

# Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

Como ya aprendimos el paso de parámetro por referencia, podemos modificar **el tipo Conjunto (en forma modular con .h y .cpp)** que habíamos hecho con solo paso por valor, lo cual era ineficiente y teníamos que hacer cosas como:

```
conj = insertar(conj, i);
```

Ahora podemos insertar un elemento a un conjunto **utilizando paso de parámetros por referencia.**

Veamos....

## El archivo conjunto\_ref.h:

```
#include <vector>

using namespace std;

//typedef vector<int> Conjunto; es igual a using
using Conjunto = vector<int> ;

void insertar(Conjunto & , int); // note que el parámetro es
                                // por referencia

void imprimir(const Conjunto & ) ; // const pues no se modifica

bool pertenece(const Conjunto & , int );
```

## El archivo conjunto\_ref.cpp:

```
#include <iostream>
#include "conjunto_ref.h"

void insertar(Conjunto &v, int i){
    // Verificar que no esté para
    // poderlo insertar
    for (int j=0; j<v.size(); j++){
        if (i==v[j]) return;
    }
    v.push_back(i);
}
```

```
bool pertenece(const Conjunto &v, int i){
    // Devuelve true sii i esta en v
    for (int j=0; j<v.size(); j++){
        if (i==v[j]) return true;
    }
    return false;
}

void imprimir(const Conjunto &conj){
    for (int j=0; j<conj.size(); j++)
        std::cout << conj[j] << " ";
}
```

## El archivo de prueba prueba.cpp: Note que **NO** sabemos como se ha implementado el tipo **Conjunto**

```
#include <iostream>
#include "conjunto_ref.h"
int main(){
    Conjunto conj; //crea un conjunto vacío
    insertar(conj,0); // se modifica la variable conj
    insertar(conj,1);
    insertar(conj,2);
    imprimir(conj);
    int i=3;
    std::cout << "¿Está " << i << " en el conjunto?: "
        << std::boolalpha << pertenece(conj,i);
}
```

## Funciones de orden superior: Podemos pasar una función como un parámetro de la función, pasando un apuntador a la función.

- Ejemplo:

```
#include <iostream>
int sumar(int x, int y) {
    return x + y;
}
int multiplicar(int x, int y) {
    return x * y;
}
int evaluar(int (*f)(int, int), int x, int y) {
    return f(x, y);
}
```

```
int main() {
    std::cout << sumar(2, 3) << '\n';
    std::cout << multiplicar(2, 3) << '\n';
    std::cout << evaluate(&sumar, 2, 3) << '\n';
    std::cout << evaluate(&multiplicar, 2, 3) << '\n';
}
```

**Ejercicio para la casa:**

**Hacer una función que ordene un vector de enteros. Se le pasa la función que permite ordenar (ejemplo: ascendentemente, descendientemente)**

**¿Saben ordenar un vector?**

**Hablemos un poco de arreglos (que usaré poco):**

**Los arreglos son como los vectores y es un tipo estructurado primitivo de C.**

- Podemos declarar un arreglo de la siguiente forma:

**int list[25];**

- **Se DEBE dar el tamaño en la declaración mediante un entero (25) o una constante.**  
No se puede modificar el tamaño durante todo el programa.
- La declaración **int arreglo[0];** es ilegal pues un arreglo **NO puede tener tamaño 0**
- Para determinar el número de elementos de un arreglo:

**sizeof(list) / sizeof(list[0]).**



## Ejemplos:

```
int x;
```

```
std::cin >> x;
```

```
int list_1[x];
```

```
std::vector<int> list_2(x);
```

**ILEGAL.** El tamaño  
debe ser constante

Legal para un vector

```
double collection[] = { 1.0, 3.5, 0.5, 7.2 };
```

ó 

```
double collection[] { 1.0, 3.5, 0.5, 7.2 };
```

**Legal:** crea un arreglo de 4 elementos tipo double.  
Su tamaño será 4 durante todo el programa

## Podemos pasar un arreglo como parámetro, pero en realidad se pasa “por referencia”

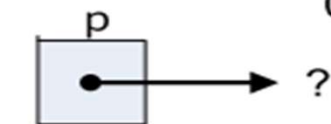
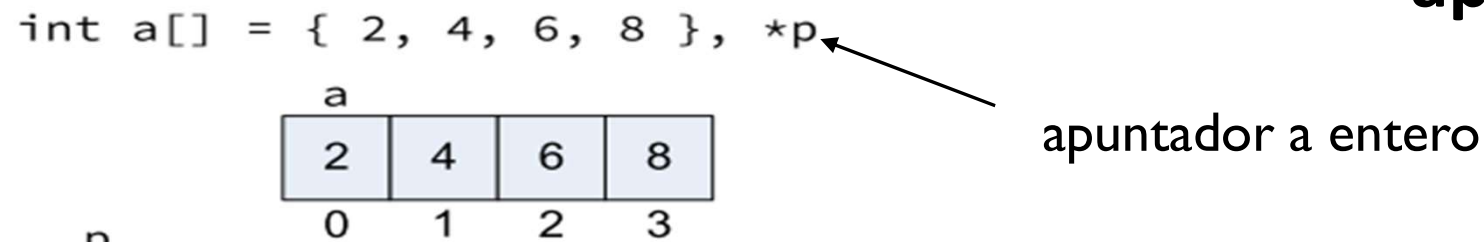
```
#include <iostream>
void imprimir(const int b[], int n)
{ // todo el arreglo es constante
    for (int i = 0; i < n; i++)
        std::cout << b[i] << " ";
    std::cout << '\n';
}
void borrar(int a[], int n) {
    for (int i = 0; i < n; i++)
        a[i] = 0;
}
```

```
int main() {
    int list [ ] = { 2, 4, 6, 8 };
    imprimir(list, 4);
    borrar(list, 4);
    imprimir(list, 4);
}
```

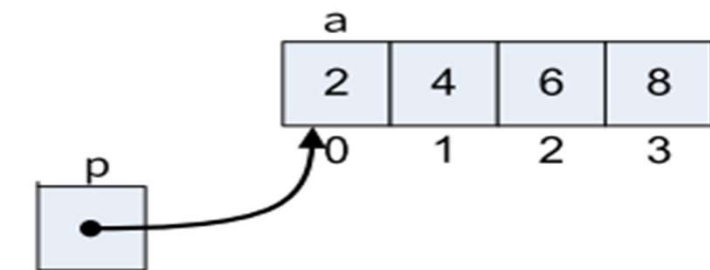
**El nombre de un arreglo se comporta como un apuntador al primer elemento del arreglo.**

Cuando **list** es pasado a las funciones **imprimir( )** y **borrar( )**, en realidad **se pasa la dirección del primer elemento del arreglo, SE DEBE PASAR EL TAMAÑO**

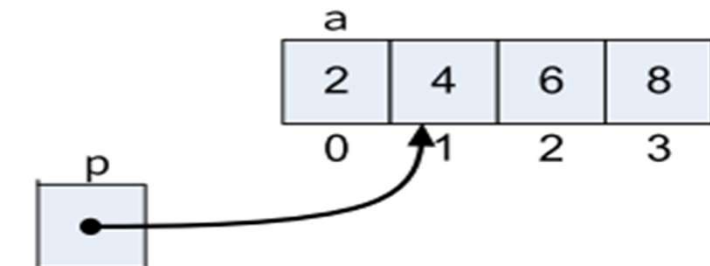
**Para ilustrar que el nombre de un arreglo es en realidad un  
apuntador al primero.**



`p = a;`



`p++;`



Incluso **p[2]** es **válido** e indica el valor (6) del elemento en la posición 2 del arreglo original **a**

Si `p = a` entonces `p++` apunta al elemento en la posición 1 de **a**. Es decir **\*(p++)** es igual a 4 . Incluso podemos usar **p[1]**

**No podemos copiar un arreglo en otro:**

```
int a[10];  
int b[10];
```

```
b = a; // ERROR
```

**Con vectores podemos:**

```
vector<int> a(10, 88);  
vector<int> b;  
b = a; // crea una copia de a y la coloca en b
```

## Arreglos dinámicos:

Podemos crear **arreglos dinámicos**, es decir que podemos definir su tamaño en tiempo de ejecución (y no de compilación):

```
int x = 3;
```

```
int *p = new int[ x+3 ]; // note que el tamaño se determina en  
                        // tiempo de ejecución y no de compilación  
                        // aunque permanecerá con ese tamaño el resto  
                        // de la ejecución del programa
```

**p** es un apuntador a un arreglo de 6 elementos.

Para liberar el espacio de este arreglo usamos: **delete [ ] p;**

## **Ejemplo de uso de arreglos dinámicos:**

**Un programa que guarda en un arreglo N números introducidos por teclado y hace el promedio de ellos.**

**Se pide inicialmente la cantidad N de números a promediar.**

```
#include <iostream>
```

```
int main() {
```

```
    double suma = 0.0;
```

```
    double *numeros; // numeros es un apuntador, no un arreglo
```

```
    int tam; // Cantidad de números a leer
```

```
    // Leer la cantidad de números a procesar
```

```
    std::cout << "Coloque la cantidad de números a procesar: ";
```

```
    std::cin >> tam;
```

```
    if (tam > 0) { // verifica que sea positivo
```

```
std::cout << "Escriba " << tam << " numeros: ";
// Reservar la cantidad exacta requerida
numeros = new double[tam]; // arreglo dinámico, números es apuntador
// El usuario entra los números
for (int i = 0; i < tam; i++) {
    std::cin >> numeros[i];
    suma += numeros[i];
}
std::cout << "El promedio de ";
for (int i = 0; i < tam - 1; i++)
    std::cout << numeros[i] << ", ";
// sin coma al final, el último número escrito
std::cout << numeros[tam - 1] << " es " << suma/tam << '\n';
delete [] numeros; // Liberar el espacio del arreglo numeros
```



**Hacer lo mismo que el programa anterior pero sin saber cuántos números vamos a colocar, solo que no son negativos.**

**Se empieza con un arreglo dinámico de un cierto tamaño y si el siguiente número leído NO cabe en el arreglo, se “agrandar” el arreglo....**

**El arreglo “puede entonces crecer” ....**

```
#include <iostream>
// Calcular promedio de varios números no negativos introducidos por teclado
int main() {
    double suma = 0.0, // Suma de los elementos a teclear
           *numeros,    // arreglo dinámico de números reales
           numero;      // el numero a leer
    // Tamaño inicial del arreglo (capacidad)
    // cantidad a agrandar (TROZO) y cantidad de números en arreglo (num_elem)
    const int TROZO = 3;
    int num_elem = 0, // cantidad actual de números en arreglo
        capacidad = TROZO; // tamaño inicial del arreglo

    numeros = new double[capacidad]; // asignar un primer tamaño al arreglo

    std::cout << "Coloque cualquier cantidad de numeros "
               << "(un número negativo finaliza la entrada): ";
    std::cin >> numero;
```

```
while (numero >= 0) { // mientras no sea negativo
    if (num_elem == capacidad) { // si no hay espacio en el arreglo numeros
        capacidad += TROZO; // aumentar la capacidad
        double *temp = new double[capacidad]; // asignar espacio
        for (int i = 0; i < num_elem; i++)
            temp[i] = numeros[i]; // copiar al nuevo arreglo
        delete [] numeros; // liberar el viejo arreglo
        numeros = temp; // el arreglo numeros ahora
                        // está donde apunta temp
        std::cout << "Expandiendo " << TROZO << '\n';
    }
    numeros[num_elem] = numero; // agregar numero en la ultima posicion
    num_elem++; // actualizar el numero de elementos actual
    suma += numero; // sumar el elemento a suma
    std::cin >> numero; // obtener el proximo elemento
}
```

```

if (num_elem > 0) {      // si se leyó al menos un numero
    std::cout << "El promedio de ";

    for (int i = 0; i < num_elem - 1; i++)
        std::cout << numeros[i] << ", ";

    // el ultimo elemento sin coma al final
    std::cout << numeros[num_elem - 1] << " es "
               << suma/num_elem << '\n';
}
else
    std::cout << "no colocó un número no negativo\n";
delete [] numeros; // liberar números
}

```

**El tipo vector se construye con arreglos primitivos y utiliza la técnica anterior para agrandar su tamaño.**

**Los arreglos en C++ permiten for basado en rango:**

**for (auto elem: arreglo) .....**

## Sobre apuntadores y manejo de memoria:

Algunos componentes de la memoria que asigna el Sistema operativo para ejecutar un programa son:

- **Código:** almacena el Código ejecutable del programa.
- **Datos:** almacena variables globales y variables locales estáticas.

Continúa....

## Otros Datos:

**Heap (montículo):** almacena la memoria dinámica. Es decir, la memoria adicional que se requiere al crear nuevos objetos en la ejecución del programa con el operador **new** y su correspondiente operador **delete** para liberar memoria utilizada en el heap.

**Stack (pila):** almacena las variables locales y parámetros de las funciones. Cada vez que se llama a una función se crean **las variables locales y parámetros y desaparecen del stack al terminar la función** su ejecución.

## Volvamos sobre new y delete:

Supongamos tenemos una función **calculo ( )** que crea un arreglo dinámico:

```
int * calculo(int n) {
    ....
    int *p = new int[n]; // // en el Heap el operador new crea un arreglo de tamaño n
    ....
}
```

Después que se ejecuta **calculo( )** se elimina **p** por ser local y el **espacio reservado en el Heap** queda reservado pero **sin poder acceder a él. Es una memoria perdida**

**Antes de terminar calculo( ) se debe liberar esa memoria o devolver su apuntador al programa que la llamó y este se encargará de liberarla con el operador: delete [ ] p**

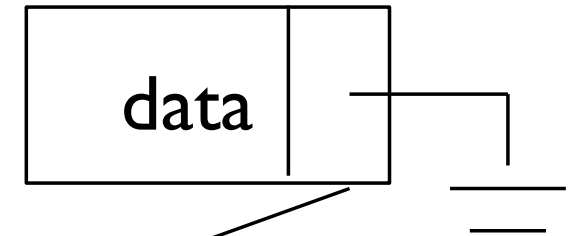


## Creemos estructuras enlazadas (listas enlazadas) creando nuevos objetos:

```
#include <iostream>
```

```
using namespace std;
```

```
struct Nodo {  
    int data; // información guardada en nodo  
    Nodo *proximo; // apuntador  
                // a otro nodo  
};
```



**Continúa.....**

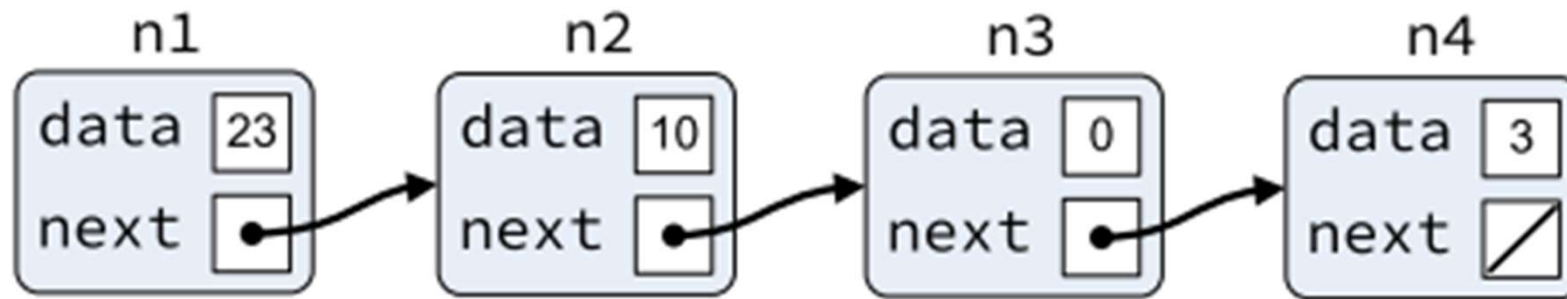
## Creemos estructuras enlazadas (listas enlazadas) creando nuevos objetos:

```
int main() {
    // Variables (u objetos) tipo Nodo
    Nodo n4 = {3, nullptr}, // declarar un nodo
    n3 = {0, &n4}, // enlace al nodo anterior
    n2 = {10, &n3}, // enlace al nodo anterior
    n1 = {23, &n2}; // enlace al nodo anterior

    // Imprimir la lista enlazada creada
    for (Nodo *cursor = &n1; cursor != nullptr;
         cursor = cursor -> proximo)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}
```

Es lo mismo que  
(\*cursor).proximo

Esta es una LISTA ENLAZADA de elementos tipo Nodo

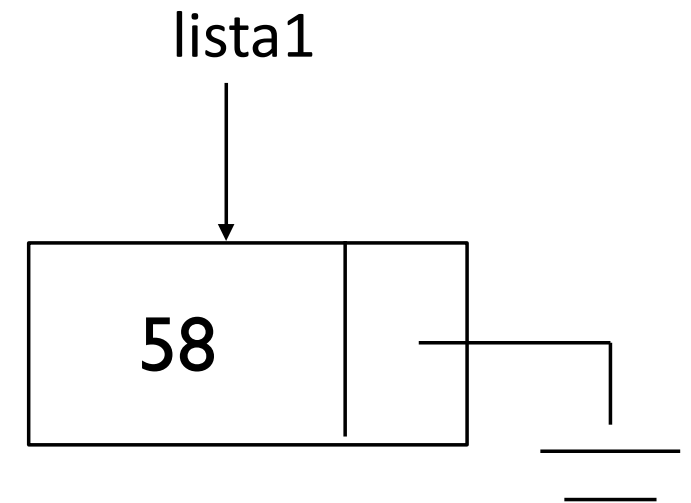


## Podemos crear nuevos objetos tipo Nodo usando new:

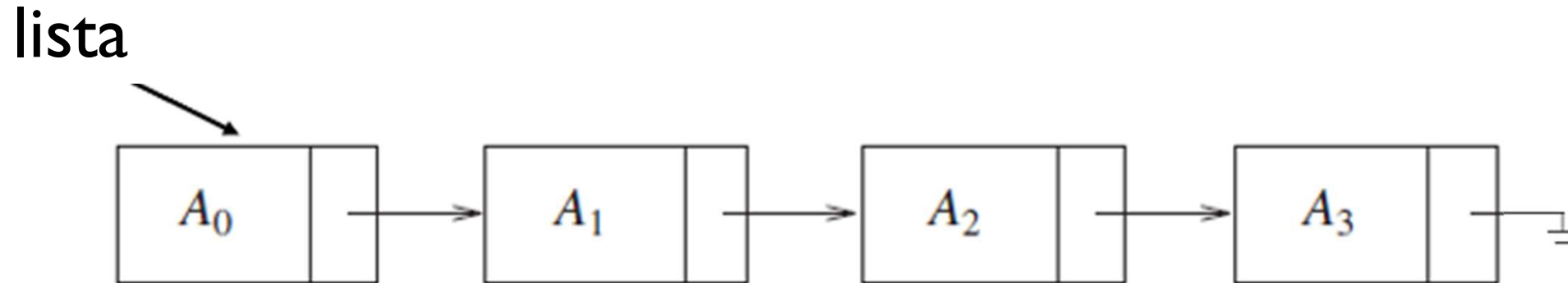
```
Nodo *lista = new Nodo{58,nullptr};  
std::cout << lista->data << '\n';
```

// y también...

```
Nodo *lista1 = new Nodo;  
lista1 -> data = 58;  
lista1 -> proximo = nullptr;  
std::cout << lista1->data << '\n';
```

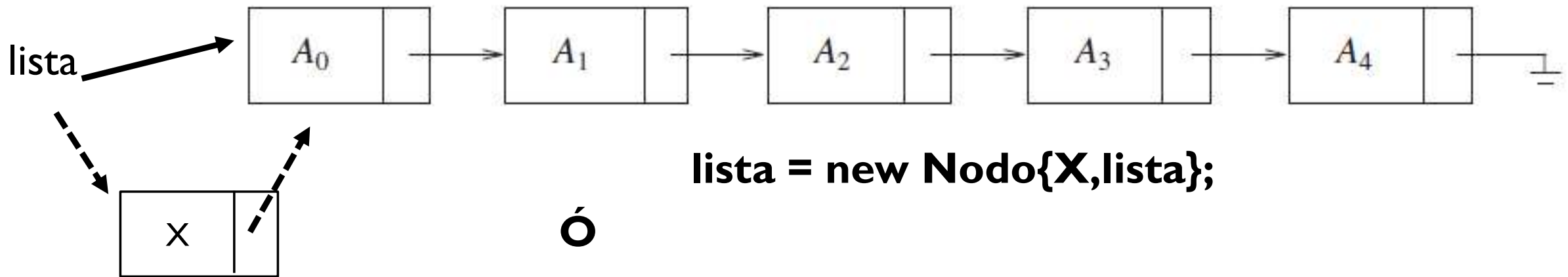


Hagamos un programa que cree una lista enlazada con 4 elementos, creando 4 nodos nuevos (el próximo del último nodo contendrá nullptr):



En una lista enlazada **al primer elemento se le llama cabeza** de la lista y **al último se le llama cola** de la lista

¿Cómo insertar un nodo de primero en la lista que contenga al elemento X?:



**lista = new Nodo{X,lista};**

Ó

**Nodo \*temp = new Nodo;**

**Nodo.data = X;**

**Nodo.proximo = lista;**

**lista = temp;**

**Note que la lista puede estar vacía  
(lista = nullptr) y funciona**

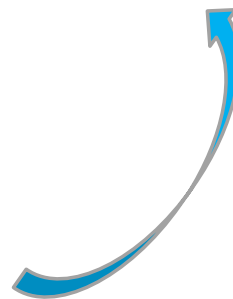
## Crear una lista enlazada con 4 elementos que contienen 1 2 3 4...

```
struct Nodo {
    int data; // informacion del nodo
    Nodo *proximo; // enlace a otro
                // nodo
};
//crear lista de 4 elementos
// insertando por el primero
int n = 4;
Nodo *lista = nullptr; //lista vacía
// insertar n nodos a la lista
for (int i=0 ; i< n ; ++i)
    // lo inserta de primero
    lista = new Nodo{i,lista};
```

// imprimir **lista**

```
for (Nodo *cursor = lista; cursor != nullptr;
     cursor = cursor->proximo)
    std::cout << cursor->data << ' ';
std::cout << '\n';
```

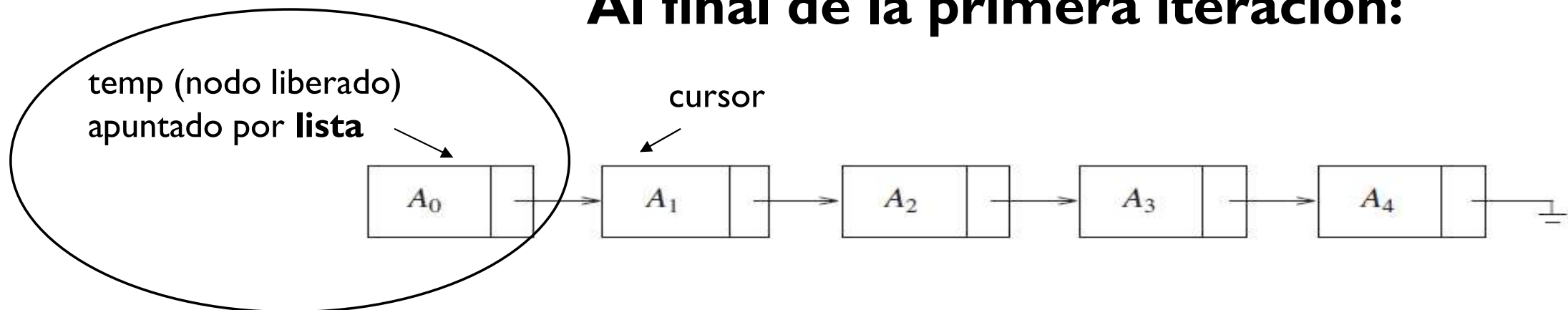
**delete lista; // BORRA SOLO EL PRIMERO**  
**//Y LOS OTROS 3 QUEDAN**  
**// PERDIDOS!!! NO SIRVE**



## Correcta liberación de espacio (borrar) de los nodos de la lista anterior:

```
for (Nodo *cursor = lista, *temp; // note el * delante de temp
     cursor != nullptr;    ) {
    temp = cursor;
    cursor = cursor->proximo;
    delete temp;
}
lista = nullptr;
```

### Al final de la primera iteración:





**Ilustremos (con dibujos) juntos como se insertarían paso a paso los elementos 56, 45 partiendo de una lista vacía. Y luego como se liberaría el espacio.**

```
for (Nodo *cursor = lista, *temp; // note el * delante de temp
    cursor != nullptr;    ) {
    temp = cursor;
    cursor = cursor->proximo;
    delete temp;
}
lista = nullptr;
```

```
for (Nodo *cursor = lista, *temp; // note el * delante de temp  
    cursor != nullptr;    ) {  
    temp = cursor;  
    cursor = cursor->proximo;  
    delete temp;  
}  
lista = nullptr;
```

**Podemos crear dos funciones, una para insertar un elemento en una lista simple enlazada y otra para imprimir sus elementos:**

Por referencia pues  
modificamos la lista

```
void insertar(Nodo* &lista, int n){
    lista = new Nodo{n, lista};
}
```

```
void imprimir(Nodo* const& lista){
    // no permite const Nodo*
    // pues lo asignamos a "cursor"
    // y podríamos modificar la
    // constante a través de cursor
    for (Nodo *cursor = lista; cursor !=
        nullptr; cursor = cursor->proximo)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}
```

**Y una función “vaciar” que elimina todos los nodos de una lista:**

Por referencia pues  
modificamos la lista

```
void vaciar_lista(Nodo* &lista){
    for (Nodo *cursor = lista, *temp;
         cursor != nullptr; ) {
        temp = cursor;
        cursor = cursor->proximo;
        delete temp;
    }
    lista = nullptr;
}
```

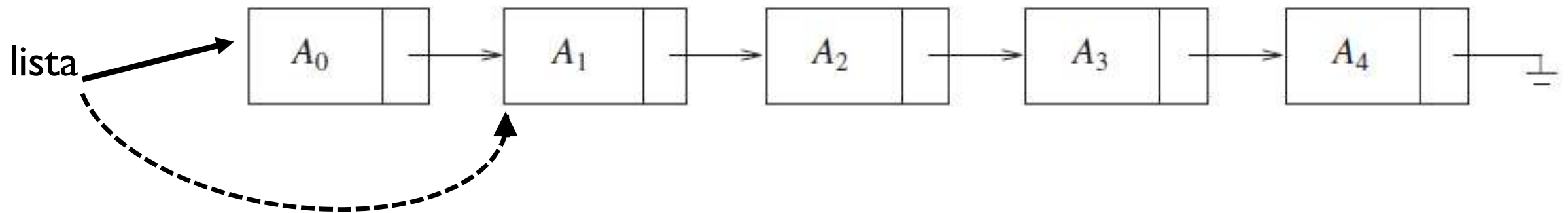
De esta forma nuestro programa de crear una lista con cuatro elementos quedaría (modularización...):

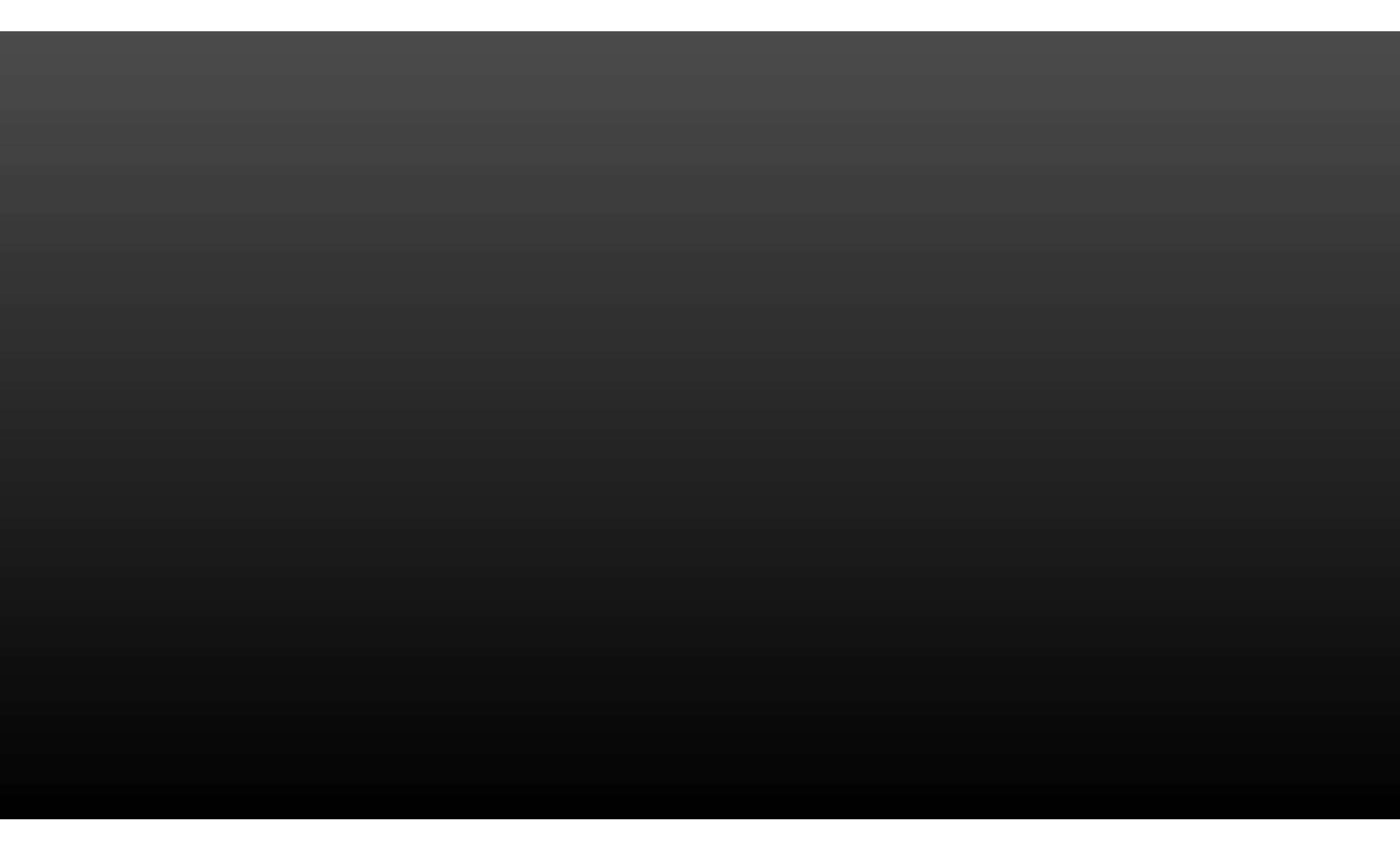
```
int main(){  
    int n = 4;  
    Nodo* lista = nullptr; //lista vacía  
    // insertar n nodos a la lista  
    for (int i=0 ; i< n ; ++i)  
        insertar(lista,i);  
    // imprimir lista  
    imprimir(lista);  
    vaciar(lista);  
}
```

**Hagamos juntos ....**

**Hacer una función para eliminar el primer nodo de una lista enlazada. Pasamos como parámetro el apuntador al primero. La lista puede estar vacía.**

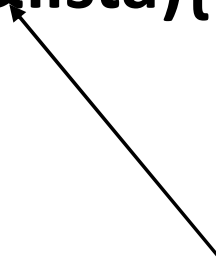
**¿Cómo eliminar el primer nodo de una lista enlazada?:**





```
void eliminar_primer(Nodo* &lista){  
    if (lista!=nullptr){  
        Nodo* temp = lista;  
        lista = lista->proximo;  
        delete temp;  
    }  
}
```

¿Por qué?





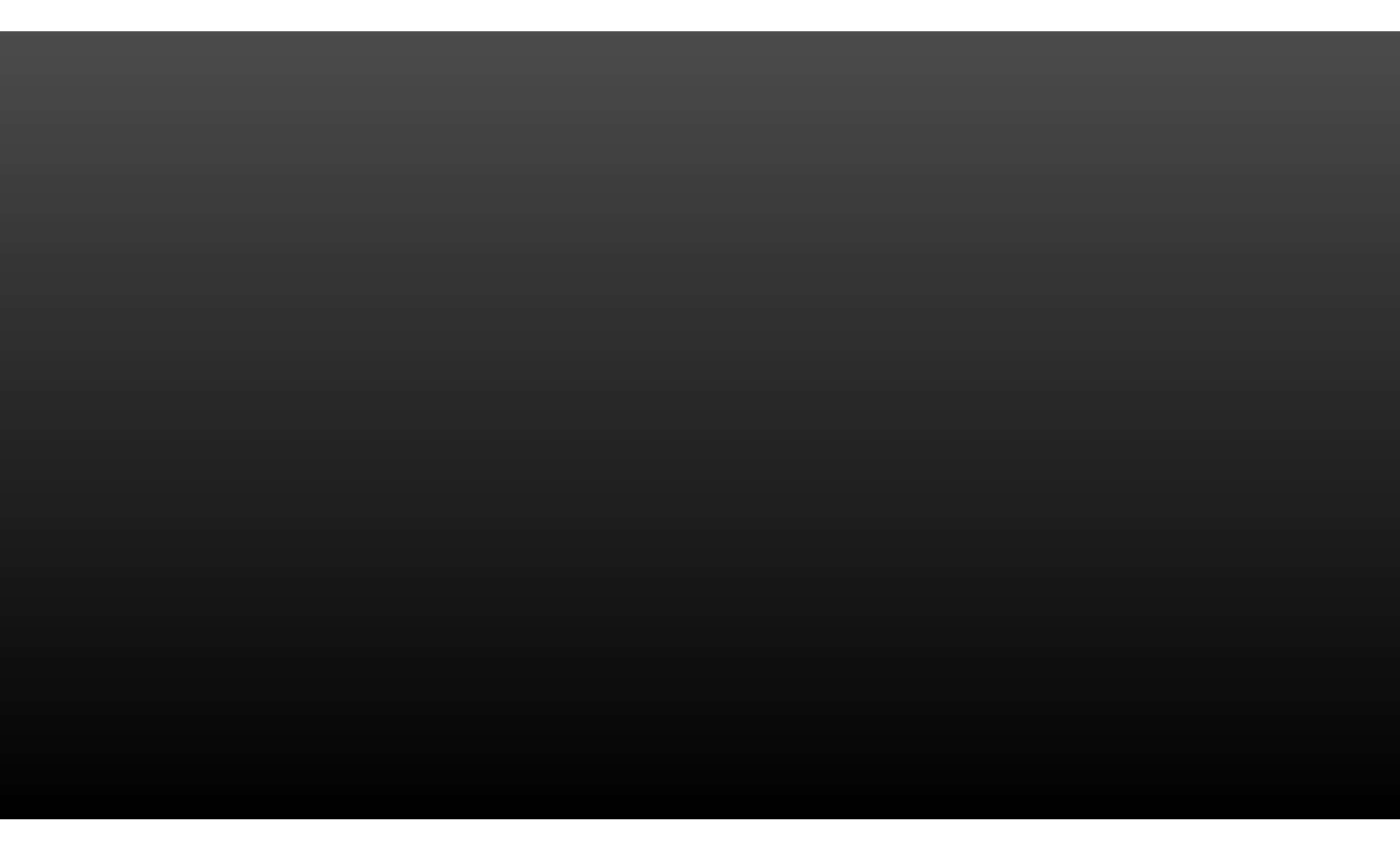
**Y Podemos incorporar esta función a nuestro programa de prueba:**

```
int main(){
    int n = 4;
    Nodo* lista = nullptr;    //lista vacía
    // insertar n nodos a la lista
    for (int i=0 ; i< n ; ++i)
        insertar(lista,i);
    // imprimir lista
    eliminar_primero(lista);
    imprimir(lista);
    vaciar(lista);
    if (lista==0) // pudimos preguntar por nullptr
        std::cout << "La lista está vacía \n";
}
```

¿ Como sería la llamada con apuntadores en lugar de por referencia?

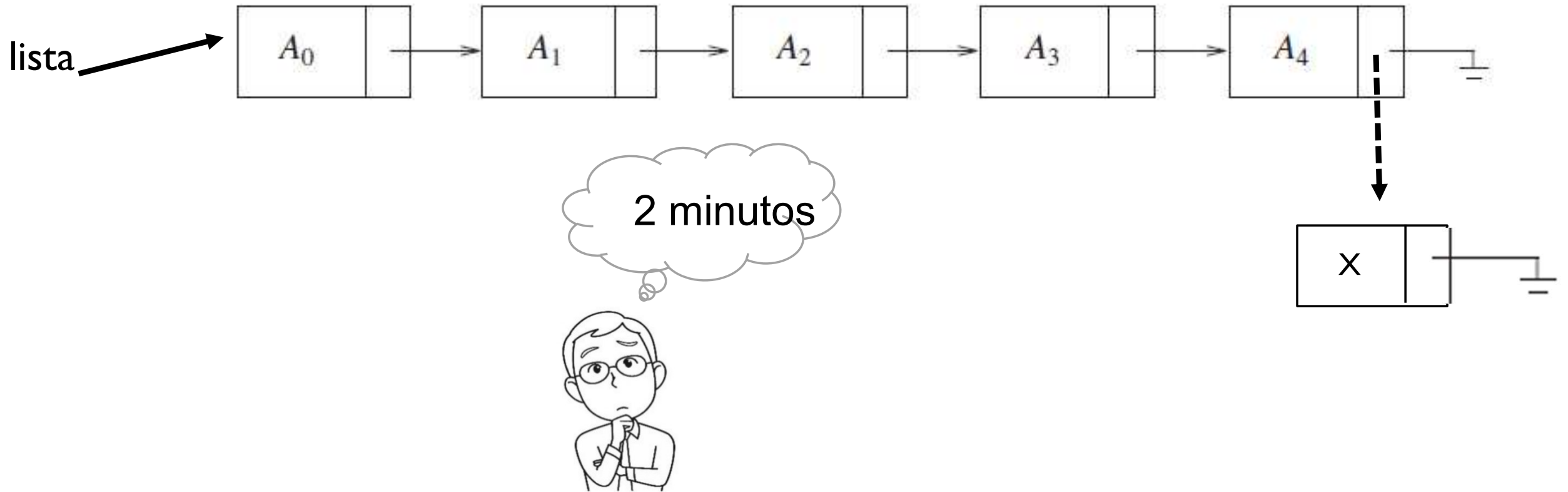
```
void eliminar_primero(Nodo * &lista){  
    if (lista!=nullptr){  
        Nodo* temp = lista;  
        lista = lista->proximo;  
        delete temp;  
    }  
}
```

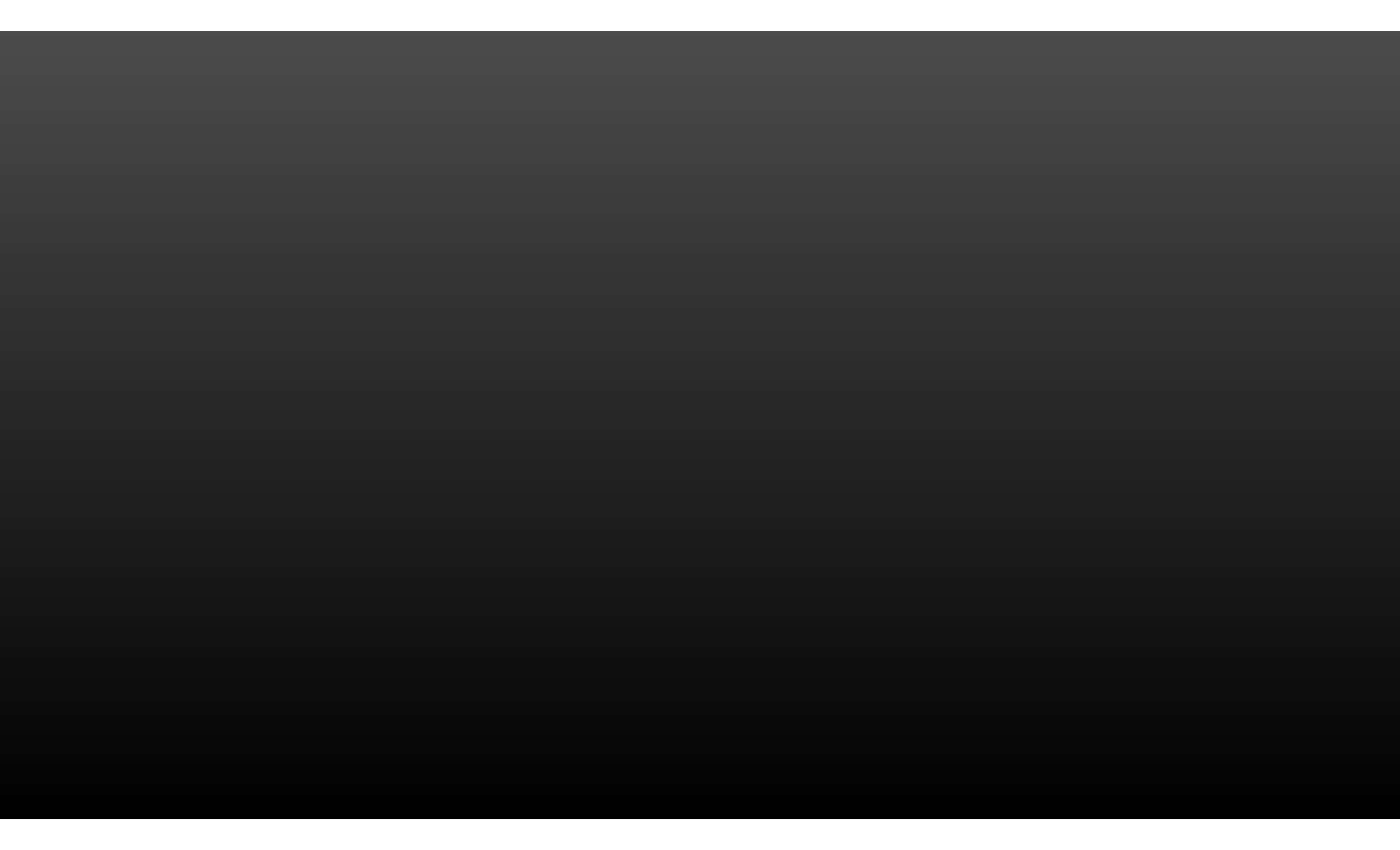




```
void eliminar_primer(Nodo* *lista){  
    if ((*lista)!=nullptr){  
        Nodo* temp = (*lista);  
        (*lista) = (*lista)->proximo;  
        delete temp;  
    }  
}  
    // más engorroso..... Y en la llamada habría que  
    // colocar eliminar_primer(&lista)
```

**Creemos ahora una función “insertar cola” que inserte un elemento X al final de la lista (se debe crear un nodo cuya información es X)**





```
void insertar_cola(Nodo* &lista, int n){
    if (lista == nullptr)
        lista = new Nodo{n,nullptr} ;
    else {    // lo inserta de ultimo
        Nodo *cursor = lista;
        while (cursor->proximo != nullptr)
            cursor = cursor->proximo;
        cursor->proximo = new Nodo{n,nullptr} ;
    }
}
```

Ahora veamos como modularizamos para utilizar una posible implementación del tipo **Lista\_enlazada** (con numero de elementos)  
Primero el archivo header Lista\_Enlazada.h

```
#include <iostream>
struct Nodo {
    int data; // información del nodo
    Nodo *proximo; // enlace al
                  // próximo nodo
};
struct Lista_enlazada {
    Nodo *apun_cabeza;
    int num_elem; // mantenemos el
                  // número de elementos
} ;
```

```
//necesito & porque modifiko lista
Lista_enlazada crear_vacia( );
int num_elem(const Lista_enlazada &);
void vaciar_lista(Lista_enlazada & );
void insertar_primerio(Lista_enlazada &, int );
void eliminar_primerio(Lista_enlazada &);
void imprimir(const Lista_enlazada &);
```



Hagamos juntos:

```
int num_elem(const Lista_enlazada &lista);
```

```
void vaciar_lista(Lista_enlazada &lista);
```

Que estarán en Lista\_enlazada.cpp

```
#include <iostream>
struct Nodo {
    int data; // informacion del nodo
    Nodo *proximo; // enlace a otro
                // próximo nodo
};
struct Lista_enlazada {
    Nodo *apun_cabeza;
    int num_elem; // mantenemos el
                  // número de elementos
} ;
```

Solución:

```
#include <iostream>
#include "Lista_enlazada.h"

Lista_enlazada crear_vacia( ){
    return {nullptr,0};
}

int num_elem(const Lista_enlazada
&lista){
    return lista.num_elem;
}
```

```
void vaciar_lista(Lista_enlazada &lista){
    //necesito & porque modifiko lista
    for (Nodo *cursor = lista.apun_cabeza,
        *temp; cursor != nullptr;      ) {
        temp = cursor;
        cursor = cursor->proximo;
        delete temp;
    }
    lista.apun_cabeza = nullptr;
    lista.num_elem = 0;
}
```

## Mas funciones en el archivo Lista\_Enlazada.cpp

```
void insertar_primerro (Lista_enlazada
                        &lista, int n){
    lista.apun_cabeza =
        new Nodo{n, lista.apun_cabeza};

    lista.num_elem++;
}
```

```
void eliminar_primerro(Lista_enlazada
                        &lista){
    if (lista.apun_cabeza!=nullptr){
        Nodo* temp = lista.apun_cabeza;
        lista.apun_cabeza =
            (lista.apun_cabeza)->proximo;
        delete temp;

        lista.num_elem--;
    }
}
```

En el archivo Lista\_enlazada.cpp

```
void imprimir(const Lista_enlazada &lista){  
    for (Nodo *cursor = lista.apun_cabeza; cursor != nullptr;  
        cursor = cursor->proximo)  
        std::cout << cursor->data << ' ' ;  
    std::cout << '\n';  
}
```

Programa de prueba, prueba.cpp (NOTE QUE NO NOS ENTERAMOS COMO SE IMPLEMENTÓ Lista\_enlazada, ha podido ser con un vector)

```
#include <iostream>
#include "Lista_enlazada.h"
int main(){
    Lista_enlazada lista = crear_vacia( ); //lista vacía
    // insertar n nodos a la lista
    int n = 4; // crear cuatro elementos
    for (int i=0 ; i< n ; ++i)
        insertar_primeros(lista,i);
    eliminar_primeros(lista);
    // imprimir lista
    imprimir(lista);
    vaciar_lista(lista);
}
```

- **Ejercicio: hacer función para eliminar la cola de la lista de una lista tipo Lista\_enlazada**

## PREGUNTAS???

