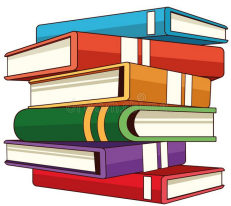
CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

# Clase 1: STL
# CC4001-CC4005

Otoño 2023

# ¿Qué es la STL?

Es la abreviación para *Standard Template Library*, una **librería** de C++ que nos proporciona muchas herramientas útiles en la Programación Competitiva.



# ¿Cómo la importamos?

Se pueden importar uno por uno los *headers* para tener acceso a cosas específicas de la librería, sin embargo, se suele trabajar con el siguiente esquema:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    // su código
}
```

que resulta ser más simplificado, y nos da acceso a todas las herramientas.

# Motivación

Existe la *complejidad algorítmica*, que tiene relación con la eficiencia de un algoritmo para resolver una tarea.

Todos los problemas tienen restricciones de tiempo y espacio.


- **Complejidad de tiempo:** Es la cantidad de tiempo que se demora un algoritmo en resolver un problema. *Si nuestro algoritmo es ineficiente, nos dará TLE (Time Limit Exceeded).*
- **Complejidad de espacio:** Es la cantidad de memoria requerida por el algoritmo para solucionar el problema. *Casi nunca suele excederse este límite.*

# Motivación

Ambas complejidades están en función del *input*. Por lo tanto, es importante revisar qué tan grandes pueden ser los números que nos dan.

En general, una buena cota al evaluar complejidad de tiempo es $10^8$. Es importante considerar **el peor caso**.

La notación usada para la complejidad es $\mathcal{O}(\cdot)$. En general, lo que nos añadirá complejidad son los ciclos (`for`, `while`).

**La STL es una herramienta potente que posee algoritmos que nos servirán para manejar esta eficiencia.**

# Preámbulo: Herramientas

Dentro de las herramientas que nos da la librería estándar, ocuparemos en el curso:

- Estructuras de datos.

**Ejemplos**

Vectores, strings, *queues* (colas), *priority queues* (colas de prioridad), stacks, maps, sets.

- Algoritmos, como `sort` o `find`.
- Iteradores.

# Preámbulo: Iteradores

También, necesitamos definir lo básico de los **iteradores**.

- Son objetos que nos permiten recorrer los elementos de un contenedor.
- En particular, los contenedores son las estructuras que veremos a continuación.

# Preámbulo: Iteradores

- El iterador `end()` apunta a una ubicación después del último elemento. Nos sirve para detectar elementos que no existen en el contenedor.
- El iterador `begin()` nos entrega un iterador que apunta al primer elemento.
- Los métodos `.prev()` (antes) y `.next()` (después) sirven para recorrer de 1 en 1 el contenedor.
- Para encontrar el iterador asociado a un elemento, se ocupa el método `.find(elemento)`.
- Para obtener el valor asociado a un iterador (llamémosle `itr`), se antepone un asterisco (`*`), por ejemplo, `*itr`.

# Vectores

- Son listas de elementos indexados.
- Son mejores que los arreglos nativos de C++.

Pensemos en un `vector` de *strings* (cadenas de texto). Queremos crear un `vector` de largo 5 cuyos valores sean "`hola`".

## Estructura general

`vector<tipo de dato> nombre_vector(n.° de elems., valor default)`
*Obs.: El valor por defecto* ***no*** *es necesario especificarlo.*

Entonces, `vector<string> v(5, "hola")` nos genera un vector de nombre `v` definido como:

`v` $\longrightarrow$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| "hola" | "hola" | "hola" | "hola" | "hola" |

# Vectores

`v` $\longrightarrow$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| ”hola” | ”hola” | ”hola” | ”hola” | ”hola” |

Notemos que la indexación parte del 0. Esto es importante, porque para acceder a elementos del vector, debemos hacer lo siguiente:

**Propiedades importantes:**

- `v[i]` nos devuelve el elemento del $i$-ésimo índice. En particular, `v[0]` es el primer elemento del vector.
- Si el vector es de largo $N$, `v[N-1]` nos devuelve el último elemento.
- Esta estructura es mutable, es decir, puedo cambiar el valor de cualquier índice. Por ejemplo, `v[2] = "patito"` nos deja:

`v` $\longrightarrow$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| ”hola” | ”hola” | ”patito” | ”hola” | ”hola” |

# Vectores

- `v.push_back(s)` AGREGA el string `s` a nuestro vector, colocándolo en el último índice. Por ejemplo, `v.push_back("alo")` nos deja:

`v` ⟶

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| "hola" | "hola" | "patito" | "hola" | "hola" | "alo" |

Complejidad: $\mathcal{O}(1)$.

- `v.pop_back()` ELIMINA el último elemento de nuestro vector. Si lo aplicamos al ejemplo, queda:

`v` ⟶

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| "hola" | "hola" | "patito" | "hola" | "hola" |

Complejidad: $\mathcal{O}(1)$.

# Vectores

- `sort(v.begin(), v.end())` ORDENA el vector de **menor** a **mayor**. Si queremos el orden inverso (mayor a menor), podemos agregar un tercer parámetro: `greater<int>()`.

  Complejidad: $\mathcal{O}(n \log n)$.
- `reverse(v.begin(), v.end())` da vuelta el vector, de tal manera que el último elemento pasa a ser el primero, el penúltimo el segundo, etc.

  Complejidad: $\mathcal{O}(n)$.
- `v.assign(n.° de elems., valor default)` cambia el largo del vector y el valor por defecto que tienen los elementos. Para el ejemplo, `v.assign(2, "adiós")` genera:

v ⟶

| 0 | 1 |
|---|---|
| "adiós" | "adiós" |

  Complejidad: $\mathcal{O}(n)$.

# Strings

- Es una lista de caracteres (`char`, representados por comillas simples).
- Cada caracter está ligado al código ASCII.

Ejemplo: `string s = "saludos"` nos genera:

| `s` ⟶ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 's' | 'a' | 'l' | 'u' | 'd' | 'o' | 's' |
| | 115 | 97 | 108 | 117 | 100 | 111 | 115 |

De esta forma, podemos trabajar cada letra de un `string` como número sin problema. O sea, `s[1] += 1` nos deja con:

| `s` ⟶ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 's' | 'b' | 'l' | 'u' | 'd' | 'o' | 's' |
| | 115 | 98 | 108 | 117 | 100 | 111 | 115 |

*Obs.: El código ASCII está ordenado, por eso es que al sumarle uno a la letra* `a` *nos da* `b`.

# Strings

**Propiedades importantes:**

- `s.pop_back()` nos ELIMINA el último caracter del string. En nuestro ejemplo, quedaría:

`s` $\longrightarrow$

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 's' | 'b' | 'l' | 'u' | 'd' | 'o' |
| 115 | 98 | 108 | 117 | 100 | 111 |

- Para cambiar una letra en el índice $i$, sin necesidad de usar números, usamos `s[i] = 'letra'`. Por ejemplo, `s[1] = 'a'` nos deja con:

`s` $\longrightarrow$

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 's' | 'a' | 'l' | 'u' | 'd' | 'o' |
| 115 | 97 | 108 | 117 | 100 | 111 |

# Strings

Como sabemos que en el código ASCII las letras minúsculas, mayúsculas y números se encuentran adyacentes, podemos comprobar si el $i$-ésimo caracter de un string `s` es una minúscula de la siguiente forma:

```
'a' <= s[i] and s[i] <= 'z'
```

Es análogo para las mayúsculas. Dicha línea retornara `true` si `s[i]` es minúscula y `false` en caso contrario.

# Queues (colas)

- Funciona como una fila (lista de espera).
- Sólo podemos acceder al primer y último elemento.
- Si añadimos un elemento, se va al final de la cola.
- Cuando retiramos un elemento, sacamos el primero.

## Estructura general

```
queue<tipo de dato> nombre_cola;
```

# Queues (colas)

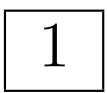
## Ejemplo

Declaremos una cola vacía de enteros.

```
queue<int> cola;
```

Para agregar elementos, debemos ocupar la función `push(elemento)`.

```
cola.push(1);
```



# Queues (colas)

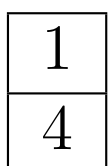
## Ejemplo

Declaremos una cola vacía de enteros.

```
queue<int> cola;
```

Para agregar elementos, debemos ocupar la función `push(elemento)`.

```
cola.push(1);
cola.push(4);
```



# Queues (colas)

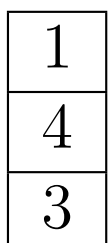
## Ejemplo

Declaremos una cola vacía de enteros.

```
queue<int> cola;
```

Para agregar elementos, debemos ocupar la función `push(elemento)`.

```
cola.push(1);
cola.push(4);
cola.push(3);
```



# Queues (colas)

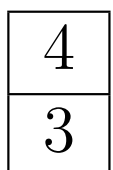
## Ejemplo

Declaremos una cola vacía de enteros.

```
queue<int> cola;
```

Para agregar elementos, debemos ocupar la función `push(elemento)`.

```
cola.push(1);
cola.push(4);
cola.push(3);
cola.front(); // nos entrega el primer valor (1)
cola.pop(); // elimina el primer valor (1)
```



# Priority queues (colas de prioridad)

- Funciona como una cola, pero a medida que ingresan elementos los va ordenando.
- Sus operaciones tomarán $\mathcal{O}(\log n)$ en ejecutarse.

## Estructura general

```
priority_queue<tipo de dato> nombre_cola;
```

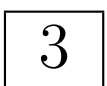
# Priority queues (colas de prioridad)

## Ejemplo

Hagamos una cola de prioridad de enteros:

```
priority_queue<int> cola;
```

```
cola.push(3);
```



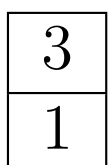
# Priority queues (colas de prioridad)

## Ejemplo

Hagamos una cola de prioridad de enteros:

```
priority_queue<int> cola;
```

```
cola.push(3);
cola.push(1);
```



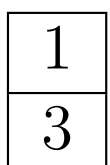
# Priority queues (colas de prioridad)

## Ejemplo

Hagamos una cola de prioridad de enteros:

```
priority_queue<int> cola;
```

```
cola.push(3);
cola.push(1);
```



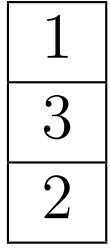
# Priority queues (colas de prioridad)

## Ejemplo

Hagamos una cola de prioridad de enteros:

```
priority_queue<int> cola;
```

```
cola.push(3);
cola.push(1);
cola.push(2);
```



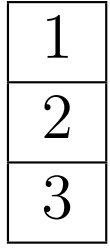
# Priority queues (colas de prioridad)

## Ejemplo

Hagamos una cola de prioridad de enteros:

```
priority_queue<int> cola;
```

```
cola.push(3);
cola.push(1);
cola.push(2);
```



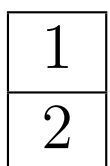
# Priority queues (colas de prioridad)

## Ejemplo

Hagamos una cola de prioridad de enteros:

```
priority_queue<int> cola;
```

```
cola.push(3);
cola.push(1);
cola.push(2);
cola.top(); // entrega el mayor valor (3)
cola.pop(); // elimina el mayor valor (3)
```



# Stacks

- Funciona como una pila (elementos apilados).
- Sólo podemos acceder al último elemento.
- Cuando añadimos un elemento, este se va al final de la pila.
- Cuando sacamos un elemento, sacamos el último de la pila.

## Estructura general

```
stack<tipo de dato> nombre_stack;
```

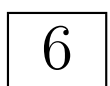
# Stacks

## Ejemplo

Hagamos una pila de enteros:

```
stack<int> pila;
```

```
pila.push(6);
```



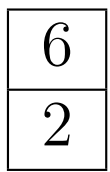
# Stacks

## Ejemplo

Hagamos una pila de enteros:

```
stack<int> pila;
```

```
pila.push(6);
pila.push(2);
```



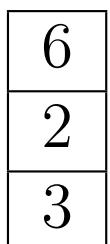
# Stacks

## Ejemplo

Hagamos una pila de enteros:

```
stack<int> pila;
```

```
pila.push(6);
pila.push(2);
pila.push(3);
```



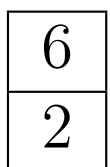
# Stacks

## Ejemplo

Hagamos una pila de enteros:

```
stack<int> pila;
```

```
pila.push(6);
pila.push(2);
pila.push(3);
pila.top(); // entrega el último valor (3)
pila.pop(); // elimina el último valor (3)
```



# Sets

- Es muy parecido a un conjunto matemático.
- Por cómo funciona, los `set` nos garantizan operaciones en $\mathcal{O}(\log N)$ donde $N$ es la cantidad de elementos del `set`.
- Los elementos se ordenan de menor a mayor dentro del conjunto. Para el orden inverso, podemos usar `set<T, greater<T>>`, donde `T` tiene que ser comparable.

## Estructura general

```
set<tipo de dato> nombre_set;
```

# Sets

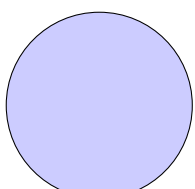
## Ejemplo

Creemos un `set` de enteros llamado `conjunto`:

```
set<int> conjunto;
```

Matemáticamente, representémoslo como $\mathcal{S}$.

Inicialmente, $\mathcal{S} = \emptyset$



# Sets

## Ejemplo

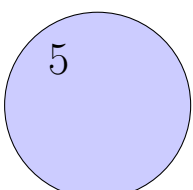
Creemos un `set` de enteros llamado `conjunto`:

```
set<int> conjunto;
```

Matemáticamente, representémoslo como $\mathcal{S}$.

$\mathcal{S} = \emptyset \cup \{5\} = \{5\}$

```
conjunto.insert(5); // funciona como unión
```



# Sets

## Ejemplo

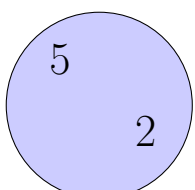
Creemos un `set` de enteros llamado `conjunto`:

```
set<int> conjunto;
```

Matemáticamente, representémoslo como $\mathcal{S}$.

$\mathcal{S} = \{5\} \cup \{2\} = \{2, 5\}$

```
conjunto.insert(5);
conjunto.insert(2);
```



# Sets

## Ejemplo

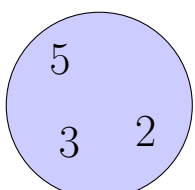
Creemos un `set` de enteros llamado `conjunto`:

```
set<int> conjunto;
```

Matemáticamente, representémoslo como $\mathcal{S}$.

$\mathcal{S} = \{5, 2\} \cup \{3\} = \{2, 3, 5\}$

```
conjunto.insert(5);
conjunto.insert(2);
conjunto.insert(3);
```



# Sets

## Ejemplo

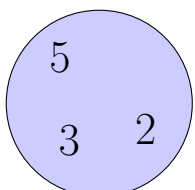
Creemos un `set` de enteros llamado `conjunto`:

```
set<int> conjunto;
```

Matemáticamente, representémoslo como $\mathcal{S}$.

$\mathcal{S} = \{2, 3, 5\} \cup \{5\} = \{2, 3, 5\}$ ¡No cambia!

```
conjunto.insert(5);
conjunto.insert(2);
conjunto.insert(3);
conjunto.insert(5);
```



# Sets

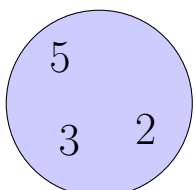
## Ejemplo

Creemos un `set` de enteros llamado `conjunto`:

```
set<int> conjunto;
```

Matemáticamente, representémoslo como $\mathcal{S}$.

- Para trabajar con iteradores, se ocupa la función `.find(elemento)`. Por ejemplo, `conjunto.find(5)` nos entrega un iterador al elemento `5`.



# Sets

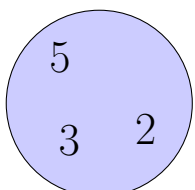
## Ejemplo

Creemos un `set` de enteros llamado `conjunto`:

```
set<int> conjunto;
```

Matemáticamente, representémoslo como $\mathcal{S}$.

- `conjunto.find(1)` nos devuelve el iterador `end` (`conjunto.end()`), pues `1` no está en el `set`.



# Sets

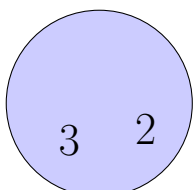
## Ejemplo

Creemos un `set` de enteros llamado `conjunto`:

```
set<int> conjunto;
```

Matemáticamente, representémoslo como $\mathcal{S}$.

- Para borrar elementos, se ocupa la función `.erase(iterador)`. Por ejemplo, `conjunto.erase(conjunto.find(5))` nos deja con el set:



# Sets: Lower Bound

A veces queremos encontrar el primer elemento que es MAYOR O IGUAL a un cierto valor en un rango del `set`. Para eso existe la función `lower_bound()`. Aplicado al ejemplo anterior, su uso es:

**Uso**

```
conjunto.lower_bound(first, last, val)
```

- `first` es el iterador que define el inicio del intervalo a revisar (se incluye).
- `last` es el iterador que define el fin del intervalo a revisar. **No** se incluye, i.e., trabajamos en [*first*, *last*).
- `val` es el valor fijo que quiero comparar.

*Obs.: Si sólo paso un parámetro, entonces se revisará* ***todo*** *el* `set` *(i.e., se omiten* `first` *y* `last`*).*

# Sets: Lower Bound

Por ejemplo, consideremos el `set` de enteros (llamémosle `conjunto`):

$$\mathcal{S} = \{1, 4, 6, 7, 10, 15, 19\}$$

Y queremos ver cuál es el valor más cercano a `5` que esté en el set, con la condición que tiene que ser mayor o igual. Entonces `conjunto.lower_bound(5)` me dará un `6`.

# Sets: Lower Bound

Una pregunta interesante es: ¿qué pasa si todos los valores son menores?

- En dicho caso, nos devuelve el iterador `last` que especificamos. Si no lo hicimos, nos devuelve el iterador `end()`.

Por ejemplo, para el `set` anterior:

$$\mathcal{S} = \{1, 4, 6, 7, 10, 15, 19\}$$

`conjunto.lower_bound(20)` nos devuelve `conjunto.end()`.

# Sets: Upper Bound

Notemos que `lower_bound()` nos devuelve una comparación si el valor es mayor o **igual**. Pero, ¿qué pasa si no nos interesa la existencia de `val`?

En ese caso, existe la función `upper_bound()`, que funciona de la misma forma, salvo que ahora solo nos da información sobre la existencia de valores **mayores estrictos** a `val` ($>$).

*Obs.: La otra diferencia es que cuando queremos revisar en un intervalo de valores,* ***`upper_bound()`*** *incluye ambos extremos* ***`first`*** *y* ***`last`*** *($[first, last]$).*

# Maps

- Funcionan como un diccionario de Python.
- Son pares de elementos compuestos por una llave y un valor.
- Si el mapa tiene $N$ elementos, esta estructura nos garantiza operaciones en $\mathcal{O}(\log N)$.
- Se puede ver como la extensión de un `set`.

## Estructura general

```
map<T llave, T valor> nombre_mapa;
```

donde `T` es el tipo de dato.

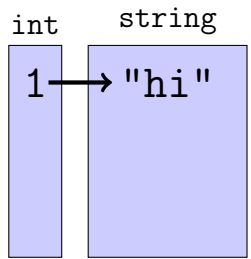
# Maps

## Ejemplo

Si queremos hacer un mapa que relacione enteros (llaves) con strings (valores), la forma es la siguiente:

```
map<int, string> mapa;
```

```
mapa[1] = "hi";
```



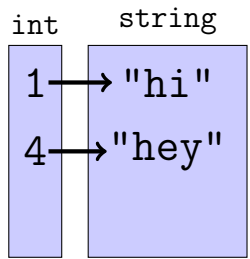
# Maps

## Ejemplo

Si queremos hacer un mapa que relacione enteros (llaves) con strings (valores), la forma es la siguiente:

```
map<int, string> mapa;
```

```
mapa[1] = "hi";
mapa[4] = "hey";
```



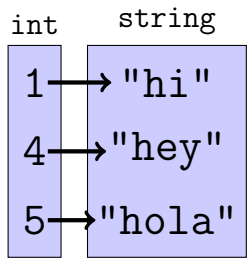
# Maps

## Ejemplo

Si queremos hacer un mapa que relacione enteros (llaves) con strings (valores), la forma es la siguiente:

```
map<int, string> mapa;
```

```
mapa[1] = "hi";
mapa[4] = "hey";
mapa[5] = "hola";
```



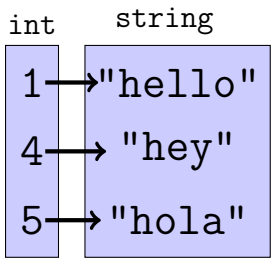
# Maps

## Ejemplo

Si queremos hacer un mapa que relacione enteros (llaves) con strings (valores), la forma es la siguiente:

```
map<int, string> mapa;
```

```
mapa[1] = "hi";
mapa[4] = "hey";
mapa[5] = "hola";
mapa[1] = "hello";
```



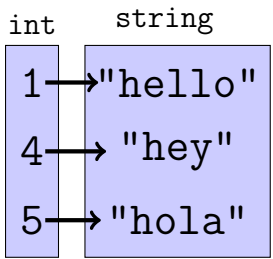
# Maps

## Ejemplo

Si queremos hacer un mapa que relacione enteros (llaves) con strings (valores), la forma es la siguiente:

```
map<int, string> mapa;
```

- Para buscar el valor de una llave, se usa `mapa[llave]`. Por ejemplo, `mapa[4]` devuelve `"hey"`.



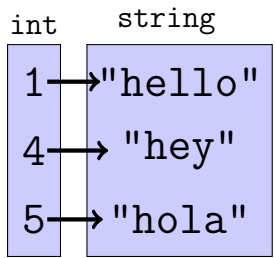
# Maps

## Ejemplo

Si queremos hacer un mapa que relacione enteros (llaves) con strings (valores), la forma es la siguiente:

```
map<int, string> mapa;
```

- Para los iteradores, se usa la función `.find(llave)`. Por ejemplo, `mapa.find(1)` nos devuelve el iterador asociado a la llave `1`. Con las llaves inexistentes, nos devolverá el iterador `mapa.end()`.



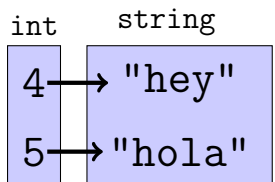
# Maps

## Ejemplo

Si queremos hacer un mapa que relacione enteros (llaves) con strings (valores), la forma es la siguiente:

```
map<int, string> mapa;
```

- Para borrar una llave y su valor, hay que usar la función `.erase(iterador)`. Por ejemplo, `mapa.erase(mapa.find(1))` nos deja con el siguiente `map`:



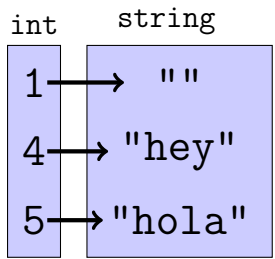
# Maps

## Ejemplo

Si queremos hacer un mapa que relacione enteros (llaves) con strings (valores), la forma es la siguiente:

```
map<int, string> mapa;
```

- *Obs.:* Si le pasamos el string vacío a una llave (por ejemplo: `mapa[1] = ""`), ésta **NO** se borrará, quedará como el mapa de la derecha:



# Y lo último...

- Para todas las estructuras vistas, existe el método `.size()` que nos entrega la cantidad de elementos que tiene (o tamaño).
- A diferencia de los `sets`, existen los `multisets` donde **sí** se pueden repetir elementos.