# Funciones en c++

**1. ¿Qué es una función?**
Una función en C++ es un bloque de código que realiza una tarea específica. Las funciones permiten dividir un programa en partes más manejables y reutilizables, mejorando la legibilidad y facilitando el mantenimiento del código.

**2. Sintaxis de una Función**

La sintaxis básica de una función en C++ es la siguiente:

```
tipo_de_retorno nombre_de_la_función(parámetros) {
     // cuerpo de la función
     return valor; // opcional si el tipo de retorno no es void
}
```

**tipo_de_retorno**: Especifica el tipo de dato que la función devolverá. ***Si la función no devuelve un valor, se usa void***.
**nombre_de_la_función:** Es el nombre dado a la función. **Debe ser único y descriptivo**.
**parámetros**: Son los valores de entrada que la función recibe y utiliza para realizar su tarea. Son opcionales.
**cuerpo de la función**: Contiene las declaraciones y sentencias que definen lo que la función hace.
**return**: Se usa para devolver un valor desde la función al lugar donde fue llamada. Solo se usa si la función tiene un tipo de retorno distinto de **void**.

**3. Declaración y Definición de Funciones**

**Declaración:** Informa al compilador que la función existe, pero no proporciona la implementación. Se usa generalmente cuando la función está definida después de ser llamada o en otro archivo.

```
int sumar(int, int);
```
**Definición:** Proporciona la implementación real de la función.

```
int sumar(int a, int b) {
     return a + b;
}
```

**4. Tipos de Funciones**

a.-Funciones **SIN** parámetros y **sin** valor de retorno:

```
void saludar() {
     std::cout << "¡Hola!" << std::endl;
}
```

b.-Funciones **CON** parámetros y **con** valor de retorno:

```
int multiplicar(int a, int b) {
     return a * b;
}
```

c.-Funciones **con** parámetros por valor y por referencia:

```
void intercambiar(int &a, int &b) {
     int temp = a;
     a = b;
     b = temp;
}
```

**5. Parámetros por Valor y por Referencia**

**Por valor**: Se *pasa una copia del valor al parámetro* de la función. ***Cualquier cambio hecho dentro de la función*** *no afecta la variable original*.

```
void incrementar(int a) {
     a++;
}
```

**Por referencia**: Se pasa la referencia a la variable. *Cualquier cambio afecta la variable original*.

```
void incrementar(int &a) {
     a++;
}
```

# Ampliación del Tema

**1. Paso por Valor**

*¿Qué es el paso por valor?*

En el paso por valor, cuando llamas a una función con argumentos, se pasa una copia de los valores de las variables originales a la función. Esto significa que cualquier modificación que se haga a los parámetros dentro de la función no afectará a las variables originales.

Ejemplo:

```
#include <iostream>

void cambiarValor(int x) {
     x = 10; // Modifica solo la copia local de x
}

int main() {
     int a = 5;
     cambiarValor(a);
     std::cout << "Valor de a: " << a << std::endl; // a sigue siendo 5
     return 0;
}
```

**Explicación:**

En este ejemplo, se define una función cambiarValor que toma un parámetro int x.
Cuando se llama a cambiarValor(a) en la función main, el valor de a (que es 5) se copia en x.
Dentro de cambiarValor, se cambia el valor de x a 10, pero esto no afecta a a en main porque x es solo una copia.
Al imprimir a en main, el valor sigue siendo 5.

**Ventajas y Desventajas del Paso por Valor:**

**Ventajas:**

**La seguridad:** La función no puede modificar los valores originales.
Útil para tipos de datos pequeños (como int, char, etc.) donde la sobrecarga de copia es mínima.

**Desventajas:**

**Ineficiencia**: Para estructuras de datos grandes, como *objetos o arrays*, copiar datos puede ser costoso en términos de tiempo y memoria.

**2. Paso por Referencia**

**¿Qué es el paso por referencia?**

En el paso por referencia, ***en lugar de pasar una copia del valor, se pasa una referencia a la variable original***. Esto significa que **cualquier modificación** realizada a los parámetros dentro de la función afectará directamente a las variables originales.

Ejemplo:

```
#include <iostream>

void cambiarValor(int &x) {
     x = 10; // Modifica el valor original de x
}

int main() {
     int a = 5;
     cambiarValor(a);
     std::cout << "Valor de a: " << a << std::endl; // a ahora es 10
     return 0;
}
```

**Explicación:**

En este ejemplo, la función cambiarValor toma un parámetro int &x. El símbolo & indica que x es una referencia a una variable.
Cuando se llama a cambiarValor(a) en main, no se copia el valor de a; en su lugar, x se convierte en una referencia directa a **a**.
Dentro de cambiarValor, cambiar el valor de x a 10 modifica directamente a.
Al imprimir a en main, el valor ahora es 10.

**Ventajas y Desventajas del Paso por Referencia:**

**Ventajas:**

**Eficiencia:** No se hace una copia de la variable, lo que es más eficiente para estructuras de datos grandes.
Permite modificar directamente las variables originales desde la función.

**Desventajas:**

**Menos seguro:** La función puede modificar las variables originales, lo que podría llevar a errores si no se maneja con cuidado.

**3. Comparación: Paso por Valor vs. Paso por Referencia**

**Modificación de Variables Originales:**

**Paso por Valor:** No modifica las variables originales.
**Paso por Referencia:** Puede modificar las variables originales.

**Uso de Memoria:**

**Paso por Valor:** Puede ser menos eficiente para datos grandes debido a la copia.
**Paso por Referencia:** Más eficiente, ya que no se hacen copias.

**Seguridad:**

**Paso por Valor:** Es más seguro porque protege las variables originales.
**Paso por Referencia:** Menos seguro, ya que permite modificaciones directas.

**4. Paso por Puntero (Una Variante del Paso por Referencia)**
Otra forma de pasar una referencia en C++ es usando punteros. Al pasar un puntero a una función, se permite que la función modifique el contenido de la dirección de memoria apuntada.

Ejemplo:

```
#include <iostream>

void cambiarValor(int *x) {
     *x = 10; // Modifica el valor en la dirección de memoria de x
}

int main() {
     int a = 5;
     cambiarValor(&a);
     std::cout << "Valor de a: " << a << std::endl; // a ahora es 10
     return 0;
}
```

**Explicación:**

En este caso, cambiarValor toma un puntero int *x.
Cuando se llama a cambiarValor(&a), se pasa la dirección de a a x.
Dentro de la función, *x = 10 *modifica el valor en la dirección de memoria de x*, que es la dirección de a.

**Resumen**

**-Paso por Valor** es útil cuando no se necesita modificar la variable original y es más seguro, pero puede ser ineficiente para datos grandes.
**-Paso por Referencia** permite modificar directamente las variables originales, siendo más eficiente para datos grandes, pero requiere un manejo cuidadoso para evitar errores.
**-Paso por Puntero** es otra forma de pasar referencias, comúnmente utilizada en el manejo de estructuras dinámicas de datos.

**6. Funciones Inline**

Las funciones **inline** son aquellas en las que *el compilador intenta insertar el código de la función directamente en el lugar donde se llama*, para evitar la sobrecarga de una llamada a función.

```
inline int cuadrado(int x) {
     return x * x;
}
```

```cpp
#include <iostream>

inline int min(int x, int y) // La palabra clave inline significa que esta función es una función en línea
{
	return (x < y) ? x : y;
}

int main()
{
	std::cout << min(5, 6) << '\n';
	std::cout << min(3, 2) << '\n';
	return 0;
}
```

**7. La recursividad**

La recursividad es una técnica de programación en la que *una función* se llama a sí misma para resolver un problema *dividiéndolo en subproblema*s más pequeños del mismo tipo. La recursividad es muy útil para problemas que tienen una estructura naturalmente repetitiva, **como los árboles**, **gráficos, y problemas matemáticos como factoriales**, **secuencias de Fibonacci**, y algoritmos como el **de Euclides para el máximo común divisor.**

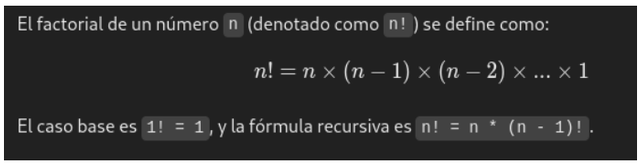
***¿Cómo funciona la recursividad?***

En cada llamada recursiva, se trabaja en una versión más pequeña del problema original. La clave de la recursividad es:

**Condición base (o caso base):** El caso base *define cuándo la función recursiva debe detenerse*. Esto es esencial para evitar llamadas recursivas infinitas.
**Llamada recursiva**: La función *se llama a sí misma con una versión más pequeña* o simplificada del problema.
Estructura básica de una función recursiva

```cpp
// Función recursiva genérica
TipoDeRetorno funcionRecursiva(Parametros) {
	if (CondicionBase) {
		// Caso base: resolver y retornar directamente
		return ResultadoBase;
	} else {
		// Llamada recursiva con problema más pequeño
		return funcionRecursiva(ParametrosModificados);
	}
}
```

**Ejemplo 1: Factorial de un número**
El factorial de un número n (denotado como n!) se define como:




**Código en C++:**

```cpp
#include <iostream>
using namespace std;

// Función recursiva para calcular el factorial
int factorial(int n) {
	// Caso base: factorial de 0 o 1 es 1
	if (n == 0 || n == 1) {
		return 1;
	}
	// Llamada recursiva
	return n * factorial(n - 1);
}

int main() {
	int num;
	cout << "Introduce un número: ";
	cin >> num;

	// Llamada a la función recursiva
	cout << "El factorial de " << num << " es: " << factorial(num) << endl;
```

```
    return 0;
}
```

**Explicación del proceso de recursión:**

**Llamada inicial:** *Si llamas factorial(5), la función no cumple con el caso base, por lo que llama a factorial(4).*
**Siguiente nivel de recursión**: *La función factorial(4) se llama y, de nuevo, no cumple con el caso base, por lo que llama a factorial(3)*.
**Continúa hasta el caso base**: Esta cadena de llamadas continúa hasta llegar a factorial(1), que retorna 1.
**Resolviendo el problema**: Luego, las llamadas se van resolviendo en orden inverso. factorial(2) retorna 2 * factorial(1), que es 2 * 1 = 2, y así sucesivamente hasta que factorial(5) retorne 120.

**Ejemplo 2**: ***Secuencia de Fibonacci***
La secuencia de Fibonacci es otro ejemplo clásico de recursividad. La secuencia se define como:

F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2) para n > 1

Esto significa que el número en la posición n es la suma de los dos números anteriores en la secuencia.

**Código en C++**:

```
#include <iostream>
using namespace std;

// Función recursiva para calcular el enésimo número de Fibonacci
int fibonacci(int n) {
    // Caso base: F(0) = 0 y F(1) = 1
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // Llamada recursiva
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int num;
    cout << "Introduce la posición en la secuencia de Fibonacci: ";
    cin >> num;

    // Llamada a la función recursiva
    cout << "El número Fibonacci en la posición " << num << " es: " << fibonacci(num) << endl;

    return 0;
}
```

**Funcionamiento de la recursividad en Fibonacci**:
Si se llama a fibonacci(5):

Se llama a fibonacci(4) y fibonacci(3).
Cada una de esas llamadas, a su vez, llama a las funciones con fibonacci(3), fibonacci(2), etc.
Eventualmente, las llamadas llegan al caso base, donde se resuelven los valores fibonacci(1) y fibonacci(0), y después las funciones empiezan a retornar sumas hasta que se obtiene el resultado.
Importancia de la condición base
Es fundamental que toda función recursiva tenga una condición base, porque sin ella la recursión no tendría un punto de parada, lo que llevaría a una recursión infinita y, eventualmente, a un desbordamiento de *pila (stack overflow)*.

**Recursividad frente a iteración**
**Recursividad**: Es elegante y se parece a la definición matemática del problema. Sin embargo, consume más memoria porque cada llamada recursiva se almacena en la pila de llamadas.
**Iteración**: Una alternativa a la recursión, es más eficiente en términos de memoria porque no requiere múltiples llamadas anidadas. Un problema recursivo como la secuencia de Fibonacci o el factorial puede resolverse también de manera iterativa.

Ejemplo iterativo del factorial:

```
int factorial_iterativo(int n) {
    int resultado = 1;
```

```
    for (int i = 1; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}
```

**Ventajas y desventajas de la recursividad**
**Ventajas:**
**Simplicidad**: Hace que el código sea más limpio y fácil de entender para problemas naturalmente recursivos.
**Adecuado para problemas complejos**: Algunos algoritmos *como el recorrido de árboles binarios* o la *búsqueda en profundidad son más fáciles de implementar recursivamente*.
**Desventajas**:
**Uso de memoria**: Cada llamada recursiva se almacena en la pila de ejecución. Si las llamadas son demasiadas (especialmente en problemas grandes), puedes agotar la memoria.
**Menor eficiencia**: Puede ser más lento que las versiones iterativas debido a la sobrecarga de las llamadas a funciones.
**Optimización de la recursividad**: Memorization o Programación Dinámica
En problemas donde una solución recursiva recalcula los mismos valores múltiples veces (como Fibonacci), se puede optimizar usando memorización, donde los resultados de las sub-problemas ya resueltos se almacenan en una tabla para evitar recalculaciones.

**El algoritmo de Euclides** es uno de los ejemplos más clásicos de recursividad. Su propósito es encontrar el máximo común divisor (MCD) de dos números, es decir, el número más grande que divide a ambos sin dejar residuo(RESTO). Este algoritmo es particularmente eficiente y tiene una implementación recursiva sencilla y elegante.

## ¿Cómo funciona el Algoritmo de Euclides?

El algoritmo de Euclides se basa en una observación clave:

El MCD de dos números a y b (con a >= b) es igual al MCD de b y el residuo de la división de a entre b. Es decir:

$$\text{MCD}(a, b) = \text{MCD}(b, a\%b)$$

Si b es 0, entonces el MCD es a. Este es el caso base de la recursión.

**Desarrollo paso a paso del algoritmo de Euclides recursivo**:
Caso base: Si b == 0, entonces el MCD es a, porque cualquier número dividido por 0 tiene como MCD al número mayor.
**Llamada recursiva:** Si b != 0, se llama a la función recursivamente con los valores b y a % b.

Código en C++
Aquí tienes la implementación recursiva del algoritmo de Euclides en C++:

```
#include <iostream>
using namespace std;

// Función recursiva para calcular el MCD usando el Algoritmo de Euclides
int mcd(int a, int b) {
    // Caso base: si b es 0, el MCD es a
    if (b == 0) {
        return a;
    }
    // Llamada recursiva
    return mcd(b, a % b);
}

int main() {
    int num1, num2;

    // Solicitar dos números al usuario
    cout << "Introduce el primer número: ";
    cin >> num1;
    cout << "Introduce el segundo número: ";
    cin >> num2;

    // Llamar a la función recursiva para calcular el MCD
    cout << "El MCD de " << num1 << " y " << num2 << " es: " << mcd(num1, num2) << endl;

    return 0;
}
```

**Ejemplo de Ejecución:**
Imagina que deseas calcular el MCD de 48 y 18. *A continuación se detalla cómo se ejecuta la recursión:*

**Primera llamada**: mcd(48, 18):

48 % 18 = 12, por lo que se llama recursivamente a mcd(18, 12).

**Segunda llamada**: mcd(18, 12):

18 % 12 = 6, por lo que se llama recursivamente a mcd(12, 6).

**Tercera llamada: mcd(12, 6)**:

12 % 6 = 0, por lo que se llama recursivamente a mcd(6, 0).

**Cuarta llamada (caso base)**: mcd(6, 0):

Como b == 0, la función retorna 6, que es el MCD.

Finalmente, el resultado 6 se devuelve a través de todas las llamadas recursivas y se muestra como el MCD de 48 y 18.

**Análisis detallado:**
**Condición base**: La función deja de llamarse a sí misma cuando `b == 0`, lo cual es necesario para detener la recursión.
**Caso recursivo**: La función sigue llamándose con un problema más pequeño en cada iteración (`b` y `a % b`), lo que garantiza que eventualmente alcanzará el caso base.

**Ventajas del algoritmo recursivo de Euclides**:
**Simplicidad**: El algoritmo es muy simple y sigue directamente la definición matemática del MCD.
**Eficiencia**: Es extremadamente eficiente. El número de llamadas recursivas está relacionado con el tamaño de los números, pero el tamaño se reduce rápidamente debido al uso de la operación módulo (`a % b`).
**Usa propiedades matemáticas fundamentales**: El algoritmo se basa en el hecho de que el MCD de dos números no cambia si el mayor de los dos se reemplaza por su residuo al dividir por el menor.
Comparación con una versión iterativa:

Aunque la versión recursiva es elegante, el mismo problema puede resolverse de manera iterativa, eliminando las llamadas recursivas y reemplazándolas con un bucle. Aquí está la versión iterativa del algoritmo de Euclides:

```
int mcd_iterativo(int a, int b) {
     while (b != 0) {
          int temp = b;
          b = a % b;
          a = temp;
     }
     return a;
}
```

**Diferencias entre recursión e iteración**:
**Recursión**: Puede ser más intuitiva para problemas como el MCD, ya que sigue la lógica matemática paso a paso, pero *consume más memoria debido al uso de la pila de llamadas*.
**Iteración**: Puede ser más eficiente en cuanto al uso de memoria porque no necesita crear múltiples marcos de pila (stack frames), pero puede no ser tan clara en su lógica.
**Aplicaciones del algoritmo de Euclides**:
Reducción de fracciones: *El MCD se utiliza para simplificar fracciones*. Por ejemplo, para simplificar la fracción 48/18, el MCD de 48 y 18 es 6, por lo que la fracción puede reducirse a

$$\frac{48}{18} = \frac{8}{3}.$$

**Algoritmos de criptografía**: El algoritmo de Euclides es clave en la criptografía, especialmente en el cálculo de inversos modulares, que es un concepto central en algoritmos como RSA.
Mínimo común múltiplo (MCM): Como hemos visto antes, el MCD se utiliza para calcular el MCM usando la fórmula:

$$\text{MCM}(a, b) = \frac{|a \times b|}{\text{MCD}(a, b)}$$

**Optimización de la recursividad con "Tail Recursion" (Recursión en la cola)**:
Algunas funciones recursivas pueden optimizarse para utilizar recursión en la cola. Este tipo de recursión ocurre cuando la llamada recursiva es la última operación en la función, permitiendo que el compilador optimice la memoria reutilizando el marco de la pila en lugar de crear uno nuevo.

Aunque el algoritmo de Euclides ya es eficiente, su versión recursiva es un buen candidato para esta optimización.

En C++, muchos compiladores modernos ya optimizan las funciones de "tail recursion", pero la versión iterativa sigue siendo más directa en cuanto a eficiencia.

**8. Sobrecarga de Funciones**

C++ permite tener múltiples funciones con el mismo nombre pero con diferentes tipos o números de parámetros. Esto se llama sobrecarga de funciones.

```
int sumar(int a, int b) {
	return a + b;
}

double sumar(double a, double b) {
	return a + b;
}
```

**9. Funciones Lambda**
Las funciones lambda son funciones anónimas que se pueden definir en línea y se utilizan principalmente en funciones como argumentos.
Una expresión lambda (también llamada lambda o cierre) nos permite definir una función anónima dentro de otra función.

El anidamiento es importante, ya que nos permite evitar la contaminación de la nomenclatura del espacio de nombres y definir la función lo más cerca posible de donde se usa (proporcionando contexto adicional).

La sintaxis de las lambdas es una de las cosas más extrañas de C++ y requiere un poco de tiempo acostumbrarse a ella. Las lambdas toman la forma:

```
[ cláusula de captura ] ( parametros ) -> tipo de valor devuelto // es opcional
{
	declaraciones;
}
```

*La cláusula de captura puede estar vacía si no se necesitan capturas.*
*La lista de parámetros puede estar vacía si no se requiere ningún parámetro.* También se puede omitir por completo a menos que se especifique un tipo de valor devuelto.
*El tipo de valor devuelto es opcional* y, si se omite, se asumirá (por lo tanto, se utilizará la deducción de tipo utilizada para determinar el tipo de valor devuelto). Aunque anteriormente señalamos que se debe evitar la deducción de tipos para los tipos de retorno de función, en este contexto, está bien usarla (porque estas funciones suelen ser muy triviales).auto

```
auto suma = [](int a, int b) -> int {
	return a + b;
};
```

**10. Funciones de Miembro**

Las funciones miembro son funciones definidas dentro de una clase. Tienen acceso a los datos de la instancia de la clase.

```
class Rectangulo {
public:
	int ancho, alto;
	int area() {
		return ancho * alto;
	}
};
```

**11. Funciones Amigas (friend)**
Las funciones amigas son funciones que no son miembros de una clase, pero tienen acceso a sus miembros privados.

```
class Caja {
	int ancho;
public:
	Caja() : ancho(0) {}
	friend void mostrarAncho(Caja &c);
};

void mostrarAncho(Caja &c) {
	std::cout << "Ancho: " << c.ancho << std::endl;
}
```

**12. Función main**

La función main es el punto de entrada de cualquier programa en C++. Es donde comienza la ejecución.

```
int main() {
	std::cout << "Hola, Mundo!" << std::endl;
	return 0;
}
```

**13. Paso de Argumentos y Retorno**

El tipo de dato de los parámetros y el valor de retorno deben coincidir con lo especificado en la declaración de la función. Los argumentos se pueden pasar por valor, referencia o puntero.

**14. Plantillas de Funciones**

Las plantillas permiten crear funciones genéricas que pueden operar con diferentes tipos de datos sin cambiar el código de la función.

```
template <typename T>
T sumar(T a, T b) {
	return a + b;
}
```

# AMPLIACIÓN DEL TEMA

**1. ¿Qué es una Función Template?**

Una función template en C++ es una función genérica que permite trabajar con diferentes tipos de datos sin escribir múltiples versiones de la misma función. En lugar de definir funciones separadas para cada tipo de dato, puedes escribir una única función template que funcionará para cualquier tipo.

**2. Sintaxis Básica de una Función Template**

La sintaxis básica de una función template es la siguiente:

```
template <typename T>
T nombreDeLaFuncion(T parametro1, T parametro2) {
	// cuerpo de la función
}
```

**template:** Indica que se va a definir una plantilla.
**typename** T: *T es un parámetro* de tipo genérico que se utilizará en la función. También se puede usar la palabra clave class en lugar de typename (ambos son equivalentes en este contexto).
**T nombreDeLaFuncion(...):** Aquí T representa el tipo de dato que será determinado cuando se llame a la función.

**3. Ejemplo de una Función Template**

Aquí tienes un ejemplo básico de una función template que suma dos valores:

```
#include <iostream>
using namespace std;

template <typename T>
T sumar(T a, T b) {
	return a + b;
}

int main() {
	int x = 5, y = 10;
	double m = 2.5, n = 3.7;

	cout << "Suma de enteros: " << sumar(x, y) << endl;	// Suma de enteros
	cout << "Suma de dobles: " << sumar(m, n) << endl;	// Suma de dobles

	return 0;
}
```

**Explicación:**

La *función sumar es una función template* que toma dos parámetros de tipo T. El tipo T se determina en tiempo de compilación según los argumentos que se le pasen.
Cuando llamas a sumar(x, y), T se convierte en int, y cuando llamas a sumar(m, n), T se convierte en double.
La función funciona para ambos tipos sin necesidad de escribir dos versiones distintas de la función.

**4. Múltiples Parámetros de Tipo**

Puedes definir una función template con múltiples parámetros de tipo:

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

En este caso, T y U pueden ser de tipos diferentes. Esto permite mayor flexibilidad, como mostrar en el siguiente ejemplo:

```
#include <iostream>
using namespace std;

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

int main() {
	int x = 10;
	double y = 5.5;

	cout << "Menor valor: " << menor(x, y) << endl;   // Funciona con tipos diferentes

	return 0;
}
```

**Explicación:**

En este ejemplo, menor(x, y) compara un int y un double. La función retorna el menor de los dos valores.
T se convierte en int y U en double según los tipos de x y y.

**5. Plantillas de Funciones con Valores Predeterminados**
Puedes proporcionar valores predeterminados para los parámetros de tipo en una plantilla de función:

```
template <typename T = int>
T multiplicar(T a, T b) {
	return a * b;
}
```

En este caso, si no se especifica el tipo de T al llamar a la función, se usará int por defecto.

**6. Especialización de Plantillas de Función**

C++ permite la especialización de plantillas, lo que significa que puedes proporcionar una implementación específica para un tipo particular:

```
template <typename T>
T cuadrado(T x) {
	return x * x;
}

// Especialización para tipo char

template <>
char cuadrado(char x) {
	return x;
}
```

En este caso, la función cuadrado está especializada para el **tipo char**. En lugar de multiplicar el carácter, simplemente devuelve el mismo valor.

**7. Plantillas de Funciones vs. Sobrecarga de Funciones**

**Plantillas de Función:** Permiten escribir una sola función que funcione con cualquier tipo de dato.

**Sobrecarga de Función:** Requiere escribir múltiples versiones de la misma función, cada una con un tipo de dato específico.

Ejemplo de Sobrecarga:

```
int sumar(int a, int b) { return a + b; }
double sumar(double a, double b) { return a + b; }
```

Ejemplo de Plantilla de Función:

```
template <typename T>
T sumar(T a, T b) { return a + b; }
```

**8. Ventajas de las Funciones Template**

**Reusabilidad:** Escribes una función una sola vez, y puede ser utilizada con cualquier tipo de dato.
**Flexibilidad:** Las funciones template pueden adaptarse a tipos de datos personalizados o definidos por el usuario.
**Eficiencia en el Código:** Reduce la duplicación de código, lo que facilita el mantenimiento.

**9. Desventajas de las Funciones Template**

**Complejidad:** La sintaxis y la lógica pueden ser más complejas que las funciones regulares.
**Errores en Tiempo de Compilación:** Los errores suelen aparecer en tiempo de compilación, lo que a veces puede ser difícil de depurar.

**Resumen**

Las funciones template son una poderosa característica de C++ que permite crear funciones genéricas, reutilizables y flexibles que pueden trabajar con cualquier tipo de dato. Son especialmente útiles cuando se quiere evitar la duplicación de código para diferentes tipos de datos y se desean soluciones más generales.

**Funciones Const** .

Las funciones *const son aquellas que no modifican el estado del objeto* en el que se llaman, lo que garantiza que los miembros del objeto no serán alterados.

```
class Rectangulo {
     int ancho, alto;
public:
     Rectangulo(int a, int b) : ancho(a), alto(b) {}
     int area() const {
          return ancho * alto;
     }
};
```

**Resumen**
Las funciones en C++ son un componente fundamental que permiten modular el código, mejorando la organización, la legibilidad y la reutilización. Comprender cómo declarar, definir y utilizar funciones es esencial para escribir programas eficientes y mantenibles.