



**CS2013**

**Programación III**

**Unidad 2 - Semana 3 - Templates y  
Metaprogramación**

Rubén Rivas

---

# Objetivos

Al finalizar la sesión, los alumnos podrán entender las características básicas de la **programación genérica**, el uso de los **templates**, sus **especializaciones**, los **variadic template**, el uso de los **constexpr** y un ejemplo básico de **meta programación** en el lenguaje C++ .



# Temas semana 3 - templates y meta programación

- 1.Programación genérica
- 2.Templates
- 3.Variadic templates
- 4.Constexpr
- 5.Meta programación



# Programación genérica

¿Qué es programación genérica?



# ¿Qué es programación genérica?

Es un estilo de programación en donde los **algoritmos** y **estructuras** son **escritos** en "**tipos**" que serán **definidos posteriormente** y que se instancian cuando se provee el tipo a través de un **parámetro**.



# Programación genérica

¿Cuáles son sus ventajas?



## ¿Cuáles son sus ventajas?

- **Reduce el tamaño** del código fuente.
- **Verificación estática en tiempo de compilación**, por lo tanto genera código binario más **optimizado**.
- **Reduce el casting** entre variables.
- Permite **mejorar la abstracción** del programa con **algoritmos genéricos independientes del tipo**.
- Son muy útiles combinándolos con **herencia y sobrecarga**.





# Programación genérica

¿Cuáles son sus desventajas?



## ¿Cuáles son sus desventajas?

- **Sintaxis compleja** que reduce la claridad del código.
- Reduce el **tamaño** del **código fuente** eso no asegura que el código binario sea pequeño.
- Los **mensajes de error** son **pocos comprensibles**.
- La **meta programación** define un estilo muy diferente a la **sintaxis heredada del lenguaje C**, por lo que se debe lidiar con “dos lenguajes” dentro de un mismo lenguaje.



# Herramientas en C++

¿Qué herramientas nos brinda C++?



# Herramientas en C++

- La principal herramienta que brinda C++ son los templates:
  - Templates de funciones
  - Templates de clases
  - Templates de variables
  - Templates de alias
- Los templates se complementan con herramientas como **lambda** que permiten desarrollar funciones un estilo **static duck typing** y herramientas de meta programación como las expresiones **constexpr**.



# Templates en C++

## Template de funciones

```
template <typename T>
void template_funcion (T param1)
{
    cout << param1 << endl;
}
```

*Cabecera de función*

## Template de clases

```
template <typename T>
class clase_template
{
    T attr1;
public:
    clase_template (T par1):
        attr1{par1} {}
};
```

*Cabecera de clase*

*Cabecera de template*

# Plantillas de Clases (Clases Genéricas)

- Sintaxis:

```
template <class|typename <id>[,...]>  
class <identificador_de_plantilla>  
{  
    // Declaración de funciones  
    // y datos miembro de la plantilla  
};
```



# Templates en C++

## Template de variables

```
template <typename T>  
T variable = T{3.1415};
```

*Cabecera de template*

## Template de alias

```
template <typename T>  
using tipoCola = queue<T>;
```

# Parámetros de template

- **Parámetros No Tipo**


```
template <typename T, double& PI>
T calculate_radius(T radio) {
    return radio * radio * PI;
}
```

- **Parámetros Tipo**

```
template <typename WType, typename HType>
auto calculate_area(WType w, HType h) {
    return w * h;
}
```

- **Parámetros Template**

```
template <typename T, template<typename> class CType>
CType<T> template_funcion (T size) {
    return CType<T>(size);
}
```





# Plantillas de Funciones (Funciones Genéricas)

## funciones template

```
template <typename T>
T max (T a, T b) {
    return a>b? a: b;
}

template <typename T1, typename T2>
auto max (T1 a, T2 b) {
    return a>b? a: b;
};
```

```
const char*max (const char* a,
                const char* b) {
    return strcmp(a,b) > 0? a: b;
}
```



# Plantillas de Clases (Clases Genéricas)

## clase template – alternativa 1 – declaración y definición separadas

```
// Declaración de template
template <class T>
class equipo_t
{
    T *data;
    int cupos; // Cupos del Equipo
    int convocados; // Numero del Convocado
public:
    equipo_t(int cupos);
    ~equipo_t();
    void add_jugador(T jugador);
    void lista_convocados();
};
```

```
// Definición de template
template <class T> equipo_t<T>::equipo_t(int cupos):
    cupos{cupos}, convocados{0} { data = new T[cupos]; }

template <class T> equipo_t<T>::~~equipo_t() { delete [] data; }

template <class T> void equipo_t<T>::addJugador(T jugador) {
    pT[convocados++] = jugador;
}

template <class T> void equipo_t<T>::listaConvocados() {
    cout<<" Convocados : "<<endl;
    for(int i=0;i<convocados;i++)
        cout<<(i+1)<<". "<<pT[i]<<" "<<endl;
    cout<<endl;
}
```



# Plantillas de Clases (Clases Genéricas)

## clase template – alternativa 1 – declaración y definición juntas

```
// Declaración y definición de template
template <class T>
class equipo_t
{
    T *data;
    int cupos; // Cupos del Equipo
    int convocados; // Numero del Convocado
public:
    equipo_t(int cupos): cupos{cupos}, convocados{0} { data = new T[cupos]; }
    ~equipo_t() { delete [] data; }
    void add_jugador(T jugador) {pT[convocados++] = jugador;}
    void lista_convocados() {
        cout<<" Convocados : "<<endl;
        for(int i=0;i<convocados;i++)
            cout<<(i+1)<<". "<<pT[i]<<" "<<endl;
        cout<<endl;
    }
};
```



# Plantillas de Clases (Clases Genéricas) ...cont

## Ejecución del programa

```
struct Futbolista {  
    string nombre;  
};  
  
ostream& operator<<(ostream &os,  
                    const Futbolista& f ) {  
    return os << f.nombre;  
}
```

```
int main() {  
  
    Equipo<int> cartas = Equipo<int>(24);  
    Equipo<Futbolista> futbol = Equipo<Futbolista>(18);  
    Futbolista paolo {"Paolo Guerrero"};  
    Futbolista cristian {"Cristian Cueva"};  
  
    cartas.addJugador(4);  
    cartas.addJugador(12);  
    cartas.listaConvocados();  
  
    futbol.addJugador(paolo);  
    futbol.addJugador(cristian);  
    futbol.listaConvocados();  
}
```



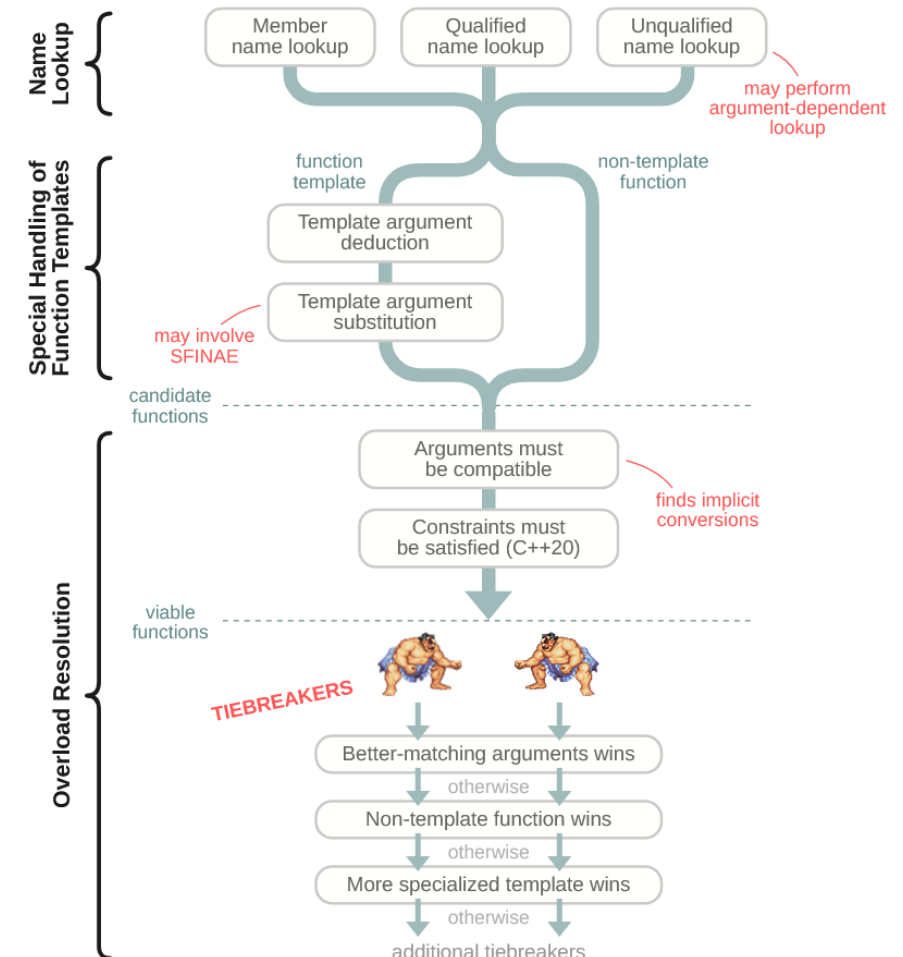
# Características de los templates

- La declaración e implementación debe realizarse **siempre** en archivos cabecera (header), y en caso de clases se sugiere que sean dentro de declaración.
- La instanciación de una clase template requiere del llamado **explícito del tipo base**, no existe una **deducción de tipo** como en el caso de **template de función**.
- Los **templates de clases** se cuenta con el mecanismo de especialización (total y parcial).
- Los **templates de funciones** no cuentan con especialización pero consiguen el mismo efecto usando la sobrecarga.



# Sobrecarga de funciones y templates de funciones

- **Name Lookup**, se determina las funciones validas basado en el ámbito de una función.
- **Special Handling of Function Templates**, Basado en los parámetros trata en 2 pasos de convertir el **template de función** en una **función** y determina las funciones candidatas.
- **Overload Resolution**, aplica las reglas regulares de selección de sobre carga. Seleccionando inicialmente **las funciones viables**. En caso de que aun exista mas de una función viable se procede a resolver el conflicto entre las funciones viables a través de 3 pasos.



# Plantillas de clases - Especialización Total

```
// Template Base
template <typename T>
class MyVector {
    T* arr;
    size_t n;
public:
    size_t size();
};
template <typename T>
size_t MyVector<T>::size() {
    return n;
}
```

```
// Especialización Total
template <>
class MyVector<int> {
    int* arr;
    size_t n;
public:
    size_t size();
};
template <>
size_t MyVector<int>::size() {
    return n;
}
```



# Plantillas de clases - Especialización Parcial

```
// template base
template <typename T, typename U, typename W>
class myClass {
public:
    size_t size();
};

// especializacion parcial
template <typename T, typename U>
struct myClass<T, U, int> {
    size_t size();
};

// especializacion parcial
template <typename T>
struct myClass<T, T*, double> {
    size_t size();
};
```

```
// template base
template <typename T>
class MyVector {};
```

```
// especialización parcial
template <typename T>
class MyVector <T*> {
    T** arr;
    size_t n;
public:
    size_t size(); // declaración
};

// definición
template <typename T>
size_t MyVector <T*>::size() {
    return n;
}
```





# Polimorfismo Dinamico

- Mejora el nivel de abstracción
- Permite generalizar estructuras de datos, pero se requiere de una jerarquía de clase.
- Permite especializar estructuras a través de la derivación de la clase base.
- Hace uso del tiempo de ejecución para elegir la implementación permitiendo tomar decisiones dinámicamente(dynamic dispatch)
- Se focaliza en las interfaces.



# Programación Genérica

- Mejora el nivel de abstracción
- Permite generalizar estructuras de datos, generalizando los tipos a través de los argumentos de un template
- Logra la especialización por medio de la sobrecarga de funciones templates y especialización de clases template.
- Hace uso del tiempo de compilación, permitiendo reducir tiempos de ejecución.
- No solo se focaliza en las interfaces sino también en los algoritmos.



Mecanismo	Generalización	Especialización
Herencia y Polimorfismo	Clase base, funciones virtuales y abstractas	Clase derivada sobreescritura (override)
Template de funciones	Template Base	Sobrecarga <i>(especialización)</i>
Template de clases	Template Base	Especialización total Especialización parcial

# Variadic template

- Template que recibe un **número indeterminado** de parámetros a través de lo que se conoce como **paquete de parámetros**.
- El **paquete de parámetros** puede incluir cualquiera de los 3 tipos de parámetros de template.

```
template<int... ParamsPack> // parametros no-tipo
```

```
template<typename... ParamsPack> // parametros tipo
```

```
template<template<class...> class... ParamsPack> // parametros template template
```

# Variadic template – operador ... (ellipsis)

- El operador **ellipsis** (...) se utiliza para representar **paquetes de parámetros** y similar al caso de otros operadores suele ser sobrecargado para diferentes propósitos.
- Definir **paquete de parámetros de template** deberá ir delante del nombre del parámetro (ver slide anterior).
- Definir un **paquete de parámetros de función**, debe ir delante del nombre del parámetro de función:

```
template<class... Types> void funcion(Types ...param); // parametro de función
```

- Desempaquetar un **paquete de parámetros de función**, debe ir después del nombre del parámetro de función:

```
funcion2(param...); // desempaquetando parametros
```



# Operator sizeof...

Permite determinar el número de parámetros empaquetados

Ejemplo:

```
template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
    std::cout << sizeof...(Types) << '\n';
    std::cout << sizeof...(args) << '\n';
}
```

# Operator sizeof...

Permite determinar el número de parámetros empaquetados

Ejemplo:

```
template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
    std::cout << sizeof...(Types) << '\n';
    std::cout << sizeof...(args) << '\n';
}
```

# Fold expressions C++17

Fold expression	Evaluation
<code>( ... op pack )</code>	<code>(( ( pack1 op pack2 ) op pack3 ) ... op packN )</code>
<code>( pack op ... )</code>	<code>( pack1 op ( ... ( packN-1 op packN ) ) )</code>
<code>( init op ... op pack )</code>	<code>(( ( init op pack1 ) op pack2 ) ... op packN )</code>
<code>( pack op ... op init )</code>	<code>( pack1 op ( ... ( packN op init ) ) )</code>



# Ejercicio # 1

- Generar la función **suma** realice la suma de todos sus parámetros.

# print

```
template<typename... Types>
void print(Types... args) {
    (cout << ... << args);
}

print("El resultado es: ", 10 + 20);
```

## Ejercicio # 2

- Generar la función **comparar** donde la cantidad de parámetros sea variada y que realice la comparación de los parámetros pares con los parámetros impares y devuelva true si todos los pares son iguales con los impares y false en caso contrario.

## Ejercicio # 3

- 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.

# Metaprogramación

- Realizar cálculos u operaciones en tiempo de compilación
- La herramienta usual de desarrollo de la metaprogramación en C++ ha sido los templates.
- Utilizando técnicas de recursividad aprovechando la capacidad de las clases template de poder especializarse.

# constexpr

- Aunque C++ 11, introduce el concepto de expresiones literales, C++17 extiende su uso mas generico.
- Pudiéndose ejecutar funciones en tiempo de compilación.
- Se definen ciertas restricciones en tiempo de compilación por ejemplo los valores de retorno y parámetros sean valores literales o que las funciones no incluyen bloques try/catch.

# factorial - C++ 11

```
template <size_t N>  
constexpr size_t fact1() { return N*fact1<N-1>(); }  
  
template <>  
constexpr size_t fact1<0>() { return 1; }
```

# factorial - C++ 17

```
constexpr int fact2(int n) {  
    return n <= 1 ? 1 : (n * fact2(n - 1));  
}
```



# Ejercicios - Template funciones

4. Write a template function that swaps the value of two data types, test with float and integer.

1. Change the types to pass by reference
2. Change the types to pass by address

5. Create the C++ Function Template named multiples so that it has three parameters sum, x, and n. The first two parameters will have the type represented by the function template type parameter WhatKind. n will always be int. The return type is void. All parameters are passed by value except for sum which is passed by reference. A Template Function created from multiples will compute...

$$\text{sum} = 1 + x + 2x + 3x + \dots + nx$$

6. Create the C++ Function Template named init so that it has three parameters whose types are determined by the function template type parameters T1 and T2. The function header is shown below. init sets the value of the parameter start to a T2-type value of 1. init returns a T1-type value which is the sum of num1 and num2.

T1 init (T1 num1, T1 num2, T2& start)

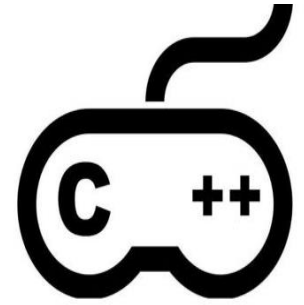
# Explorando lo aprendido

---

- ¿Qué es la programación genérica?
- ¿Mencione una ventaja?
- ¿Mencione una desventaja?
- ¿Qué es un template? Mencione los 4 tipos
- ¿Cuántos tipos de parámetros template hay en C++?
- ¿Que mecanismos permiten especializar un template en C++?
- ¿Qué es el variadic template?



# 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