



PROGRAMACIÓN III

Docente:

Ing. Jimmy Nataniel Requena Llorentty

Integrantes:

Jaquelin Priscila Bellido Duran

Materia:

Programación III

Santa Cruz - Bolivia

Clase 1: El Poder de Apuntar

¡Entendiendo Punteros y la Memoria!

Hoy aprendimos:

¿Qué es un STACK?

Un **stack** es una estructura de datos y también una región de memoria que sigue un orden muy específico para almacenar y recuperar datos.

Se caracteriza por ser: muy rápida, tamaño limitado, se libera automáticamente (cuando la función termina)

¿Qué es un HEAP?

Es una zona de memoria usada para almacenar datos dinámicos, es decir, datos que el programa reserva manualmente en tiempo de ejecución

Se caracteriza por ser: Acceso más lento Debes liberar la memoria manualmente (en C/C++ con free o delete) Riesgo de *memory leaks* (si no liberas)

Ampersand (&) → Operador de dirección

- Se usa para **obtener la dirección de memoria** de una variable.
- Es lo que necesitas para inicializar un puntero.

Asterisco (*) → Operador de indirección (o desreferenciación)

- Se usa para **acceder al valor almacenado en la dirección** a la que apunta un puntero.
- También se usa para **declarar punteros**

```
9
10  std::cout << "--- Información de 'variable' ---" << std::endl;
11  std::cout << "Valor de 'variable': " << variable << std::endl;
12  std::cout << "Dirección de 'variable' (&variable): " << &variable << std::endl;
13
14  std::cout << "\n--- Informacion de 'puntero' ---" << std::endl;
15  std::cout << "Contenido de 'puntero' (la direccion que guarda): " << puntero << std::endl;
16  std::cout << "Dirección donde esta guardado el propio 'puntero' (&puntero): " << &puntero << std::endl;
17
18  std::cout << "\n--- Accediendo al valor A TRAVES del puntero ---" << std::endl;
19  std::cout << "Valor al que apunta 'puntero' (*puntero): " << *puntero << std::endl; // DEREFERENCIA
20
21  // Modificando 'variable' A TRAVÉS del puntero
22  std::cout << "\n--- Modificando a traves del puntero ---" << std::endl;
23  *puntero = 30; // Ve a la dirección que guarda 'puntero' y cambia el valor allí a 30
24  std::cout << "Nuevo valor de 'variable' (despues de *puntero = 30): " << variable << std::endl;
25  std::cout << "Nuevo valor apuntado por 'puntero' (*puntero): " << *puntero << std::endl;
26  std::cout << "\nPriscila Bellido - Fin deCodigo" << std::endl;
27
```

input

```
Contenido de 'puntero' (la direccion que guarda): 0x7ffefceabeac
Dirección donde esta guardado el propio 'puntero' (&puntero): 0x7ffefceabeb0

--- Accediendo al valor A TRAVES del puntero ---
Valor al que apunta 'puntero' (*puntero): 20

--- Modificando a traves del puntero ---
Nuevo valor de 'variable' (despues de *puntero = 30): 30
Nuevo valor apuntado por 'puntero' (*puntero): 30

Priscila Bellido - Fin deCodigo

...Program finished with exit code 0
Press ENTER to exit console.
```

Clase 2: ¡Forjando Memoria a Voluntad!

new y delete

Hoy aprendimos:

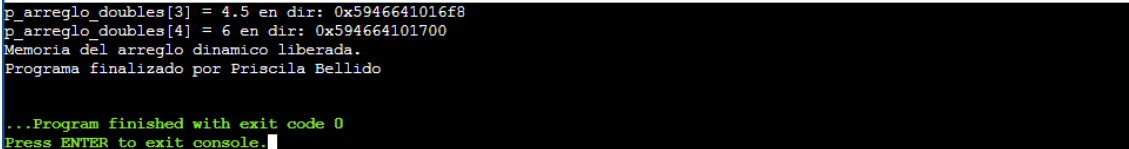
¿Qué es un operador NEW?

El **operador new** en C++ se utiliza para **asignar memoria dinámica** en el **heap** (memoria libre), a diferencia de las variables normales que se asignan en la **pila (stack)**. Es decir, cuando no sabes cuánta memoria necesitas hasta que el programa se esté ejecutando, **new** te permite reservar esa memoria en tiempo de ejecución.

¿Qué es un operador DELETE?

El **operador delete** en C++ se usa para **liberar** la memoria que fue previamente **asignada dinámicamente con new**. Liberar memoria es **crucial** para evitar **fugas de memoria (memory leaks)**, que ocurren cuando el programa pierde acceso a la memoria asignada pero nunca la libera.

```
28     p_arreglo_doubles[i] = i * 1.5; // Asigna valores al arreglo
29 }
30
31 std::cout << "Arreglo dinamico creado y llenado:" << std::endl;
32 for (int i = 0; i < tamano_arreglo; ++i) {
33     std::cout << "p_arreglo_doubles[" << i << "] = " << p_arreglo_doubles[i]
34         << " en dir: " << (p_arreglo_doubles + i) << std::endl;
35 }
36
37 delete[] p_arreglo_doubles; // ¡IMPORTANTE! Usar delete[] para arreglos
38 p_arreglo_doubles = nullptr; // Buena práctica
39 std::cout << "Memoria del arreglo dinamico liberada." << std::endl;
40 } else {
41     std::cout << "ERROR: No se pudo asignar memoria para p_arreglo_doubles." << std::endl;
42 }
43
44 // Intentar usar un puntero nulo (solo para demostrar, usualmente causa error o comportamiento indefinido)
45 // if (p_entero == nullptr) {
46 //     std::cout << "\np_entero es ahora nullptr." << std::endl;
47 //     // *p_entero = 789; // ¡Esto causaría un error de segmentación! (Descomentar con precaución)
48 // }
49 std::cout << "Programa finalizado por Priscila Bellido" << std::endl;
50
51 return 0;
52
```



```
p_arreglo_doubles[3] = 4.5 en dir: 0x5946641016f8
p_arreglo_doubles[4] = 6 en dir: 0x594664101700
Memoria del arreglo dinamico liberada.
Programa finalizado por Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream> // Para std::cout, std::endl
```

```
int main() {
    // 1. Asignar memoria para un solo entero
    int *p_entero = nullptr; // Siempre inicializar punteros
    p_entero = new int;      // Solicita memoria en el Heap para un int

    if (p_entero != nullptr) { // Buena práctica: verificar si new tuvo
        éxito (aunque suele lanzar excepción)
    }
}
```

```

        *p_entero = 123;        // Asigna un valor a la memoria recién
reservada
        std::cout << "Entero dinamico creado. Valor: " << *p_entero
                << " en direccion: " << p_entero << std::endl;

        delete p_entero;        // Libera la memoria
        p_entero = nullptr;    // ¡Buena práctica! Evita puntero colgante.
        std::cout << "Memoria del entero dinamico liberada." << std::endl;
    } else {
        std::cout << "ERROR: No se pudo asignar memoria para p_entero." <<
std::endl;
    }

    // 2. Asignar memoria para un arreglo de doubles
    std::cout << "\n--- Arreglo Dinamico ---" << std::endl;
    double *p_arreglo_doubles = nullptr;
    int tamano_arreglo = 5;
    p_arreglo_doubles = new double[tamano_arreglo]; // Solicita memoria para
5 doubles

    if (p_arreglo_doubles != nullptr) {
        for (int i = 0; i < tamano_arreglo; ++i) {
            p_arreglo_doubles[i] = i * 1.5; // Asigna valores al arreglo
        }

        std::cout << "Arreglo dinamico creado y llenado:" << std::endl;
        for (int i = 0; i < tamano_arreglo; ++i) {
            std::cout << "p_arreglo_doubles[" << i << "] = " <<
p_arreglo_doubles[i]
                << " en dir: " << (p_arreglo_doubles + i) <<
std::endl;
        }

        delete[] p_arreglo_doubles; // ¡IMPORTANTE! Usar delete[] para
arreglos
        p_arreglo_doubles = nullptr; // Buena práctica
        std::cout << "Memoria del arreglo dinamico liberada." << std::endl;
    } else {
        std::cout << "ERROR: No se pudo asignar memoria para
p_arreglo_doubles." << std::endl;
    }

    // Intentar usar un puntero nulo (solo para demostrar, usualmente causa
error o comportamiento indefinido)
    // if (p_entero == nullptr) {
    //     std::cout << "\np_entero es ahora nullptr." << std::endl;
    //     // *p_entero = 789; // ¡Esto causaría un error de segmentación!
    // }
    // (Descomentar con precaución)
    std::cout << "Programa finalizado por Priscila Bellido" << std::endl;

    return 0;

```

Clase 3: ¡Construyendo Cadenas de Datos!

Introducción a Listas Enlazadas

Hoy aprendimos:

¿Qué es un ARREGLO (ARRAY)?

Un **arreglo** (también llamado *array*) es una **estructura de datos** que almacena una **colección de elementos del mismo tipo** en posiciones contiguas de memoria.

¿Qué es una lista enlazada simple?

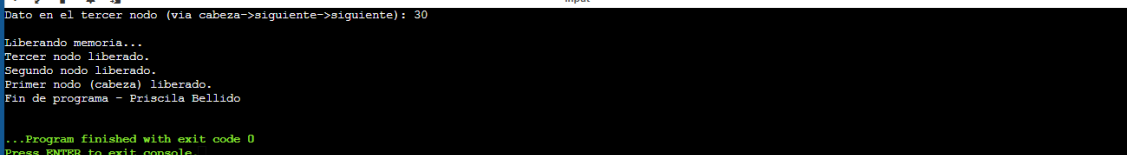
Una **lista enlazada simple** es una estructura de datos **dinámica** que consiste en **nodos conectados entre sí** mediante punteros. A diferencia de los arreglos, **no requiere tamaño fijo**, y puedes insertar/eliminar elementos fácilmente en cualquier posición.

¿Qué es un nodo simple?

Un **nodo simple** es la **unidad básica de una lista enlazada simple**. Contiene dos cosas:

1. **Un dato** (el valor que guarda).
2. **Un puntero al siguiente nodo**.

```
34 std::cout << "Dato en cabeza: " << cabeza->dato << std::endl;
35 std::cout << "Dato en el segundo nodo (via cabeza->siguiente): " << cabeza->siguiente->dato << std::endl;
36 std::cout << "Dato en el tercer nodo (via cabeza->siguiente->siguiente): "
37     << cabeza->siguiente->siguiente->dato << std::endl;
38
39 // ¡IMPORTANTE! Liberar la memoria dinámica cuando ya no se necesite
40 // Se debe hacer en orden inverso a con cuidado para no perder punteros
41 std::cout << "\nLiberando memoria..." << std::endl;
42 delete cabeza->siguiente->siguiente; // Borra el tercer nodo (tercerNodo)
43 cabeza->siguiente->siguiente = nullptr; // Buena práctica
44 std::cout << "Tercer nodo liberado." << std::endl;
45
46 delete cabeza->siguiente; // Borra el segundo nodo (segundoNodo)
47 cabeza->siguiente = nullptr; // Buena práctica
48 std::cout << "Segundo nodo liberado." << std::endl;
49
50 delete cabeza; // Borra el primer nodo
51 cabeza = nullptr; // Buena práctica
52 std::cout << "Primer nodo (cabeza) liberado." << std::endl;
53
54 std::cout << "Fin de programa - Priscila Bellido" << std::endl;
55
```



```
#include <iostream>
```

```
struct Nodo {
    int dato;
    Nodo* siguiente;
```

```
    Nodo(int valor_dato) : dato(valor_dato), siguiente(nullptr) {} //
Constructor conciso
};
```

```
int main() {
```

```

    // 1. Crear el primer nodo (cabeza de nuestra mini-lista)
    Nodo* cabeza = new Nodo(10); // Usamos 'new' porque queremos memoria
dinámica
    std::cout << "Creado primer nodo (cabeza) con dato: " << cabeza->dato <<
std::endl;

    // 2. Crear un segundo nodo
    Nodo* segundoNodo = new Nodo(20);
    std::cout << "Creado segundo nodo con dato: " << segundoNodo->dato <<
std::endl;

    // 3. ¡ENLAZARLOS!
    // El puntero 'siguiente' del primer nodo (cabeza) ahora apunta al
segundoNodo
    cabeza->siguiente = segundoNodo;
    std::cout << "Enlazando cabeza->siguiente con segundoNodo." <<
std::endl;

    // 4. Crear un tercer nodo
    Nodo* tercerNodo = new Nodo(30);
    std::cout << "Creado tercer nodo con dato: " << tercerNodo->dato <<
std::endl;

    // 5. Enlazar el segundo nodo con el tercero
    segundoNodo->siguiente = tercerNodo; // 0 cabeza->siguiente->siguiente =
tercerNodo;
    std::cout << "Enlazando segundoNodo->siguiente con tercerNodo." <<
std::endl;

    // ¿Cómo accedemos a los datos ahora?
    std::cout << "\nRecorriendo la mini-lista:" << std::endl;
    std::cout << "Dato en cabeza: " << cabeza->dato << std::endl;
    std::cout << "Dato en el segundo nodo (via cabeza->siguiente): " <<
cabeza->siguiente->dato << std::endl;
    std::cout << "Dato en el tercer nodo (via cabeza->siguiente->siguiente):
"
    << cabeza->siguiente->siguiente->dato << std::endl;

    // ¡IMPORTANTE! Liberar la memoria dinámica cuando ya no se necesite
    // Se debe hacer en orden inverso o con cuidado para no perder punteros
    std::cout << "\nLiberando memoria..." << std::endl;
    delete cabeza->siguiente->siguiente; // Borra el tercer nodo
(tercerNodo)
    cabeza->siguiente->siguiente = nullptr; // Buena práctica
    std::cout << "Tercer nodo liberado." << std::endl;

    delete cabeza->siguiente; // Borra el segundo nodo (segundoNodo)
    cabeza->siguiente = nullptr; // Buena práctica
    std::cout << "Segundo nodo liberado." << std::endl;

    delete cabeza; // Borra el primer nodo
    cabeza = nullptr; // Buena práctica
    std::cout << "Primer nodo (cabeza) liberado." << std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;

    return 0;
}

```

Clase 4: ¡Funciones con Múltiples Talentos!

La Sobrecarga de Funciones

Hoy aprendimos:

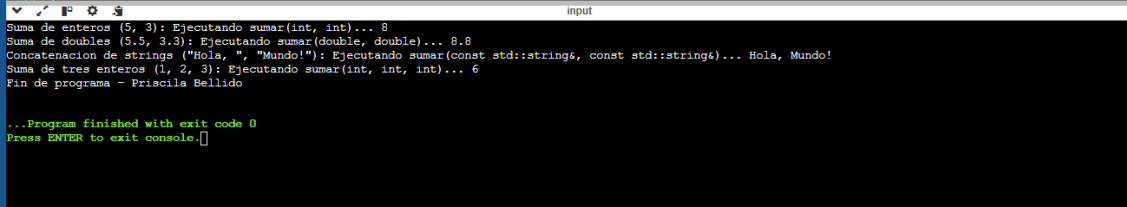
¿Qué es SOBRECARGA?

La **sobrecarga** es una técnica en programación (especialmente en C++) que permite **usar el mismo nombre** para **funciones** o **operadores**, pero con **diferentes comportamientos** según los **argumentos** o el **tipo de datos**.

Ventajas

- Código más **limpio y legible**
- Permite **usar el mismo nombre** para realizar tareas similares
- Esencial en programación orientada a objetos (POO)

```
21     return a + b;
22 }
23
24 // Versión 4: Suma tres enteros
25 // ¡Sobrecargada! Mismo nombre, diferente número de parámetros.
26 int sumar(int a, int b, int c) {
27     std::cout << "Ejecutando sumar(int, int, int)... ";
28     return a + b + c;
29 }
30
31 int main() {
32     std::cout << "Suma de enteros (5, 3): " << sumar(5, 3) << std::endl;
33     std::cout << "Suma de doubles (5.5, 3.3): " << sumar(5.5, 3.3) << std::endl;
34     std::cout << "Concatenacion de strings (\\"Hola, \\", \\"Mundo!\"): "
35     << sumar(std::string("Hola, "), std::string("Mundo!")) << std::endl;
36     std::cout << "Suma de tres enteros (1, 2, 3): " << sumar(1, 2, 3) << std::endl;
37
38     std::cout << "fin de programa - Priscila Bellido" << std::endl;
39     return 0;
40 }
```



```
#include <iostream> // Para std::cout, std::endl
#include <string>    // Para std::string

// Versión 1: Suma dos enteros
int sumar(int a, int b) {
    std::cout << "Ejecutando sumar(int, int)... ";
    return a + b;
}

// Versión 2: Suma dos números de punto flotante (double)
// ¡Sobrecargada! Mismo nombre, diferente tipo de parámetros.
double sumar(double a, double b) {
    std::cout << "Ejecutando sumar(double, double)... ";
    return a + b;
}

// Versión 3: Concatena dos cadenas (std::string)
// ¡Sobrecargada! Mismo nombre, diferente tipo de parámetros.
std::string sumar(const std::string& a, const std::string& b) {
```

```

        std::cout << "Ejecutando sumar(const std::string&, const
std::string&)... ";
        return a + b;
    }

    // Versión 4: Suma tres enteros
    // ¡Sobrecargada! Mismo nombre, diferente número de parámetros.
    int sumar(int a, int b, int c) {
        std::cout << "Ejecutando sumar(int, int, int)... ";
        return a + b + c;
    }

    int main() {
        std::cout << "Suma de enteros (5, 3): " << sumar(5, 3) << std::endl;
        std::cout << "Suma de doubles (5.5, 3.3): " << sumar(5.5, 3.3) <<
std::endl;
        std::cout << "Concatenacion de strings (\"Hola, \", \"Mundo!\"): "
        << sumar(std::string("Hola, "), std::string("Mundo!")) <<
std::endl;
        std::cout << "Suma de tres enteros (1, 2, 3): " << sumar(1, 2, 3) <<
std::endl;

        // Ejemplo de llamada ambigua (si no tuviéramos una versión exacta)
        // Si solo tuviéramos sumar(double, double) y llamáramos sumar(5, 3),
        // los 'int' se promocionarían a 'double'. ¡Pero aquí tenemos una
        exacta!
        // std::cout << "Llamada con promocion (si no hubiera int,int): " <<
sumar(5, (int)3.0) << std::endl;
        // El casteo (int)3.0 no es necesario aquí, solo es para ilustrar.

        std::cout << "Fin de programa - Priscila Bellido" << std::endl;
        return 0;
    }

```


Clase 5: ¡La Sobrecarga en Acción!

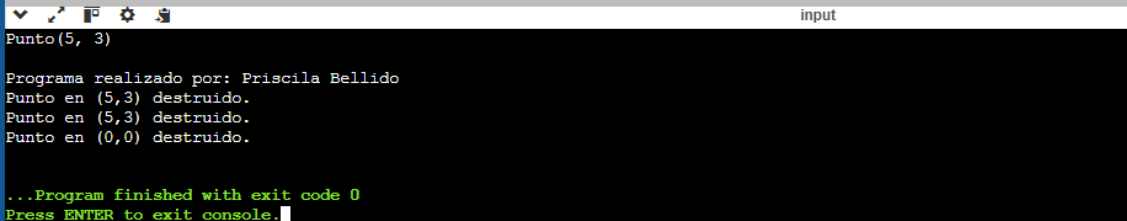
Aplicaciones y Utilidades practicas

Hoy aprendimos:

¿Qué son constructores?

Un **constructor** es una función **especial** dentro de una **clase** que se **ejecuta automáticamente** cuando se **crea un objeto**. Sirve para **inicializar atributos** y preparar el objeto para su uso.

```
30 }
31 };
32
33 // Función main para probar la clase
34 int main() {
35     std::cout << "=== Demostrando constructores ===" << std::endl;
36
37     // Constructor por defecto
38     Punto p1;
39
40     // Constructor con coordenadas
41     Punto p2(5.0, 3.0);
42
43     // Constructor de copia
44     Punto p3(p2);
45
46     std::cout << "\n=== Mostrando puntos ===" << std::endl;
47     p1.mostrar();
48     p2.mostrar();
49     p3.mostrar();
50
51     std::cout << "\nPrograma realizado por: Priscila Bellido" << std::endl;
52 }
```



```
#include <iostream> // ¡Esta línea es la que faltaba!
```

```
class Punto {
public:
    double x, y;

    // Constructor 1: Por defecto (en el origen)
    Punto() : x(0.0), y(0.0) {
        std::cout << "Punto creado en el origen (0,0) por constructor por defecto." << std::endl;
    }

    // Constructor 2: Con coordenadas específicas
    Punto(double coord_x, double coord_y) : x(coord_x), y(coord_y) {
        std::cout << "Punto creado en (" << x << ", " << y << ") por constructor con coords." << std::endl;
    }
}
```

```

    // Constructor 3: Copia (se genera uno por defecto, pero podemos hacerlo
    explícito)
    Punto(const Punto& otroPunto) : x(otroPunto.x), y(otroPunto.y) {
        std::cout << "Punto copiado de (" << otroPunto.x << ", " <<
otroPunto.y << ")." << std::endl;
    }

    // Destructor (opcional, para demostrar el ciclo de vida)
    ~Punto() {
        std::cout << "Punto en (" << x << ", " << y << ") destruido." <<
std::endl;
    }

    // Método para mostrar las coordenadas
    void mostrar() const {
        std::cout << "Punto(" << x << ", " << y << ")" << std::endl;
    }
};

// Función main para probar la clase
int main() {
    std::cout << "=== Demostrando constructores ===" << std::endl;

    // Constructor por defecto
    Punto p1;

    // Constructor con coordenadas
    Punto p2(5.0, 3.0);

    // Constructor de copia
    Punto p3(p2);

    std::cout << "\n=== Mostrando puntos ===" << std::endl;
    p1.mostrar();
    p2.mostrar();
    p3.mostrar();

    std::cout << "\nPrograma realizado por: Priscila Bellido" << std::endl;

    return 0;
}

```

¿Por qué sobrecargar?

La **sobrecarga** permite **definir varias versiones de una misma función** (o constructor) con **diferentes parámetros**. Esto mejora la **flexibilidad, claridad del código** y **reutilización**.

¿Cuándo deberías sobrecargar?

- Quieres **una misma acción** que varía según los **argumentos**.
- Buscas **hacer tu clase o función más flexible**.
- Necesitas versiones con **valores por defecto** o inicializaciones distintas.

¿Qué son las colecciones polimórficas?

Una **colección polimórfica** es una **estructura de datos** (como una lista, vector, arreglo, etc.) que puede **almacenar objetos de diferentes tipos derivados** de una misma clase base, y **tratarlos de forma genérica** usando **polimorfismo**.

Esto permite trabajar con objetos de distintas clases (que comparten herencia) **a través de punteros o referencias a la clase base**.

¿Qué son los smart pointers?

Los smart pointers o punteros inteligentes son objetos de la STL (Standard Template Library) que gestionan automáticamente la memoria dinámica en C++. Su objetivo es evitar fugas de memoria (memory leaks) y errores como dobles liberaciones o accesos a memoria liberada.

Mostrar completo:

```
42
43 void mostrar(double valor) {
44     std::cout << "Tipo Decimal (double): " << valor << std::endl;
45 }
46
47 void mostrar(const std::string& valor) {
48     std::cout << "Tipo Cadena (std::string): \"" << valor << "\"" << std::endl;
49 }
50
51 void mostrar(char valor) {
52     std::cout << "Tipo Caracter (char): '" << valor << "'" << std::endl;
53 }
54
55 void mostrar(const std::vector<int>& miVector) {
56     std::cout << "Tipo Vector de Enteros (std::vector<int>): [ ";
57     for (size_t i = 0; i < miVector.size(); ++i) {
58         std::cout << miVector[i] << (i == miVector.size() - 1 ? "" : ", ");
59     }
60     std::cout << "]" << std::endl;
61 }
```

input

```
--- Demostracion de 'mostrar' sobrecargado ---
Tipo Entero (int): 100
Tipo Decimal (double): 3.14159
Tipo Cadena (std::string): "Hola desde Programacion III!"
Tipo Caracter (char): 'Z'
Tipo Vector de Enteros (std::vector<int>): [ 10, 20, 30, 40, 50 ]
Tipo Cadena (std::string): "Esto es un literal de C-string."
Tipo Cadena (std::string): "Priscila Bellido - Fin del codigo"

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
#include <string>
#include <vector>

// Declaraciones de las funciones 'mostrar' (prototipos)
void mostrar(int valor);
void mostrar(double valor);
void mostrar(const std::string& valor);
void mostrar(char valor);
void mostrar(const std::vector<int>& miVector);

int main() {
    std::cout << "--- Demostracion de 'mostrar' sobrecargado ---" <<
    std::endl;
```

```

mostrar(100);
mostrar(3.14159);
mostrar(std::string("Hola desde Programacion III!"));
mostrar('Z');

std::vector<int> numeros = { 10, 20, 30, 40, 50 };
mostrar(numeros);

// Llamada con un literal de cadena de C (const char*)
// El compilador puede convertirlo a std::string si no hay otra
sobrecarga mejor
mostrar("Esto es un literal de C-string.");
mostrar("Priscila Bellido - Fin del codigo");

return 0;
}

// Implementaciones de las funciones 'mostrar'
void mostrar(int valor) {
    std::cout << "Tipo Entero (int): " << valor << std::endl;
}

void mostrar(double valor) {
    std::cout << "Tipo Decimal (double): " << valor << std::endl;
}

void mostrar(const std::string& valor) {
    std::cout << "Tipo Cadena (std::string): \"" << valor << "\"" <<
std::endl;
}

void mostrar(char valor) {
    std::cout << "Tipo Caracter (char): '" << valor << "'" << std::endl;
}

void mostrar(const std::vector<int>& miVector) {
    std::cout << "Tipo Vector de Enteros (std::vector<int>): [ ";
    for (size_t i = 0; i < miVector.size(); ++i) {
        std::cout << miVector[i] << (i == miVector.size() - 1 ? "" : ", ");
    }
    std::cout << " ]" << std::endl;
}

```

Clase 6: ¡El Arte de las Funciones que se Piensan a Sí Mismas

Hoy aprendimos:

¿Qué es la recursividad?

La **recursividad** es una técnica donde una **función se llama a sí misma** para resolver un problema más grande dividiéndolo en **subproblemas más pequeños**.

¿Por qué se usa?

Se utiliza cuando un problema puede **dividirse en partes similares a sí mismo**, lo que permite una solución elegante y más natural en algunos casos (como estructuras de árbol, matemáticas, etc.).

Función Recursiva: Sus Partes Vitales

Una función recursiva tiene 3 componentes esenciales que garantizan su funcionamiento correcto y seguro:

Caso Base (o condición de parada)

Es la condición que detiene la recursión.

Evita que la función se llame infinitamente.

Siempre debe resolverse sin llamadas recursivas.

Paso Recursivo (la llamada a sí misma)

- Es el corazón de la recursividad.
- La función se **llama a sí misma**, pero con un problema **más pequeño o más simple**.

Progresión hacia el caso base

- Es la **garantía de que nos acercamos** al caso base en cada llamada.
- Si no hay progresión, la recursión será infinita.

¡La Recursividad en Acción! Calculando el Factorial

```
17     long long resultadoRecurcion = factorial(n - 1); // Llamada recursiva
18     long long resultadoFinal = n * resultadoRecurcion;
19
20     std::cout << " factorial(" << n << ") -> Retornando " << n << " * "
21     << resultadoRecurcion << " = " << resultadoFinal << std::endl;
22
23     return resultadoFinal;
24 }
25 }
26
27 int main() {
28     int numero = 4; // Probar con 4!
29     std::cout << "Iniciando calculo del factorial de " << numero << "." << std::endl;
30
31     long long resultado = factorial(numero);
32
33     std::cout << "\nEl factorial de " << numero << " es: " << resultado << std::endl;
34
35     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
36
37     return 0;
}
```

input

```
Calculando factorial(1)...
factorial(1) -> Caso Base! Retorna 1.
factorial(2) -> Retornando 2 * 1 = 2
factorial(3) -> Retornando 3 * 2 = 6
factorial(4) -> Retornando 4 * 6 = 24

El factorial de 4 es: 24
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>

// Función recursiva para calcular el factorial
long long factorial(int n) {
    std::cout << "Calculando factorial(" << n << ")..." << std::endl;

    // Caso Base
    if (n == 0 || n == 1) {
        std::cout << " factorial(" << n << ") -> Caso Base! Retorna 1." <<
std::endl;
        return 1;
    }
    // Paso Recursivo
    else {
        std::cout << " factorial(" << n << ") -> Paso Recursivo. Llama a
factorial("
            << (n - 1) << ")." << std::endl;

        long long resultadoRecurcion = factorial(n - 1); // Llamada
recursiva
        long long resultadoFinal = n * resultadoRecurcion;

        std::cout << " factorial(" << n << ") -> Retornando " << n << " * "
            << resultadoRecurcion << " = " << resultadoFinal <<
std::endl;

        return resultadoFinal;
    }
}
```

```
int main() {
    int numero = 4; // Probar con 4!
    std::cout << "Iniciando calculo del factorial de " << numero << "." <<
std::endl;

    long long resultado = factorial(numero);

    std::cout << "\nEl factorial de " << numero << " es: " << resultado <<
std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;

    return 0;
}
```

Clase 7: ¡La Recursividad se Expande!

Aplicaciones y Análisis de Eficiencia

Hoy aprendimos:

¿Qué es la secuencia de Fibonacci?

La **secuencia de Fibonacci** es una serie de números donde cada número es la **suma de los dos anteriores**.

Fibonacci: La Naturaleza Hecha Números (y Recursión)

```
13     return fibonacci(n - 1) + fibonacci(n - 2);
14 }
15
16 int main() {
17     int terminos = 7; // Calcular hasta F(6)
18     std::cout << "Secuencia de Fibonacci (primeros " << terminos << " terminos):" << std::endl;
19
20     for (int i = 0; i < terminos; ++i) {
21         std::cout << fibonacci(i) << " ";
22     }
23
24     std::cout << std::endl;
25
26     // Prueba adicional
27     std::cout << "F(7) es: " << fibonacci(7) << std::endl;
28
29     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
30
31     return 0;
32 }
```

input

```
Secuencia de Fibonacci (primeros 7 terminos):
0 1 1 2 3 5 8
F(7) es: 13
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
```

```
// Función Fibonacci recursiva
```

```
int fibonacci(int n) {
    // Casos base
    if (n <= 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // Paso recursivo
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

```
int main() {
    int terminos = 7; // Calcular hasta F(6)
    std::cout << "Secuencia de Fibonacci (primeros " << terminos << "
terminos):" << std::endl;
```



```

    for (int i = 0; i < terminos; ++i) {
        std::cout << fibonacci(i) << " ";
    }

    std::cout << std::endl;

    // Prueba adicional
    std::cout << "F(7) es: " << fibonacci(7) << std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;

    return 0;
}

```

Sumando en Cadena: Recursión con Arreglos

```

5 // Caso Base: Si el índice está fuera de los límites del vector,
6 // significa que no hay más elementos que sumar.
7 if (idx >= arr.size()) {
8     return 0;
9 }
10 // Paso Recursivo: Suma el elemento actual (arr[idx])
11 // con la suma del resto del arreglo (desde idx + 1).
12 else {
13     return arr[idx] + sumarArreglo(arr, idx + 1);
14 }
15 }
16 int main() {
17     std::vector<int> misNumeros = {10, 5, 15, 20, 50}; // Suma = 100
18     int sumaTotal = sumarArreglo(misNumeros, 0); // Empezar desde el índice 0
19     std::cout << "La suma recursiva del arreglo es: " << sumaTotal <<
20     std::endl;
21
22     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
23
24     return 0;
25 }

```

input

```

La suma recursiva del arreglo es: 100
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.

```

```

#include <iostream>
#include <vector>
// Suma los elementos de 'arr' desde el índice 'idx' hasta el final
int sumarArreglo(const std::vector<int>& arr, int idx) {
    // Caso Base: Si el índice está fuera de los límites del vector,
    // significa que no hay más elementos que sumar.
    if (idx >= arr.size()) {
        return 0;
    }
    // Paso Recursivo: Suma el elemento actual (arr[idx])
    // con la suma del resto del arreglo (desde idx + 1).
    else {
        return arr[idx] + sumarArreglo(arr, idx + 1);
    }
}
int main() {

```

```
std::vector<int> misNumeros = {10, 5, 15, 20, 50}; // Suma = 100
int sumaTotal = sumarArreglo(misNumeros, 0); // Empezar desde el índice 0
std::cout << "La suma recursiva del arreglo es: " << sumaTotal <<
std::endl;

std::cout << "Fin de programa - Priscila Bellido" << std::endl;

return 0;

}
```

Clase 8: ¡Arquitectos de la Recursión!

Diseñando Nuestras Propias Soluciones

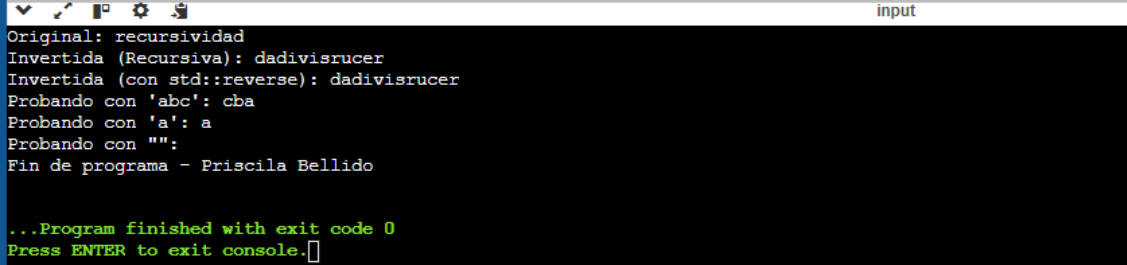
Hoy aprendimos:

¿Qué es Diseño Recursivo?

El **diseño recursivo** es una técnica para resolver problemas dividiéndolos en subproblemas más pequeños y similares, hasta llegar a un caso base trivial que puede resolverse directamente.

```
15 // std::cout << " Primer caracter: '" << primerCaracter
16 // << "'", Resto: '\"' << restoDeLaCadena << "\"' << std::endl;
17 std::string restoInvertido = invertirRecursiva(restoDeLaCadena); // ¡Fe recursiva!
18 // std::cout << " Resto invertido: '\"' << restoInvertido << "\"' << std::endl;
19 return restoInvertido + primerCaracter; // Combinar
20 }
21 }
22 int main() {
23     std::string original = "recursividad";
24     std::string invertida = invertirRecursiva(original);
25     std::cout << "Original: " << original << std::endl;
26     std::cout << "Invertida (Recursiva): " << invertida << std::endl;
27     // Para comparar (no es parte de la solución recursiva)
28     std::string comparacion = original;
29     std::reverse(comparacion.begin(), comparacion.end());
30     std::cout << "Invertida (con std::reverse): " << comparacion << std::endl;
31     std::cout << "Probando con 'abc': " << invertirRecursiva("abc") << std::endl;
32     std::cout << "Probando con 'a': " << invertirRecursiva("a") << std::endl;
33     std::cout << "Probando con '\"': " << invertirRecursiva("") << std::endl;
34     |
35     std::cout << "Fin de programa - Priscila Bellido" << std::endl;

```



```
Original: recursividad
Invertida (Recursiva): dadivisruocer
Invertida (con std::reverse): dadivisruocer
Probando con 'abc': cba
Probando con 'a': a
Probando con '\"': 
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.

```

```
#include <iostream>
#include <string>
#include <algorithm> // Para std::reverse (solo para comparar)
std::string invertirRecursiva(const std::string& s) {
    // std::cout << "Llamada con: '\"' << s << "\"' << std::endl; // Para traza
    // Caso Base: Cadena vacía o de un solo carácter
    if (s.length() <= 1) {
        // std::cout << " Caso Base, retorna: '\"' << s << "\"' << std::endl;
        return s;
    }
    // Paso Recursivo:
    else {
        char primerCaracter = s[0];
        std::string restoDeLaCadena = s.substr(1);
        // std::cout << " Primer caracter: '" << primerCaracter
        // << "'", Resto: '\"' << restoDeLaCadena << "\"' << std::endl;
        std::string restoInvertido = invertirRecursiva(restoDeLaCadena); // ¡Fe
recursiva!
        // std::cout << " Resto invertido: '\"' << restoInvertido << "\"' <<
std::endl;
        return restoInvertido + primerCaracter; // Combinar
    }
}
```

```

    }
}
int main() {
    std::string original = "recursividad";
    std::string invertida = invertirRecursiva(original);
    std::cout << "Original: " << original << std::endl;
    std::cout << "Invertida (Recursiva): " << invertida << std::endl;
    // Para comparar (no es parte de la solución recursiva)
    std::string comparacion = original;
    std::reverse(comparacion.begin(), comparacion.end());
    std::cout << "Invertida (con std::reverse): " << comparacion << std::endl;
    std::cout << "Probando con 'abc': " << invertirRecursiva("abc") <<
std::endl;
    std::cout << "Probando con 'a': " << invertirRecursiva("a") << std::endl;
    std::cout << "Probando con \"\": " << invertirRecursiva("") << std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;
    return 0;
}

```