Informe de Implementación y Resultados (C++)

Joshua Mendez

Sistemas Operativos — **Fecha:** Octubre 2025

Resumen

Se implementó un conjunto de ejercicios cortos en C++ para practicar direccionamiento de memoria: variables, punteros, referencias, arreglos y memoria dinámica (matrices 2D). El objetivo general fue identificar segmentos de memoria (stack/heap/data/code) y manipular datos a través de punteros y referencias, en línea con la guía de "Trabajo de clase".

Entorno y forma de ejecución

- **Compilador:** `g++ 13` — `-std=c++17 -O2 -Wall -Wextra -pedantic`
- **Sistema operativo:** Linux x86_64 (tested en entorno tipo GCC/docker)
- **Cómo compilar todo (ejemplos):**

Listing 1: Comandos de compilación por ejercicio

```
# Ejercicio 1
g++ -std=c++17 -O2 -Wall -Wextra -pedantic -o actividad1 actividad1.cpp
# Ejercicio 2
g++ -std=c++17 -O2 -Wall -Wextra -pedantic -o actividad2 actividad2.cpp
# Ejercicio 3
g++ -std=c++17 -O2 -Wall -Wextra -pedantic -o actividad3 actividad3.cpp
# Ejercicio 4
g++ -std=c++17 -O2 -Wall -Wextra -pedantic -o actividad4 actividad4.cpp
```

Notas: No se requieren librerías externas. Las direcciones de memoria impresas variarán en cada ejecución.

Mapa de ejercicios

Ejercicio	Archivo	Objetivo breve
1	`actividad1.cpp`	Variable entera, puntero para modificar su valor e impresión de segmentos de memoria.
2	`actividad2.cpp`	Puntero y referencia a una variable; comparación de direcciones y segmentos de memoria.
3	`actividad3.cpp`	Arreglo `int[5]`, acceso con punteros, modificación *in-place* y reporte de direcciones/tamaños.
4	`actividad4.cpp`	Matriz 2D dinámica (layout contiguo) con `unique_ptr`, llenado/impresión y análisis de direcciones; muestra de segmentos.

1. Ejercicio 1 — Variable y puntero (segmentos básicos)

Archivo: `actividad1.cpp`

Descripción del problema

Declarar una variable entera, mostrar su valor y dirección; crear un puntero a esa variable para modificar su valor indirectamente; imprimir ejemplos de direcciones correspondientes a stack, heap y sección estática.

Enfoque de implementación

- **Estructuras/algoritmos:** variables escalares, puntero `int*`, función auxiliar `memoria()` que reserva en heap y usa `static` para evidenciar segmentos.
- **Pasos clave:**
  - `int x = 42;` → imprime valor y `&x`.
  - `int* p = &x; *p = 99;` → modifica indirectamente.
  - `memoria()` crea `stackVar`, `heapVar = new int(20)` y `static int staticVar`, imprime direcciones y libera `heapVar`.
- **Complejidad:** Tiempo $O(1)$; Espacio $O(1)$.

Fragmento ilustrativo

```
int x = 42;
int* p = &x;
*p = 99; // modificaci n indirecta
```

Casos de prueba y salidas

Ejecución:

```
./actividad1
```

Salida típica (las direcciones varían):

```
Valor inicial de x: 42
Direccion de x: 0x7ffc...
Nuevo valor de x (via puntero): 99
Direccion de x (igual que antes): 0x7ffc...

=== Segmentos de memoria ===
stackVar (Stack): 0x7ffc... Valor: 10
heapVar (Heap):  0x55ab... Valor: 20
staticVar (Code/Data): 0x55ab... Valor: 30
```

Posibles mejoras

- Imprimir tamaño de tipos con `sizeof` para reforzar conceptos.
- Verificar `new` con `try/catch (std::bad_alloc)` cuando sea pertinente.

2. Ejercicio 2 — Puntero y referencia; comparación de direcciones

Archivo: `actividad2.cpp`

Descripción del problema

Crear un programa con puntero y referencia a una variable; modificar el valor a través de ambos y mostrar direcciones de `x`, del puntero y de la referencia; además, ilustrar segmentos de memoria.

Enfoque de implementación

- **Estructuras/algoritmos:** `int* ptr`, `int& ref`, función `memoria()` similar al ejercicio 1.
- **Pasos clave:**
  - `ptr = &x; *ptr = 20;` → cambio vía puntero.
  - `int& ref = x; ref = 30;` → cambio vía referencia.
  - Imprime `&x`, `ptr`, `&ptr`, `&ref` (igual a `&x`).
  - `memoria()`: muestra stack/heap/data y libera lo reservado.
- **Complejidad:** Tiempo $O(1)$; Espacio $O(1)$.

Fragmento ilustrativo

```
int* ptr = &x;
*ptr = 20;
int& ref = x;
ref = 30; // misma direcci n que &x
```

Casos de prueba y salidas

Ejecución:

```
./actividad2
```

Salida típica:

```
Valor inicial de x: 10
Valor via *ptr: 10
Nuevo valor de x tras *ptr=20: 20
Nuevo valor de x tras ref=30: 30

--- Direcciones ---
Direccion de x (&x):        0x7ffc...
Contenido de ptr (direccion de x): 0x7ffc...
Direccion del puntero (&ptr): 0x7ffc...
Direccion de la referencia (&ref): 0x7ffc... (igual a &x)

=== Segmentos de memoria ===
...
```

Posibles mejoras

- Añadir comentarios sobre *aliasing* y *const-correctness*: `int* const` vs `const int*`.

3. Ejercicio 3 — Arreglo y aritmética de punteros

Archivo: `actividad3.cpp`

Descripción del problema

Declarar un arreglo de 5 enteros, modificar sus elementos usando un puntero con aritmética (`*(p+i)`), e imprimir contenido y direcciones relevantes (incluyendo `sizeof`).

Enfoque de implementación

- **Estructuras/algoritmos:** arreglo estático `int a[5]`, puntero `int* p = a`.
- **Pasos clave:**
  - Multiplicar por 10 cada elemento usando `*(p + i)`.
  - Imprimir `a[i]`, `a` (decay a `&a[0]`), `&a[0]`, `&p`, `p`, `&a`.
  - Comparar `sizeof(a)` (tamaño total del arreglo) vs `sizeof(*p)` (tamaño de `int`).
- **Complejidad:** Tiempo $O(n)$ con $n = 5$; Espacio $O(1)$.

Fragmento ilustrativo

```
int a[5] = {1,2,3,4,5};
int* p = a;
for (int i = 0; i < 5; ++i) *(p + i) *= 10;
```

Casos de prueba y salidas

Ejecución:

```
./actividad3
```

Salida típica:

```
Contenido del array:
a[0] = 10
a[1] = 20
a[2] = 30
a[3] = 40
a[4] = 50

--- Direcciones ---
Direccion de a (decay)    : 0x7ffc...   (== &a[0])
Direccion de a[0]         : 0x7ffc...
Direccion del puntero p   : 0x7ffc...
Contenido de p            : 0x7ffc...
Direccion de todo el array: 0x7ffc...

sizeof(a)   = 20 bytes
sizeof(*p)  = 4 bytes (tama o de un int)
```

Posibles mejoras

- Ejemplificar desbordamiento por índice y cómo prevenirlo.
- Mostrar `std::array<int,5>` como alternativa más segura.

4. Ejercicio 4 — Matriz 2D dinámica (layout contiguo) y segmentos

Archivo: `actividad4.cpp`

Descripción del problema

Reservar una matriz 2D dinámica usando un bloque contiguo (`std::unique_ptr<int[]>data`) y un arreglo de punteros a filas (`std::unique_ptr<int*[]>A`). Llenar la matriz, imprimirla y reportar direcciones clave. Además, una función `memoria` imprime direcciones representativas de code/data/stack/heap.

Enfoque de implementación

- **Estructuras/algoritmos:** administración de memoria con `std::unique_ptr`, confección de vista 2D sobre bloque contiguo, E/S con `cin`.
- **Pasos clave:**
  - Leer `filas` y `cols` (valida > 0).
  - `data = std::make_unique<int[]>(filas*cols);`
  - `A = std::make_unique<int*[]>(filas);` y `A[i] = data.get() + i*cols;` (filas apuntan dentro del bloque contiguo).
  - Llenado secuencial `A[i][j] = i*cols + j;`

- Impresión de matriz y direcciones (`data.get()`, `A.get()`, `A[0]`, `&A[0][0]`).
- `memoria(A.get(), filas, cols)` demuestra segmentos y coherencia de punteros.

- **Complejidad:** Tiempo $O(filas \cdot cols)$; Espacio $O(filas \cdot cols)$ para el bloque contiguo + $O(filas)$ para los punteros a fila.

Fragmento ilustrativo

```
std::unique_ptr<int[]> data(new int[filas * cols]);
std::unique_ptr<int*[]> A(new int*[filas]);
for (int i = 0; i < filas; ++i) A[i] = data.get() + i * cols;
```

Casos de prueba y salidas

Entrada de ejemplo:

```
Filas: 2
Columnas: 3
```

Ejecución:

```
./actividad4
```

Salida típica (parcial):

```
Matriz:
0 1 2
3 4 5

=== Direcciones (contiguo) ===
data (bloque contiguo):    0x55ab...
A (punteros a filas):      0x55ab...
A[0]:                      0x55ab...
&A[0][0]:                  0x55ab...

=== Segmentos de memoria ===
Direccion de funcion memoria (CODE): 0x55ab...
staticVar (DATA): 0x55ab...
stackVar  (STACK): 0x7ffc...
heapVar   (HEAP):  0x55ab...
```

Observaciones y casos borde

Validación de entrada: el programa retorna temprano si `filas<=0` o `cols<=0`.

Importante (gestión de memoria): al final del `main` se hace:

```
for (int i = 0; i < filas; ++i) delete[] A[i];
delete[] A.get();
```

Esto **es incorrecto** porque:

1. `A[i]` **no** fue creado con `new[]`; cada `A[i]` apunta dentro de `data.get()`. Hacer `delete[] A[i]` produce comportamiento indefinido.

2. `A` y `data` son `unique_ptr`, por lo que no se deben liberar manualmente; el destructor se encarga. **Corrección sugerida:**

```
// Nada de delete[] manual; dejar que unique_ptr libere:
// A.reset(); data.reset();  // opcional, o dejar que salgan de scope
```

Si se quisieran filas no contiguas (cada fila con `new int[cols]`), entonces sí iría un bucle `delete[] A[i]` y luego `delete[] A`, o mejor, usar `std::vector`.

Conclusiones

Se cumplieron los objetivos del trabajo práctico sobre direccionamiento de memoria: se observó cómo cambian valores vía punteros y referencias, cómo decae el nombre de un arreglo a puntero y cómo organizar una matriz 2D sobre un bloque contiguo. Los tres primeros ejercicios son correctos y auto-contenidos. El cuarto ilustra un patrón eficiente (layout contiguo) pero presenta un **bug** de liberación de memoria que debe corregirse dejando que `unique_ptr` gestione la vida de los recursos, o reescribiendo la reserva si se desean filas independientes.

Nota: las direcciones de memoria mostradas son de ejemplo y variarán en cada ejecución.