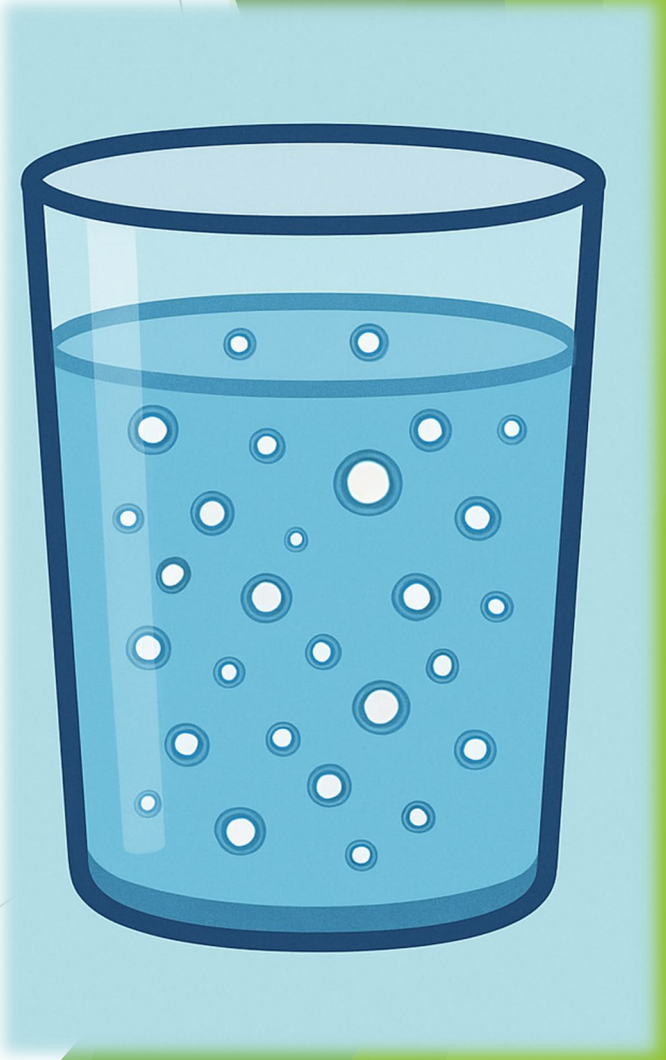
- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket Sort
- Bingo Sort Algorithm
- ShellSort
- TimSort
- Comb Sort
- Pigeonhole Sort
- Cycle Sort
- Cocktail Sort
- Strand Sort
- Bitonic Sort
- Pancake sorting
- BogoSort or Permutation Sort
- Gnome Sort
- Sleep Sort – The King of Laziness
- Structure Sorting in C++
- Stooge Sort
- Tag Sort (To get both sorted and original)
- Tree Sort
- Odd-Even Sort / Brick Sort
- 3-way Merge Sort

Bubble Sort

- El método de la burbuja busca ordenar elementos haciendo comparaciones de un elemento con su vecino siguiente.
- Supongamos entonces 5 elementos en el siguiente arreglo:

```
1   array<int, 5> numeros = {5, 3, 1, 4, 2};
```

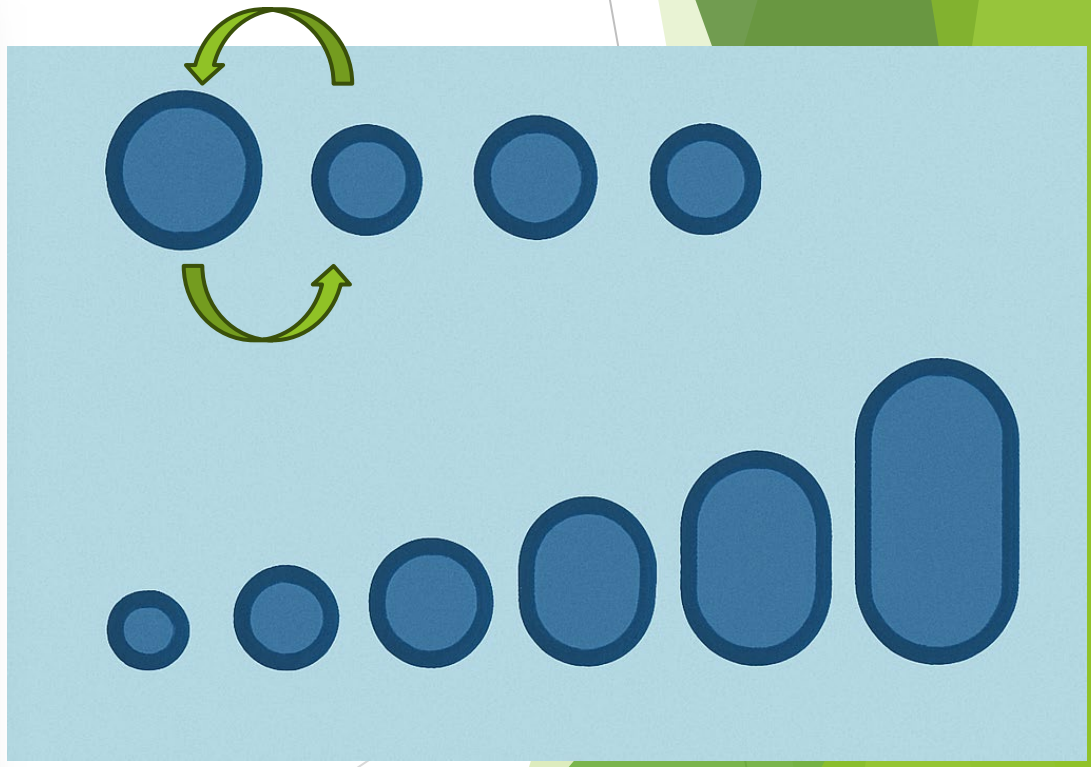
- Tomemos el elemento en la segunda posición (3) y realicemos la comparación con el siguiente elemento (1)
- Como **1 > 3** entonces realizamos un cambio utilizando la función std::swap(T elem1, T elem2)
- Realizamos este proceso una vez para todos los elementos que existen en el contenedor (n) y luego realizamos el cambio si el elemento en la iteración el elemento i es mayor que el elemento siguiente.
- Siempre que hayamos realizado por lo menos un cambio, debemos repetir el proceso



Bubble Sort >07

- El método de la burbuja busca ordenar elementos según el vecino siguiente.

```
1   array<int, 5> numeros = {5, 3, 1, 4, 2};
2   bool intercambio_hecho;
3   do{
4       intercambio_hecho = false;
5       for (int i = 0; i < numeros.size()-1; i++)
6       {
7           if (numeros[i] > numeros[i+1]){
8               swap(numeros[i], numeros[i+1]);
9               intercambio_hecho = true;
10          }
11      }
12
13  }while(intercambio_hecho);
```



Selection Sort

- El método de selección busca uno a uno los elementos más pequeños de todo el contenedor (búsqueda hacia delante)
- Supongamos entonces 5 elementos en el siguiente arreglo:

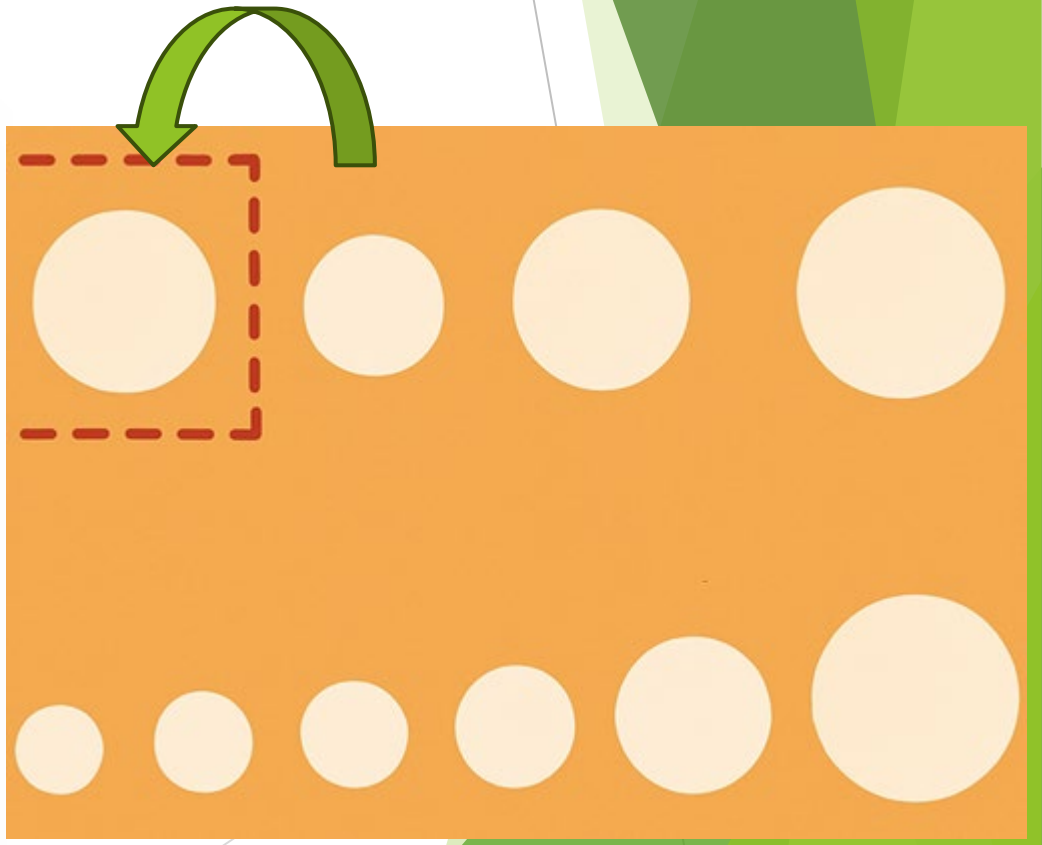
```
1   array<int, 5> numeros = {5, 3, 1, 4, 2};
```

- Comenzamos a buscar el número más pequeño posible, y lo intercambiamos con el número en la posición 0. En la siguiente iteración buscamos el siguiente más pequeño y lo intercambiamos con el elemento en la posición 1.
- Repetimos el proceso hasta que hayamos intercambiado todos los elementos.

Selection Sort

- El método de selección busca uno a uno los elementos más pequeños de todo el contenedor.

```
1  for (int i = 0; i < numeros.size(); i++)
2  {
3      int indice_menor = i;
4      for (int j = i; j < numeros.size(); j++)
5          if (numeros[j] < numeros[indice_menor])
6              indice_menor = j;
7      if (i != indice_menor)
8          swap(numeros[i], numeros[indice_menor]);
9      else
10         break;
11 }
```



Insertion Sort

- El método insertion busca insertar un elemento entre el elemento que sea un poco menor y el elemento que sea un poco mayor (intercambio hacia atrás)
- Supongamos entonces 5 elementos en el siguiente vector:

```
1   vector<int> numeros = {5, 3, 1, 4, 2};
```

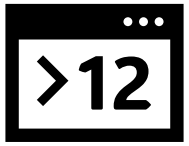
- Asumimos que el elemento 0 se encuentra en la posición correcta y realizamos un cambio en cadena si se reubica el elemento 1 en la posición 0.
- Repetimos el proceso para cada elemento i.

Insertion Sort

- El método insertion busca insertar un elemento entre el elemento que sea un poco menor y el elemento que sea un poco mayor.

```
1  vector<int> numeros = {5, 3, 1, 4, 2};
2  vector<int>::iterator a_insertar, donde;
3  for (int i = 1; i < numeros.size(); i++)
4  {
5      donde = numeros.begin();
6      a_insertar = numeros.begin() + i;
7      for (int j = 0; j <= i; j++)
8      {
9          if ( *a_insertar < *donde){
10             break;
11         }
12         advance(donde, 1);
13     }
14 int valor = *a_insertar;
15 numeros.erase(a_insertar);
16 numeros.insert(donde, valor);
17 }
```

Algoritmos de ordenación

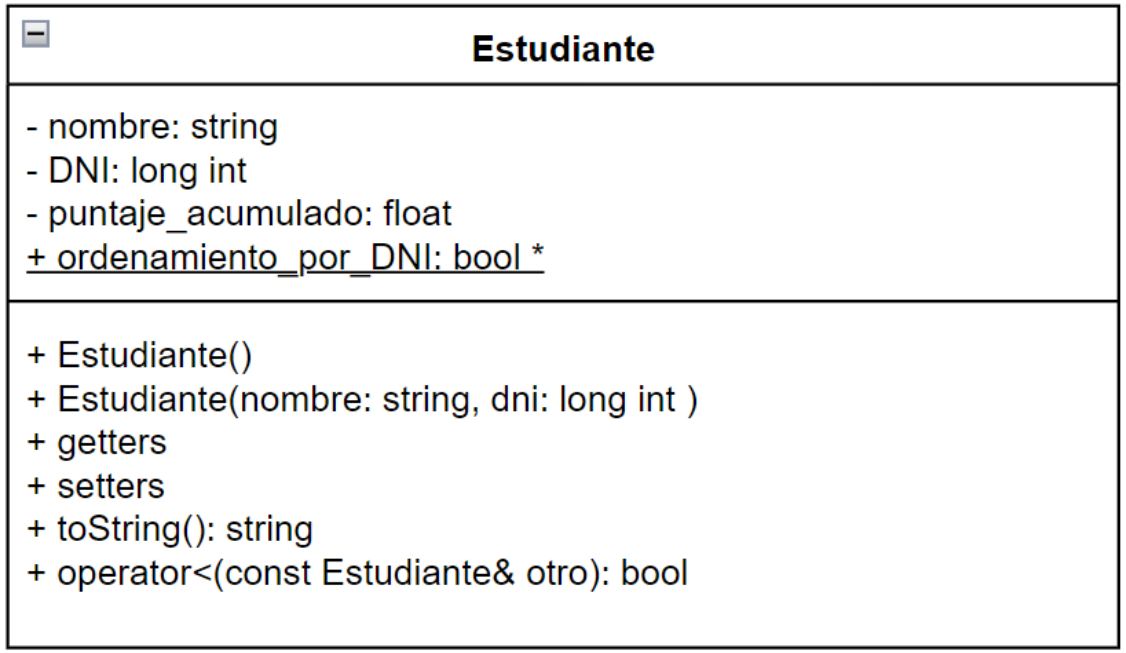


- Todos estos algoritmos han utilizado el operador < para comparación.
- A continuación, por ejemplo, podemos realizar la misma función, pero en formato de plantilla, asumiendo elementos de entrada desconocidos que corresponden a iteradores de contenedores.

```
template<class T> void ordenar(T begin, T end){
    T it, menor_posible;
    int size = begin - end;
    for(int i = 0; i < size; i++){
        menor_posible = begin + i;
        it = begin + i;
        for (int j = i; j < size; j++){
            if (*it < *menor_posible){
                menor_posible = it;
            }
            advance(it, 1);
    }
    swap(*menor_posible, *(begin + i));
    }
}
```

Algoritmos de ordenación

- Los algoritmos de ordenación, en conjunto con la definición apropiada del operador de menor que, pueden ser útiles para ordenar diferentes contenedores de acuerdo con diferentes ítems.



- Crear un array de diferentes objetos de la clase Estudiante, y ordenarlos por DNI.
- Repetir para: una list de diferentes objetos de la clase y ordenarlos por Puntaje Acumulado.
- Cuidado, Array no permite insertar objetos, y list requiere acceso iterativo.