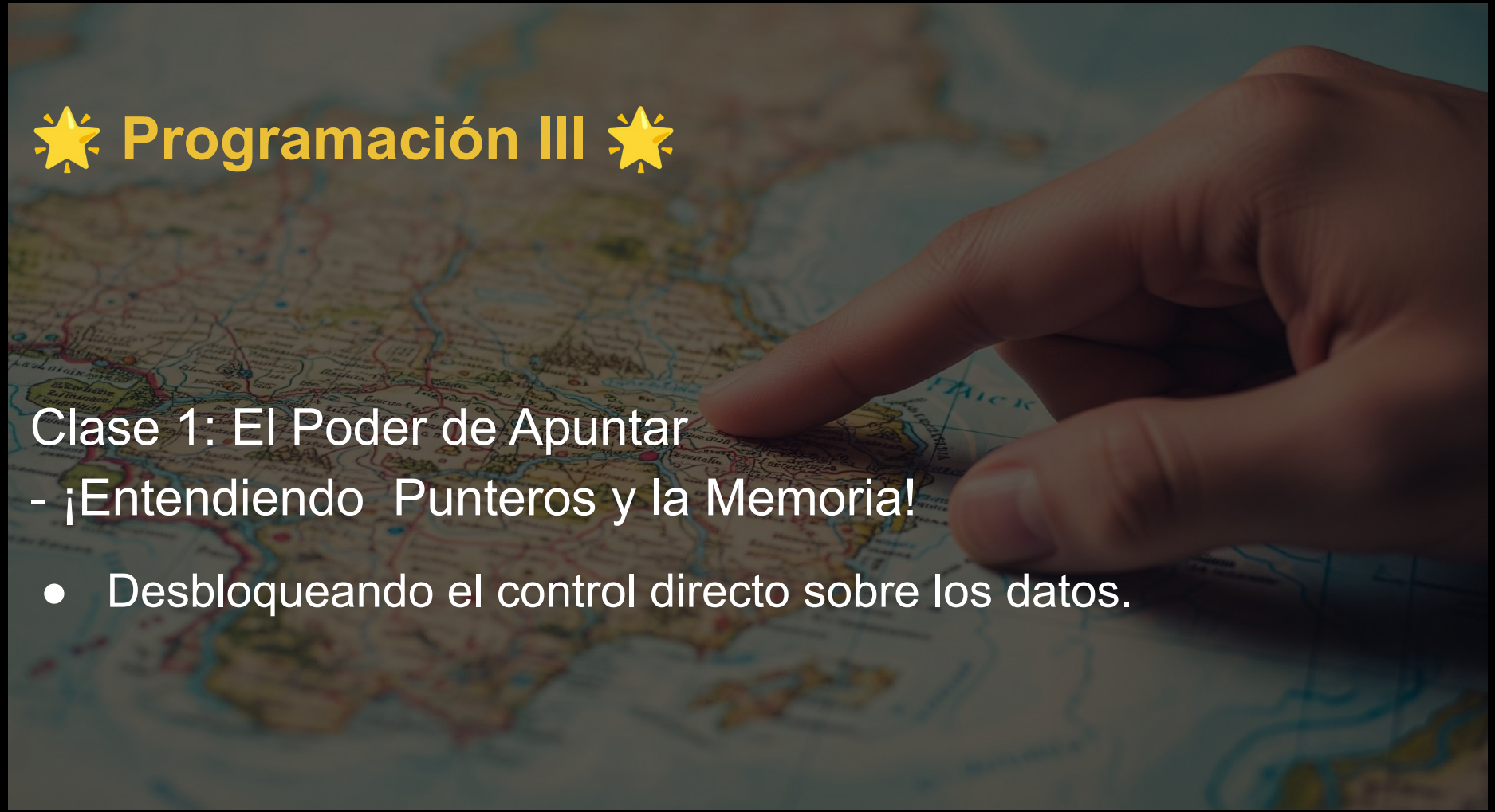


🌟 Programación III 🌟

Clase 1: El Poder de Apuntar

- ¡Entendiendo Punteros y la Memoria!

- Desbloqueando el control directo sobre los datos.

Qué Descubriremos Hoy?

Al finalizar esta clase, podrás:

- Visualizar cómo se organiza la memoria de la computadora (**stack vs. heap**).
- Definir con tus propias palabras qué es un puntero y cuál es su propósito esencial.
- Declarar e **inicializar punteros** en C++ correctamente.
- Manejar con confianza los operadores de dirección (**&**) y de indirección o de referencia (*).
- Comprender la diferencia crucial entre la dirección de memoria y el valor almacenado en ella.

El Lienzo de la Computadora: La Memoria

Organización de la Memoria (Simplificado):
Imagina casilleros numerados. Cada casillero tiene una dirección única. Ahí se guardan los datos (nuestros valores, variables).

Variable: *Un nombre simbólico para una ubicación de memoria donde guardamos un valor. (Ej: int edad = 30; -> 'edad' es el nombre, 30 es el valor, y está en una dirección específica).*

Stack (Pila): *Memoria estática, organizada, rápida. Variables locales.*

Heap (Montón): *Memoria dinámica, más flexible, la gestionamos nosotros (¡aquí entrarán los punteros a jugar un papel clave más adelante!).*

¿Qué es un Puntero?

¡El Mapa del Tesoro!

- *Un puntero es una variable especial.*
- *Su valor no es un dato común (como un número o texto), sino una **DIRECCIÓN DE MEMORIA** de otra variable.*
- *Nos dice dónde encontrar otro dato.*

Las Herramientas del Explorador: Operadores & y *

El Operador de Dirección `&` (Ampersand): Dame la Dirección

- Se usa con una variable normal.
- Devuelve la **dirección de memoria** donde esa variable está almacenada.
- Ej: `&miVariable` -> nos da la dirección de `miVariable`.

El Operador de Indirección (o Dereferencia) `*` (Asterisco): Ve a esa Dirección y Dame el Valor

- Se usa con una variable **puntero**.
- Accede al **valor almacenado en la dirección de memoria** a la que apunta el puntero.
- Ej: `*miPuntero` -> nos da el valor que está en la dirección que `miPuntero` guarda.

- **¡Cuidado!** El `*` se usa para declarar un puntero (`int *ptr;`) y también para dereferenciarlo (`valor = *ptr;`). El contexto lo diferencia.

Comprobemos si comprendimos

El Operador de Dirección `&` (Ampersand): Dame la Dirección

- `int numero = 25;`
- `int *punteroANumero; //` Declaración de un puntero
- `punteroANumero = &numero; //` Asignación: puntero guarda la DIRECCIÓN de numero
- `std::cout << punteroANumero; //` Imprimiría una dirección (ej. 0x7ffc...)
- `std::cout << *punteroANumero; //` Imprimiría el VALOR 25

- Es importante notar algo aquí. El asterisco `*` lo usamos para dos cosas:
  a. Cuando **declaramos** un puntero: `int *puntero;` Aquí le dice al compilador '**esta variable va a guardar direcciones de enteros**.
  b. Cuando **dereferenciamos** un puntero que ya tiene una dirección: `int valor = *puntero;` Aquí significa **ve a la dirección que guarda `puntero` y tráeme el valor que está allí**.
- No se confundan, el contexto nos dirá qué está haciendo el `*`. Es como la palabra **banco**', que puede ser un asiento o una institución financiera; sí el contexto de la oración nos lo aclara.

- **Si tengo `int x = 5; int *p = &x;`, ¿qué valor tiene `p`? ¿Y qué valor tiene `*p`?**
- **¿Cuál es la diferencia fundamental entre `p` y `*p` en este caso?**

¡Manos a la Obra! Punteros en Acción

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

int main() {
    int variable = 20;    // Una variable entera normal
    int *puntero;         // DECLARACIÓN de un puntero a un entero

    // ASIGNACIÓN: 'puntero' ahora guarda la dirección de 'variable'
    puntero = &variable;

    std::cout << "--- Información de 'variable' ---" << std::endl;
    std::cout << "Valor de 'variable': " << variable << std::endl;
    std::cout << "Direccion de 'variable' (&variable): " << &variable << std::endl;

    std::cout << "\n--- Informacion de 'puntero' ---" << std::endl;
    std::cout << "Contenido de 'puntero' (la direccion que guarda): " << puntero << std::endl;
    std::cout << "Direccion donde esta guardado el propio 'puntero' (&puntero): " << &puntero << std::endl;

    std::cout << "\n--- Accediendo al valor A TRAVES del puntero ---" << std::endl;
    std::cout << "Valor al que apunta 'puntero' (*puntero): " << *puntero << std::endl; // DEREFERENCIA

    // Modificando 'variable' A TRAVÉS del puntero
    std::cout << "\n--- Modificando a traves del puntero ---" << std::endl;
    *puntero = 30; // Ve a la dirección que guarda 'puntero' y cambia el valor allí a 30
    std::cout << "Nuevo valor de 'variable' (despues de *puntero = 30): " << variable << std::endl;
    std::cout << "Nuevo valor apuntado por 'puntero' (*puntero): " << *puntero << std::endl;

    return 0;
}
```

Descifrando el Código: Sintaxis y Propósito

Sintáxis clave:

- Declarar un puntero: `tipo_dato *nombre_puntero;`
  - Ej: `int *ptr_edad;` (puntero a un entero)
  - Ej: `char *ptr_letra;` (puntero a un caracter)
- Obtener dirección: `&nombre_variable`
- Asignar dirección a puntero: `nombre_puntero = &nombre_variable;`
- Dereferenciar (acceder al valor): `*nombre_puntero`
- Modificar valor a través de puntero: `*nombre_puntero = nuevo_valor;`

Propósito del Ejemplo:

- Demostrar cómo se declara un puntero.
- Mostrar cómo se le asigna la dirección de otra variable.
- Ilustrar cómo se accede y se modifica el valor de una variable indirectamente usando su puntero.
- Ver la diferencia entre la dirección almacenada por el puntero y el valor al que apunta.

¿Qué Vimos en la Consola?

```
--- Información de 'variable' ---
Valor de 'variable': 20
Direccion de 'variable' (&variable): 0x7ffe61922cfc

--- Informacion de 'puntero' ---
Contenido de 'puntero' (la direccion que guarda): 0x7ffe61922cfc
Direccion donde esta guardado el propio 'puntero' (&puntero): 0x7ffe61922d00

--- 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
```

Tu Laboratorio Virtual: OnlineGDB

URL: https://www.onlinegdb.com/
Ventajas:

- Gratis y accesible desde cualquier navegador.
- No requiere instalación.
- Soporta C++ (y muchos otros lenguajes).
- Permite escribir, compilar y ejecutar código rápidamente.
- Ideal para probar los ejemplos de clase y experimentar.

Pasos Rápidos:

1. Visita la URL.
2. Selecciona C++ como lenguaje (usualmente en la esquina superior derecha).
3. Escribe o pega tu código en el editor.
4. Presiona el botón Run (verde).
5. Observa la salida en la ventana inferior.

PUNTEROS EN ACCIÓN

¡Sigue Explorando el Universo de los Punteros!

Temas para Curiosos:

- **Punteros a `void` (`void*`):** Punteros genéricos que pueden apuntar a cualquier tipo de dato (¡pero hay que tener cuidado al usarlos!).
- **Aritmética de Punteros:** ¿Qué sucede si sumas 1 a un puntero? (Pista: no siempre avanza 1 byte).
- **Punteros y Arreglos:** Una relación muy íntima y poderosa.
- **Punteros a Punteros:** ¡Sí, puedes tener un puntero que apunta a otro puntero!
- **Puntero Nulo (`nullptr` en C++11 en adelante):** Un puntero que no apunta a ninguna dirección válida. ¡Muy importante para la seguridad!

Recursos Recomendados:

- Documentación de C++ (ej. cppreference.com)
- Tutoriales online (buscar C++ pointers tutorial for beginners)
- .ellibro.net Abstracción y estructura de datos en C++

🧠 Momento de Reflexión: ¿Por Qué y Para Qué?

Thinking ……. 

- Hemos visto que un puntero guarda una dirección y que podemos acceder/modificar un valor a través de él.
- Considerando esto,
  ¿en qué tipo de situaciones creen que esta capacidad de **manejo indirecto** de los datos podría ser **MÁS ÚTIL** o incluso **NECESARIA** para un programa real, en comparación con simplemente usar variables normales, comunes y corrientes?

Pistas:

- Funciones que necesitan modificar variables originales
- Estructuras de datos que cambian de tamaño
- Compartir grandes bloques de datos sin copiarlos todos
- ¿En qué más tú crees que sea necesaria esta capacidadl de manejo indirecto?.

🎮 ¡Juguemos con Punteros! El Cartero y las Cartas

https://bit.ly/prog3upds

- **El Profesor es el CPU/Memoria**
- **Variables (Cartas en Buzones)**

```
int edad = 25;
```

- **Punteros (Notas del Cartero)**

```
int *ptr_edad;

ptr_edad = &edad;
```

- Si les muestro la nota 'ptr_edad' (que dice @1001), ¿qué hay en la dirección que indica?

```
?
```

- Si ahora digo: 'Cartero, ve a la dirección de 'ptr_edad

```
*ptr_edad = 30;
```

- ¿Cuál es la dirección de la nota 'ptr_edad' misma?

```
?
```

Objetivo: Visualizar de forma lúdica la diferencia entre dirección y valor, y la indirección.

Recapitulación y Mirando Hacia Adelante

Hoy Descubrimos:

- La memoria como un conjunto de ubicaciones con direcciones.
- Los punteros como variables que guardan esas direcciones.
- Los operadores `&` (obtener dirección) y `*` (obtener valor en la dirección).
- ¡El poder de modificar datos indirectamente!

Próxima Clase: ¡Nos sumergimos en la **Memoria Dinámica**! Aprenderemos a solicitar memoria en tiempo de ejecución usando `new` y a liberarla con `delete`. Los punteros serán nuestros protagonistas.

Tarea/Sugerencia: Revisa el código de hoy. Experimenta con OnlineGDB creando tus propios ejemplos simples de punteros. ¡Intenta romperlo y entender por qué!

🌟 Programación III 🌟

Clase 2: ¡Forjando Memoria a Voluntad! new y delete

- El arte de solicitar y liberar memoria en tiempo de ejecución.

Dominando la Memoria Dinámica

Al finalizar esta clase, serás capaz de:

- **Explicar** la necesidad y el propósito de la asignación dinámica de memoria **(el Heap).**
- **Utilizar** el operador `new` para solicitar y asignar memoria en el heap en tiempo de ejecución.
- **Manejar** el operador `delete` para liberar correctamente la memoria asignada y prevenir fugas.
- **Distinguir** cuándo usar `delete` vs. `delete[]` (para arreglos dinámicos).
- **Reconocer** la importancia de manejar punteros nulos (`nullptr`) para la seguridad del código.

El Reino del Heap

Recordatorio Rápido: Stack vs. Heap

- **Stack (Pila):**
  - Memoria gestionada automáticamente (variables locales, parámetros de función).
  - Tamaño fijo conocido en tiempo de compilación.
  - Rápida, ordenada (LIFO - Last In, First Out).
- **Heap (Montón):**
  - Región grande de memoria disponible para el programa.
  - Asignación y liberación **manual** por el programador en **tiempo de ejecución**.
  - Flexible en tamaño, pero más lenta que el Stack.
  - ¡Aquí es donde usamos `new` y `delete`!

¿Por Qué Necesitamos el Heap?

- Cuando no sabemos el tamaño de los datos de antemano (ej. leer N elementos de un usuario).
- Cuando necesitamos que los datos existan más allá del ámbito de una función (que no se destruyan al salir de la función).
- Para estructuras de datos complejas que crecen y decrecen (listas, árboles, etc.).

El Poder de Crear: El Operador new

¿Qué Hace `new`?

1. Solicita un bloque de memoria del tamaño necesario desde el **Heap**.
2. Si tiene éxito, **devuelve un puntero** (la dirección) al inicio de ese bloque de memoria.
3. Si falla (ej. no hay suficiente memoria), puede lanzar una excepción (`std::bad_alloc`) o, en versiones más antiguas/configuraciones, devolver `nullptr`.

Sintaxis:

- Para un solo objeto: `puntero = new tipo_dato;`
  - Ej: `int *ptr_num = new int;` (reserva espacio para un entero)
  - Ej: `MiClase *ptr_obj = new MiClase();` (reserva espacio y llama al constructor)
- Para un arreglo de objetos: `puntero = new tipo_dato[tamaño];`
  - Ej: `double *ptr_arreglo_doubles = new double[10];` (reserva para 10 doubles)

¡Importante! La memoria asignada con `new` **persiste** hasta que sea explícitamente liberada con `delete` (o `delete[]`). No se libera automáticamente al salir del ámbito como la memoria del Stack.

La Responsabilidad de Liberar: El Operador delete

¿Qué Hace `delete`?

1. Recibe un puntero que apunta a memoria previamente asignada con `new`.
2. Libera ese bloque de memoria en el Heap, devolviéndolo al sistema para que pueda ser reutilizado.
3. **¡OJO!** `delete` NO elimina la variable puntero en sí, solo la memoria a la que apunta. El puntero aún existe y ahora apunta a memoria inválida (puntero colgante o dangling pointer).

Sintaxis:

- Para un solo objeto (asignado con `new tipo_dato`): `delete puntero;`
  - Ej: `delete ptr_num;`
  - Si `puntero` apunta a un objeto de clase, `delete` llama primero al destructor del objeto.
- Para un arreglo de objetos (asignado con `new tipo_dato[tamaño]`): `delete[] puntero_arreglo;`
  - Ej: `delete[] ptr_arreglo_doubles;`
  - ¡Es CRUCIAL usar `delete[]` para arreglos! Si no, solo se libera el primer elemento y se llama solo al destructor del primer objeto (si aplica), causando fugas y comportamiento indefinido.

Buenas Prácticas:

- Después de `delete puntero;`, es buena idea asignar `puntero = nullptr;`. Esto evita que se use accidentalmente el puntero colgante.
- Solo usar `delete` en punteros que apuntan a memoria asignada con `new`.
- No usar `delete` dos veces sobre el mismo puntero (doble liberación).

¡Manos a la Obra! Gestión Dinámica Completa (parte 1)

```cpp
#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
```

¡Manos a la Obra! Gestión Dinámica Completa (parte 2)

```
25
26      if (p_arreglo_doubles != nullptr) {
27          for (int i = 0; i < tamano_arreglo; ++i) {
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
50      return 0;
51  }
```

Código Desglosado: Buenas Prácticas Esenciales

Resumen

1. **Inicializar punteros a `nullptr`:** Antes de asignarles memoria con `new`.
2. **Solicitar memoria con `new` o `new[]`:** Guardar la dirección devuelta en un puntero.
3. **(Opcional) Verificar si `new` tuvo éxito:** (Más relevante en entornos con memoria muy limitada o sin excepciones).
4. **Usar la memoria:** A través del puntero (con `*` o `[]`).
5. **Liberar memoria con `delete` o `delete[]`:** ¡Absolutamente crucial! Usar la forma correcta.
6. **Asignar `nullptr` al puntero liberado:** Para evitar punteros colgantes.

Errores Comunes a Evitar:

- **Fugas de Memoria (Memory Leaks):** Olvidar `delete`/`delete[]`.
- **Punteros Colgantes (Dangling Pointers):** Usar un puntero después de que la memoria a la que apuntaba ha sido liberada.
- **Doble Liberación (Double Free):** Llamar a `delete`/`delete[]` dos veces sobre la misma memoria.
- **Usar `delete` en lugar de `delete[]` (o viceversa).**
- **Intentar `delete` memoria del Stack.**

La Huella de Nuestra Gestión de Memoria

```
Entero dinamico creado. Valor: 123 en direccion: 0x647edeb372b0
Memoria del entero dinamico liberada.
--- Arreglo Dinamico ---
Arreglo dinamico creado y llenado:
p_arreglo_doubles[0] = 0 en dir: 0x647edeb376e0
p_arreglo_doubles[1] = 1.5 en dir: 0x647edeb376e8
p_arreglo_doubles[2] = 3 en dir: 0x647edeb376f0
p_arreglo_doubles[3] = 4.5 en dir: 0x647edeb376f8
p_arreglo_doubles[4] = 6 en dir: 0x647edeb37700
Memoria del arreglo dinamico liberada.
...Program finished with exit code 0
```

Tu Laboratorio Virtual: OnlineGDB (Recordatorio)

URL: https://www.onlinegdb.com/

Recordatorio:

- GestiónDinamicaCompleta.cpp (revise en plataforma)

- Seleccionar C++, pegar código, Run, observar salida

Profundizando en la Dinámica de la Memoria

Temas Avanzados / Relacionados:

- **`std::bad_alloc`:** La excepción que `new` lanza cuando no puede asignar memoria. ¿Cómo se maneja? (try-catch).
- **Placement `new`:** Una forma especializada de `new` para construir objetos en memoria ya asignada.
- **Punteros Inteligentes (Smart Pointers en C++ v11 y posteriores):**
  - `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`.
  - ¡Grandes aliados para un manejo de memoria dinámico MÁS SEGURO y automático! (Previenen muchas fugas y errores).
- **Fugas de Memoria (Memory Leaks):** ¿Cómo detectarlas? (Herramientas como Valgrind en Linux/macOS).

Recursos:

- cppreference.com (para `new`, `delete`, `std::bad_alloc`, punteros inteligentes).
- Tutoriales sobre C++ smart pointers.

🤔 Reflexionando Sobre la Responsabilidad

1. Si `new` puede fallar (aunque raramente en ejemplos pequeños), ¿qué debería hacer un programa robusto si intenta crear un objeto dinámico esencial y `new` falla? ¿Simplemente dejar que el programa termine con error, o hay mejores estrategias?

2. Hemos dicho que es mala idea usar un puntero después de `delete` y que deberíamos asignarle `nullptr`. Si están trabajando en un equipo grande, y una parte del código libera memoria, ¿cómo se aseguran otras partes del código de no usar accidentalmente ese puntero ahora inválido?

🎮 ¡Juego de Roles! El Administrador de Memoria del Hotel

Objetivo: Visualizar las consecuencias de una mala gestión de memoria.

1. **El Profesor es el Recepcionista del Hotel Heap**.
2. **Estudiantes son Programas Huéspedes**.
3. **Habitaciones son bloques de memoria**.
4. **Proceso:**
   - Un Huésped (estudiante) pide una habitación: `int *huesped1_hab = new int;`
   - El Recepcionista le da una llave (dirección, ej. Habitación 101) y anota que está ocupada.
   - El Huésped usa la habitación: `*huesped1_hab = 5;`
   - Otro Huésped pide 3 habitaciones juntas: `char *huesped2_buff = new char[3];`
   - El Recepcionista dá llaves (ej. 201, 202, 203) y anota.
   - **El Desafío:**
     a. Si un Huésped se va (termina su función) **sin hacer `delete`** (sin devolver la llave)... ¿Qué pasa con la habitación? (¡Sigue marcada como ocupada! -> Fuga de memoria).
     b. Si un Huésped devuelve la llave (`delete huesped1_hab;`), pero luego intenta usarla de nuevo (`*huesped1_hab = 10;`)... ¿Qué pasa? (¡Intenta entrar a una habitación que ya no le pertenece! -> Puntero colgante).
     c. Si devuelve la llave de las 3 habitaciones con `delete huesped2_buff;` en lugar de `delete[] huesped2_buff;`... ¿Qué pasa? (El recepcionista solo libera la primera, ¡las otras dos siguen ocupadas a nombre del programa pero sin forma fácil de devolverlas!).

¿Qué fue lo que Logramos Hoy?

- Solicitar memoria del Heap cuando la necesitamos (`new` y `new[]`).
- La importancia crítica de liberar esa memoria (`delete` y `delete[]`).
- Las buenas prácticas para evitar errores comunes (fugas, punteros colgantes, `nullptr`).

¡Ustedes son responsables de la memoria que solicitan!

Mañana, veremos más **Aplicaciones de Punteros y Memoria Dinámica**, empezando a construir los cimientos para estructuras de datos flexibles como las listas enlazadas.

Tarea: Tomen el código de hoy. Intenten crear un pequeño programa que pida al usuario un tamaño para un arreglo, creen ese arreglo dinámicamente, lo llenan y luego lo liberan. ¡Experimenten!

🌟 Programación III 🌟

Clase 3: ¡Construyendo Cadenas de Datos!
Introducción a Listas Enlazadas

- Usando punteros para crear estructuras de datos flexibles y dinámicas.

Nuestro Plan de Enlace para Hoy

Al finalizar esta sesión, lograrás:

- **Comprender** por qué los arreglos tradicionales a veces no son suficientes y cuándo necesitamos estructuras de datos dinámicas.
- **Definir** qué es una lista enlazada y cuáles son sus componentes básicos (nodos).
- **Visualizar** cómo los punteros se utilizan para conectar nodos y formar una lista.
- **Implementar** la estructura de un nodo simple en C++ usando `struct` (o una clase básica).
- **Crear y enlazar manualmente** un par de nodos usando `new` y punteros.

¿Por Qué Necesitamos Algo Más que Arreglos?

Recordemos los Arreglos (Arrays):

- Colección de elementos del mismo tipo.
- Almacenados en memoria de forma **contigua** (uno después del otro).
- Acceso rápido a elementos por índice (ej. `miArreglo[5]`).

Sus Desventajas / Limitaciones:

1. **Tamaño Fijo:**
   - Debes declarar su tamaño al momento de la compilación (para arreglos estáticos en el Stack).
   - Si usas `new tipo[TAMAÑO];` en el Heap, el `TAMAÑO` se fija al momento de la creación.
   - **Problema:** ¿Qué pasa si no sabes cuántos elementos necesitarás?
     - Si declaras uno muy grande: Desperdicio de memoria.
     - Si declaras uno pequeño: Te quedas sin espacio.
2. **Inserción y Eliminación Costosas (en medio del arreglo):**
   - Para insertar un elemento en una posición específica (que no sea el final), debes desplazar todos los elementos siguientes.
   - Para eliminar un elemento, debes desplazar los elementos para rellenar el hueco.
   - ¡Esto puede ser muy lento si el arreglo es grande!

La Cadena Mágica: Listas Enlazadas Simples

¿Qué es una Lista Enlazada?

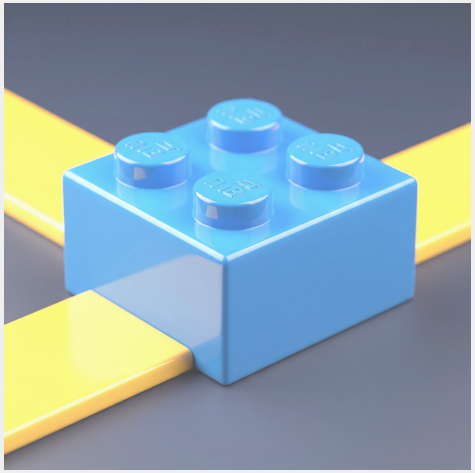
- Una colección **lineal** de elementos de datos llamados **nodos**.
- A diferencia de los arreglos, los nodos **NO necesariamente están contiguos en memoria**.
- Cada nodo contiene dos partes principales:
  1. **Dato(s):** La información que queremos almacenar (un entero, un string, un objeto complejo, etc.).
  2. **Puntero al Siguiente Nodo (`next`):** Una dirección de memoria que apunta al siguiente nodo en la secuencia. ¡Este es el enlace de la cadena!
- El último nodo de la lista apunta a **`nullptr`** (o **`NULL`**) para indicar el final.
- Se accede a la lista a través de un puntero al primer nodo, comúnmente llamado **`cabeza`** (o **`head`**).

Visualización:

- (Diagrama simple: [Dato|Siguiente*] -> [Dato|Siguiente*] -> [Dato|nullptr])
- `cabeza*` -> apunta al primer nodo.

El Bloque Constructor: El Nodo en C++

Definiendo un Nodo Simple (para una lista de enteros):



```
// Usaremos una struct para agrupar el dato y el puntero
struct Nodo {
    int dato;       // El valor que almacenamos en este nodo
    Nodo* siguiente; // Puntero al PRÓXIMO nodo en la lista
                     // ¡Un puntero a un objeto de su mismo tipo!

    // Constructor (opcional pero útil) para inicializar un nodo
    Nodo(int valor_dato) {
        dato = valor_dato;
        siguiente = nullptr; // Por defecto, un nuevo nodo no apunta a nada
    }
};
```

Puntos Clave:

- La estructura `Nodo` se contiene a sí misma (un puntero a `Nodo`). ¡Esto es lo que permite el enlace!
- El constructor simplifica la creación de nuevos nodos.
- `siguiente = nullptr;` es una buena práctica al crear un nodo aislado.

¡A Construir Nuestra Primera Mini-Cadena!

```cpp
#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; // O 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;

    return 0;
}
```

```
Creado primer nodo (cabeza) con dato: 10
Creado segundo nodo con dato: 20
Enlazando cabeza->siguiente con segundoNodo.
Creado tercer nodo con dato: 30
Enlazando segundoNodo->siguiente con tercerNodo.

Recorriendo la mini-lista:
Dato en cabeza: 10
Dato en el segundo nodo (via cabeza->siguiente): 20
Dato en el tercer nodo (via cabeza->siguiente->siguiente): 30

Liberando memoria...
Tercer nodo liberado.
Segundo nodo liberado.
Primer nodo (cabeza) liberado.


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

¿Qué Hicimos?

Pasos Clave Realizados:

1. **Definimos la Estructura `Nodo`:** Con un `dato` y un puntero `Nodo* siguiente`.

2. **Creamos Nodos Individuales Dinámicamente:** Usando `new Nodo(valor);` y guardando sus direcciones en punteros (`cabeza`, `segundoNodo`, etc.).

3. **Establecimos los Enlaces:** Modificamos el miembro `siguiente` de un nodo para que apuntara a la dirección de otro nodo.
   - `nodo_anterior->siguiente = nodo_actual;`

4. **Accedimos a Datos Anidados:** Usando la sintaxis de flecha (`->`) repetidamente para saltar de nodo en nodo: `cabeza->siguiente->siguiente->dato`.

5. **Liberamos la Memoria Dinámica:** Usando `delete` para cada nodo creado con `new`, ¡con cuidado de no perder referencias!

Nuestra Mini-Cadena en la Consola

```
Creado primer nodo (cabeza) con dato: 10
Creado segundo nodo con dato: 20
Enlazando cabeza->siguiente con segundoNodo.
Creado tercer nodo con dato: 30
Enlazando segundoNodo->siguiente con tercerNodo.

Recorriendo la mini-lista:
Dato en cabeza: 10
Dato en el segundo nodo (via cabeza->siguiente): 20
Dato en el tercer nodo (via cabeza->siguiente->siguiente): 30

Liberando memoria...
Tercer nodo liberado.
Segundo nodo liberado.
Primer nodo (cabeza) liberado.


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

Laboratorio Virtual: OnlineGDB
(¡Tu Mejor Amigo para Experimentar!)

URL: https://www.onlinegdb.com/

Recuerda:

- ★ Perfecto para estos ejemplos.
- ★ Experimenta:
  - ¿Qué pasa si olvidas un `delete`?
  - ¿Si intentas acceder a `cabeza->siguiente->siguiente->siguiente` (que sería `nullptr`)?

    (¡Con cuidado!).

¡La Aventura de las Listas Apenas Comienza!

Próximos Conceptos (que construiremos sobre esto):

- Clase ListaEnlazada Completa: Con métodos para insertarAlInicio, insertarAlFinal, eliminarNodo, buscarNodo, etc.
- Listas Doblemente Enlazadas: Nodos con punteros a siguiente Y anterior.
- Listas Circulares: El último nodo apunta de nuevo al primero.
- Otras Estructuras Basadas en Nodos y Punteros: Árboles, Grafos.

Recursos:

- Tutoriales: C++ linked list tutorial.
- Libro del Curso (referencia): Estructura de Datos en C por Joyanes & Zahonero (Capítulos sobre listas enlazadas, ej. Cap 4 o 5).
- Otro Libro del referencia: Programación en C++: algoritmos, estructuras de datos y objetos (2a. ed.) por Luis Joyanes Aguilar (17.2 Operaciones en Listas enlazadas. Pg. 630).

💡 Desafío Mental: Ventajas y Desventajas

- Hemos visto que las listas enlazadas parecen resolver el problema del tamaño fijo de los arreglos y facilitan (conceptualmente) la inserción/eliminación en medio (solo reasignando punteros, sin mover muchos datos).

- Pero, ¿pueden pensar en alguna desventaja que podrían tener las listas enlazadas en comparación con los arreglos? ¿Alguna situación donde un arreglo seguiría siendo mejor? (Pistas: Acceso a elementos, uso de memoria).

Eslabones del Conocimiento: Lo Aprendido Hoy

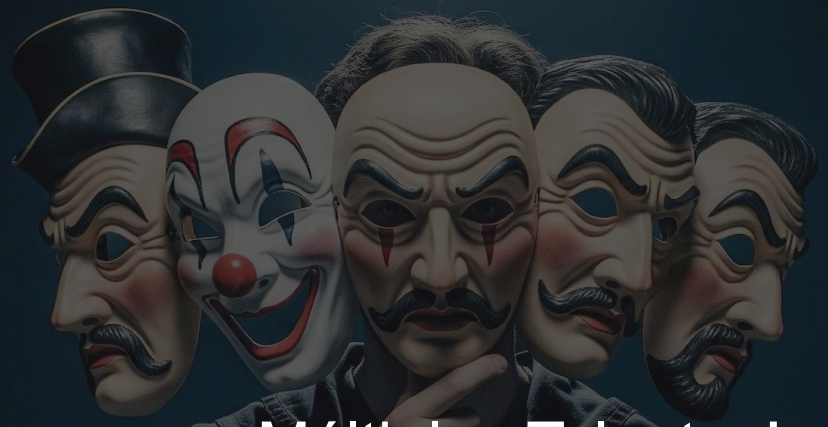
Hoy Logramos:

- Entender las limitaciones de los arreglos para datos de tamaño variable.
- Definir y visualizar el concepto de nodo y lista enlazada.
- Implementar la estructura de un nodo en C++.
- Crear y enlazar manualmente nodos usando `new` y punteros.
- Reflexionar sobre ventajas y desventajas.

Próxima Clase: ¡**Sobrecarga de Funciones**! Aprenderemos cómo una misma función puede tener múltiples personalidades para trabajar con diferentes tipos de datos o diferentes números de argumentos.

Desafío Personal: Dibuja en papel una lista enlazada de 4 nodos. Luego, simula (dibujando) cómo eliminarías el segundo nodo y cómo re-enlazarías el primero con el tercero. ¿Qué punteros necesitas modificar?

🌟 Programación III 🌟



Clase 4: ¡Funciones con Múltiples Talentos!

La Sobrecarga de Funciones

- Escribiendo código más intuitivo y flexible con un solo nombre para diferentes tareas.

Los Talentos que Descubriremos Hoy

Al finalizar esta clase, podrán:

- **Definir** qué es la sobrecarga de funciones (function overloading) en C++.
- **Identificar** las reglas precisas que el compilador de C++ utiliza para permitir la sobrecarga de funciones.
- **Distinguir** entre una función sobrecargada y una redefinición errónea de una función.
- **Comprender** cómo el compilador selecciona la versión correcta de una función sobrecargada a llamar (resolución de sobrecarga).
- **Valorar** los beneficios de la sobrecarga en términos de claridad y flexibilidad del código.

Un Nombre, Múltiples Personalidades

Definición de Sobrecarga de Funciones:

- En C++, la sobrecarga de funciones permite que **múltiples funciones en el mismo ámbito compartan el mismo nombre**, siempre y cuando sus **listas de parámetros (firmas)** sean diferentes.
- El compilador utiliza estas diferencias en los parámetros para distinguir qué función específica llamar.

¿Qué es la Firma de una Función (para sobrecarga)?

- En C++, para propósitos de sobrecarga, la firma de una función se compone de:
  1. El **nombre** de la función.
  2. El **número, tipo y orden** de sus parámetros.
- **Importante:** El **tipo de retorno de la función NO forma parte de la firma** para la sobrecarga. ¡No puedes sobrecargar una función basándote solo en un tipo de retorno diferente!

Ejemplo Conceptual:

- `void imprimir(int numero);`
- `void imprimir(double numeroFlotante);`
- `void imprimir(std::string texto);`
- `void imprimir(int num1, int num2);`
- Todas estas funciones se llaman `imprimir`, pero son distintas para el compilador debido a sus parámetros.

Las Reglas del Juego de la Sobrecarga

Para que un conjunto de funciones se considere sobrecargado correctamente, deben:

1. **Compartir el Mismo Nombre.** (Obvio, pero fundamental).
2. **Tener una Lista de Parámetros Diferente.** Esta diferencia puede ser en:
   - **Número de parámetros:**
     - `void func();`
     - `void func(int a);`
     - `void func(int a, int b);`
   - **Tipo de parámetros:**
     - `void procesar(int dato);`
     - `void procesar(double dato);`
     - `void procesar(std::string dato);`
   - **Orden de los tipos de parámetros:**
     - `void configurar(int id, std::string nombre);`
     - `void configurar(std::string nombre, int id);`
   - **Calificadores `const` y `volatile` en parámetros (para punteros y referencias):**
     - `void operar(int* ptr);`
     - `void operar(const int* ptr);` (Consideradas diferentes para sobrecarga).

NO se puede sobrecargar basándose ÚNICAMENTE en:

- El tipo de retorno.
- Convenciones de nombres de parámetros (ej. `void f(int x);` vs `void f(int y);` son la misma).
- Uso de `typedef` que resulten en el mismo tipo subyacente.

¡Talentos en Acción! Sobrecarga de **sumar**

```cpp
#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;
}
```

```cpp
// 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.

    return 0;
}
```

El Código Bajo la Lupa

Sintaxis Utilizada:

- Múltiples definiciones de función con el **mismo nombre** (`sumar`).
- Cada definición varía en el **tipo** y/o **número** de sus parámetros.
- El tipo de retorno puede variar entre funciones sobrecargadas (ej. `int` para suma de enteros, `std::string` para concatenación), pero esto *no* es lo que las diferencia para la sobrecarga.

Propósito del Ejemplo:

- Ilustrar cómo se definen funciones sobrecargadas para diferentes tipos de datos y diferente cantidad de argumentos.
- Mostrar cómo el compilador selecciona la función apropiada en tiempo de compilación basándose en los argumentos de la llamada (esto se llama **enlace estático** o **resolución de sobrecarga en tiempo de compilación**).
- Demostrar la flexibilidad que ofrece la sobrecarga para crear interfaces de función más intuitivas.

La Magia del Compilador: Salida Detallada

```
Suma de enteros (5, 3): Ejecutando sumar(int, int)... 8
Suma de doubles (5.5, 3.3): Ejecutando sumar(double, double)... 8.8
Concatenacion de strings ("Hola, ", "Mundo!"): Ejecutando sumar(const std::string&, const std::string&)... Hola, Mundo!
Suma de tres enteros (1, 2, 3): Ejecutando sumar(int, int, int)... 6


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

Suma de enteros (5, 3): Ejecutando sumar(int, int)... 8

Suma de doubles (5.5, 3.3): Ejecutando sumar(double, double)... 8.8

Concatenacion de strings (Hola, , Mundo!): Ejecutando sumar(const std::string&, const std::string&)... Hola, Mundo!

Suma de tres enteros (1, 2, 3): Ejecutando sumar(int, int, int)... 6

¡A Sobrecargar se Ha Dicho! OnlineGDB

URL: https://www.onlinegdb.com/

Sugerencia de Práctica:

- Toma el ejemplo de `sumar`.
- Intenta añadir OTRA versión sobrecargada. Por ejemplo, una que sume dos `float`.
- Intenta crear una llamada ambigua a propósito (ej. si solo tuvieras `sumar(int, double)` y `sumar(double, int)` y llamarás con `sumar(5, 5)` que son dos `int`) y observa qué error te da el compilador.

Ampliando el Repertorio de Talentos

Temas Relacionados y Avanzados:

- **Resolución de Sobrecarga y Ambigüedad:** ¿Qué pasa si una llamada podría coincidir con múltiples funciones sobrecargadas mediante conversiones de tipo? Reglas de mejor coincidencia.
- **Sobrecarga de Operadores:** ¡Sí, también puedes cambiar cómo funcionan operadores como `+`, `-`, `<<`, `>>` para tus propias clases! (Muy poderoso).
- **Argumentos por Defecto y Sobrecarga:** ¿Cómo interactúan? A veces pueden llevar a ambigüedades.
- **Plantillas de Funciones (Function Templates):** Una forma aún más genérica de escribir funciones que funcionan con muchos tipos, a menudo usada junto o en lugar de la sobrecarga manual extensa.

Recursos:

- cppreference.com (buscar function overloading, operator overloading).
- Tutoriales sobre C++ overload resolution rules.

🧐 Sobrecarga: ¿Cuándo Sí y Cuándo Quizás No?

- Hemos visto que la sobrecarga puede hacer el código más intuitivo al usar el mismo nombre para operaciones conceptualmente similares.

- Pero, ¿podría haber situaciones donde usar la sobrecarga en exceso o de manera inapropiada podría hacer el código *más confuso* o *difícil de entender*?

- Den un ejemplo o describan un escenario donde quizás sería mejor usar nombres de función diferentes, aunque las funciones hagan algo vagamente parecido.

🕵️ ¡Detective de Firmas!

- A ver, **void print(int x)**; y **void print(double y);**. ¿Son sobrecargas válidas? ¿Sí o No?

- **void process()**; y **void process(int count = 0);** (¡Cuidado con argumentos por defecto!)

El de argumentos por defecto (`process(int count = 0);`) es interesante porque `process();` podría llamar a esa versión, lo que podría generar **ambigüedad** si también existe `void process();` sin argumentos.

Hoy Aprendimos Múltiples Talentos

- Qué es la sobrecarga de funciones: mismo nombre, diferentes firmas de parámetros.
- Las reglas exactas para sobrecargar (número, tipo, orden de parámetros; el tipo de retorno NO cuenta).
- Cómo el compilador elige la función correcta.
- Los beneficios de claridad y flexibilidad que aporta.

Próxima Clase: Continuaremos explorando las Aplicaciones y Utilidades de la Sobrecarga de Funciones, viendo ejemplos más complejos y cómo se usa en constructores de clases.

Desafío: Piensa en una clase que podrías diseñar (ej. Calculadora, FiguraGeometrica). ¿Qué métodos de esa clase podrían beneficiarse de la sobrecarga?