UNIDAD 3

PROGRAMACIÓN IMPERATIVA

Contenido

Programación Imperativa

Unidad 3

1. Punteros

Un puntero es un tipo especial de variable que contiene la dirección de memoria de otra variable. Los punteros permiten acceder directamente a la memoria, lo que puede ser muy útil para gestionar recursos dinámicos y realizar operaciones a bajo nivel.

En C++, el uso de punteros es fundamental para trabajar con estructuras de datos dinámicas y optimizar ciertas operaciones. Finalmente, en esa dirección de memoria (almacenada en el puntero) puede haber cualquier tipo de variable: char, int, float, arreglo, struct, otro puntero, etc.

Declaración de punteros

Para declarar un puntero, se debe especificar el tipo de datos al que el puntero hará referencia seguido del operador *****. Por ejemplo, del siguiente modo:

```
<tipo> * <identificador>;
```

Veamos el siguiente código:

```
int * p_entero;
char * p_caracter;

struct Punto {
    float x, y;
};

Punto * p_punto;
```

En el código anterior se declaran dos variables de tipo punteros, el primero p_entero contiene una dirección de memoria que sólo puede almacenar valores tipo int. El segundo puntero, p_caracter, contiene una dirección de memoria que almacena caracteres. Luego, p_punto será otro puntero.

Entonces, podríamos decir que existen tantos tipos diferentes de punteros como tipos de datos puedan ser referenciados mediante punteros.

Si tenemos esto en cuenta, los punteros a tipos de datos distintos tendrán tipos diferentes. Por ejemplo, no podemos asignarle un puntero a char a un puntero a int y viceversa.

En la siguiente imagen el puntero palabra podría tener, por ejemplo, la dirección 3. En ese caso, la posición de memoria apuntada por palabra tendría el valor *"si"* (y palabra sería un puntero a string).

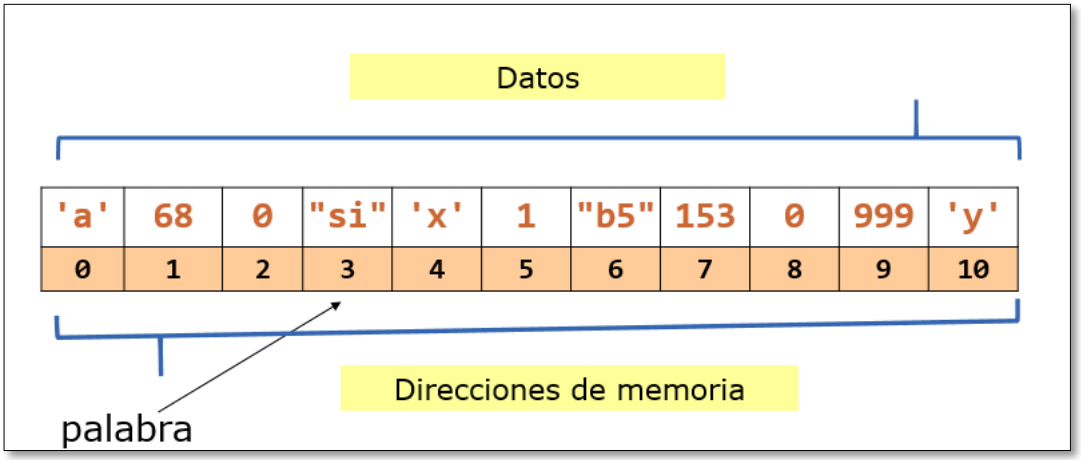


Ilustración 1: representación gráfica de una variable de tipo puntero que apunta a una dirección de memoria que contiene un string.

Al **declarar** un puntero no se le asigna automáticamente una **dirección de memoria válida**, por lo que normalmente es necesario **inicializarlo**.

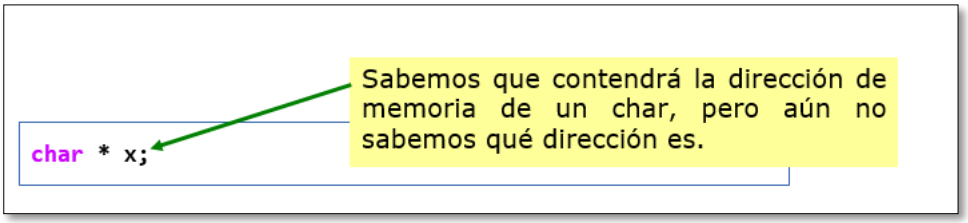


Ilustración 2: declaración de una variable de tipo puntero que apunta a una dirección de memoria que contiene un char.

1.1. Uso de punteros

Una vez declarado, el puntero puede ser asignado a la dirección de memoria de una variable utilizando el operador **&**, que devuelve la dirección de la variable. Posteriormente, puedes acceder al valor almacenado en esa dirección mediante el operador de **desreferenciación** *.

Ejemplo:

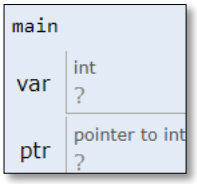
```
int var = 10; // se declara e inicializa una variable de tipo entero
int* ptr = &var;  // se declara e inicializa la variable ptr que almacena la dirección de la variable var
```

Ahora:

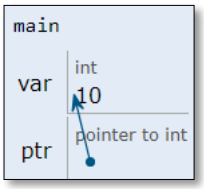
- **&var** obtiene la dirección de la variable var y la almacena en ptr.
- Mientras que ***ptr** accede al valor contenido en la dirección de memoria que ptr apunta, es decir, el valor de var.
- Analicemos el siguiente código:

```
#include <iostream>
int main() {

  int var = 10;
int* ptr = &var;  // ptr almacena la dirección de var

// Acceder al valor de var mediante ptr
std::cout << "Valor de var: " << *ptr << std::endl;
std::cout << "Qué contiene la variabel puntero? " << ptr << std::endl;

}
```

Ilustración 3



Memoria



Memoria

En la *Ilustración 4*, se observa cómo quedan almacenadas en la memoria las variables “var” y “ptr”. Luego, con cada “cout” el programa imprimirá lo que se observa en la *Ilustración 5*.

```
Print output (drag lower right corner to resize)
Valor de var: 10
Qué contiene la variabel puntero? 0xfff000bd4
```

Ilustración 4

Observemos que 0xfff000bd4 es una dirección de memoria que será la que contenga al valor entero 10.

1.2. Contenido basura

Como cualquier otra variable, mientras no se le asigne un valor, un puntero contiene "**datos basura**".

Si queremos decir "este puntero no apunta a nada", no es lo mismo que dejar que apunte a cualquier cosa. Para decir "no apunta a nada" usamos el valor **nullptr**.

Entonces, cuando se declara un puntero, antes de inicializarlo, no está definido a qué dirección de memoria apunta: el contenido de esa memoria será "basura" (no es un dato que nos interese).

```
int* ptr = nullptr;  // ptr no apunta a nada
if (ptr == nullptr) {
    std::cout << "El puntero es nulo" << std::endl;
```

Ilustración 5

En la ilustración anterior **nullptr** facilita la comprobación de punteros y evita errores comunes de acceso a memoria.

2. Modelo de la memoria de un programa

La segmentación de memoria de un programa se divide en varias zonas, que gestionan cómo se almacenan y acceden los datos mientras el programa se ejecuta. Estas áreas son clave para entender cómo funcionan variables, punteros, memoria dinámica y el ciclo de vida de los objetos. Estas secciones son Stack, Heap, Data Segment, Text Segment y Registers.

Cada uno de estos segmentos cumple un papel crucial en cómo C++ gestiona la memoria. El Stack se usa para datos temporales y locales, el Heap para datos dinámicos, el Data Segment para variables globales y estáticas, y el Text Segment contiene el propio código del programa.

En esta asignatura nos concentraremos en las secciones Stack y Heap.

2.1. Stack (Pila)

Una característica de la stack es que las variables que se almacenan en ella tienen un identificador (un nombre), y así nos desentendemos de sus direcciones de memoria.

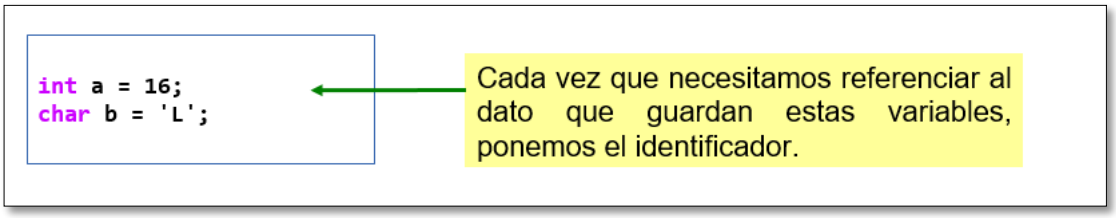


Ilustración 6

La pila es donde se almacenan las variables locales y los datos asociados con las llamadas a funciones. Cada vez que una función es llamada, sus variables y parámetros se colocan en la pila. Una vez que la función termina, el espacio en la pila se libera automáticamente. Dicho de otro modo, las variables dentro de la stack son gestionadas automáticamente (cuando una función retorna, sus variables son desalojadas de la memoria), desligando al programador de esta tarea.

Ejemplo:

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

void functionA() {
    int a = 10;  // Variable local de functionA, almacenada en el stack
    int b = 20;  // Otra variable local de functionA
    cout << "En functionA, a = " << a << ", b = " << b << endl;
}

void functionB() {
    int x = 30;  // Variable local de functionB, almacenada en el stack
    int y = 40;  // Otra variable local de functionB
    cout << "En functionB, x = " << x << ", y = " << y << endl;
}

int main() {
    char a = 'U';  // Variable local de main, almacenada en el stack
    char b = 'N';  // Otra variable local de main
    cout << "En la función main , a = " << a << ", b = " << b << endl;
    functionA();  // Llamada a functionA,
    // las variables locales se colocan en el stack
    functionB();  // Llamada a functionB,
    // usa su propio espacio en el stack
    return 0;
}
```

Ilustración 7

Analicemos el código anterior paso a paso.

```cpp
int main() {
    char a = 'U';  // Variable local de main, almacenada en el stack
    char b = 'N';  // Otra variable local de main
    cout << "En la función main , a = " << a << ", b = " << b << endl;
    functionA();  // Llamada a functionA,
    // las variables locales se colocan en el stack
    functionB();  // Llamada a functionB,
    // usa su propio espacio en el stack
    return 0;
}
```

Ilustración 8

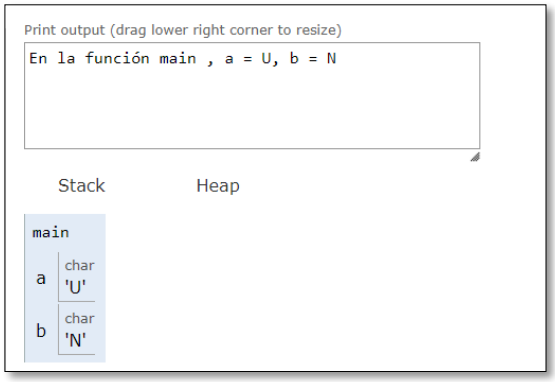


Ilustración 9

Cuando se ejecutan las tres líneas señaladas en la *Ilustración 8*, se puede observar en la *Ilustración 9* cómo se reservaron dos lugares en la memoria Stack, una para cada variable de tipo “char”.

```cpp
int main() {
    char a = 'U';  // Variable local de main, almacenada en el stack
    char b = 'N';  // Otra variable local de main
    cout << "En la función main , a = " << a << ", b = " << b << endl;
    functionA();  // Llamada a functionA,
    // las variables locales se colocan en el stack
    functionB();  // Llamada a functionB,
    // usa su propio espacio en el stack
    return 0;
}
```

Ilustración 10

Después cuando se invoca a functionA() (*Ilustración 10*), en la Ilustración 11 puede observarse que se reservan nuevos espacios en la memoria Stack exclusivos para esa función.

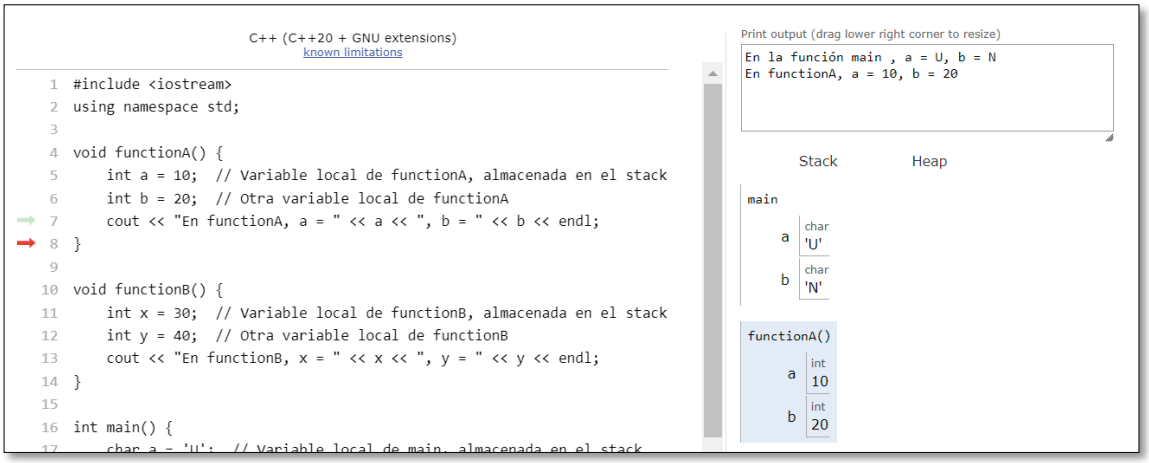


Ilustración 11

Luego, cuando la función termina, ese espacio se destruye, y la memoria usada por a y b se libera.

Lo mismo sucede cuando se invoca a functionB():

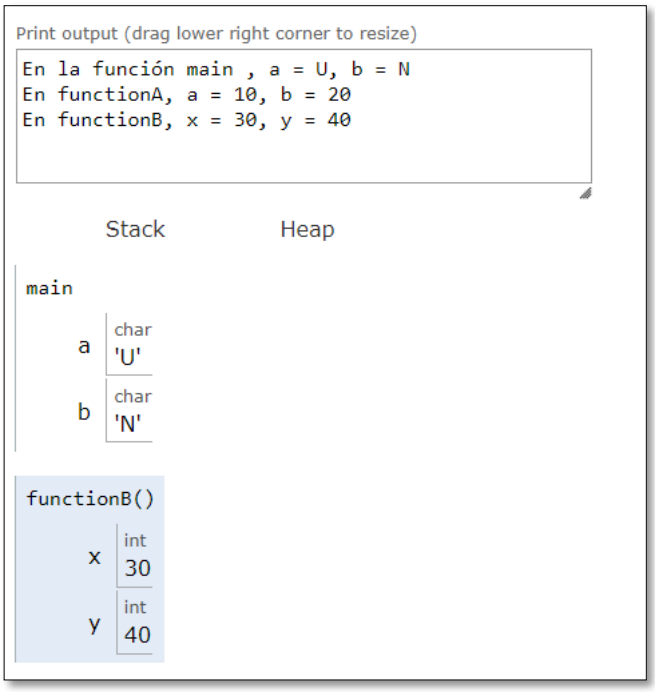


Ilustración 12

Al finalizar functionB(), el espacio en la pila también se libera.

Para probar el código acceder al *LINK*

En cuanto a este tipo de memoria podemos decir que es rápida y automática, lo cual es muy ventajoso; pero tiene un tamaño limitado y no permite manejar memoria dinámica.

2.2. Heap (Montón)

Aquí es donde se almacena la memoria dinámica que se gestiona explícitamente en el programa usando operadores como new y delete. Los datos acá sólo pueden accederse mediante punteros, ya que no tienen un nombre. Para poder reservar un espacio en la memoria heap se usa la instrucción new, y para desalojar un dato de la memoria se usa delete.

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

int main() {
  int* c = new int;
  *c = 72;

  return 0;
}
```

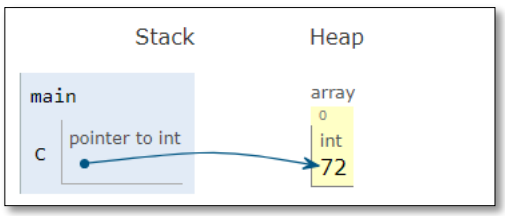


Ilustración 13

En la *Ilustración 13*, se observa que se reserva un espacio en la memoria dinámica con la instrucción new. La variable “c” es de tipo puntero y apunta a una dirección de la memoria que sólo puede almacenar un entero.

Y, ¿por qué querríamos guardar cosas en la memoria heap?

El problema de la stack es que es estática y limitada, el espacio que se le reserva al iniciarse el programa no puede cambiar durante la ejecución.

En cambio, la memoria heap es dinámica, si se necesita más espacio, el sistema operativo se encargará de otorgarlo.

Pero la memoria heap no es gestionada automáticamente (a diferencia de la stack que sí lo es) sino que es responsabilidad del programador, esto significa que las instrucciones para reservar (new) y desalojar (delete) espacio deben ser explícitas.

La stack, además, almacena todo en posiciones contiguas, por lo que la reserva y desalojo de memoria son operaciones que se pueden realizar rápidamente.

La memoria heap, al ir reservando y liberando memoria a lo largo de la ejecución del programa, almacena los datos donde encuentra un espacio lo suficientemente grande. Pero esto significa que, entre un dato y otro, pueden haber "huecos". Las operaciones de reservar y liberar memoria en la heap son más lentas que en la stack.

En ese sentido, se debe estar atento a posibles errores como:

- ✓ Fugas de memoria: Si asignamos memoria dinámicamente (con new) y olvidamos liberarla (con delete), el programa consumirá más memoria de la necesaria.
- ✓ Punteros colgantes: Que ocurren cuando un puntero sigue apuntando a una dirección de memoria que ha sido liberada.

2.3. Fuga de memoria

Si un dato que reside en la heap deja de estar apuntado por algún puntero (es decir, si ya nada apunta a él) se vuelve inaccesible y no habrá forma de desalojarlo hasta que el programa finalice.

La existencia de datos inaccesibles en la memoria heap es llamada "fuga de memoria" (ó "memory leak") y puede llevar a que el programa falle inesperadamente debido a la falta de memoria disponible.

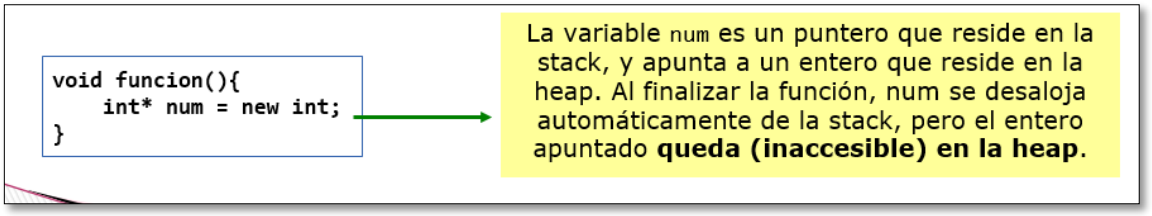


Ilustración 14

2.4. Puntero colgante

El problema opuesto a la fuga de memoria es lo que se llama "puntero colgante" ("dangling pointer"): un puntero queda apuntando a una dirección de memoria de la heap que ya fue desalojada. Esto causa errores del tipo “segmentation fault”.

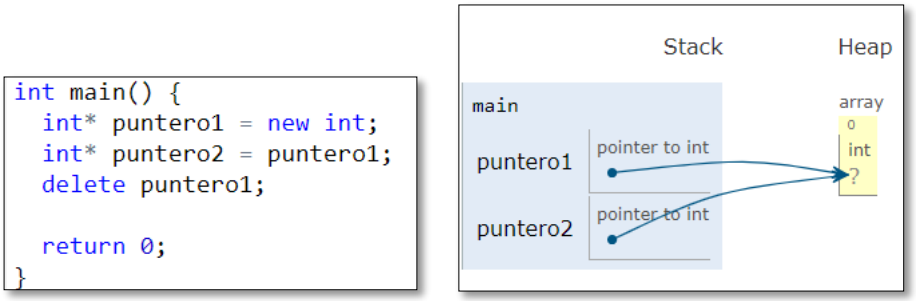


Ilustración 15

En el ejemplo que se visualiza en la *Ilustración 15*, la instrucción new int reserva, en la heap, un espacio suficiente para almacenar un entero, y su dirección de memoria queda guardada en la variable puntero1 (que reside en la stack).

Luego, la variable puntero2 (también en la stack) contendrá la misma dirección que puntero1. Pero, con el uso de delete, se elimina el espacio de memoria heap que es apuntado por puntero1 (que, casualmente, es el mismo al que apunta puntero2).

De ese modo, puntero2 es un dangling pointer que referencia a una dirección de memoria que ha quedado desalojada.

Finalmente podemos resumir la comparación de ambas memorias en el siguiente cuadro:

STACK	HEAP
Las variables se desalojan automáticamente.	Manejada manualmente por el programador (new / delete).
La alocación se realiza más rápidamente.	Alocación más lenta que en la stack.
Los datos almacenados pueden accederse con nombres de variables.	Los datos sólo se acceden mediante punteros.
Normalmente tiene un espacio limitado y fijo al iniciar la ejecución del programa (estática).	Puede crecer a medida que se requiera más espacio (dinámica).

3. Punteros. Objetos dinámicos

Como ya se mencionó, el lenguaje C++ dispone de dos operadores para manejar la memoria dinámica:

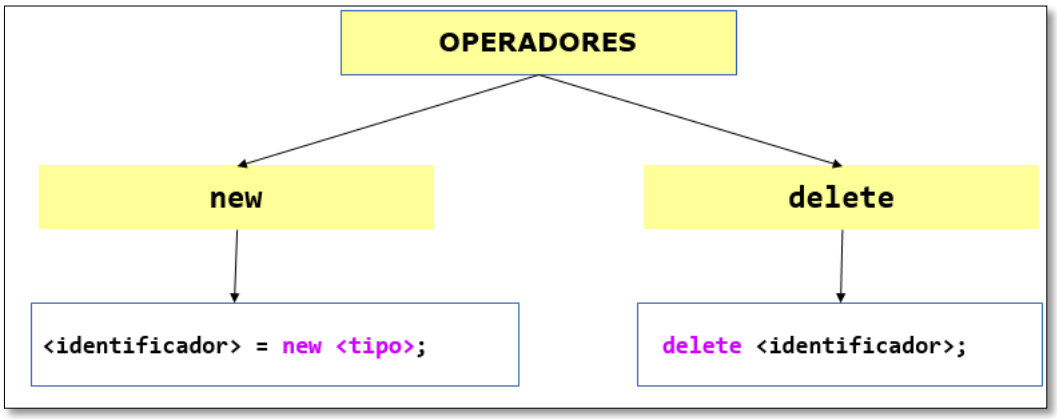


Ilustración 16

Como se observa en la ilustración anterior, el operador **new**, no sólo reserva un espacio en memoria heap, sino que además retorna la dirección del espacio que reservó. Es por eso que normalmente hacemos algo con esa dirección. Por ejemplo, almacenarla en una variable.

Esa variable deberá ser de tipo puntero, ya que son los punteros quienes pueden almacenar direcciones de memoria. Observar la Ilustración 17:

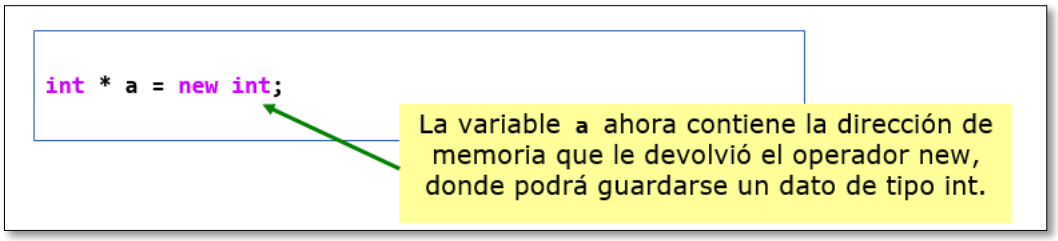


Ilustración 17

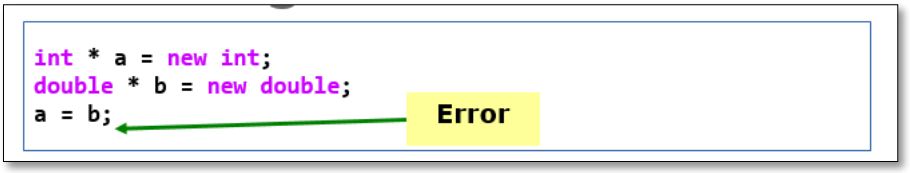


Ilustración 18

Si se declara a la variable "a" como puntero a int, entonces sólo podrá guardarse en ella la dirección de memoria donde resida un dato de tipo int. Es por ello que se presentará un error en la última línea de la *Ilustración 18*, ya que se intenta guardar una dirección destinada a otro tipo de dato (double).

Recordemos que es C++ un lenguaje fuertemente tipado, y por ello nos impide almacenar en una variable un dato que no corresponda a su tipo (a menos que se haga una conversión de tipos). En ese sentido, los punteros no son la excepción, y sólo podemos guardar en un puntero una dirección de memoria, y esa dirección sólo puede contener un dato del tipo indicado por el puntero.

Como los punteros son variables, el valor que almacenan puede cambiar: pero siempre deberemos guardar en ellos una dirección de memoria (que corresponda a un dato del tipo indicado por el puntero).

Como sucede con cualquier otra variable, podemos copiar el valor que guarda un puntero (una dirección de memoria), en otra variable del mismo tipo. Así, tendremos dos punteros que apuntan a la misma dirección.

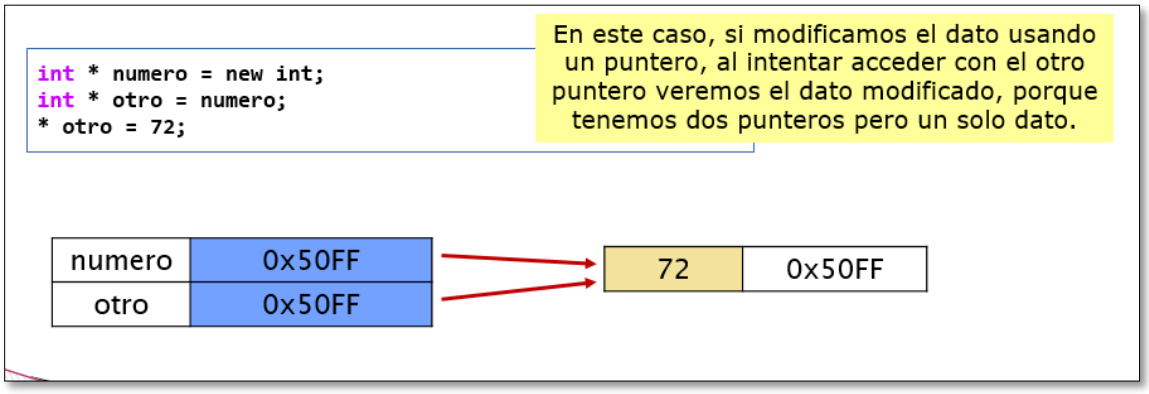
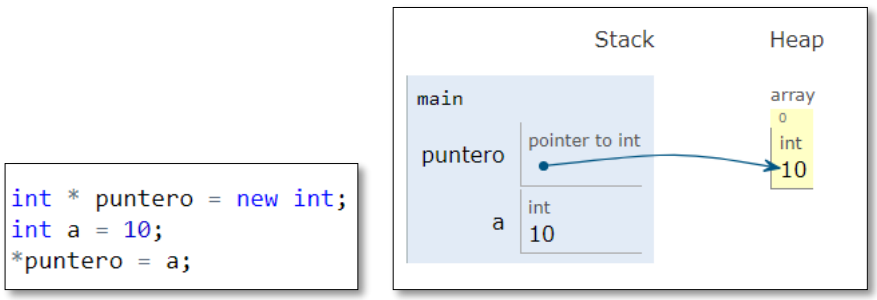


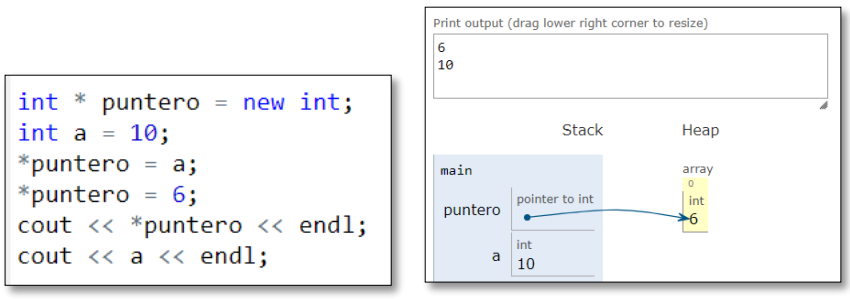
Ilustración 19

También como sucede con otras variables, el valor que guardemos en el espacio referenciado por el puntero puede provenir de cualquier expresión: una variable, un literal, el retorno de una función, un ingreso por teclado.



Si analizamos el ejemplo anterior, podemos ver que asignamos al dato apuntado por "puntero" el mismo valor que está guardado en "a". Ahora tenemos el mismo número (10) guardado en dos espacios de la memoria: uno en la stack (referenciado por la variable a) y otro en la heap, referenciado de forma indirecta por el puntero.

Ahora, si sumamos algunas líneas de código:



El programa va a imprimir 6 y 10, dado que sólo cambió el dato apuntado por "puntero" y no la variable "a".

Para recordar:

- Siempre que aparezca un **asterisco** (*) en una **definición** de variable, **ésta es una variable puntero**.
- Siempre que aparezca un **asterisco** (*) **delante de una variable puntero, se accede a la variable referenciada por el puntero**. Es decir, se accede al valor contenido en la dirección de memoria al que se apunta.
- C++ requiere que las variables puntero direccionen realmente variables del mismo tipo de dato que está ligado a los punteros en sus declaraciones.

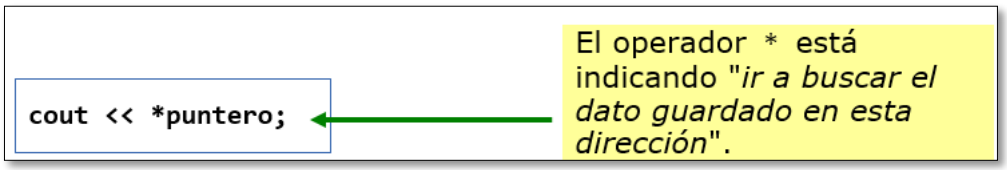
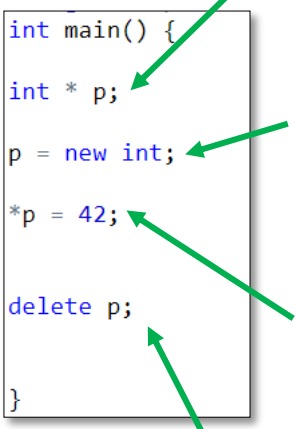
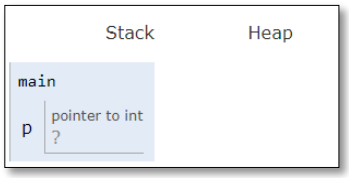


Ilustración 20

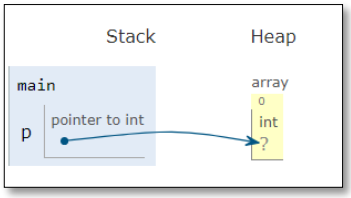
Entonces:



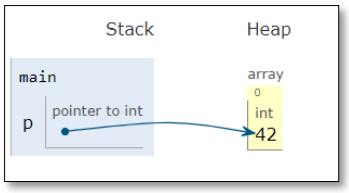
Se crea un puntero a entero, sin inicializar (no apunta a nada).

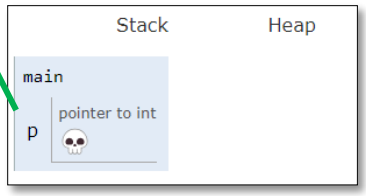


Reserva en la heap espacio para un int y devuelve la dirección de ese entero, la cual asigna a p.



Como **p** apunta a un entero, se lo puede **desreferenciar** y asignarle valores a la dirección que apunta.





Libera la memoria (p sigue teniendo la misma dirección, pero ya no es un puntero válido).

Una vez más hagamos hincapié en la siguiente cuestión “Toda la memoria que se reserve durante la ejecución del programa debería liberarse, a lo sumo, cuando el programa finalice (en general, antes).”

4. Puntero a struct

New puede usarse para reservar memoria para cualquier tipo de C++ (incorporado o definido por el usuario). Por ejemplo, en el siguiente código el puntero señala una dirección de memoria que almacenará un “Struct”:

```
struct Persona {
    string nombre;
    int edad;
};

int main(){
    Persona *p = new Persona;
}
```

Ilustración 21

Para acceder a los campos de una estructura hay que desreferenciar al puntero.

Posteriormente, puedes acceder al valor almacenado en esa dirección mediante el operador de desreferenciación *.

Repasemos este concepto: Desreferenciar un puntero en C++ significa acceder al valor o a los datos a los que apunta el puntero. En el caso de un puntero que apunta a una estructura (struct), desreferenciarlo permite acceder a los miembros de esa estructura. Supongamos que se tiene una estructura llamada Persona y un puntero que apunta a una instancia de esa estructura.

El operador **&** devuelve la dirección de la variable **p**.

```
#include <iostream>
#include <string>
using namespace std;
struct Persona {
    string nombre;
    int edad; };
int main() {
    Persona p = {"Juan", 25};    // Crear una instancia de Persona
    Persona* ptr = &p;           // Puntero que apunta a la estructura p

    // Desreferenciamos el puntero para acceder
    // a los miembros de la estructura
    cout << "Nombre: " << (*ptr).nombre << endl;
    cout << "Edad: " << (*ptr).edad << endl;

    // Alternativa más común: usando el operador '->'
    cout << "Nombre: " << ptr->nombre << endl;
    cout << "Edad: " << ptr->edad << endl;
     return 0;
}
```

Los **()** son necesarios, porque "." tiene mayor precedencia que "*".

El operador "*" desreferencia el puntero, mientras que el punto "." accede al campo de una estructura.

El operador "->" desreferencia al puntero y accede a un campo de la struct, todo junto.

Ilustración 22

5. Punteros como parámetros

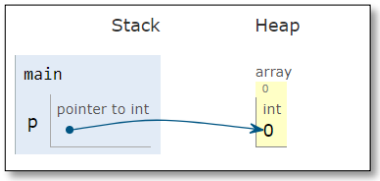
Un puntero puede pasarse como parámetro a una función. Si es por copia, se copia la dirección de lo que apunta.

En este caso, lo que se pasa por copia es el puntero, pero el dato apuntado en la heap se modifica dentro de la función y esta modificación subsiste luego de retornar.

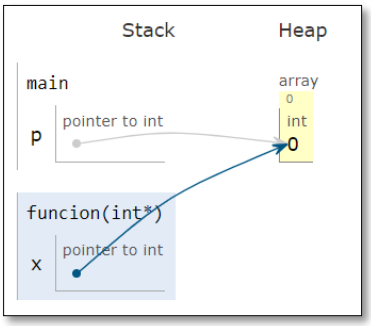
Veamos un ejemplo:

```
#include <iostream>
void funcion (int* x) {
    *x = 10;
}

int main () {
    int* p = new int;
    *p = 0;
    funcion(p);
}
```

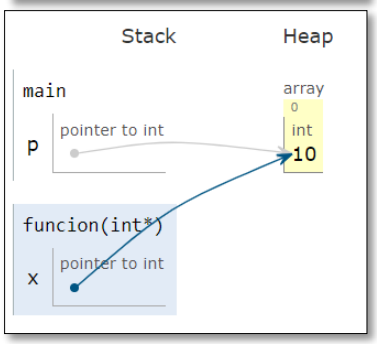
Estado de la memoria antes de invocar a la función.



Ejecutando la función. En este punto se recibe la variable puntero por copia, pero apunta al dato original.



Entonces, cuando se ejecute la línea ***x = 10** se modificará el dato original.



Otro ejemplo:

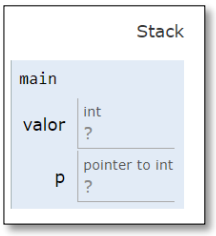
```cpp
#include <iostream>
#include <string>
using namespace std;
void modificarPorValor(int* ptr) {
    *ptr = 20;  // Cambiamos el valor al que apunta el puntero
    ptr = nullptr;  // Intentamos cambiar el puntero a null
    //(solo afecta dentro de la función)
}

int main() {
    int valor = 10;
    int* p = &valor;

    std::cout << "ANTES " << std::endl;
    std::cout << "Valor: " << valor << std::endl;
    std::cout << "Puntero: " << p << std::endl;

    modificarPorValor(p);

    std::cout << "DESPUÉS " << std::endl;
    std::cout << "Valor: " << valor << std::endl;
    // Se ve afectado, es 20
    std::cout << "Puntero: " << p << std::endl;
    // No se afectó, sigue apuntando a valor
}
```

Ilustración 23

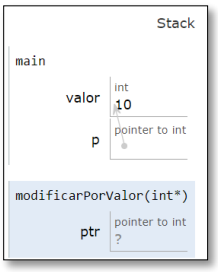
```
Print output (drag lower right corner to resize)
ANTES
Valor: 10
Puntero: 0xfff000bd4
DESPUÉS
Valor: 20
Puntero: 0xfff000bd4
```

Ilustración 24: salida del programa anterior.

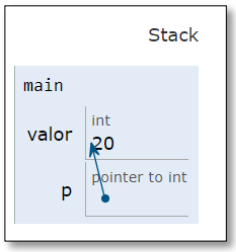
Como se aprecia en la *Ilustración 23*, dentro de la función modificarPorValor, se puede modificar el valor al que apunta el puntero (*ptr = 20;), y este cambio afectará el valor en la memoria. Sin embargo, cuando se intenta cambiar el puntero mismo (ptr = nullptr;), solo se está cambiando la copia local del puntero, y no afecta al puntero original en el contexto que lo llamó. A continuación, se grafica el estado de la memoria:



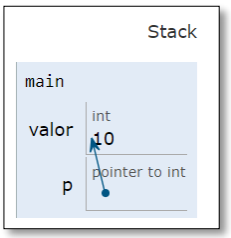
Al iniciar la función **main.**

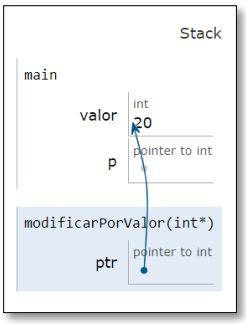


Al invocarse la función **modificarPorValor.**



Finalmente cuando regresa a la función **main**, el valor apuntado se modificó pero la variable puntero no.





Cuando pasas un puntero por referencia, estás pasando una referencia al puntero original, lo que permite modificar tanto el valor al que apunta como el propio puntero (la dirección que almacena).

Veamos un ejemplo:

```cpp
#include <iostream>
#include <string>
using namespace std;
void modificarPorReferencia(int*& ptr) {
    *ptr = 30;  // Cambiamos el valor al que apunta el puntero
    ptr = nullptr;  // Cambiamos el puntero a null
}

int main() {
    int valor = 10;
    int* p = &valor;

    std::cout << "Valor: " << valor << std::endl;
    std::cout << "Puntero: " << p << std::endl;
    modificarPorReferencia(p);

    std::cout << "Valor: " << valor << std::endl;  // Es 30
    std::cout << "Puntero: " << p << std::endl;    // Ahora es null
}
```

Ilustración 24

En este caso, pasar el puntero por referencia permite que la función modifique tanto el valor al que apunta el puntero como la propia dirección almacenada en el puntero. Por lo tanto, después de la función *modificarPorReferencia*, el puntero p será *nullptr*.

6. Lista enlazada con punteros.

El concepto de lista es bastante intuitivo. Encontramos varios ejemplos en la vida cotidiana, tal es el caso de una lista de pasajeros que vuelan en un avión, la lista del supermercado cuando vamos de compras, entre otros tantos ejemplos.

En programación refiere a una estructura de datos que permite almacenar una colección de elementos en un orden específico, que puede modificarse dinámicamente. A diferencia de los arreglos, las listas suelen ser más flexibles porque permiten agregar y eliminar elementos de manera dinámica sin la necesidad de definir un tamaño fijo al inicio.

Además, las listas pueden almacenar datos de distintos tipos, aunque en lenguajes como C++ es común trabajar con listas de un solo tipo de dato.

6.1. Características generales de las listas

Orden: Cada elemento tiene una posición dentro de la lista. En este contexto los elementos se denomina nodos.

Lineal: Cada nodo apunta a otro nodo.

Dinámicas: Se pueden modificar en tiempo de ejecución, agregando o eliminando elementos sin necesidad de redefinir su tamaño.

Acceso secuencial: A diferencia de un arreglo, donde podemos acceder directamente a cualquier elemento usando un índice, en las listas es común acceder a los elementos de manera secuencial, recorriendo uno a uno.

Memoria: Los nodos se almacenan en memoria dinámica. La ocupación de memoria se resuelve en tiempo de ejecución.

6.1.1. Nodos

Como se mencionó en el punto anterior, las listas están compuestas por nodos. Esta estructura tiene dos campos: *Dato* y *Enlace*. Si quisiéramos graficarlo, podríamos hacerlo del siguiente modo:

Ilustración 25: Estructura "Nodo".

Entonces, un nodo incluye el dato que almacena el valor en sí y un puntero que apunta al siguiente nodo de la lista.

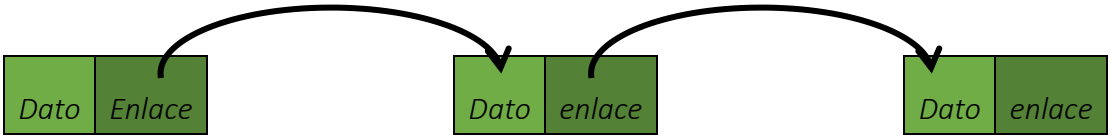


Ilustración 26: Lista enlazada

En la *Ilustración 26* se observa una lista enlazada simple ya que cada nodo apunta sólo al siguiente nodo. Sin embargo, existen otras variantes como las "listas dobles" en donde los nodos tienen un puntero tanto al nodo anterior como al siguiente y las "listas circulares" en cuyo caso el último nodo apunta al primero, formando una suerte de círculo.

En C++, una lista enlazada "simple" podría ejemplificarse del siguiente modo:

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

struct Nodo {
    int dato;
    Nodo * siguiente;
};
```

Ilustración 27

Observemos que se declara un struct llamado "Nodo" que tiene dos campos:

- **dato**, de tipo integer;
- **siguiente**, que es un puntero al siguiente nodo.

Cuando declaras el **primer elemento** (o nodo) de una lista enlazada en C++, este es típicamente un **puntero** a un nodo de la lista. Dicho puntero se declara en la memoria stack, porque es una variable local dentro de la función o el bloque de código donde estás trabajando. Luego, el resto de los nodos y sus enlaces estarán en la memoria heap.

Veamos un ejemplo:

```
#include <iostream>
using namespace std;
struct Nodo {
    int dato;         // Dato que almacena el nodo
    Nodo* siguiente; // Puntero al siguiente nodo
};
void insertarAlInicio(Nodo*& cabeza, int valor) {
    Nodo* nuevoNodo = new Nodo(); // Crear un nuevo nodo
    nuevoNodo->dato = valor;      // Asignar el valor al nodo
    nuevoNodo->siguiente = cabeza;
    // El nuevo nodo apunta al antiguo inicio
    cabeza = nuevoNodo;
    // El nuevo nodo se convierte en el nuevo inicio
}
void mostrarLista(Nodo* cabeza) {
    Nodo* actual = cabeza;
    // Puntero temporal para recorrer la lista
    while (actual != nullptr) {
        cout << actual->dato << " -> ";
        actual = actual->siguiente; // Avanzar al siguiente nodo
    }
    cout << "NULL" << endl;
}
int main() {
    Nodo* lista = nullptr; // Puntero al inicio de la lista
    // Insertar algunos elementos
    insertarAlInicio(lista, 10);
    insertarAlInicio(lista, 20);
    insertarAlInicio(lista, 30);
    mostrarLista(lista);
    return 0;}
```

Función **insertarAlInicio**: Inserta un nuevo nodo al comienzo de la lista. Esto requiere reasignar el puntero de inicio para que apunte al nuevo nodo.

Función **mostrarLista**: Recorre la lista desde el nodo de inicio hasta el último (cuando el puntero al siguiente nodo es nullptr), mostrando los valores almacenados.

Función **main**: Crea una lista vacía y luego inserta tres nodos. Finalmente, imprime el contenido de la lista.

Ilustración 28

Link al simulador

Es fundamental conservar el **puntero inicial** y **no perderlo** ya que será la única variable que tendremos para referenciar a la lista. Al ser un contenedor de acceso secuencial, siempre se debe comenzar recorriéndolo por el principio.

Resumiendo, el puntero inicial de una lista almacena la dirección del primer nodo y está en el stack. Mientras que los nodos, se crean dinámicamente en el heap para que permanezcan en la memoria hasta que el programador los libere explícitamente.

Este diseño es importante porque la memoria stack es limitada en espacio y no se puede usar para grandes cantidades de datos dinámicos. En cambio, en la heap, es más flexible y permite gestionar estructuras complejas que pueden crecer y reducirse dinámicamente en tiempo de ejecución.

6.2. Operaciones en listas enlazada con punteros.

Para comenzar con listas veremos qué operaciones se pueden realizar.

6.2.1. Crear una lista vacía

Para iniciar una lista se declara un puntero del tipo de nodo que hayamos definido. Además, se le asigna nullptr, recordemos que esto significa que "el puntero no apunta a nada", de este modo evitamos el contenido basura. Entonces, para indicar "no apunta a nada" usamos el valor **nullptr**.

```
    Nodo* lista = nullptr; // Puntero al inicio de la lista
```

La lista vacía aún no tiene nodos. Por cada elemento que se agregue, se generará un nuevo nodo y se lo enlazará en alguna parte de la lista (y el puntero inicial contendrá la dirección del primer nodo). Observar las siguientes líneas de código:

```
    Nodo* nuevoNodo = new Nodo(); // Crear un nuevo nodo
    nuevoNodo->dato = valor;      // Asignar el valor al nodo
    nuevoNodo->siguiente = cabeza;
    // El nuevo nodo apunta al antiguo inicio
    cabeza = nuevoNodo;
    // El nuevo nodo se convierte en el nuevo inicio
}
```

Ilustración 29

Entonces, cada vez que se necesite agregar un nuevo nodo a una lista, previamente se deberá generar ese nodo en memoria y almacenar en él los datos necesarios.

A continuación, se enlazará ese nodo con el resto de la lista, según el criterio de inserción, es decir al comienzo, al final o en algún lugar intermedio de la lista.

6.2.2. Insertar un elemento al principio de una lista

Supongamos que queremos agregar un elemento en una lista de nodos, insertando siempre al principio (efecto pila).

En primer lugar, se deberá generar un nuevo nodo para el elemento a agregar y almacenar en dicho nodo el dato. A continuación, se enlazará ese nodo al principio de la lista y dejaremos a este nuevo nodo como el primero. El puntero inicial ahora apuntará al nuevo nodo.

Si observamos la Ilustración 29, veremos que la variable **cabeza** que apuntaba al comienzo de la lista ahora pasa a un segundo lugar y se reemplaza por el nuevo elemento (en el ejemplo la variable se denomina **nuevoNodo**).

Para una mejor modularización, podríamos tener una función que, dado el puntero inicial de la lista y el nuevo nodo o elemento, sólo se encargue de crear el nodo, agregarle el o los elementos correspondientes y por último insertarlo en la lista.

```
void insertarAlInicio(Nodo*& cabeza, int valor) {
    Nodo* nuevoNodo = new Nodo(); // Crear un nuevo nodo
    nuevoNodo->dato = valor;      // Asignar el valor al nodo
    nuevoNodo->siguiente = cabeza;
    // El nuevo nodo apunta al antiguo inicio
    cabeza = nuevoNodo;
    // El nuevo nodo se convierte en el nuevo inicio
}
```

Ilustración 30

Esto nos sirve para que esta función de inserción sea más reutilizable.

6.2.3. Recorrer una lista

Para recorrer una lista será necesario empezar por el primer nodo y "visitar" cada uno de los demás nodos, realizando las operaciones deseadas (por ejemplo, imprimir los datos).

```
void mostrarLista(Nodo* cabeza) {
    Nodo* actual = cabeza;
    // Puntero temporal para recorrer la lista
    while (actual != nullptr) {
        cout << actual->dato << " -> ";
        actual = actual->siguiente; // Avanzar al siguiente nodo
    }
    cout << "NULL" << endl;
}
```

Ilustración 31

Como es indispensable no perder el puntero al nodo inicial, se utilizará un nuevo puntero, que comenzará apuntando al nodo inicial y luego se irá moviendo por el resto de la lista, hasta llegar al final (identificado con nullptr). Se puede observar en la *Ilustración 31*, que el nodo inicial es la variable **cabeza**.

6.2.4. Buscar un elemento en una lista.

La búsqueda de un elemento en una lista se realiza recorriéndola hasta encontrarlo o hasta llegar al final.

```
bool buscar(Nodo* inicio, int datoBuscado)
{
    for (Nodo* aux = inicio; aux != nullptr; aux = aux->siguiente){
      if (aux->dato == datoBuscado)
        {return true;}
    } return false;
}
```

Ilustración 32

En el ejemplo de la *Ilustración 32*, se observa una función que recibe el puntero inicial de la lista (inicio) y el dato a buscar (datoBuscado), y recorre la lista hasta encontrar un nodo que contenga a ese dato.

La función retorna true o false de acuerdo a si encontró al dato o no.

Otra opción podría retornar un puntero al nodo que contiene el elemento, en lugar de sólo indicar si se lo halló o no. Esto dependerá de las necesidades de nuestro programa. Observar la *Ilustración 33*:

```
Nodo* buscar(Nodo* inicio, int datoBuscado)
{   Nodo* aux = inicio;
    while (aux != nullptr && aux->dato != datoBuscado)
    {aux = aux->siguiente;}
    return aux;
}
```

Ilustración 33

En el caso anterior la función retorna un puntero. Si se encontró el dato buscado, el puntero retornado tendrá la dirección del nodo que contiene a ese dato. Si no se encontró, el puntero retornado será nullptr.

6.2.5. Insertar un elemento al final de una lista

Supongamos ahora que se desea agregar un nuevo elemento, pero esta vez, al final de la lista, de manera que se mantenga el orden en que los datos fueron ingresados (efecto cola).

En ese sentido, deben hacerse dos cosas antes de insertar: crear un nuevo nodo para el dato y buscar el final de la lista. Finalmente, se enlaza el nuevo nodo.

```
Nodo* insertar_final(Nodo* inicio, Nodo* nuevo)
{
    if (inicio == nullptr)
        inicio = nuevo;
    else {
        Nodo* aux = inicio;
        while (aux->siguiente != nullptr) {
            aux = aux->siguiente;
        }
        aux->siguiente = nuevo;
    }
    return inicio;
}
```

Si la lista está vacía, el último nodo será también el primero.

Si la lista tiene elementos, buscamos el último nodo e insertamos a continuación

Usamos un puntero auxiliar para recorrer, para evitar perder la dirección del inicio de la lista.

Ilustración 34

6.2.6. Insertar un elemento en una lista ordenada

Supongamos que tenemos la siguiente lista de números enteros, ordenada en forma creciente, tal como se observa en la siguiente imagen:

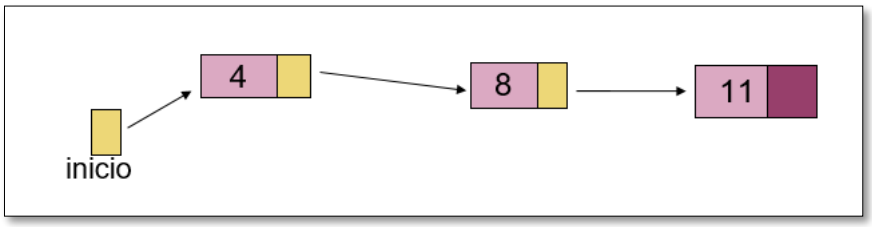


Ilustración 35

Ahora queremos insertar los valores 2, 6 y 15; de modo que quede de la siguiente manera:

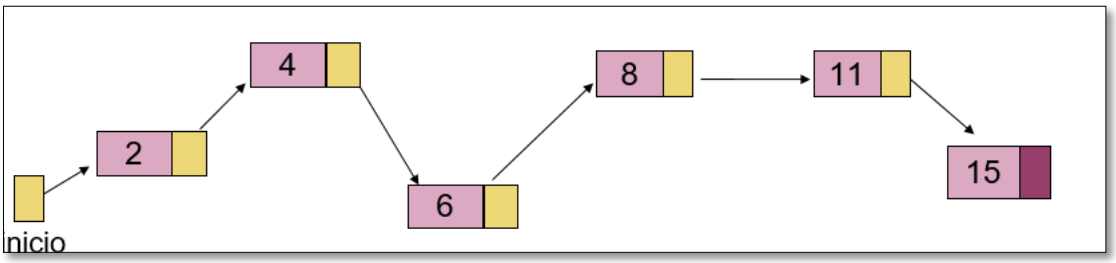


Ilustración 36

Entonces, al momento de insertar un nodo, tendremos que ver qué hacer, de acuerdo a distintos casos posibles:

1. **Si la lista está vacía**: el nodo a insertar será el primero (y también último).
2. **Si no está vacía**:
    a. Si el dato del nodo nuevo es más chico que el menor elemento de la lista: el nodo se insertará como primero de la lista.

b. Si el dato del nodo nuevo es más grande que el último elemento de la lista: el nodo se insertará al final de la lista.
c. Si el dato del nodo nuevo es mayor que el primero de la lista, pero menor que el último, se insertará en medio de dos nodos.

Analizando lo anterior se pueden observar que algunos casos parecen ser similares, entonces ¿podemos unir casos?

1. Si la lista está vacía o el dato es más chico que el menor elemento:
   a. Agregar el nodo como comienzo de la lista.
2. Si no:
   a. Buscar el último nodo que contenga un valor menor al que se quiere insertar.
   b. Hacer los enlaces necesarios. Hay dos casos posibles:
      i. El nuevo nodo va en medio de dos nodos existentes.
      ii. El nuevo nodo va al final de la lista.

```
Nodo* insertar_ordenado(Nodo* inicio, Nodo* nuevo)
{
    if (inicio == nullptr || nuevo->dato < inicio->dato)
    {
        nuevo->siguiente = inicio;
        inicio = nuevo;
    }
    else
    {
        Nodo* aux = inicio;
        while (aux->siguiente != nullptr && aux->siguiente->dato < nuevo->dato) {
            aux = aux->siguiente;
        }
        nuevo->siguiente = aux->siguiente;
        aux->siguiente = nuevo;
    }
    return inicio;
}
```

Si la lista está vacía o el dato es más chico que el menor elemento, se inserta al principio.

Si no se insertó al principio, buscamos dónde debe insertarse.

- Si el siguiente de aux es `nullptr`, estamos en el último nodo de la lista, entonces el nuevo debe ir a final (*el nuevo será el siguiente de aux*).
- Si el siguiente de aux no es nullptr, se enlaza al nuevo entre dos nodos: *el siguiente del nuevo ahora será el que antes era siguiente de aux, y el siguiente de aux ahora es el nuevo.*

Ilustración 37

6.2.7. Eliminar un elemento de la lista

Al eliminar es posible que se deban modificar enlaces (dependiendo de si el nodo eliminado es el primero, el último o si está entre otros dos).

Supongamos que necesitamos eliminar el nodo con el valor 11:

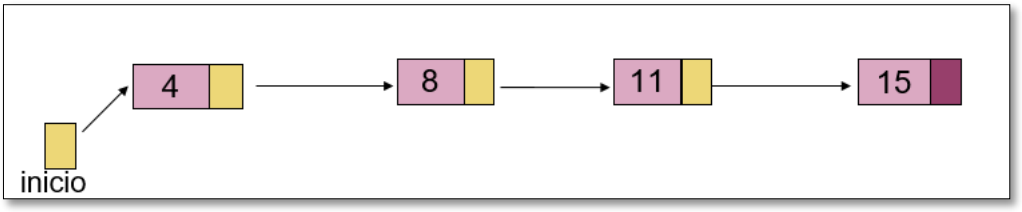


Ilustración 38

Ahora bien, será importante mantener al nodo a eliminar apuntado mediante algún puntero, ya que de otra forma quedaría inaccesible y sería imposible aplicarle la operación delete.

Entonces, para eliminar un elemento de una lista enlazada será necesario hacer tres cosas:

1. Buscar el nodo que contiene al elemento a eliminar (si existe)
2. Realizar el nuevo enlace, de ser necesario
3. Eliminar el puntero al nodo eliminado

El último paso es muy importante ya que, aunque un nodo deje de ser apuntado por un puntero, el espacio en la memoria heap **sigue estando ocupado e inutilizado** (ninguna otra variable se podrá almacenar en él).

Para realmente liberar el espacio en la memoria y no generar fugas de memoria, es necesario utilizar la instrucción **delete**.

```
Nodo* eliminar(Nodo* inicio, int datoABorrar) {
    if (inicio != nullptr) {
        Nodo* aux = inicio;
        if (inicio->dato == datoABorrar) {
            inicio = inicio->siguiente;
            delete aux;
        }
        else {
            while (aux->siguiente != nullptr && aux->siguiente->dato != datoABorrar) {
                aux = aux->siguiente;
            }
            if (aux->siguiente->dato == datoABorrar){
                Nodo* aEliminar = aux->siguiente;
                aux->siguiente = aEliminar->siguiente;
                delete aEliminar;
            }
        }
    }
    return inicio;
}
```

Sólo proceder si la lista tiene nodos

Se debe eliminar el primer nodo

Si no se eliminó el primer nodo, buscamos a partir del segundo

Ilustración 39

6.2.8. Eliminar varias ocurrencias de un elemento de la lista

Con el algoritmo anterior, si el elemento a eliminar se encuentra repetido, se elimina sólo la primera ocurrencia.

Si quisiéramos eliminar **todas las ocurrencias** deberíamos continuar iterando, cuidando de no avanzar en caso de haber hecho una eliminación.

Por lo tanto, será necesario contemplar algunos **nuevos casos posibles**. Ya no basta con verificar que la lista pueda estar vacía, que el elemento a borrar no exista, que exista al inicio, que exista en medio o que exista al final. Ahora también podría suceder que haya que eliminar el nodo inicial, el final y alguno en medio a la vez. O

que haya varios nodos consecutivos con el elemento a borrar. O que la lista esté compuesta sólo por nodos con el elemento a borrar.

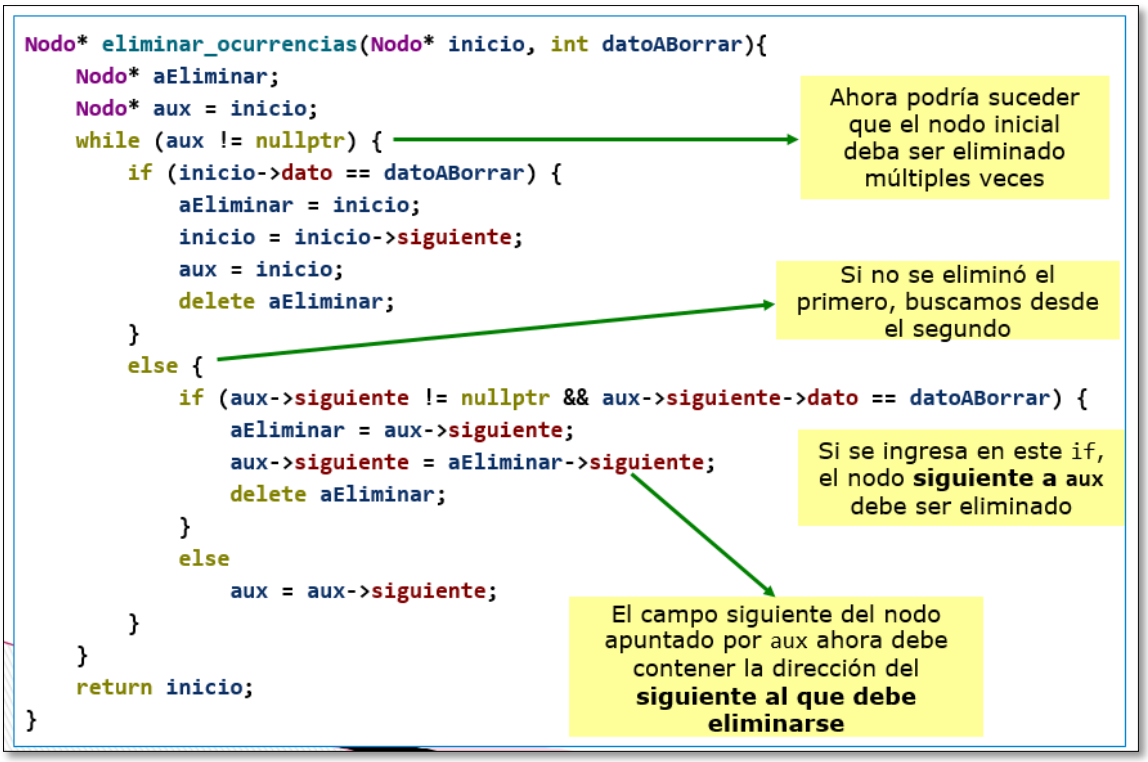


Ilustración 40

6.2.9. Dividir una lista en dos o más.

Supongamos que, dada una lista, obtener nuevas listas reutilizando sus nodos. Entonces, ¿cómo debe hacerse?

- Se debe tener algún criterio por el cual separar a la lista original (por ejemplo: dividir una lista de números en una lista con los pares y otra con los impares).
- Se reutilizan los nodos de la lista original, generando nuevos enlaces.
- La lista original quedará vacía (puntero inicial en nullptr).

Si se ha modularizado correctamente la inserción, bastará con usar el puntero inicial de la nueva lista y el nodo de la lista original que deseamos insertar en ella.

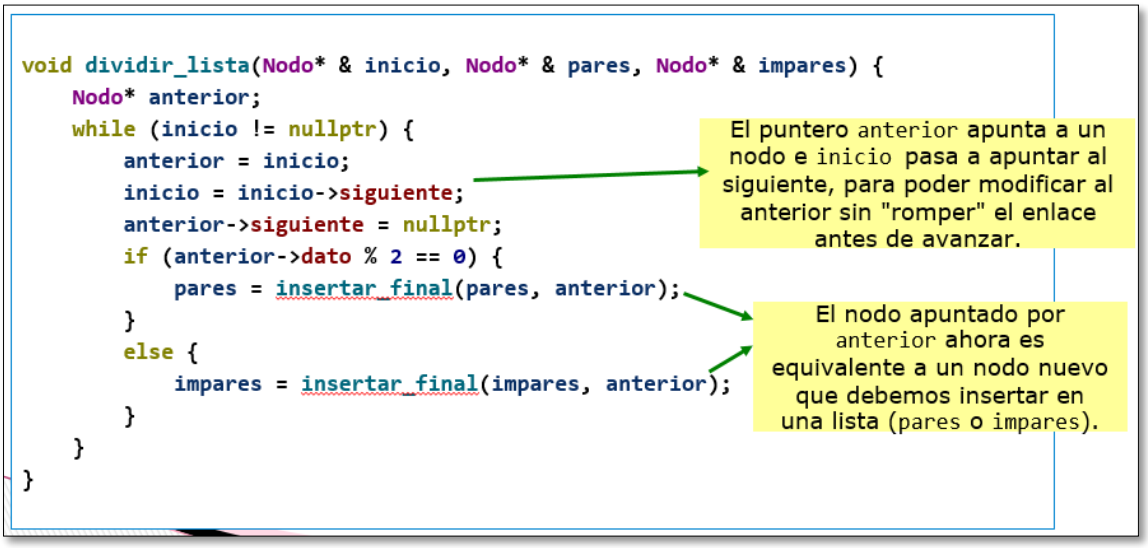


Ilustración 41

6.2.10. Combinar dos listas ordenadas, formando una nueva lista ("merge")

Se trata de combinar dos listas con sus datos ordenados de acuerdo a cierto criterio, para obtener una nueva lista que también esté ordenada por el mismo criterio.

```
lista1 = 10 15 20 25 30
lista2 = 6 12 14
merge = 6 10 12 14 15 20 25 30+
```

¿Cómo hacerlo?

- Si una de las dos listas está vacía, retornamos la otra.
- Si ninguna está vacía, recorremos ambas a la vez, insertando el nodo con el valor menor en una tercera lista y luego avanzando.
- Si una lista termina antes que la otra (pueden ser de distintas longitudes), continuamos procesando la restante.

Para obtener una lista ordenada que sea una copia unificada de las otras dos (es decir, sin modificar las listas originales), podemos iterar por cada lista, creando un nuevo nodo y llamando a la función insertar_ordenado por cada nodo.

Pero en este caso veremos cómo unificar dos listas reutilizando sus nodos. Es decir, con un algoritmo destructivo, que reenlaza los nodos de las listas originales. Al finalizar, los punteros iniciales de las listas originales quedan ambos en nullptr.

En pseudocódigo podríamos definir:

```
funcion merge(A, B):
    C = nullptr

    mientras A no sea nullptr y B no sea nullptr:
        si dato en nodo inicial de A ≤ dato en nodo inicial de B entonces:
            insertar nodo inicial de A en C
            avanzar A al siguiente nodo
        si no
            insertar nodo inicial de B en C
            avanzar B al siguiente nodo

    //En este punto se terminó de recorrer A o B. Seguimos procesando la otra.
    mientras A no sea nullptr:
        insertar nodo inicial de A en C
        avanzar A al siguiente nodo
    mientras B no sea nullptr:
        insertar nodo inicial de B en C
        avanzar B al siguiente nodo

    retornar C
```

Ilustración 42

Como en la mayoría de los casos, no existe un único algoritmo para unificar dos listas. En ese sentido habrá que buscar siempre más opciones.

```
Nodo* merge(Nodo* & A, Nodo* & B) {
    Nodo* C = nullptr; Nodo* anterior;

    while (A != nullptr && B != nullptr) {
        if (A->dato <= B->dato) {
            anterior = A;
            A = A->siguiente;
        }
        else {
            anterior = B;
            B = B->siguiente;
        }
        anterior->siguiente = nullptr;
        C = insertar_final(C, anterior);
    }
    while (A != nullptr) {
        anterior = A;
        A = A->siguiente;
        anterior->siguiente = nullptr;
        C = insertar_final(C, anterior);
    }
    while (B != nullptr) {
        anterior = B;
        B = B->siguiente;
        anterior->siguiente = nullptr;
        C = insertar_final(C, anterior);
    }
    return C;
}
```

Se intercalan los nodos, reutilizando la función de inserción al final. El menor de ambas listas se inserta al final de la nueva

Si se llegó al final de una de las listas (que tenía menos elementos), continuamos con la otra

Si observamos, los dos bucles while finales hacen lo mismo, con distintas variables, y también coincide con el primer bucle while. Podríamos colocar el algoritmo en otra función y llamarla cuando se la necesite...

78

Ilustración 43

```
void reinsertarNodo(Nodo* & lista_original, Nodo* & lista_nueva) {
    Nodo* anterior = lista_original;
    lista_original = lista_original->siguiente;
    anterior->siguiente = nullptr;
    lista_nueva = insertar_final(lista_nueva, anterior);
}

Nodo* merge(Nodo* & A, Nodo* & B) {
    Nodo* C = nullptr;

    while (A != nullptr && B != nullptr) {
        if (A->dato <= B->dato) {
            reinsertarNodo(A, C);
        }
        else {
            reinsertarNodo(B, C);
        }
    }
    while (A != nullptr) {
        reinsertarNodo(A, C);
    }
    while (B != nullptr) {
        reinsertarNodo(A, C);
    }
    return C;
}
```

La función reenlaza el primer nodo de **lista_original** para colocarlo al final de **lista_nueva**. Luego avanza el puntero de **lista_original** para no perder su inicio.

Ilustración 44

7. Listas de struct.

Los ejemplos vistos hasta el momento fueron de listas cuyos nodos contienen un dato simple (de tipo int). Pero nada impide que el dato sea de cualquier otro tipo, incluso una struct, tal es el caso del siguiente código:

```
struct Articulo {
    string descripcion;
    float precio;
};

struct Nodo {
    Articulo articulo;
    Nodo * siguiente;
};
```

Ilustración 45

En esta lista anterior, si desreferenciamos un nodo y obtenemos su campo articulo, estaremos obteniendo una struct de tipo Articulo, y se podrá trabajar con ella como con cualquier struct (usando el operador "." para acceder a cada campo).

Otro modo de resolver la misma situación sería representar a cada nodo con los datos necesarios como campos (sin crear otra struct), por ejemplo:

```
struct Nodo {
    string descripcion;
    float precio;
    Nodo * siguiente;
};
```

Ilustración 46

En el caso de la Ilustración 46, se complicarían algunos algoritmos, por ejemplo, en el caso en que fuese necesario copiar los elementos de la lista a un arreglo, colocando los datos de cada artículo en cada elemento del arreglo.

Si hacemos que los elementos del arreglo sean de tipo Nodo, estaremos incluyendo un campo puntero al siguiente, algo que en un arreglo no tiene sentido.

Pero, en cambio, tenemos una struct Articulo dentro de cada nodo, podremos hacer un arreglo de Articulo y sólo copiar este dato.

Cuando están por separado, es importante diferenciar las dos structs: una representa cada nodo de la lista y la otra representa el dato contenido en cada nodo. Además, se debe recordar que el puntero inicial es sólo una dirección de memoria, no una struct.

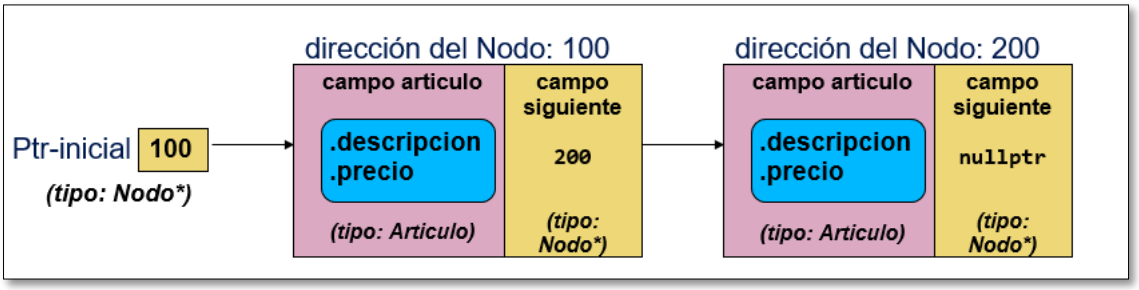


Ilustración 47

El ejemplo anterior se intenta graficar una lista con dos nodos, donde puede verse que el puntero inicial sólo es un dato de tipo Nodo* con la dirección del primer nodo de la lista. Cada Nodo tiene dos campos: uno de tipo Articulo (que es una struct con todos los campos que componen a un Articulo) y otro de tipo Nodo* que tiene la dirección del siguiente nodo. En la lista, el último nodo tiene como siguiente a nullptr, indicando que es el último.

Un punto importante a tener en cuenta es que sólo se debe reservar espacio en memoria (“new Nodo”) cuando se sabe efectivamente que se necesitará para insertar datos. Si se reserva y no se usa, se ocasionará una fuga de memoria.

En el ejemplo a continuación se señala el lugar correcto para reservar memoria para un nuevo nodo.

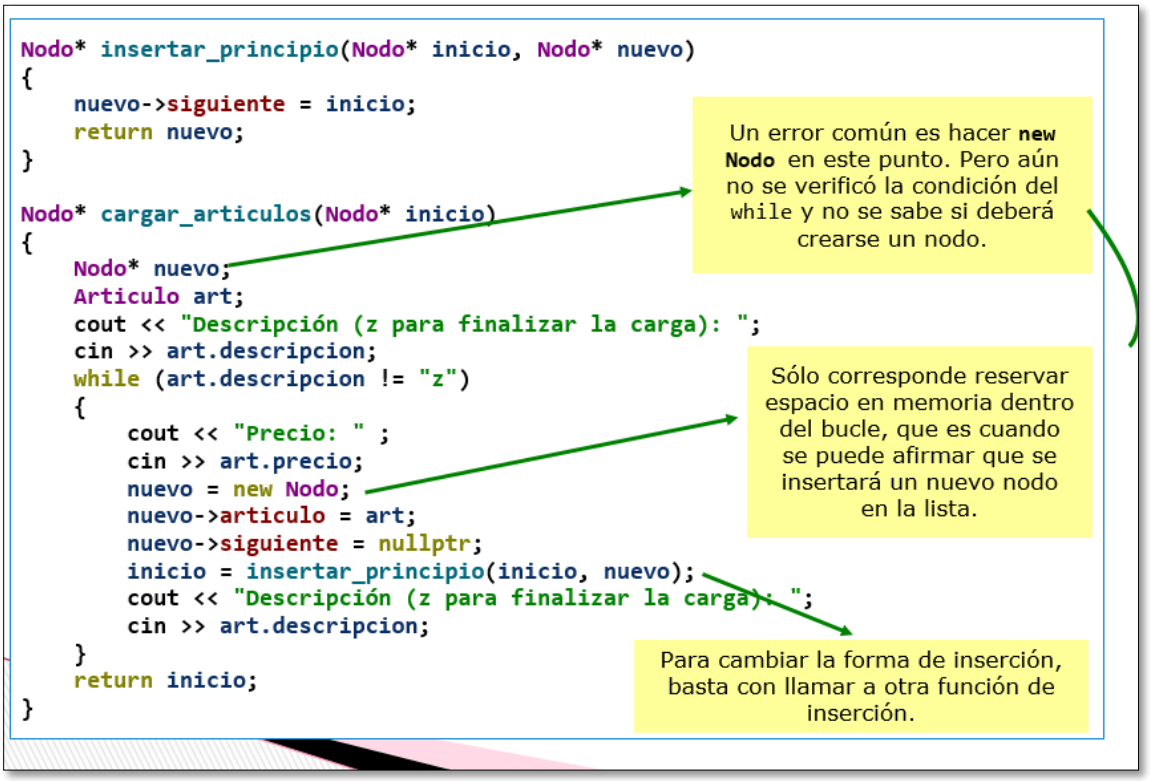


Ilustración 48

Para permitir mayor reutilización de código, es apropiado hacer funciones por separado: la carga de los datos en una y el algoritmo de inserción en otra. Esto permitirá cambiar fácilmente el tipo de inserción (adelante, al final u ordenada) con sólo modificar la llamada a la función de inserción.

Yendo más allá, una modularización óptima separaría la operación de generar un nuevo nodo (con los datos correspondientes y el puntero siguiente inicializado como nullptr) en una tercera función. Así, podremos generar nodos sólo cuando lo requiera el problema a resolver.

```
Nodo* insertar_principio(Nodo* inicio, Nodo* nuevo) {
    nuevo->siguiente = inicio;
    return nuevo;
}
Nodo* generar_nodo(Articulo art) {
    Nodo* nuevo = new Nodo;
    nuevo->articulo = art;
    nuevo->siguiente = nullptr;
    return nuevo;
}
Nodo* cargar_articulos(Nodo* inicio) {
    Articulo art;
    Nodo* nuevo;
    cout << "Descripción (z para finalizar la carga): ";
    cin >> art.descripcion;
    while (art.descripcion != "z") {
        cout << "Precio: " ;
        cin >> art.precio;
        nuevo = generar_nodo(art);
        inicio = insertar_principio(inicio, nuevo);
        cout << "Descripción (z para finalizar la carga): ";
        cin >> art.descripcion;
    }
    return inicio;
}
```

Este ejemplo es muy similar al anterior, pero cada tarea bien identificada se ha colocado en una función por separado: una para pedir al usuario los datos, una para generar el nuevo nodo y una para insertarlo en la lista.

De esta forma, cada función es atómica y permite utilizar cada operación sólo cuando se la necesita.

Ilustración 49

8. Listas circulares.

Son muy similares a las listas simples, sólo que no tienen un valor nullptr al final ya que, al recorrer todos los elementos, se vuelve al nodo inicial.

En ese sentido, si la lista tiene un solo nodo, su puntero siguiente referenciará al mismo nodo.

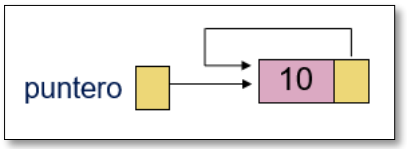


Ilustración 50

Si la lista está vacía, al igual que en las listas simples, el puntero contendrá nullptr.

Ilustración 51

Son útiles para implementar estructuras como la lista de procesos en ejecución en un sistema operativo: todos los procesos se colocan en una lista circular y el sistema operativo le da una fracción de tiempo del procesador a cada uno.

Cuando un proceso finalizó su tiempo, se ejecuta el siguiente y el anterior queda a la espera de que se le otorgue nuevamente el procesador. Cuando se termina con el último, se vuelve a dar lugar al primero hasta que termine su ejecución.

Podemos notar que no conviene tener un puntero al *primer* elemento de la lista, sino al *último*, ya que desde él podemos acceder también al siguiente (que es el primero).

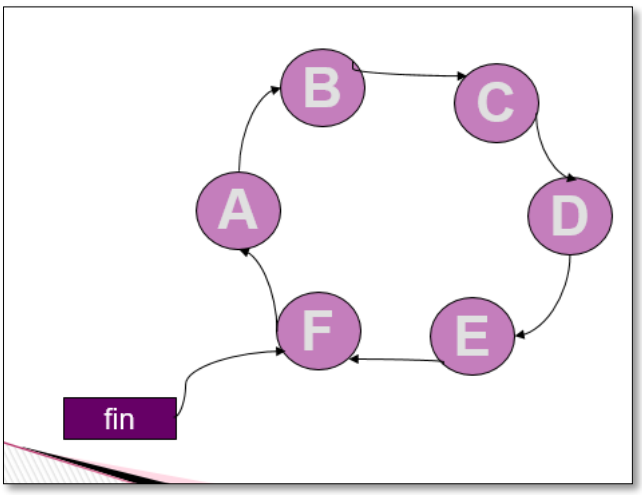


Ilustración 52

Si sólo hay un puntero al primer nodo, para insertar un elemento al final (es decir, entre el nodo F y el nodo A), se deberá recorrer toda la lista hasta encontrar el final.

En cambio, desde el último puede accederse al primero mediante fin->siguiente.

8.1. Listas circulares: declaración y operaciones.

La declaración de los nodos de una lista circular será igual que en las listas simples, y las operaciones que pueden realizarse son muy similares a las de una lista simple.

```
struct Nodo {
    int dato;
    Nodo* siguiente;
};
```

También podría almacenar una struct, o cualquier otro tipo de dato

Ilustración 53

La diferencia es que ahora tendremos un puntero que, aunque inicialmente contendrá nullptr mientras la lista no tenga nodos (vacía), luego apuntará siempre al último nodo de la lista.

```
Nodo* fin = nullptr;
```

8.2. Insertar un valor al principio de la lista.

Cuando la lista está vacía, el nodo insertado será también el último y el único. Y, al tratarse de una lista circular, ese único nodo debe apuntarse a sí mismo.

Si la lista tiene elementos, el nodo nuevo debe apuntar ahora al primero y el final apuntará al nuevo.

```
Nodo* insertar_principio(Nodo* fin, Nodo* nuevo)
{
    if (fin == nullptr)
    {
        nuevo->siguiente = nuevo;
        return nuevo;
    }
    else
    {
        nuevo->siguiente = fin->siguiente;
        fin->siguiente = nuevo;
        return fin;
    }
}
```

Esta función siempre retorna un puntero al último nodo de la lista.

Sabemos que el siguiente del nodo final es el primero de la lista. Ahora el nuevo nodo debe ser el primero de la lista, por lo que `fin` lo apuntará.

Ilustración 54

8.3. Insertar un valor al final de la lista.

El algoritmo es muy similar, sólo que ahora el puntero fin deberá apuntar al nuevo nodo insertado.

```
Nodo* insertar_final(Nodo* fin, Nodo* nuevo)
{
    if (fin == nullptr)
    {
        nuevo->siguiente = nuevo;
    }
    else
    {
        nuevo->siguiente = fin->siguiente;
        fin->siguiente = nuevo;
    }
    return nuevo;
}
```

Esta función también retorna un puntero al último nodo de la lista.

Ilustración 55

8.4. Recorrer la lista.

Como sabemos que tenemos un puntero al nodo final y queremos recorrer desde el principio, entonces comenzamos por el nodo siguiente al final (previamente se debe verificar que exista un siguiente al final).

```
void imprimir(Nodo* fin)
{
    if (fin != nullptr)
    {
        Nodo* aux = fin->siguiente;
        do {
            cout << aux->dato << endl;
            aux = aux->siguiente;
        } while (aux != fin->siguiente);
    }
}
```

En este ejemplo se recorre la lista para imprimirla, pero podría recorrerse con cualquier otro objetivo.

Ilustración 56

Iteramos hasta volver a encontrarnos con el primer nodo.

8.5. Eliminar un elemento.

Para eliminar un nodo deberemos recorrer la lista (en caso de no estar vacía), buscando el nodo a eliminar.

En caso de encontrarlo, será necesario reenlazar el nodo anterior al que queremos borrar, para que ahora apunte al siguiente al que queremos borrar.

Al igual que con las listas simples, una forma de acceder al nodo anterior al que queremos eliminar es con un puntero que apunte a un nodo mientras "miramos" si el nodo siguiente es el que queremos eliminar.

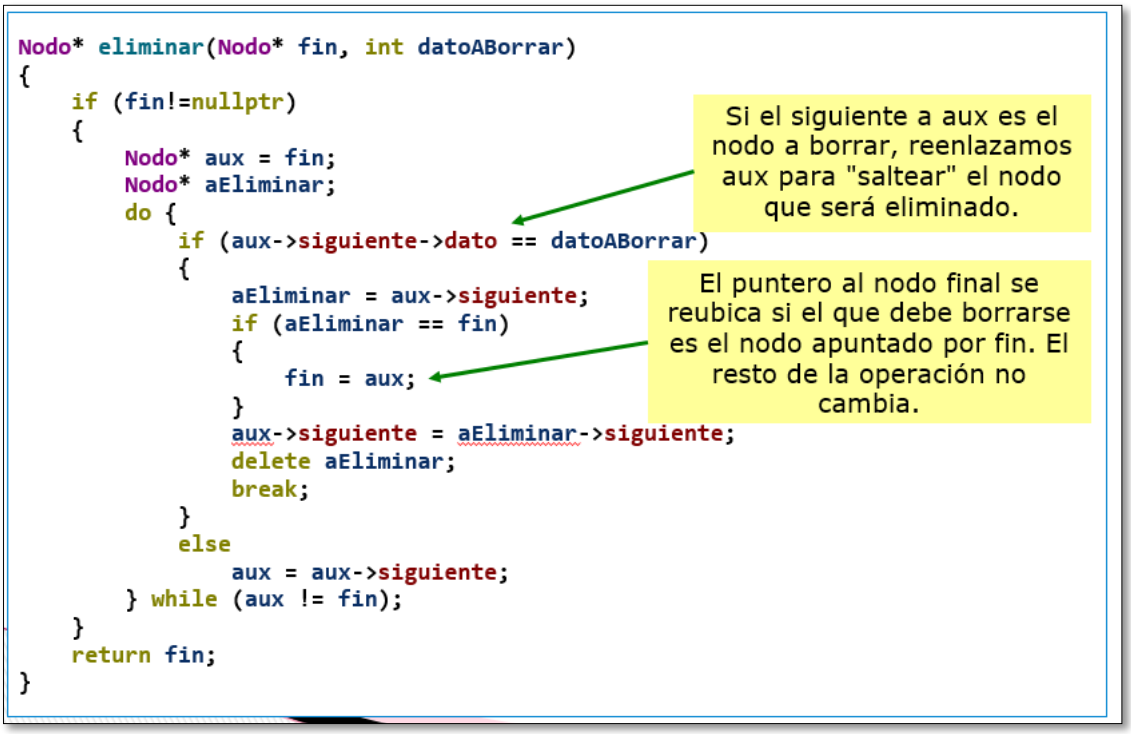


Ilustración 57

9. Resumen de la unidad

Para sintetizar esta unidad temática podemos decir que se enfoca al uso de punteros en C++, tema fundamental para gestionar memoria y estructuras de datos dinámicas en este lenguaje.

Se aborda el uso de punteros en C++, explicando cómo estos permiten gestionar la memoria de forma dinámica al almacenar direcciones de variables. Se detalla el funcionamiento del “Stack” y el “Heap”, donde el primero maneja memoria de forma automática y el segundo requiere gestión manual, lo que puede derivar en errores como fugas de memoria o punteros colgantes.

Además, se describe cómo los punteros pueden pasarse a funciones, permitiendo modificar datos en la memoria original. También se profundiza en las listas enlazadas, una estructura de datos dinámica que utiliza punteros para gestionar elementos, y se explican las operaciones básicas para crear, insertar y eliminar nodos en diferentes tipos de listas, como las circulares.

Finalmente, el texto combina teoría y práctica para ayudar a comprender cómo gestionar la memoria eficientemente y cómo aplicar punteros en estructuras dinámicas en C++.