

# CS1103

## Programación Orientada a Objetos 2

### Unidad 3: Semana 6 - Templates y Metaprogramación

**Estanislao Contreras**  
**Rubén Rivas**

# Function Templates

# Ejercicio # 1

Existen 3 tipos de funciones de valor absoluto **abs**, **labs**, y **fabs**. Esas funciones se diferencian solamente por el tipo, sería mejor tener una sola función template para el valor absoluto que se llame **absolute**. de modo que pueda ser usada por tipos como int, long y double.

## Ejercicio # 2

Generar una función template **split\_range**, que reciba como parámetros un **contenedor** y un número **n** y permita dividir el contenedor en **n** contenedores, si el valor **n** no es múltiplo del tamaño del contenedor, los ítems restantes de la división deberán ser almacenados en el último contenedor. el resultado o valor de retorno deberá ser un contenedor que contenga los contenedores devueltos, siendo **vector** el contenedor por defecto, el template tendrá la posibilidad de personalizar en contenedor de resultado.

Ejemplo:

```
deque<int> v1 = {1, 2, 3, 4, 5, 6, 7};  
auto resultado = split_range(v1, 3);  
    // resultado[0] = {1, 2};  
    // resultado[1] = {3, 4};  
    // resultado[2] = {5, 6, 7};
```

## Ejercicio # 3

Escribir una función template **sumar\_rango** que permita la suma de los valores de 2 contenedores, por ejemplo.

```
vector<int> v1 = {1, 3, 4};
```

```
vector<int> v2 = {4, 5, 6};
```

```
auto v3 = sumar_rango(v1, v2); // {5, 8, 10}
```

si uno de los contenedores es de menor tamaño, el contenedor de menor tamaño se sumará repetidamente con otro contenedor hasta completar el tamaño del mayor. ejemplo

```
list<int> v1 = {1, 2, 3, 4, 5};
```

```
list<int> v2 = {10, 20};
```

```
auto v3 = sumar_rango(v1, v2); // {11, 22, 13, 24, 15}
```

# Ejercicio # 4

Escribir una función template **delete\_items** que permita eliminar un valor específico o una lista de valores de un contenedor secuencial:

**Ejemplo # 1:**

```
vector<int> v1 = {1, 3, 4, 1, 3, 2, 3, 4, 6, 5};  
auto v3 = delete_items(v1, 1); // {3, 4, 3, 2, 3, 4, 6, 5}
```

**Ejemplo # 2:**

```
list<int> v1 = {1, 3, 4, 1, 3, 2, 3, 4, 6, 5};  
auto v3 = delete_items(v1, {1, 4}); // {3, 3, 2, 3, 6, 5}
```

# Ejercicio # 5

Escribir una función template **delete\_duplicated** que permita eliminar todos los valores duplicados:

**Ejemplo # 1:**

```
vector<int> v1 = {1, 3, 4, 1, 3, 2, 3, 4, 6, 5};  
auto v3 = delete_duplicated(v1);    // {1, 3, 4, 2, 6, 5}
```

**Ejemplo # 2:**

```
list<int> v1 = {1, 1, 1, 1, 3, 2, 2, 2, 2, 5};  
auto v3 = delete_duplicated(v1);    // {1, 3, 2, 5}
```

# Ejercicio # 6

Escribir una función template **rotate\_range** que permita rotar los valores de un contenedor una cantidad determinada, esta función recibe 2 parametros un contenedor, y un valor entero, en caso sea positivo los valores rotará hacia la derecha, en caso negativo los valores rotaran a la izquierda:

**Ejemplo # 1:**

```
vector<int> v1 = {1, 2, 3, 4, 5, 6};  
auto v3 = rotate_range(v1, 2);           // {5, 6, 1, 2, 3, 4}
```

**Ejemplo # 2:**

```
list<int> v1 = {10, 20, 30, 40, 50};  
auto v3 = rotate_range(v1, -2);         // {30, 40, 50, 10, 20}
```



# Ejercicio # 7

La librería estándar de C++ brinda un contenedor simple conocido como `std::pair`, que permite almacenar solo 2 valores de diferentes. Su sintaxis es `std::pair<type1, type2>` ejemplo:

```
std::pair<int, string> p1 = {1321, "Jose Perez"};
```

Usualmente este tipo de container que se utiliza en `std::maps` o para 2 valores de una función, uno de las desventajas es que para acceder a los dos valores `std::pair` se utilizan 2 campos (`first` & `second`) que usualmente no tienen un significado en el contexto de un problema.

```
std::cout << p1.first << " " << p1.second;
```

De otro lado lenguajes como python brindan un mecanismo conocido como `unpack` que permite asignar valores de un contenedor a variable con nombres significativos.

Desarrollar un function template que permite asignar los valores de un `std::pair` directamente a variables con nombres significativos, Ejemplo.

```
int key; string name;  
unpack(key, name) = p1;  
std::cout << key << " " << name;
```

# Ejercicio # 8

Extender la función **unpack** del ejercicio #10 para la estructura `std::tuple` (investigar el funcionamiento de `tuple`):

```
std::tuple<int, string, string, double> t1 = {1321, "Jose", "Perez", 1.68};
```

El acceso no tiene valores significativos.

```
std::cout << std::get<0>(t1) << " "  
          << std::get<1>(t1) << " " << std::get<2>(t1) << " "  
          << std::get<3>(t1);
```

De otro lado lenguajes como python brindan un mecanismo conocido como **unpack** que permite asignar valores de un contenedor a variable con nombres significativos.

Ejemplo de **unpack** usando `tuple`:

```
int key; string first_name; string last_name; double height;  
unpack(key, first_name, last_name, height) = t1;  
std::cout << key << " " << first_name << last_name << height;
```

# Ejercicio # 9

Escribir un function template que permita leer un número de parámetros variados y que genere a partir de esa listas de parámetros un vector (por defecto) o un contenedor seleccionado.

Ejemplo:

```
auto c1 = generar_contenedor(1, 2, 3, 4);  
for (const auto& item: c1)  
    std::cout << item << " ";  
std::cout << endl;
```

```
auto c2 = generar_contenedor<list>(10, 20, 30, 40);  
for (const auto& item: c1)  
    std::cout << item << " ";  
std::cout << endl;
```

# Ejercicio # 10

Diseñar un function template que permita leer un número variado de contenedores de un mismo tipo y que retorne el tamaño mínimo.

Ejemplo:

```
vector<int> v1 = { 11 };  
vector<int> v2 = { 21, 22, 23, 1, 2 };  
vector<int> v3 = { 31, 32, 33, 4 };  
  
cout << min_size(v2, v3) << endl;           // 4  
cout << min_size(v1, v2, v3) << endl;       // 1
```

# Ejercicio # 11

En python existe una función denominada zip que permite recibir un número variado de contenedores del mismo tipo y agrupar los valores de una misma fila generando tuplas por cada fila con los valores de cada contenedor.  
Generar un function template similar, pero en vez de tuplas genere un vector con los valores correspondientes de cada fila

Ejemplo:

```
list<int> v1 = { 11, 12, 13 };  
list<int> v2 = { 21, 22, 23 };  
list<int> v3 = { 31, 32, 33 };  
auto result = zip(v1, v2, v3);  
for (const auto& row : result) {  
    for (const auto& value : row)  
        cout << value << " ";  
    cout << endl;  
}
```

El tipo de **result** sería un **list<vector<int>>** y lo impreso sería:

```
11 21 31  
12 22 32  
13 23 33
```

# Ejercicio # 12

Escribir un template para la versión iterativa de la búsqueda binaria.

# Ejercicio # 13

Escribir un template para la versión recursiva de la búsqueda binaria.

## Ejercicio # 14

- Generar la función **sum\_product** de cantidad de parámetros variado y que realice la multiplicación de los parámetros pares con los parámetros impares y que devuelva la suma de todos estos productos.



# **Class Templates**

# Ejercicio # 1

Definir una estructura genérica Linked List que cuente con los siguientes métodos básicos:

```
void push_front(T item); // Agrega un item en el frente
void push_back(T item);  // Agrega un item al final
T pop_front();           // Remueve un item del frente y
                        // lo retorna
T pop_back();            // Remueve un item del final y
                        // lo retorna
T front();               // Retorna el valor del frente
T back();                // Retorna el valor del final
```

## Ejercicio # 2

Definir una estructura genérica max-heap que cuente con los siguientes métodos básicos:

```
T find_max();           // Busca el valor mayor
void insert(T item);    // Inserta un item al heap
void delete();          // Remueve un item del final y
                        // lo retorna
void print();           // lista todos los elementos
                        // del heap
```

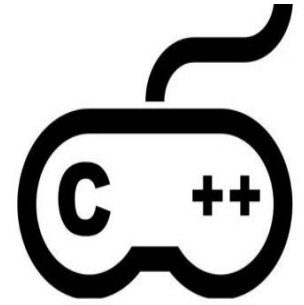
# Explorando lo aprendido

---

- ¿Qué es el variadic template?
- ¿Por qué es útil?
- ¿Que es metaprogramación?
- ¿Cuales son los mecanismos de la metaprogramación?



# Bibliografía:



- **C++ Templates, The Complete Guide; 2018;** David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor
- **C++ Primer, Fifth Edition; 2013;** Stanley B. Lippman, Josée Lajoie, Barbara E. Moo