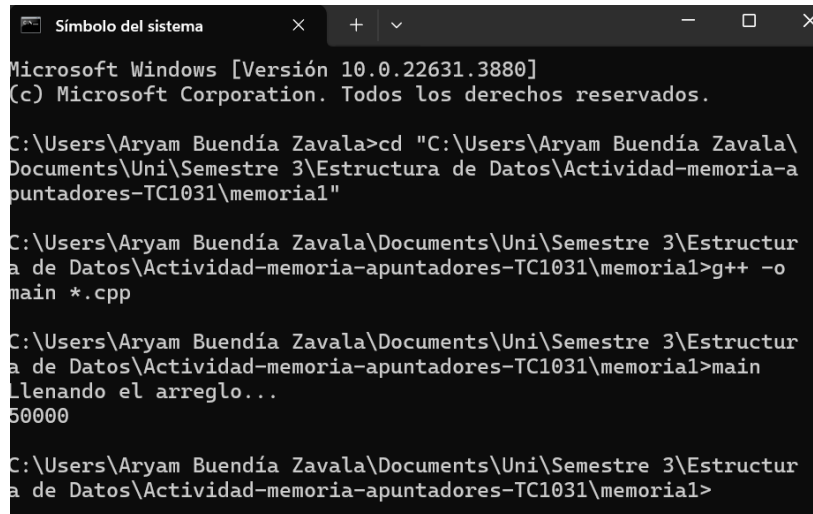


Aryam Lizette Buendía Zavala A01659465

Actividad Apuntadores y Memoria

Caso 1: Stack

Compilando y ejecutando el Código como está, salió esto:



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.22631.3880]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Aryam Buendía Zavala>cd "C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memorial"

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memorial>g++ -o main *.cpp

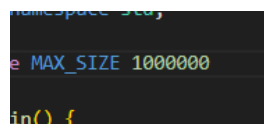
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memorial>main
llenando el arreglo...
50000

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memorial>
```

Tiene sentido porque si al inicio del programa el MAX_SIZE es 100000, el cout del MAX_SIZE/2 es 50000.

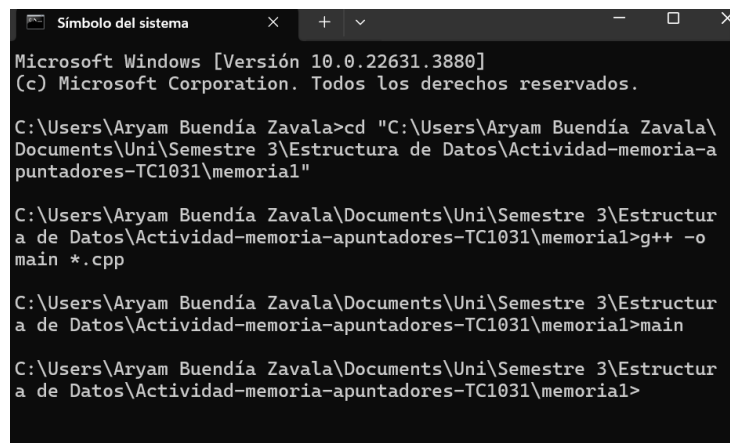
Prueba 1:

Después le agregué al MAX_SIZE un 0 más y yo no ejecutó.



```
namespace std;
const int MAX_SIZE = 1000000;

int main() {
```



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.22631.3880]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Aryam Buendía Zavala>cd "C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memorial"

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memorial>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memorial>main

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memorial>
```

Ya no corrió el programa porque el stack ya está lleno.

Prueba 2: Probé ahora quitar ceros JAJAJA. A ver si me funcionaba.

```
<_SIZE 10000
```

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria1>g++ -o main *.cpp
```

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria1>main
Llenando el arreglo...
5000
```

Con 10000 sí funcionó todo bien, y si quitaba ceros también sin problema.

Caso 2: Heap

```
11 #include <iostream>
12
13 using namespace std;
14
15 #define MAX_SIZE 1000000000
16
17 int main() {
18     long int* arrayPtr = nullptr;
19
20     cout << "Asking for: "
21           << (sizeof(long int) * MAX_SIZE) / (1024 * 1024 * 1024)
22           << " Gbytes\n";
23
24     arrayPtr = new(nothrow) long int[MAX_SIZE];
25     if (!arrayPtr) {
26         cerr << "Not enough memory" << endl;
27         return 1;
28     }
29
30     cout << "Got them! Filling the array with long ints...\n";
31     for (long int i = 0; i < MAX_SIZE; i++)
32         arrayPtr[i] = i;
33
34     cout << arrayPtr[MAX_SIZE / 2] << "\n";
35
36     cout << "Revisa cuanto utiliza en el administrador de procesos de
37     cout << "Presiona ENTER para liberar la RAM y terminar";
38     cin.ignore();
39
40     delete[] arrayPtr;
41     arrayPtr = nullptr;
42
43     return 0;
44 }
```

Al inicio, compilé y corrí el programa así como estaba y salió esto:

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2>g++ -o main *.cpp
memoria2.cpp: In function 'int main()':
memoria2.cpp:24:46: error: size of array is too large
    arrayPtr = new(nothrow) long int[MAX_SIZE];
                                   ^
```

Ni siquiera me dejaba ejecutarlo, así que decidí bajar el MAX SIZE 2 ceros hasta que me dejara ejecutarlo.

```
AX_SIZE 10000000
```

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2>main
Asking for: 0 Gbytes
Got them! Filling the array with long ints...
5000000
Revisa cuanto utiliza en el administrador de procesos de tu sistema operativo
Presiona ENTER para liberar la RAM y terminar
```

Aquí ya vi que sí se ejecutó, así que a partir de aquí empecé a hacer los cambios solicitados.

Cuando cambié el long int por string me salió esto. No sale ningún número.

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2>main
Asking for: 0 Gbytes
Got them! Filling the array with long ints...
@
Revisa cuanto utiliza en el administrador de procesos de tu sistema operativo
Presiona ENTER para liberar la RAM y terminar
```

```
MAX_SIZE 100000000
```

Regresé el string a long int y agregué un cero más.

De plano con 100000000 es cuando funcionaba, con más ceros marcaba error.

```

#define MAX_SIZE 1000000000

int main() {
    long int* arrayPtr = nullptr;

    cout << "Asking for: "
          << (sizeof(long int) * MAX_SIZE) / (1024
          << " Gbytes\n";

    arrayPtr = new long int[MAX_SIZE];
    if (!arrayPtr) {
        cerr << "Not enough memory" << endl;
        return 1;
    }
}

```

Quité el nowthrow de new, compilé y ejecuté y pasó esto:

```

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2>g++ -o main *.cpp
memoria2.cpp: In function 'int main()':
memoria2.cpp:24:37: error: size of array is too large
    arrayPtr = new long int[MAX_SIZE];
                              ^

```

Lo mismo, muy grande el MAX SIZE.

```

#define MAX_SIZE 100000000

int main() {
    long int* arrayPtr = nullptr;

    cout << "Asking for: "
          << (sizeof(long int) * MAX_SIZE) / (
          << " Gbytes\n";

    arrayPtr = new long int[MAX_SIZE];
    if (!arrayPtr) {
        cerr << "Not enough memory" << endl;
        return 1;
    }
}

```

Le quité un cero.

```

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2>main
Asking for: 0 Gbytes
Got them! Filling the array with long ints...
50000000
Revisa cuanto utiliza en el administrador de procesos de tu sistema operativo
Presiona ENTER para liberar la RAM y terminar

```

No sé qué pasa, desde la vez pasada no me sale nada kjasnkfs. Pero debería salir que no hay memoria, porque finalmente no me deja ejecutarlo por eso. Y con o sin el nothrow sale lo mismo. El nothrow nos ayuda a no lanzar excepciones (evitar usar el try y catch), y nos ayuda a mejorar la seguridad del código (a que se nos vaya un try o catch, vaya).

Caso 2c:

```

#include <iostream>

using namespace std;

int main() {
    int* ptr = nullptr;

    ptr = new(nothrow) int;
    if (ptr == nullptr) {
        cout << "No hay memoria\n";
        return 0;
    }

    *ptr = 5;
    cout << "El valor de ptr es: " << ptr << endl;
    cout << "El valor en ptr es: " << *ptr << endl;

    delete ptr;
    ptr = nullptr;

    return 0;
}

```

Al compilar y ejecutar el código de memoria2c, sale esto. Al inicio del código se declara un apuntador a un entero (int) y se está inicializando con el nullptr, esto para evitar que tenga basura o que haya confusión después. El * indica que es apuntador y sin apuntador es la ubicación de memoria. Con el new(nothrow) se le está asignando memoria (new) y se están evitando lanzar excepciones (nothrow). Si el apuntador (ptr) es nullptr lanza un mensaje de que no hay memoria. Si no lanza error, significa que la

memoria fue asignada correctamente y se le da el valor de 5 a la ubicación de memoria que apunta ptr. Los couts de los valores son:

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2c>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2c>main
El valor de ptr es: 0x12e15b0
El valor en ptr es: 5

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2c>|
```

ptr (la dirección de memoria como tal)

*ptr (el valor que está en esa dirección de memoria)

Si lo ejecuto otra vez, lo que cambia es la dirección de memoria.

```
main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2c>main
El valor de ptr es: 0x12e15b0
El valor en ptr es: 5

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2c>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2c>main
El valor de ptr es: 0x12715b0
El valor en ptr es: 5

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2c>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2c>main
El valor de ptr es: 0x11a15b0
El valor en ptr es: 5
```

Caso 2d:

```
#include <iostream>

using namespace std;

#define MAX_SIZE 10

int main() {
    int* ptr = nullptr;

    ptr = new(nothrow) int[MAX_SIZE];
    if (ptr == nullptr) {
        cout << "No hay memoria\n";
        return 0;
    }

    for (unsigned int i = 0; i < MAX_SIZE; i++)
        ptr[i] = MAX_SIZE - i;

    cout << "El valor de ptr es: " << ptr << endl;
    cout << "El valor en ptr[6] es: " << ptr[6] << endl;
    cout << "El valor en *ptr + 6 es: " << *ptr + 6 << endl;
    cout << "El valor en *(ptr + 6) es: " << *(ptr + 6) << endl;

    delete [] ptr;
    ptr = nullptr;

    return 0;
}
```

Arreglo:

ptr[0] = 10

ptr[1] = 9

ptr[2] = 8

ptr[3] = 7

ptr[4] = 6

ptr[5] = 5

ptr[6] = 4

ptr[7] = 3

ptr[8] = 2

ptr[9] = 1

Este programa nos muestra apuntadores en arreglos. Primero definimos el valor de MAX SIZE a 10 (el tamaño que tendrá el arreglo, (0 a 9)). Inicializamos el apuntador con `int* ptr = nullptr`. Y lo mismo del ejemplo de arriba, inicializarlo y asignarle memoria. Con el ciclo for vamos llenando el arreglo (del 10 [0] al 1 [9]). En los couts tenemos 4. El primero nos dice la dirección de memoria del apuntador, el segundo nos dice EL VALOR del 6to elemento del arreglo (que es 4), el tercero nos dice el primer valor del arreglo + 6 (10 + 6 = 16), y el último imprime `ptr + 6` (o sea `ptr[6]`) que es 4. Si hubiera sido `ptr + 9`, sería `ptr[9]` y se imprimiría el 1. Aquí la demostración.

```

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2d>main
El valor de ptr es: 0x1186cd0
El valor en ptr[6] es: 4
El valor en *ptr + 6 es: 16
El valor en *(ptr + 6) es: 4

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2d>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria2d>main
El valor de ptr es: 0x1146cd0
El valor en ptr[6] es: 4
El valor en *ptr + 6 es: 16
El valor en *(ptr + 6) es: 4

```

Aquí vemos que al ejecutarlo varias veces, obviamente cambia la dirección de memoria, pero los valores de los apuntadores son los mismo.s.

Ejemplo del ptr + 9

```

cout << "El valor en *(ptr + 9) es: " << *(ptr + 9) << endl;

El valor en *ptr + 6 es: 16
El valor en *(ptr + 9) es: 1

```

Caso 3:

```

#include <iostream>

using namespace std;

// Definimos una clase de manera breve en el main como ejemplo
class Persona {
public:
    Persona() { cout << "Persona creada\n"; }
    ~Persona() { cout << "Persona destruida\n"; }
};

int main() {
    Persona* ptr = nullptr;

    ptr = new(nothrow) Persona();
    if (!ptr) {
        cout << "Not enough memory\n";
        return 0;
    }

    cout << "En el main, antes del delete\n";

    delete ptr;
    ptr = nullptr;

    return 0;
}

```

```

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria3>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria3>main
Persona creada
En el main, antes del delete
Persona destruida

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria3>

```

En este código podemos ver que estamos creando una clase, en los atributos públicos tenemos al constructor y al destructor. Estos constructores y destructores tienen un cout que indican si se está llamando al constructor (sale Persona Creada), y al destructor (Persona destruida).

En el main tenemos un apuntador a un objeto de la clase Persona, podemos ver esto gracias al *, esto nos dice que es un apuntador y no un objeto. Este está apuntando a la dirección de memoria del objeto Persona. En la línea de abajo estamos inicializando el apuntador y si falla esta asignación, se manda un mensaje indicando que no hay memoria. Si la memoria fue asignada correctamente, se manda a llamar al constructor que imprime su respectivo mensaje. Luego tenemos el cout de “antes de liberar memoria” y con el delete ptr; ptr = nullptr; estamos liberando la memoria (delete) y es aquí donde el destructor se manda a llamar.

Caso 4:

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

class Persona {
public:
    Persona() { cout << "Persona creada\n"; }
    ~Persona() { cout << "Persona destruida\n"; }
};

Persona* crearPersona() {
    Persona* ptr = nullptr;

    ptr = new(nothrow) Persona();
    if(!ptr) {
        cout << "Not enough memory\n";
    }
    return ptr;
}

int main() {
    Persona* ptr = nullptr;

    ptr = crearPersona();

    if(ptr) {
        delete ptr;
        ptr = nullptr;
    }

    return 0;
}
```

```
C:\Users\Aryam Buendía Zavala>cd "C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria4"

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria4>g++ -o main *.cpp

C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria4>main
Persona creada
Persona destruida
```

Lo ejecuté 3 veces y salió lo mismo, “Persona creada” y “Persona destruida”. Tenemos clase Persona, con su constructor y destructor. Tenemos diferentes variables, por así decirlo, una global y una local. crearPersona de la función es SOLO de la función, y la del main es por aparte. Se inicializa el apuntador con nullptr, se le asigna memoria, y si no fue asignada correctamente, lanza el mensaje de que no hay espacio. En el main el ptr = crearPersona(); manda llamar a la función, aquí se asigna memoria al objeto y le da un apuntador a él. Si el apuntador es nulo, la memoria se libera (delete).

Caso 5:

```
#include <iostream>

using namespace std;

class Persona {
public:
    Persona() { cout << "Persona creada\n"; }
    ~Persona() { cout << "Persona destruida\n"; }

    string nombre;
};

Persona* crearPersona() {
    Persona* ptr = new(nothrow) Persona();

    if (!ptr) {
        cout << "Not enough memory\n";
        return nullptr;
    }

    ptr->nombre = "Juan";

    return ptr;
}

int main() {
    Persona* ptr = nullptr;

    ptr = crearPersona();

    if (ptr) {
        cout << "(main) Nombre: " << ptr->nombre << "\n";

        delete ptr;
        ptr = nullptr;
    }

    cout << "terminando programa\n";
    return 0;
}
```

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria5>main
Persona creada
(main) Nombre: Juan
Persona destruida
terminando programa
```

Primero debemos ver si un objeto está en el stack o el heap. Tenemos lo mismo de los ejemplos anteriores, la clase Persona, constructor y destructor. Después tenemos la función de asignación de memoria, solo que tenemos un `ptr->nombre = "Juan";`. Cuando cambiamos `ptr->nombre = "Juan";` a `ptr.nombre = "Juan";`, nos manda error.

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria5>g++ -o main *.cpp
memoria5.cpp: In function 'Persona* crearPersona()':
memoria5.cpp:30:9: error: request for member 'nombre' in 'ptr',
which is of pointer type 'Persona*' (maybe you meant to use '->'
?)
    ptr.nombre = "Juan";
      ^~~~~~
```

Cuando es un apuntador a un objeto se utiliza `->` para acceder a los miembros del objeto apuntado. El punto es para el objeto.

Caso 6:

```
#include <iostream>

using namespace std;

class Persona {
public:
    Persona() { cout << "Persona creada\n"; }
    ~Persona() { cout << "Persona destruida\n"; }
    string getNombre() { return nombre; }
    void setNombre(string nombre) { this->nombre = nombre; }
private:
    string nombre;
};

int main() {
    Persona* ptr = new(nothrow) Persona();

    if (!ptr) {
        cout << "Not enough memory\n";
        return 0;
    }

    ptr->setNombre("Karen");
    cout << "Nombre: " << ptr->getNombre() << "\n";

    delete ptr;
    ptr = nullptr;

    cout << "terminando programa\n";
    return 0;
}
```

```
C:\Users\Aryam Buendía Zavala\Documents\Uni\Semestre 3\Estructura de Datos\Actividad-memoria-apuntadores-TC1031\memoria6>main
Persona creada
Nombre: Karen
Persona destruida
terminando programa
```

Para mandar a llamar un método a través de un apuntador se usa el ->

Con este operador podemos acceder a los miembros de los datos y métodos. El ptr es un apuntador de tipo Persona para el objeto Persona y así podemos llamar a los setter y getters.

Caso 6b:

```
#include <iostream>
using namespace std;
#define MAX_SIZE 4

class Persona {
public:
    Persona() { cout << "Persona creada\n"; }
    ~Persona() { cout << "Persona " << nombre << " destruida\n"; }
    string getNombre() { return nombre; }
    void setNombre(string nombre) { this->nombre = nombre; }
private:
    string nombre;
};

int main() {
    string nombres[MAX_SIZE] = { "Juan", "Pedro", "Luis", "Karen" };
    Persona* ptr = new(nothrow) Persona[MAX_SIZE];

    if (!ptr) {
        cout << "Not enough memory\n";
        return 0;
    }

    for (unsigned int i = 0; i < MAX_SIZE; i++)
        ptr[i].setNombre(nombres[i]);

    delete [] ptr;
    ptr = nullptr;

    cout << "terminando programa\n";
    return 0;
}
```

```
C:\Users\Aryam Buendía Zava
a de Datos\Actividad-memori
Persona creada
Persona creada
Persona creada
Persona creada
Persona Karen destruida
Persona Luis destruida
Persona Pedro destruida
Persona Juan destruida
terminando programa
```

Este código tiene más objetos, es un arreglo de objetos con apuntador. Aquí asignamos memoria al arreglo. Cada vez que se crea una persona, sale el constructor, y cada vez que sale el destructor sale el nombre (objeto), El MAX SIZE es 4, por lo que el arreglo de objetos tiene 4 objetos.

Caso 7:

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

class Persona {
public:
    Persona() { cout << "Persona creada\n"; }
    ~Persona() { cout << "Persona destruida\n"; }
    string nombre;
};

Persona* crearPersona() {
    Persona* ptr = new(nothrow) Persona();

    if (!ptr) {
        cout << "Not enough memory\n";
        return nullptr;
    }

    ptr->nombre = "Juan";
    return ptr;
}

int main() {
    Persona* ptr = nullptr;

    ptr = crearPersona();

    // Hacemos uso incorrecto del apuntador ya que:
    // 1) No verificamos si existe antes de
    //     llamar delete para regresar la memoria
    // 2) Accedemos a un atributo ya que destruimos el objeto
    delete ptr;
    cout << "(main) Nombre: " << ptr->nombre << "\n";

    cout << "terminando programa\n";
    return 0;
}
```

```
C:\Users\Aryam Buen
a de Datos\Activida
Persona creada
Persona destruida
(main) Nombre:
```

No se lee el mensaje, ni el nombre. Aquí no sé por qué para eso, no sé por qué no se lee el último mensaje.

Caso 8:

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

class Persona {
public:
    Persona() { cout << "Persona creada\n"; }
    ~Persona() { cout << "Persona destruida\n"; }

    string nombre;

    Persona* crearPersona() {
        Persona* ptr = new(nothrow) Persona();

        if (!ptr) {
            cout << "Not enough memory\n";
            return nullptr;
        }

        ptr->nombre = "Juan";

        return ptr;
    }
};

int main() {
    Persona* ptr = nullptr;

    ptr = crearPersona();

    if (!ptr)
        return 0;

    cout << "(main) Nombre: " << ptr->nombre << "\n";
    // Aca la liberamos correctamente...
    delete ptr;
    // Y acá cometemos el error al liberar por segunda vez
    delete ptr;
    cout << "terminando programa\n";

    return 0;
}
```

```
C:\Users\Aryam Buendía Z...
a de Datos\Actividad-men...
Persona creada
(main) Nombre: Juan
Persona destruida
Persona destruida
terminando programa
```

Al correr el código, no hubo problema, a comparación del ejemplo anterior, según yo en el otro como que muestra el constructor y destructor, el delete va antes como de la función, vaya. Porque no se muestra la persona o algo. Pero en este lo único que pasa como “raro” es que el constructor se manda a llamar 2 veces, pero justo es porque hay 2 delete.

Caso 9:

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

class Persona {
public:
    Persona() { cout << "Persona creada\n"; }
    ~Persona() { cout << "Persona destruida\n"; }

    string nombre;

    Persona* crearPersona() {
        Persona* p = new(nothrow) Persona();

        if (!p) {
            cout << "Not enough memory" << endl;
            return nullptr;
        }

        p->nombre = "Juan";

        return p;
    }
};

int main() {
    Persona* p = nullptr;

    p = crearPersona();

    // revisamos si si tenemos un objeto válido
    if (p) {
        cout << "(main) Nombre: " << p->nombre << endl;
        if (p) {
            delete p;
            p = nullptr;
        }
        // si revisamos p, no se liberará de nuevo
        if (p) {
            delete p;
            p = nullptr;
        }
        cout << "terminando programa" << endl;
        return 0;
    }
}
```

```
C:\Users\Aryam Buendía Z...
a de Datos\Actividad-m...
Persona creada
(main) Nombre: Juan
Persona destruida
terminando programa
```

En este código tenemos los deletes en condicionales. Estamos asegurándonos que si ya liberamos la memoria, no la volvamos a liberar.

Caso 10:

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

class Persona {
public:
    Persona() { cout << "Persona creada" << endl; }
    ~Persona() { cout << "Persona " << nombre << " destruida" << endl; }

    string nombre;

    Persona* crearPersona(string nombre) {
        Persona* p = new(nothrow) Persona();

        if (!p) {
            cout << "Not enough memory" << endl;
            return nullptr;
        }

        p->nombre = nombre;

        return p;
    }
};

int main() {
    Persona* p = nullptr;

    p = crearPersona("Juan"); // primera asignación
    p = crearPersona("Pedro"); // segunda asignación

    if (p) {
        cout << "(main) Nombre: " << p->nombre << endl;
    }
    if (p) {
        delete p;
        p = nullptr;
    }
    cout << "terminando programa" << endl;
    // Nunca se liberó la primera asignación, memoryleak
    return 0;
}
```

```
C:\Users\Aryam Buendía Za...
0>main
Persona creada
Persona creada
(main) Nombre: Pedro
Persona Pedro destruida
terminando programa
```

No se llamaron los 2 destructores, según yo solo se muestra LA SEGUNDA asignación de Pedro y no la de Juan. Como p se asigna a Juan y esa misma se reasigna a Pedro, es

por eso es que solo se muestra Pedro al final. Sí salen los 2 constructores, pero solo 1 destructor.

Caso 11:

[illegible]

Aquí sale el mensaje de persona creada y destruida, pero después se me cuatropéó todo. Aquí todavía no entiendo por qué, justo pasó lo mismo la vez pasada pero no entiendo bien qué pasa.