



UNIVERSIDAD MARIANO GÁLVEZ DE GUATEMALA
FACULTAD DE INGENIERÍA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN Y CIENCIAS DE LA
COMPUTACIÓN



JOSÉ DANIEL VÉLIZ VELÁSQUEZ – 9941-24-12576

Guatemala, febrero 2026



Parte I – Análisis de Memoria

Salida del programa:

20

20

20

Explicación:

La variable x inicialmente almacena el valor 8. El puntero p guarda la dirección de memoria de x, y el puntero q recibe la misma dirección al asignársele p. Cuando se ejecuta `*q = 20`, se modifica directamente el valor almacenado en la dirección apuntada, que corresponde a x. Por lo tanto, x cambia su valor a 20. Como p y q apuntan a la misma dirección, al desreferenciarlos también muestran 20.

¿Cuántos espacios de memoria distintos existen en el stack?

Existen tres espacios distintos en el stack: uno para la variable entera x, uno para el puntero p y uno para el puntero q. Sin embargo, solo hay un entero almacenado, ya que p y q comparten la misma dirección.

¿Existe aliasing?

Sí, existe aliasing. Esto ocurre porque dos punteros distintos (p y q) hacen referencia al mismo espacio de memoria (la dirección de x). Cualquier modificación realizada mediante uno de ellos afecta al mismo dato.

Esquema de memoria:

STACK:

x -> 20

p -> &x

q -> &x

HEAP:

(No se utiliza memoria dinámica en este fragmento)



Parte II – Memoria Dinámica

Programa propuesto:

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

int main() {
    int* p = new int; // Reserva dinámica

    *p = 50;          // Asignación por desreferenciación

    cout << "Dirección almacenada en p: " << p << endl;
    cout << "Valor almacenado en esa dirección: " << *p << endl;
    cout << "Dirección del puntero p: " << &p << endl;

    delete p;        // Liberación de memoria
    p = nullptr;      // Evitar puntero colgante

    return 0;
}
```

En este programa se reserva memoria en el heap para un entero. Se asigna el valor 50 utilizando desreferenciación. Posteriormente, se imprime la dirección apuntada, el valor contenido y la dirección propia del puntero. Finalmente, se libera correctamente la memoria y se asigna nullptr para evitar un puntero colgante.

Parte III – Análisis de Error

Código analizado:

```
int* p = new int(15);
int* q = p;

delete p;

cout << *q << endl;
```

Análisis técnico:

El error conceptual es el uso de memoria después de haber sido liberada. Cuando se ejecuta `delete p`, el espacio reservado en el heap queda invalidado. Sin embargo, el puntero `q` sigue almacenando la misma dirección, pero esa dirección ya no pertenece al programa. Esto se conoce como 'dangling pointer' (puntero colgante).



Puede funcionar aparentemente porque la memoria liberada no necesariamente es sobrescrita inmediatamente por el sistema. Mientras el contenido no cambie, podría imprimirse el valor anterior (15), pero el comportamiento es indefinido.

Se está violando el principio de comportamiento indefinido definido por el estándar de C++, específicamente el acceso a memoria ya liberada. El estándar establece que cualquier uso de un puntero después de delete produce undefined behavior.

Parte IV – Memory Leak

Código original:

```
int* p = new int(10);  
p = new int(30);
```

¿Existe memory leak?

Sí, existe un memory leak. El primer bloque de memoria reservado con `new int(10)` pierde su referencia cuando `p` es reasignado con una nueva dirección. Al no conservar la dirección original, no es posible liberarla posteriormente.

Corrección del código:

```
int* p = new int(10);  
  
delete p;      // Liberar antes de reasignar  
p = new int(30);  
  
delete p;      // Liberar al finalizar  
p = nullptr;
```

En la versión corregida, la memoria previa se libera antes de reasignar el puntero. De esta forma se evita la pérdida de memoria y se mantiene un manejo correcto del heap.