

# Programación Imperativa

**CUANDO LLEVAS ESTUDIANDO  
TRES DÍAS SEGUIDOS SIN DORMIR**



**Y TU CUERPO CONSISTE EN  
90% CAFEÍNA Y ODIO**



**Atribución - No Comercial (by-nc):** Se permite la generación de obras derivadas siempre que no se haga con fines comerciales. Tampoco se puede utilizar la obra original con fines comerciales. Esta licencia no es una licencia libre.

# UNIDAD 3

# TEMAS

**Estructuras de datos:** en esta unidad nos ocupamos de otra estructura de datos: las **listas enlazadas**.

Para esto, primero veremos una introducción a **punteros**. Posteriormente, utilizaremos punteros para implementar **listas enlazadas simples y listas enlazadas circulares**.

Detalles de implementación y principales algoritmos para manipular listas enlazadas:

✦ *insertar elementos al inicio*, ✦ *insertar elementos al final*, ✦ *insertar elementos ordenados*, ✦ *recorrer la lista*, ✦ *buscar elementos*, ✦ *eliminar elementos*, ✦ *fusionar dos listas*, ✦ *dividir una lista*.

# Punteros

# Cómo se almacenan los datos...

- La memoria está compuesta por unidades básicas llamadas **bits**. Cada bit sólo puede tomar dos posibles valores: 0 ó 1.
- Pero trabajar con bits no es práctico y por eso se agrupan. Cada grupo de 8 bits forma un **byte**.
- Cada byte tiene asignada una dirección, llamada **dirección de memoria**.
- Los microprocesadores trabajan con una unidad básica de información, a la que se denomina “palabra”. Dependiendo del tipo de microprocesador, una palabra puede estar compuesta por uno, dos, cuatro, ocho o dieciséis bytes.

# Cómo se almacenan los datos...

- Muchos de los datos que usamos en nuestros programas no caben en una dirección de memoria. La solución utilizada para manejar datos que ocupen más de un byte es usar posiciones de memoria correlativas. De este modo, la dirección de un dato es la dirección de memoria de la primera posición que contiene ese dato.
- Podemos saber qué tipo de plataforma estamos usando mediante el tamaño en bytes del tipo **int**, usando el operador **sizeof**:

```
cout << "Plataforma de " << 8 * sizeof(int) << " bits" << endl;
```

# Punteros

Un puntero se almacena en la memoria como la dirección del objeto al que apunta.

| Sistema de... | Longitud de una dirección de memoria |
|---------------|--------------------------------------|
| 32 bits       | 4 bytes                              |
| 64 bits       | 8 bytes                              |

Todos los punteros ocupan la misma cantidad de memoria.

# Punteros

Un puntero **es un tipo especial de variable que contiene la dirección de memoria de una variable.**

Almacenada a partir de esa dirección de memoria puede haber cualquier tipo de variable: un **char**, un **int**, un **float**, un arreglo, una struct, otro puntero, etc.

Indicamos ese tipo en la declaración:

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

Al declarar el puntero, ponemos un “\*” entre el **tipo** y el **identificador**, lo que indica que se trata de **un puntero a un valor del tipo dado.**



# Punteros

Ejemplos:

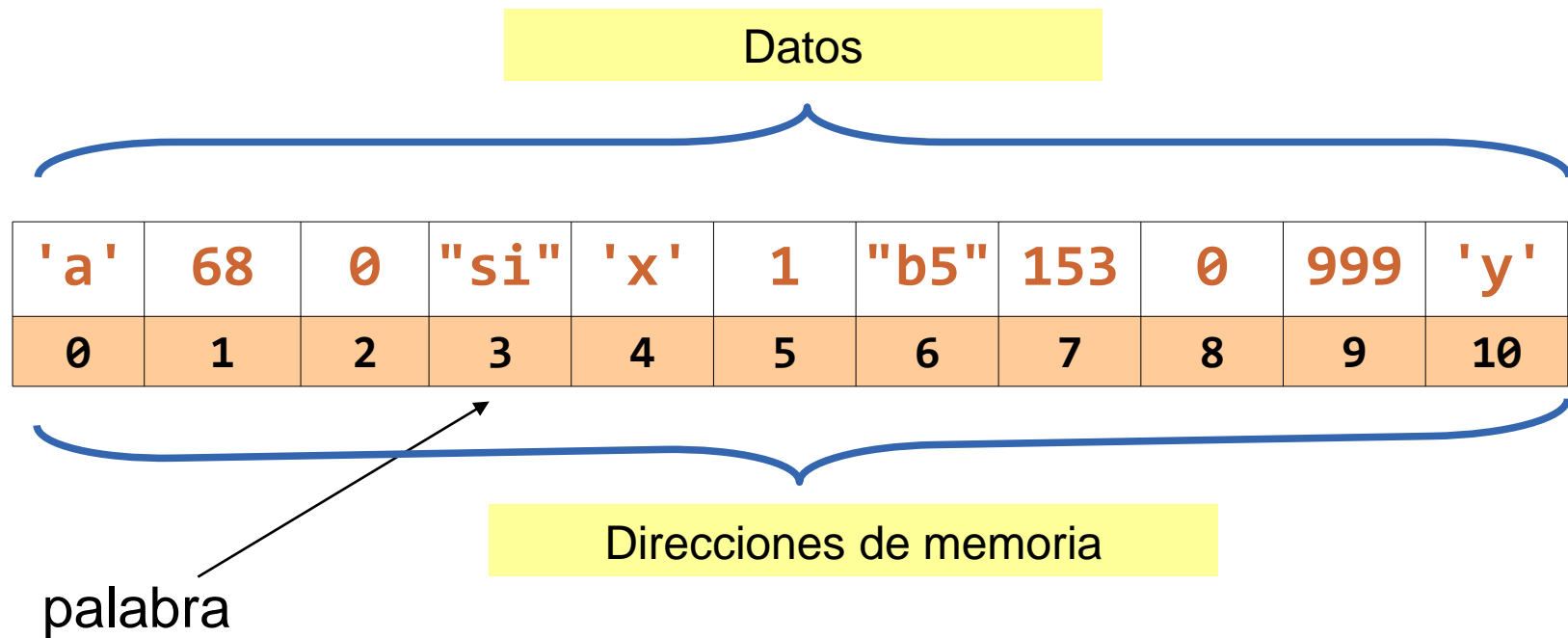
```
int * p_entero;  
char * p_caracter;  
  
struct Punto {  
    float x, y;  
};  
  
Punto * p_punto;
```

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.

# Punteros

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).



# Punteros

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

```
char * x;
```

Sabemos que contendrá la dirección de memoria de un char, pero aún no sabemos qué dirección es.

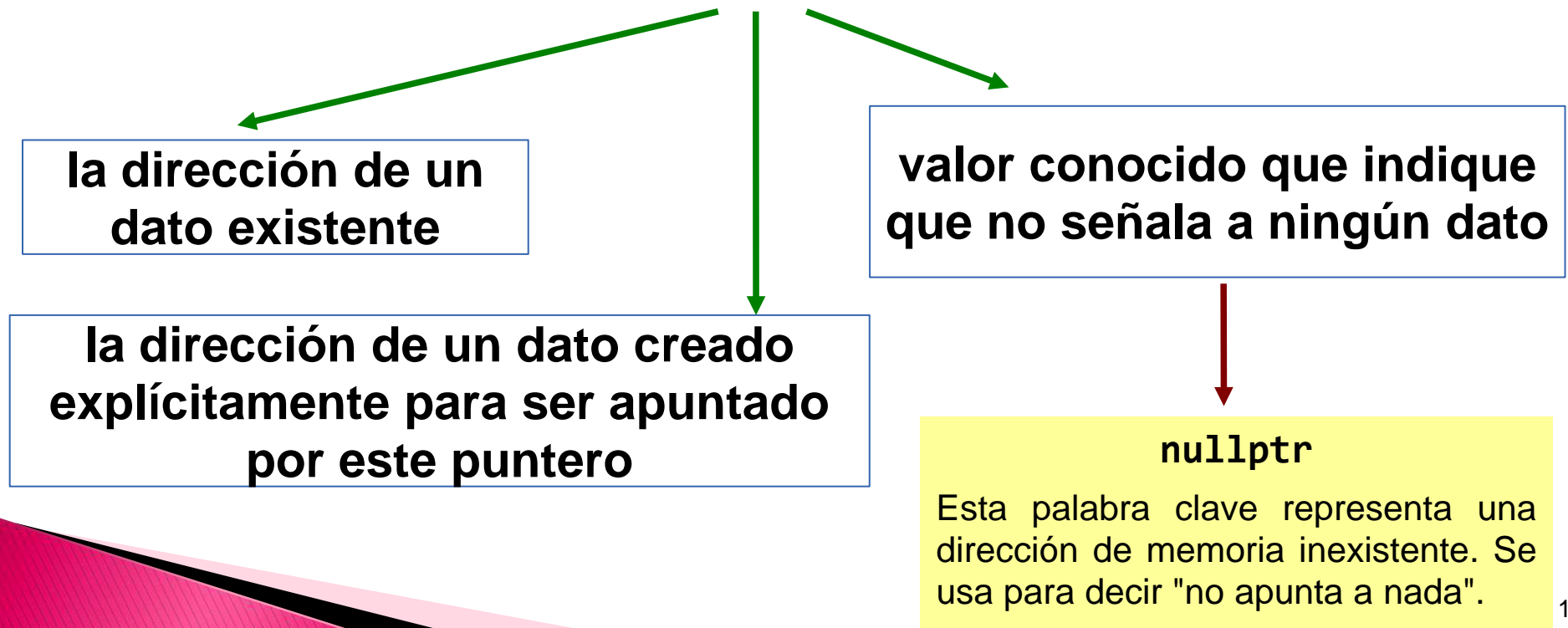
Como cualquier otra variable, mientras no se le asigne un valor, contiene "**datos basura**".

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**.

# Punteros

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).

## Valores posibles para un puntero:



# Punteros. Declaración

Veamos en detalle la definición. Ya vimos la sintaxis:

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

Es importante tener en cuenta (especialmente cuando declaremos varias variables en una sola línea) que `*` forma parte del identificador. Por ejemplo, en esta línea:

```
int * x, y;
```

se está definiendo a **x** como un “puntero a un entero”, mientras que a **y** se lo está definiendo como un entero. Para que ambos sean punteros habrá que hacer:

```
int * x, * y;
```

```
int * x,  
int * y;
```

Esta opción es preferible porque es **más fácil de leer** y no deja dudas sobre la intención.

# Punteros. Declaración

Otro detalle es que el \* puede estar: pegado al tipo de dato al que se apunta, pegado al nombre de la variable, o separado de ambos. Todas las formas son válidas y significan lo mismo.

```
int * numero;
```



Para interpretarlo, puede leerse de derecha a izquierda:  
"numero es un puntero a int"

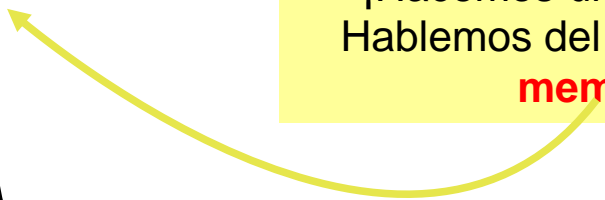
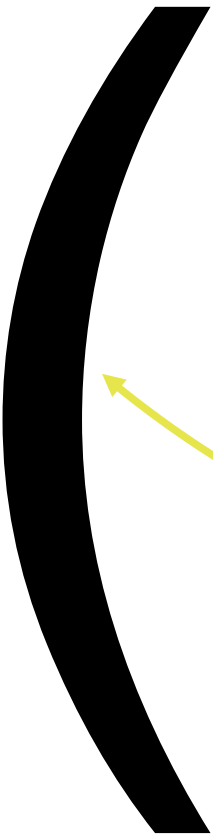
# Punteros. Desreferencia

Para obtener el dato referenciado por el puntero se usa \*, que se lee “el dato apuntado por” (operador de *indirección* o *desreferencia*):

\* puntero

Con otras variables que no son punteros, al poner su nombre se está **referenciando** directamente al dato que contienen: por ejemplo, si la variable **numero** guarda el valor 15, al poner **numero+10** se obtiene 25.

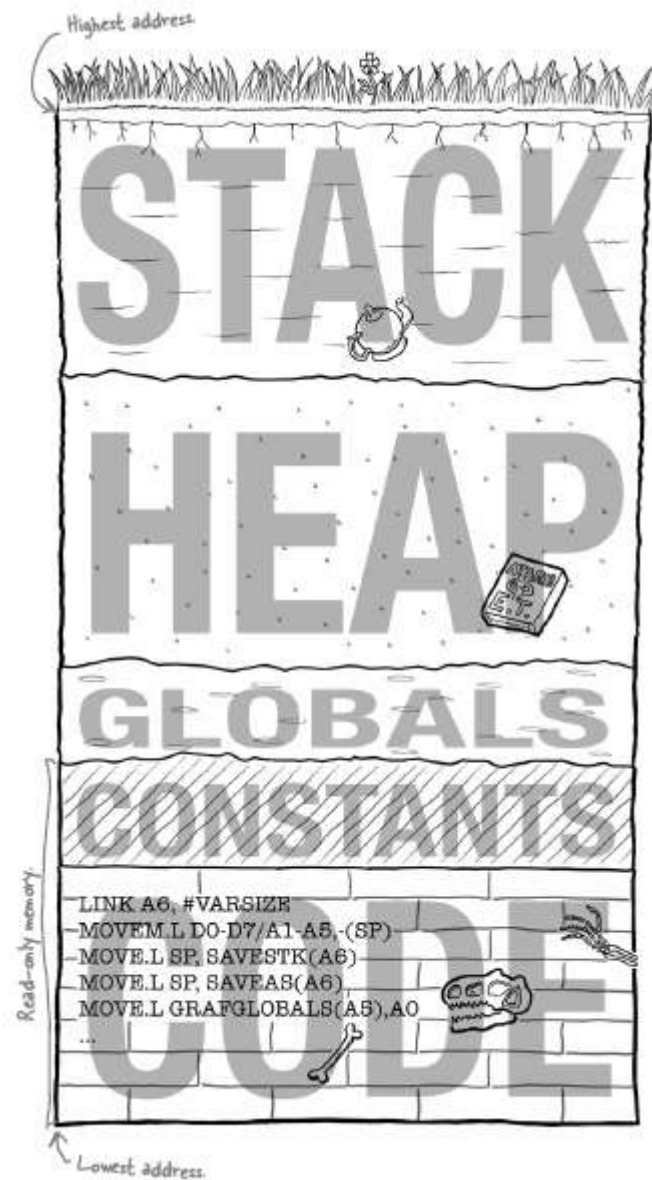
Con los punteros hay un nivel más de **indirección**: al poner su nombre no obtenemos directamente el dato, sino que obtenemos la dirección a donde hay que **ir a buscar el dato**. Para indicar “*ir a buscar el dato a esta dirección*” se usa el operador de indirección \*.



¡Hacemos un **paréntesis**!  
Hablemos del **manejo de la**  
**memoria**



# Modelo de la memoria de un programa



# Stack y heap

Se pueden identificar dos partes (entre otras) dentro de la memoria RAM asignada a nuestro programa en ejecución: "**stack**" ("pila") y "**heap**" (también llamada "montículo").

| identificador | Valor | Dirección de memoria |
|---------------|-------|----------------------|
|               |       |                      |
|               |       |                      |
|               |       |                      |
|               |       |                      |
|               |       |                      |
|               |       |                      |
|               |       |                      |
|               |       |                      |




The diagram illustrates the memory layout. A table with three columns: 'identificador', 'Valor', and 'Dirección de memoria'. The table has 8 rows. The first four rows have yellow 'Valor' cells and are grouped by a bracket on the right labeled 'HEAP'. The next four rows have blue 'Valor' cells and are grouped by a bracket on the right labeled 'STACK'. The 'identificador' and 'Dirección de memoria' columns are empty for all rows.

Todos los tipos de variables que hemos usado hasta ahora se almacenan en la memoria **stack**.

# Stack y heap

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.

```
int a = 16;  
char b = 'L';
```



Cada vez que necesitamos referenciar al dato que guardan estas variables, ponemos el identificador.

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.

# Stack y heap

Cuando se declara una variable, se le reserva un espacio en la memoria “**stack**”, dentro del sector destinado a la función donde la variable fue declarada (como hacíamos hasta ahora):

```
void funcion(int x){  
    bool y = x>18;  
}  
  
int main(){  
    int a = 16;  
    char b = 'L';  
    int * c;  
    funcion(a);  
}
```

| S T A C K     |       |                      |         |
|---------------|-------|----------------------|---------|
| identificador | Valor | Dirección de memoria |         |
| y             | false | 0xFFFFA              | funcion |
| x             | 16    | 0xFFFF9              |         |
| c             | ?     | 0xFFFF               |         |
| b             | 'L'   | 0xFFFFE              | main    |
| a             | 16    | 0xFFFFD              |         |

Si declaramos un puntero de esta forma, también se almacenará en la stack (cuidado: no es lo mismo un *puntero* que el *dato al que apunta*).

# Stack y heap

Pero también es posible almacenar datos en la memoria **heap**. Estos datos **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**.
- ♦ Para desalojar un dato de la memoria heap se usa **delete**.

```
int* c = new int;  
*c = 72;
```

| Nombre | Valor  | Dirección de memoria |
|--------|--------|----------------------|
|        | 72     | 0x50FF               |
|        |        |                      |
|        |        |                      |
|        |        |                      |
| c      | 0x50FF | 0xFFFC               |
|        |        |                      |
|        |        |                      |
|        |        |                      |

HEAP

STACK

# Stack y heap

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. La stack también es llamada "pila" (que es la traducción del inglés "stack").

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.

# Stack y heap

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.

# Stack y heap: 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 ocasionar que el programa termine de manera inesperada por falta de memoria.

```
void funcion(){  
    int* num = new int;  
}
```

La variable num es un puntero que reside en la stack, y apunta a un entero que reside en la heap.  
Al finalizar la función, num se desaloja automáticamente de la stack, pero el entero apuntado **queda (inaccesible) en la heap.**



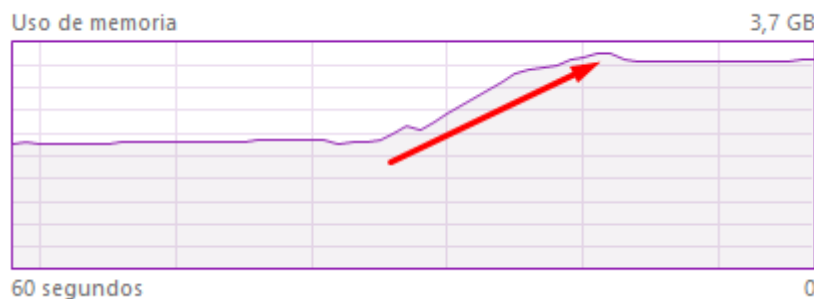
# Stack y heap: fuga de memoria

Concretamente, la **fuga de memoria** se produce cuando un programa ocupa memoria que no está utilizando.

Un ejemplo trivial pero extremo puede verse en el siguiente programa, que sólo aloca memoria (sin almacenar la dirección ni hacer nada con ella) dentro de un bucle infinito

*(cuidado: si ejecutamos este programa es posible que la máquina deje de responder por falta de memoria):*

```
int main() {  
    while (true)  
        new int;  
}
```




Si observamos el monitor de RAM durante la ejecución, veremos cómo el uso de memoria se dispara

# Stack y heap: "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".

```
int* puntero1 = new int;  
int* puntero2 = puntero1;  
delete puntero1;
```



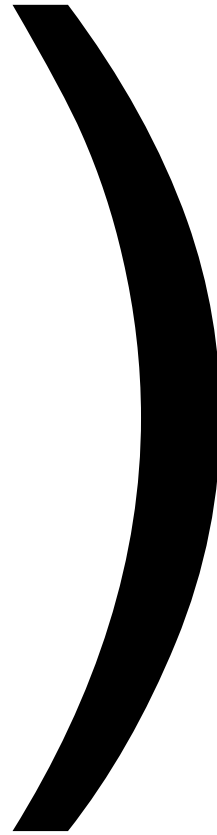
- La instrucción **new int** reserva, en la heap, un espacio suficiente para almacenar un entero, y su dirección de memoria queda guardada en la variable puntero1 (que reside en la stack).
- La variable puntero2 (también en la stack) contendrá la misma dirección que puntero1.
- Pero, con el uso de **delete**, se elimina el espacio de memoria heap que es apuntado por puntero1 –que, casualmente, es el mismo al que apunta puntero2 –.
- Así, puntero2 es un **dangling pointer** que referencia a una dirección de memoria que ha quedado *desalojada*.

# Stack y heap: comparación

Resumiendo...

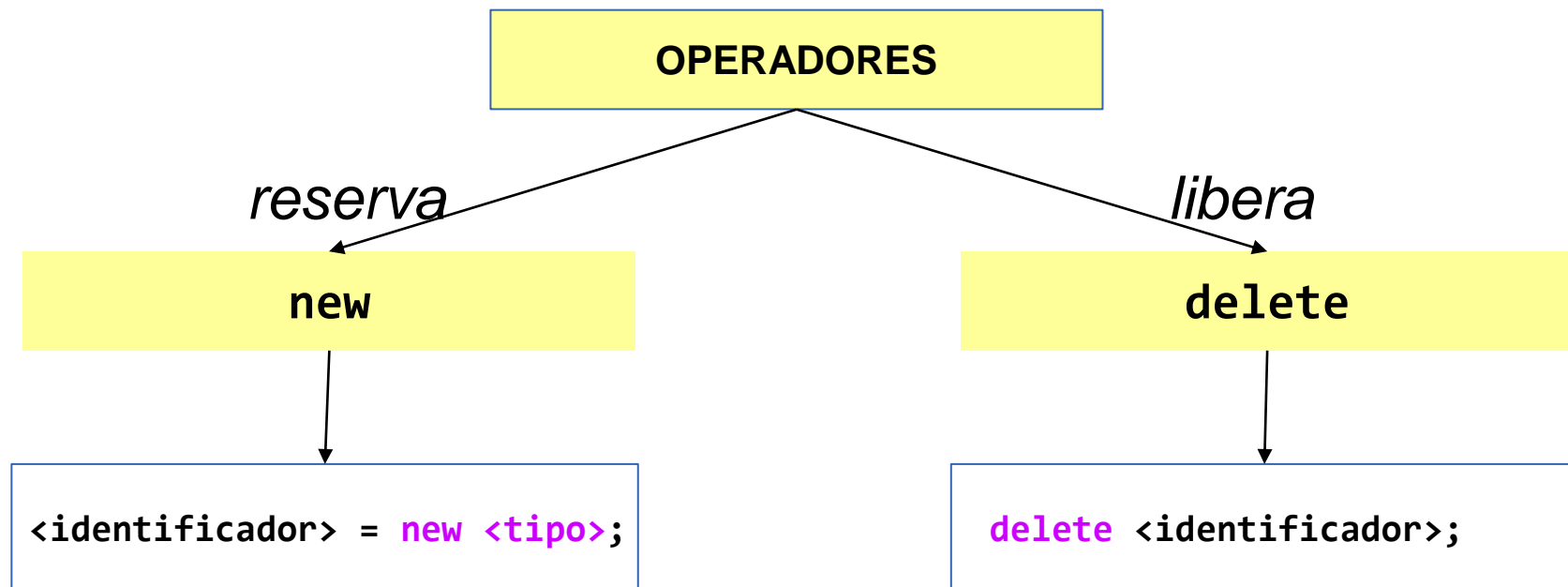
| STACK   | HEAP  |
|---|---|
| Las variables se desalojan automáticamente.   | Manejada manualmente por el programador (new / delete).       |
| La alocaión se realiza más rápidamente.   | Alocaión más lenta que en la stack.                           |
| Los datos almacenados pueden accederse con nombres 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). |

Algunos lenguajes (C++ no lo trae activado por defecto) incluyen mecanismos para evitar las fugas de memoria, llamados “recolectores de basura” (“garbage collectors”).



# Punteros. Objetos dinámicos

C++ dispone de dos **operadores** para manejar la memoria dinámica:

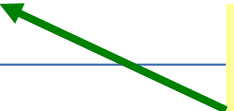


# Punteros. Asignación

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.

```
int * a = new int;
```



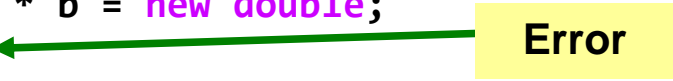
La variable **a** ahora contiene la dirección de memoria que le devolvió el operador **new**, donde podrá guardarse un dato de tipo **int**.

# Punteros. Asignación

Al ser C++ un lenguaje *fuertemente tipado*, nos impide almacenar en una variable un dato que no corresponda a su tipo (a menos que se haga una conversión de tipos).

Los punteros no son la excepción: sólo podemos guardar en un **puntero** una **dirección de memoria**, y esa dirección sólo puede contener un dato **del tipo indicado** por el puntero.

```
int * a = new int;  
double * b = new double;  
a = b;
```



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*. Si intentamos guardar una dirección destinada a otro tipo de dato, obtendremos un error.

# Punteros. Asignación

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).

```
int * a = new int;  
.  
. //más instrucciones  
.  
a = new int;
```

Siempre que tengamos cuidado de no generar fugas de memoria, podemos reutilizar el puntero **a** para guardar en él una dirección de memoria diferente.



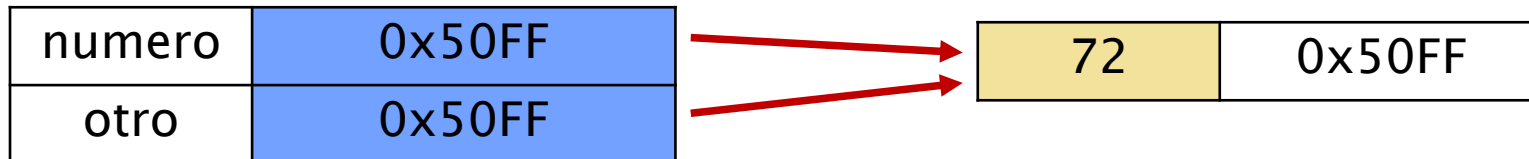
# Punteros. Asignación

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.

```
int * numero = new int;  
int * otro = numero;  
* otro = 72;
```


En este caso, si modificamos el dato usando un puntero, al intentar acceder con el otro puntero veremos el dato modificado, porque tenemos dos punteros pero un solo dato.



# Punteros. Asignación


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...

```
int * puntero = new int;  
int a = 10;  
*puntero = a;
```



Le 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.

```
*puntero = 6;  
cout << *puntero << endl;  
cout << a << endl;
```



Se imprime:  
6  
10

# Punteros

Entonces tenemos que recordar:

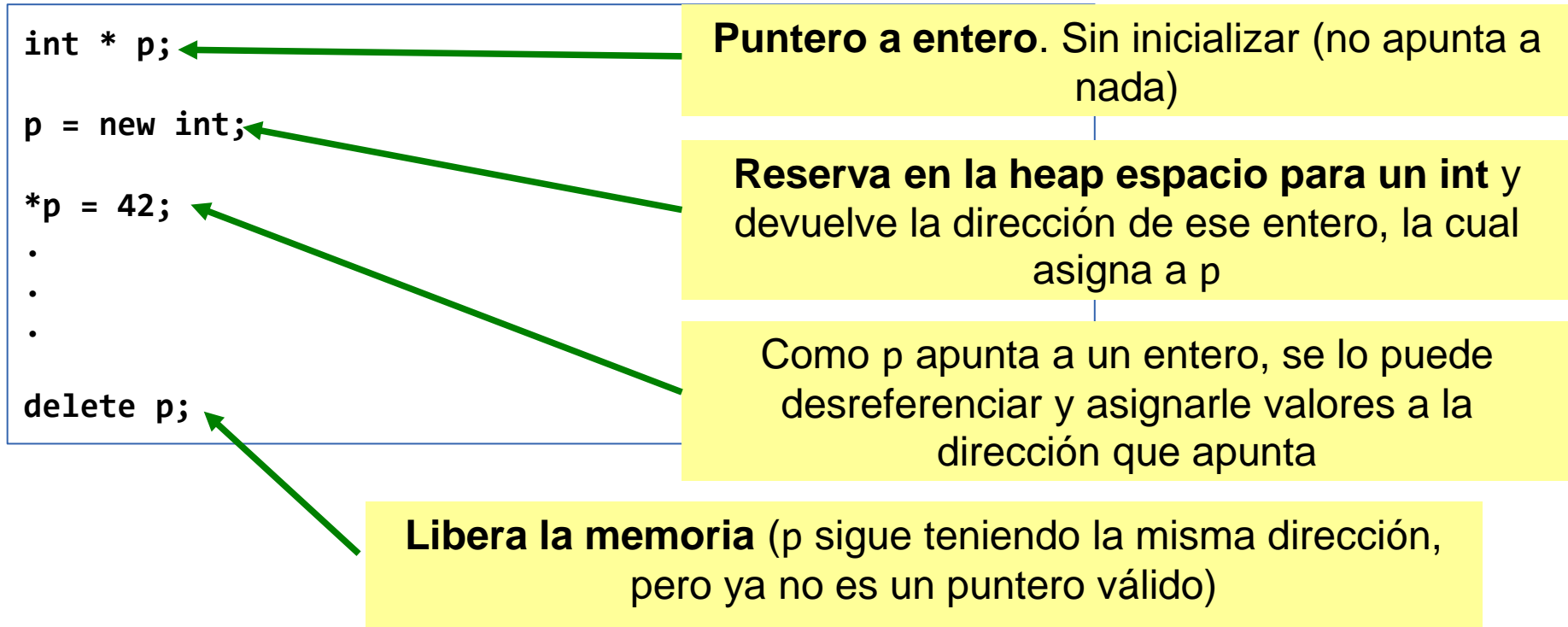
- Siempre que aparezca un asterisco (\*) en una definición de variable, ésta es una variable puntero.
- Siempre que aparezca un asterisco (\*) delante de una variable puntero, se accede a la variable referenciada por el puntero.
- C++ requiere que las variables puntero direccionen realmente variables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

```
cout << *puntero;
```

El operador \* está indicando  
*"ir a buscar el dato guardado  
en esta dirección".*

# Punteros

## Asignación dinámica de memoria



Toda la memoria que se reserve durante la ejecución del programa debería liberarse, a lo sumo, cuando el programa finalice (en general, antes).

# Punteros

**Puntero a entero.** Sin inicializar (no apunta a nada)

```
int * p;
```

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

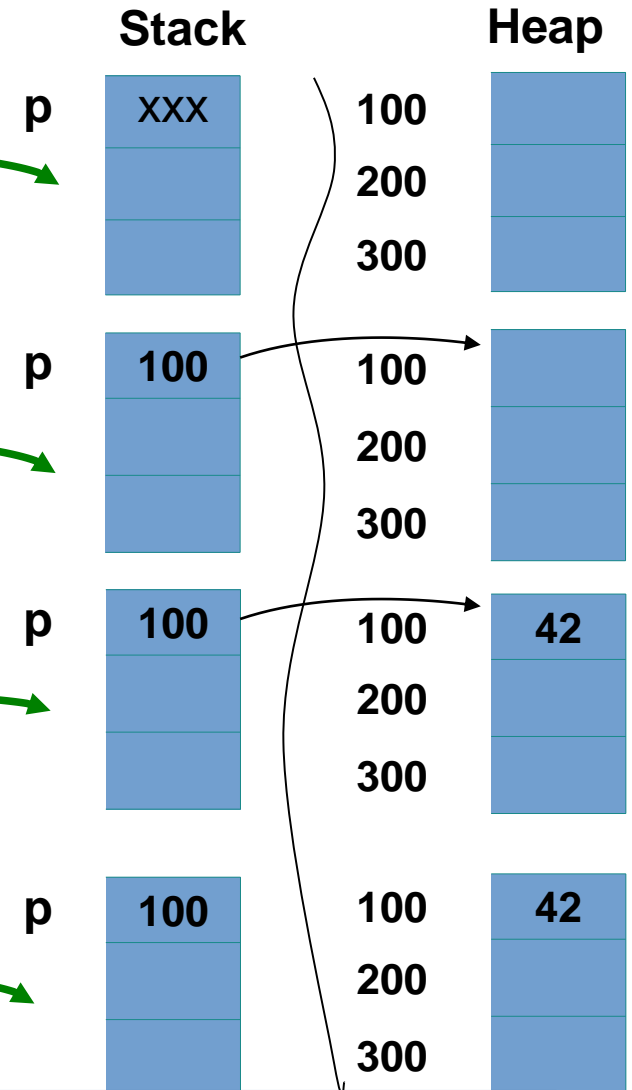
```
p = new int;
```

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

```
*p = 42;
```

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

```
delete p;
```



p sigue existiendo en **memoria estática (stack)** durante la ejecución de su ámbito, mientras que la **memoria dinámica (heap)** fue liberada

# Puntero a struct

**new** puede usarse para reservar memoria para cualquier tipo de C++ (incorporado o definido por el usuario):

```
struct Persona {  
    string nombre;  
    int edad;  
};  
  
int main(){  
    Persona *p = new Persona;  
}
```

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

```
(*p).nombre = "Ceci";
```

Los **()** son **necesarios**, porque “.” tiene **mayor precedencia** que “\*”

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

# Puntero a struct

## Operador ->

Simplifica el acceso a los campos de una estructura

```
(*p).nombre = "Ceci";
```

Es equivalente a:

```
p -> nombre = "Ceci";
```

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

# 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.

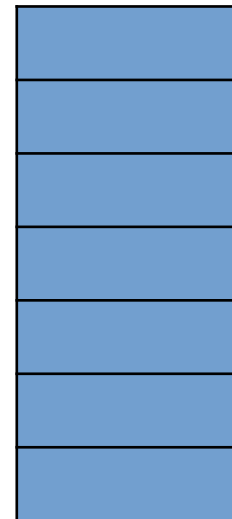
```
void funcion (int* x) {  
    *x = 10;  
}
```

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

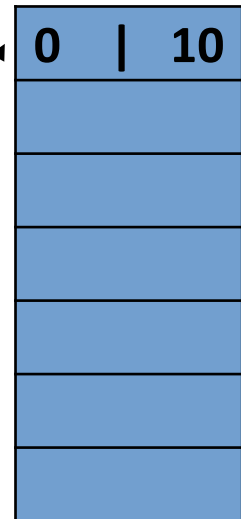
Se copia la  
dirección

x

stack



heap



p




# Punteros y referencias

Es común confundir las referencias con los punteros. Pero la diferencia sustancial es que una referencia es un "alias" para otra variable, y por ende es constante, en el sentido de que siempre va a referenciar a la misma variable.


Un puntero, por el contrario, puede cambiar la dirección a la que apunta.

```
void incremento(int & z) {  
    z++;  
}  
  
int main() {  
    int x = 15;  
    incremento(x);  
}
```



La variable **z** es un alias para la variable **x**, y esto no puede cambiar (no hay forma de indicar "ahora va a ser un alias de otra variable")

```
int * a = new int;  
//más instrucciones  
a = new int;
```



La variable **a** apunta a la dirección de un int, luego hay más instrucciones y finalmente **a** guarda una nueva dirección (en *más instrucciones* debería suceder algo que evite una fuga de memoria, por ejemplo liberar lo apuntado inicialmente por **a**).

# Estructura de datos: Lista enlazada simple

# Listas: Concepto

## El concepto de lista es bastante intuitivo.

Encontramos varios ejemplos en la vida cotidiana:

[illegible]

|    | A                | B                                  |
|----|------------------|------------------------------------|
| 1  | Lista de Alumnos |                                    |
| 2  |                  |                                    |
| 3  | Usuario          | Nombre del alumno                  |
| 4  | mes alu aba mar  | Abarca Murga Marcelino             |
| 5  | mes alu agu mig  | Aguiayo Bedolla Miguel Angel       |
| 6  | mes alu ale rod  | Alejandro Martinez Rodrigo         |
| 7  | CTC_mipequeñoalu | alumno mi pequeño                  |
| 8  | mes alu alv ang  | Alvarez Acosta Angel               |
| 9  | mes alu are san  | Arevalo Hernandez Sandra Patricia  |
| 10 | mes alu arg min  | Argon Herrera Minerva Alejandra    |
| 11 | mes alu arr mar  | Arreola Silva Maria Josefina       |
| 12 | mes alu art jai  | Arteaga Marquez Jaime              |
| 13 | mes alu bec luz  | Becerril Reynoso Luis Carlos       |
| 14 | mes alu can luz  | Cano Pardiñas Luz Del Carmen       |
| 15 | mes alu car fab  | Cardenas Espinosa Fabiola Whyteika |

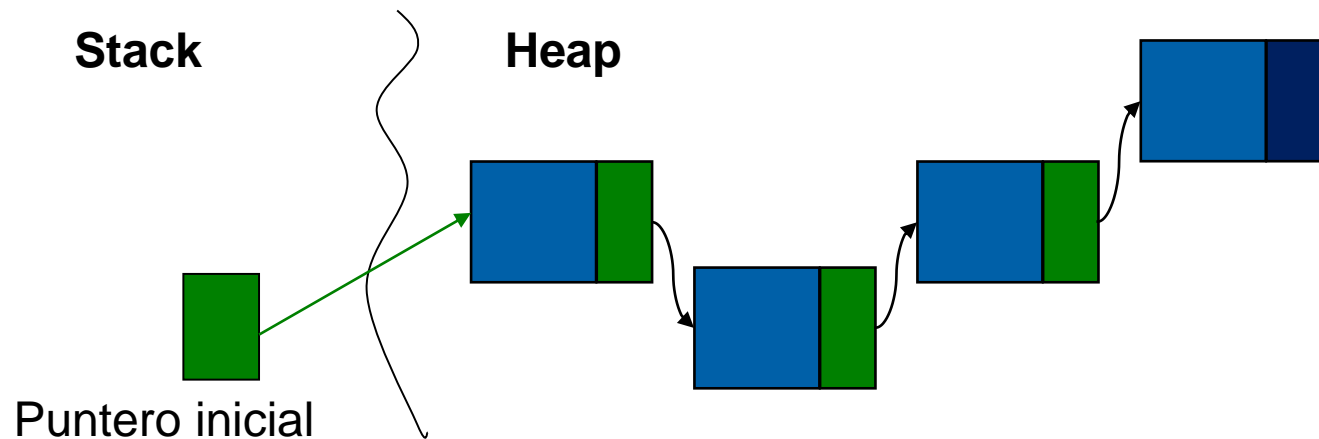


# Estructura de datos: Lista

**Lista** → estructura dinámica que se construye con **nodos**.

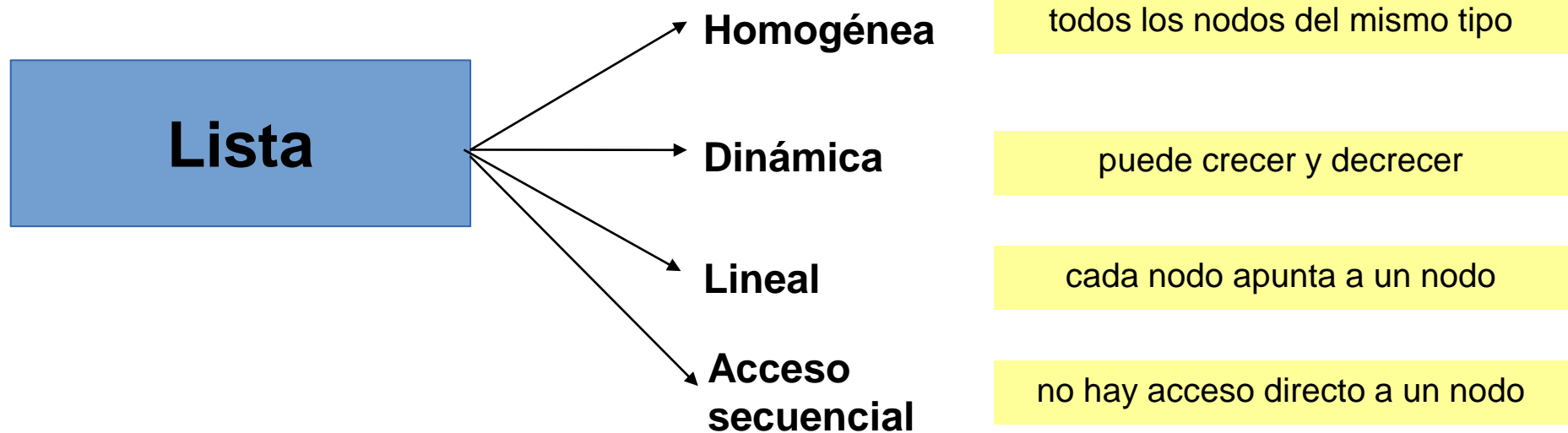
**Nodos** → estructura de dos campos: *dato* y *enlace*

Una lista enlazada es una colección de nodos que se ordena según su posición, tal que cada uno de ellos es accedido mediante el campo de enlace del nodo anterior.



# Lista: características del tipo

Los elementos que la componen **no ocupan posiciones secuenciales o contiguas de memoria**. Pueden aparecer dispersos en la memoria, pero se mantiene un orden lógico interno, dado por los enlaces.



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

# Listas: definición de nodo en C++

Definimos un tipo al que llamamos "Nodo" (podría llamarse de cualquier forma, pero Nodo es lo más descriptivo). Este nodo contendrá todos los datos necesarios, y un puntero al nodo siguiente.

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

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

Como ya hemos visto, tendremos un puntero inicial de la lista apuntando a un dato de tipo **Nodo** en memoria heap. Luego, dentro de cada nodo, un **puntero** indicará la dirección del **nodo siguiente**. Es decir: el puntero inicial está en la stack, pero todos los nodos y sus enlaces estarán en heap.



# Listas: puntero inicial

Toda lista tiene un **puntero inicial** que **no debe perderse** 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.

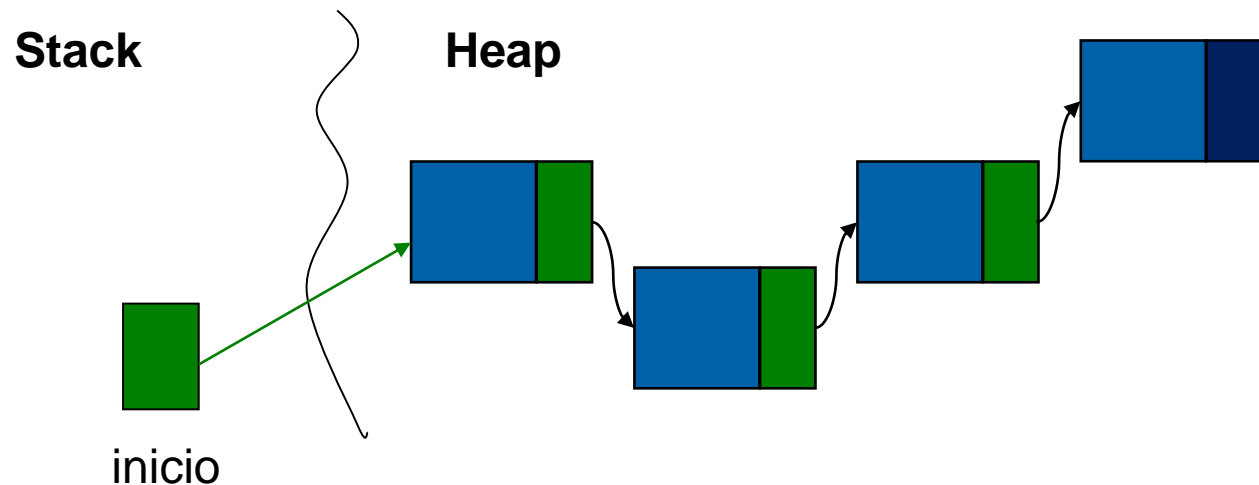
¿Cómo podría "perderse" el puntero inicial? Por ejemplo, si al puntero inicial se le asigna una nueva dirección de memoria de forma inadvertida, dejando en situación de "fuga de memoria" al nodo inicial de la lista.

Si no podemos acceder al primer nodo y no tenemos otro puntero que apunte a él, hemos perdido la lista y ya no es posible acceder a ella.

```
Nodo * inicio;
```

# Listas: nodo final

Se puede identificar al último nodo de la lista porque es el único que no apunta a ningún nodo. El valor de su puntero al siguiente es **nullptr**.



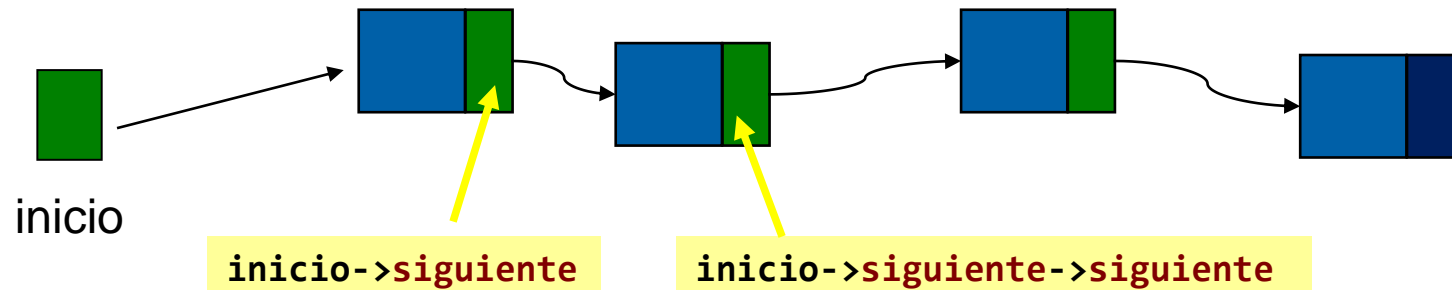


# Listas: puntero al siguiente

Dado un puntero que apunta a un nodo de la lista, es posible obtener la dirección del nodo siguiente:

```
inicio->siguiente
```

Al desreferenciar al puntero inicio se obtiene el primer nodo de la lista y podemos acceder a sus campos. El operador `->` desreferencia un puntero y accede a un campo. El campo siguiente del nodo apuntado por inicio es un *puntero a Nodo*.

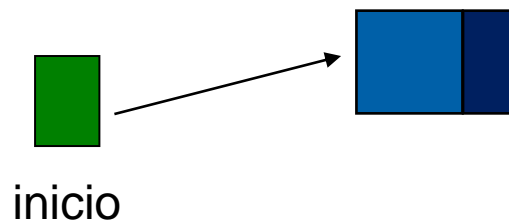


Si desreferenciamos el puntero `inicio->siguiente`, obtendremos el segundo nodo, el cual, en su campo siguiente, tendrá la dirección del tercer nodo. Y así hasta el final de la lista.

# Listas: desreferenciar punteros

Si se intentara desreferenciar un puntero que no apunta a nada, se obtendría un error.

Por ejemplo: dada una lista con un único nodo, apuntado por el puntero inicio, el puntero al siguiente del nodo contiene nullptr.



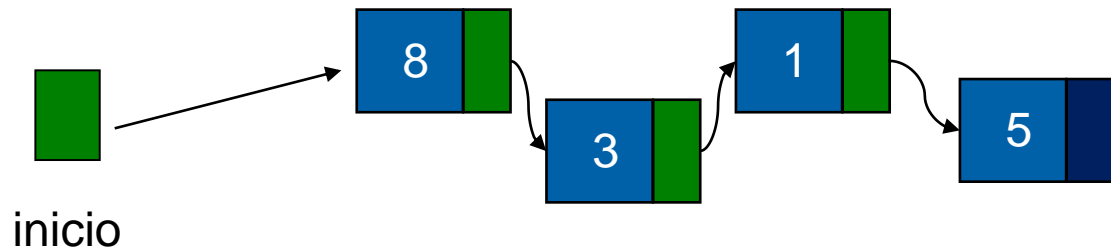
Entonces, si intentamos acceder a un campo del nodo siguiente al inicial, obtendremos un error. Esto es porque no hay nodo siguiente al inicial y no es posible desreferenciar nullptr.

```
inicio->siguiente->dato ← Error
```

Este tipo de errores son comunes al iterar por una lista.

# Listas: desreferenciar punteros

Es importante la diferencia entre un puntero y el dato al que apunta. Cuando se desreferencia un puntero, se está accediendo al dato al que apunta (si es que apunta a algo).



Sabemos que el operador **->** desreferencia un puntero y accede al campo de un struct (ambas cosas al mismo tiempo).

`inicio->siguiente`

Obtiene la **dirección de memoria** donde se encuentra el segundo nodo, ya que desreferencia al puntero inicio y accede al campo siguiente, que es un puntero a Nodo.

`inicio->siguiente->dato`

Obtiene el **dato** almacenado en el segundo nodo, ya que desreferencia al puntero inicio y luego desreferencia al puntero siguiente, obteniendo el struct que compone al segundo nodo, para luego acceder a un campo concreto.

# Operaciones

## Algunas operaciones posibles:

- ▶ Crear una lista vacía
- ▶ Insertar un elemento al principio de una lista
- ▶ Recorrer una lista
- ▶ Buscar un elemento en una lista.
- ▶ Insertar un elemento al final de una lista
- ▶ Insertar un elemento en una lista ordenada
- ▶ Eliminar un elemento de la lista
- ▶ Dividir una lista en dos o más.
- ▶ Combinar dos listas ordenadas, formando una nueva lista (“merge”)

# Operaciones

No hay una única manera de implementar estos algoritmos. Veremos algunas opciones, pero es posible realizar estas operaciones de más de una forma.

# Crear una lista vacía

Para iniciar una lista se declara un puntero del tipo de nodo que hayamos definido y se le asigna **nullptr** (que significa que no apunta a nada aún). Si no se le asignara nullptr, el puntero –como cualquier otra variable– podría contener residuos de memoria y apuntar a cualquier lado.

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

```
Nodo* inicio = nullptr;
```

Esta variable se declarará donde sea necesario (por ejemplo, en la función main).

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). Los nodos residirán en la heap.

# Crear un nuevo nodo

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.

En el código mostrado a continuación se crea un nuevo nodo, se almacena en él un determinado dato a insertar y se inicializa el puntero siguiente en nullptr (aunque es posible que este puntero sea modificado cuando se inserte el nodo en la lista).

```
Nodo * nuevo;  
nuevo = new Nodo;  
nuevo->dato = datoAInsertar;  
nuevo->siguiente = nullptr;
```

# Insertar un valor al principio de la 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.

```
nuevo->siguiente = inicio;  
inicio = nuevo;
```




# Insertar un valor al principio de la lista

Para una mejor modularización, podríamos tener una función que, dado el puntero inicial de la lista y el nuevo nodo, sólo se encargue de insertar ese nodo en la lista.

De esta manera, se desacoplan las diferentes tareas: la creación del nodo y asignación de los datos correspondientes a él constituye una tarea, y la inserción de ese nodo en la lista constituye otra. Esto nos sirve para que esta función de inserción sea más reutilizable.

```
Nodo* insertar_principio(Nodo* inicio, Nodo* nuevo)
{
    nuevo->siguiente = inicio;
    return nuevo;
}
```



La función retorna el puntero al nodo inicial (que, por haber insertado al principio, ahora es el nuevo nodo)

# Insertar un valor al principio de la lista

Supongamos que el usuario ingresa números y creamos una lista con cada uno de ellos. Podríamos hacer algo como lo que sigue:

```
Nodo* insertar_principio(Nodo* inicio, Nodo* nuevo)
{
    nuevo->siguiente = inicio;
    return nuevo;
}

Nodo* carga_de_datos(Nodo* inicio)
{
    int numero;
    Nodo* nuevo;
    cout << "Ingresar un número (0 para cortar): ";
    cin >> numero;
    while (numero != 0) {
        nuevo = new Nodo;
        nuevo->dato = numero;
        nuevo->siguiente = nullptr;
        inicio = insertar_principio(inicio, nuevo);
        cout << "Ingresar un número (0 para cortar): ";
        cin >> numero;
    }
    return inicio;
}
```

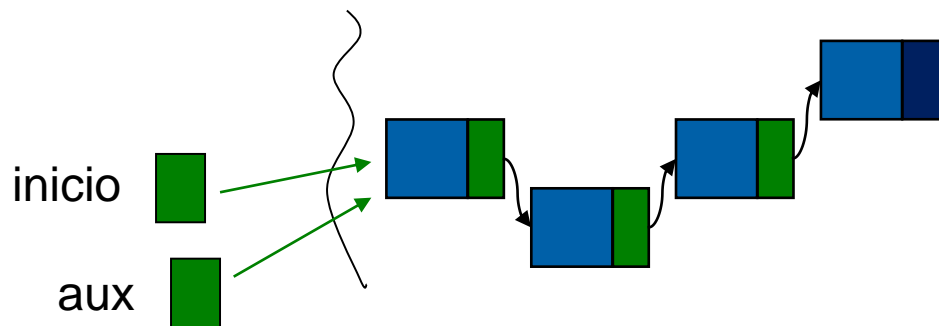
Por cada nuevo número, creamos un nodo. Ponemos como siguiente a nullptr porque podría ser el primer nodo insertado y quedar al final (la operación de crear un nuevo nodo y asignarle los valores puede estar en una función aparte, para una mejor modularización).

# 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).

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**).

```
for (Nodo* aux = inicio; aux != nullptr; aux = aux->siguiente)
    cout << aux->dato << endl;
```



Se logra avanzar por la lista porque, luego de cada iteración, el puntero aux (que tiene la dirección de un nodo) pasa a almacenar la dirección de su nodo siguiente

# Buscar un elemento en una lista

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

En este ejemplo una función recibe el puntero inicial de la lista y el dato a buscar, y recorre la lista hasta encontrar un nodo que contenga a ese dato.

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

Retorna true o false de acuerdo a si encontró al dato o no.

# Buscar un elemento en una lista

También podría retornarse 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.

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

En este ejemplo se recorre la lista con un bucle while. Es una alternativa al recorrido con for.

Esta 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.

# Buscar un elemento: circuito corto

Es interesante ver que, en la búsqueda del nodo a eliminar, la condición del while está formada por dos condiciones, y que el orden en que están expresadas es importante:

```
while (aux != nullptr && aux->dato != datoBuscado)
{
    aux = aux->siguiente;
}
```

La condición es doble porque se debe analizar ❶ si **aux apunta a un nodo** y ❷ si **el campo dato de ese nodo** contiene el dato buscado. El problema es que, si aux no apunta a un nodo válido, se obtendría un error al intentar acceder a su dato mediante `aux->dato`. La solución es evaluar antes si es posible desreferenciar aux: gracias a la semántica de **circuito corto**, la segunda condición sólo se evalúa si la primera es true, ya que un **and** requiere que ambas sean true para resultar *verdadero*.

# Insertar un valor al final de una lista

Supongamos que queremos **agregar** un nuevo elemento al final de una lista, de manera que se mantenga el orden en que los datos fueron ingresados (efecto **cola**). Se deben hacer 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

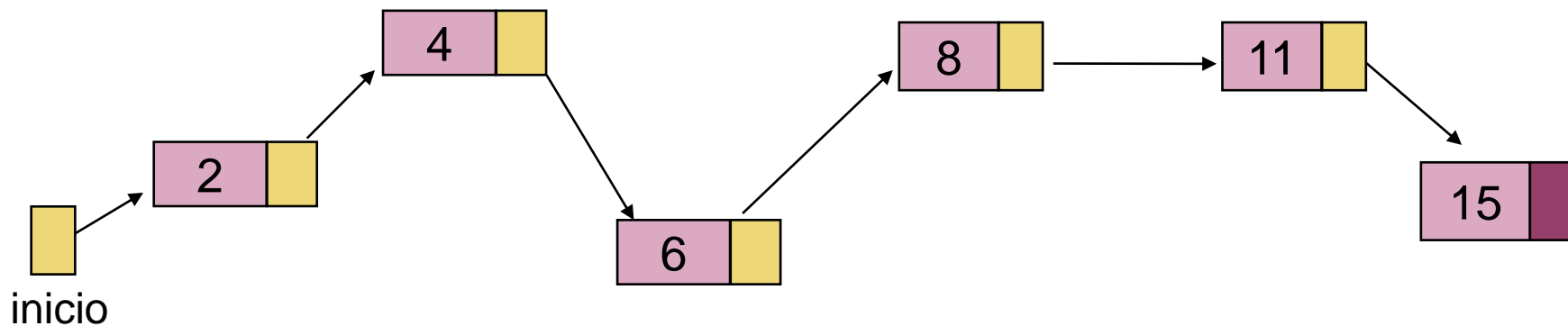
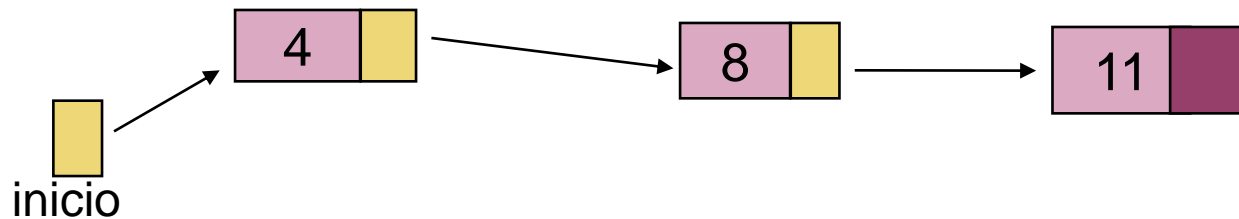
Usamos un puntero auxiliar para recorrer, para evitar perder la dirección del inicio de la lista.

Como ya hemos visto en la inserción al principio, dejaremos que esta función sólo se encargue del algoritmo de inserción, desligándola de la tarea de crear el nuevo nodo.



# Insertar un nuevo valor en una lista ordenada

Supongamos que tenemos la siguiente lista de números enteros, ordenada en forma creciente y queremos insertar los valores 2, 6 y 15:





# Pasos para insertar ordenado (menor a mayor)

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

- Si la lista está vacía: el nodo a insertar será el primero (y también último).
- Si no está vacía:
  - 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.
  - 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.
  - 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.

**Algunos casos parecen ser similares...**



**¿Podemos unir casos?**

# Pasos para insertar ordenado (menor a mayor)

En pseudocódigo podemos definir:

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

# Insertar un nuevo valor en una lista ordenada

```
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.

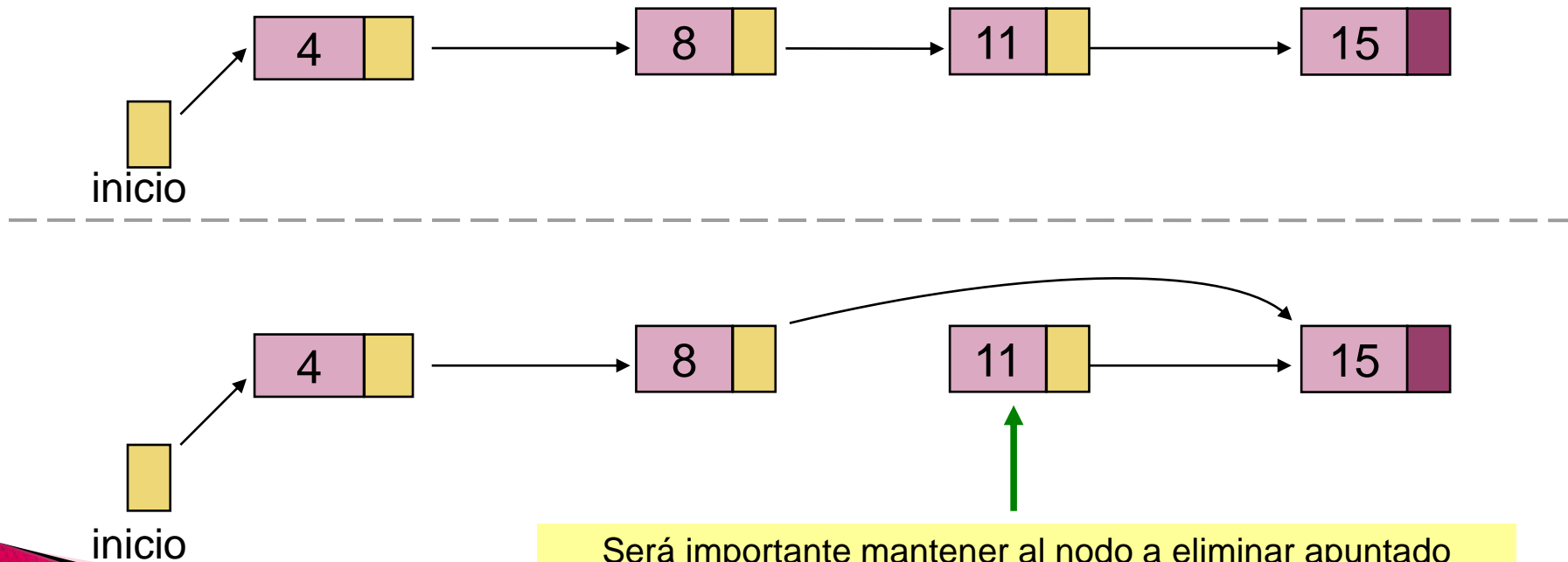
Si no se insertó al principio, buscamos dónde debe insertarse.

- 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*.

Cuando la lista está vacía no es necesario hacer `nuevo->siguiente=inicio;` pero hemos unido casos para simplificar el algoritmo.

# 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:



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.

# Eliminar un elemento de la lista




Para eliminar un elemento de una lista enlazada es 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**.

# Eliminar un elemento de la lista

```
Nodo* eliminar(Nodo* inicio, int datoABorrar) {  
    if (inicio != nullptr) {  Sólo proceder si la lista tiene nodos  
        Nodo* aux = inicio;  
        if (inicio->dato == datoABorrar) {  Se debe eliminar el primer nodo  
            inicio = inicio->siguiente;  
            delete aux;  
        }  
        else {  Si no se eliminó el primer nodo, buscamos a partir del segundo  
            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;  
}
```

# Eliminar ocurrencias de un elemento

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:

Será necesario contemplar algunos **nuevos casos posibles**. Ya no basta con contemplar 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.



# Eliminar ocurrencias de un elemento

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

Ahora podría suceder que el nodo inicial deba ser eliminado múltiples veces

Si no se eliminó el primero, buscamos desde el segundo

Si se ingresa en este if, el nodo **siguiente a aux** debe ser eliminado

El campo siguiente del nodo apuntado por aux ahora debe contener la dirección del **siguiente al que debe eliminarse**



# Dividir una lista en dos

## Idea:

Dada una lista, obtener nuevas listas reutilizando sus nodos.

## ¿Cómo hacerlo?

Se tiene 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).

# Dividir una lista en dos

Si hemos 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.

```
void dividir_lista(Nodo* & inicio, Nodo* & pares, Nodo* & impares) {  
    Nodo* anterior;  
    while (inicio != nullptr) {  
        anterior = inicio;  
        inicio = inicio->siguiente;  
        anterior->siguiente = nullptr;  
        if (anterior->dato % 2 == 0) {  
            pares = insertar_final(pares, anterior);  
        }  
        else {  
            impares = insertar_final(impares, anterior);  
        }  
    }  
}
```

El puntero anterior apunta a un nodo e inicio pasa a apuntar al siguiente, para poder modificar al anterior sin "romper" el enlace antes de avanzar.

El nodo apuntado por anterior ahora es equivalente a un nodo nuevo que debemos insertar en una lista (pares o impares).

# Unir dos listas ordenadas ("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, recorreremos 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.

# Unir dos listas ordenadas ("merge")

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`.

# Pasos para unir dos listas ordenadas

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  $\leq$  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
```

Como en la mayoría de los casos, no existe un único algoritmo para unificar dos listas.

# Unir dos listas ordenadas

```
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);  
    }
```

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

```
    while (A != nullptr) {  
        anterior = A;  
        A = A->siguiente;  
        anterior->siguiente = nullptr;  
        C = insertar_final(C, anterior);  
    }
```

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

```
    while (B != nullptr) {  
        anterior = B;  
        B = B->siguiente;  
        anterior->siguiente = nullptr;  
        C = insertar_final(C, anterior);  
    }
```

```
    return C;
```

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...

# Unir dos listas ordenadas

```
void reininsertarNodo(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);  
}
```

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.

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

# 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.

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

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



# Listas de struct

Aunque podría representarse a cada nodo con los datos necesarios como campos (sin crear otra struct), esta forma complicaría algunos algoritmos:

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

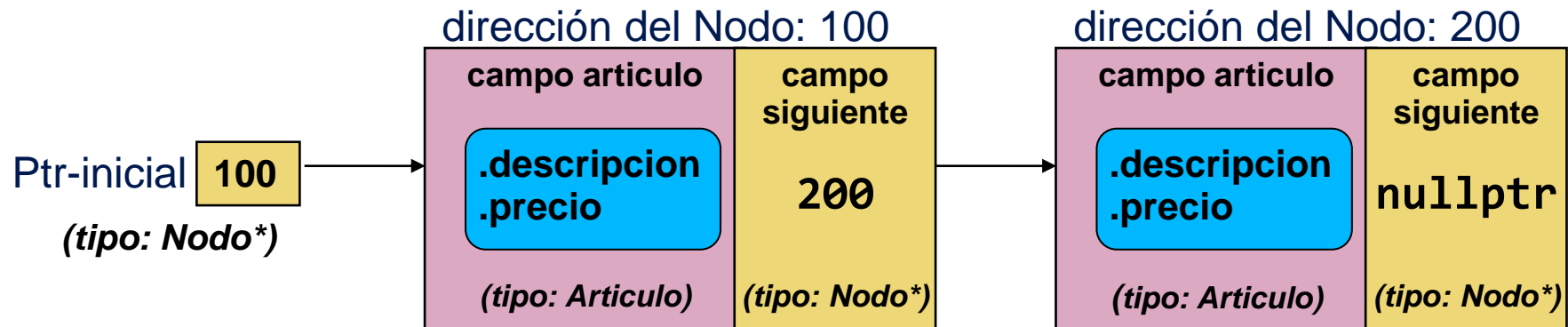
Supongamos el caso en que es necesario copiar los elementos de la lista a un arreglo, colocando los datos de cada artículo en cada elemento del arreglo.

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.

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

# Listas de struct

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.



El ejemplo grafica 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.

# Listas de struct

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.

# Listas de struct

```
Nodo* insertar_principio(Nodo* inicio, Nodo* nuevo)
{
    nuevo->siguiente = inicio;
    return nuevo;
}

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

Un error común es hacer `new Nodo` en este punto. Pero aún no se verificó la condición del `while` y no se sabe si deberá crearse un nodo.

Sólo corresponde reservar espacio en memoria dentro del bucle, que es cuando se puede afirmar que se insertará un nuevo nodo en la lista.

Para cambiar la forma de inserción, basta con llamar a otra función de inserción.

# Listas de struct

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.

# Listas de struct

```
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.

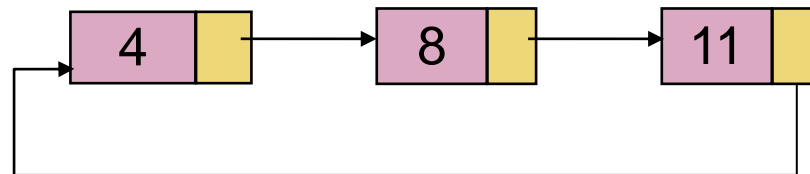
# Listas circulares

# Listas circulares

Hasta el momento sólo hemos visto listas con enlaces simples, en una única dirección, que tienen un inicio y un fin. Pero esta no es la única forma de crear una lista enlazada.

## 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.

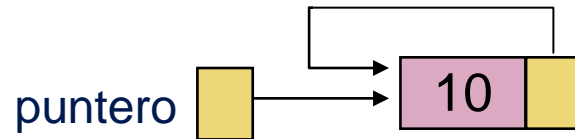


En realidad, podríamos comenzar a recorrer la lista por cualquier nodo y finalizar cuando volvamos a pasar por el mismo nodo por segunda vez.



# Listas circulares

Si la lista tiene un solo nodo, su puntero siguiente referenciará al mismo nodo.



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



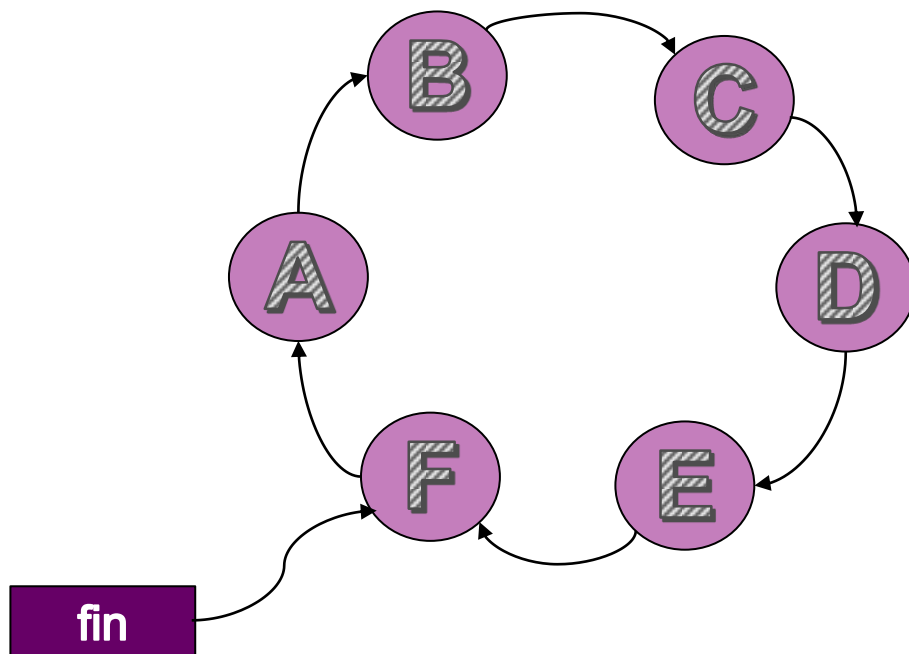
# Listas circulares

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.

# Puntero final

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).



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.

# 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



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;
```

# 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.

# 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.

# 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);
    }
}
```

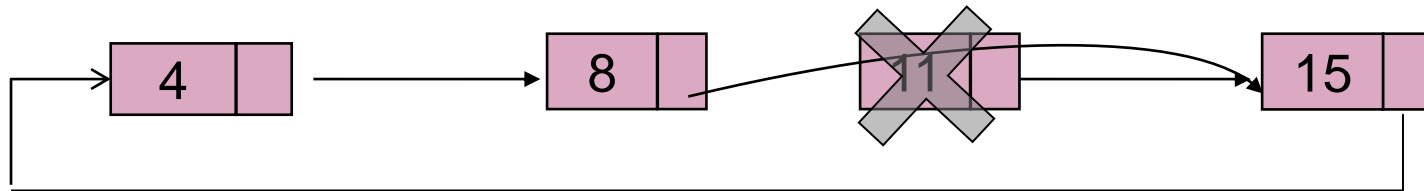
En este ejemplo se recorre la lista para imprimirla, pero podría recorrerse con cualquier otro objetivo.

Iteramos hasta volver a encontrarnos con el primer nodo.

# 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.



# Eliminar elemento

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

Si el siguiente a aux es el nodo a borrar, reenlazamos aux para "saltar" el nodo que será eliminado.

El puntero al nodo final se reubica si el que debe borrarse es el nodo apuntado por fin. El resto de la operación no cambia.

# FIN UNIDAD 3

Esta presentación fue diseñada por el siguiente equipo docente:

Dra. Claudia Russo  
Lic. Paula Lencina  
Lic. Cecilia Rastelli  
AC María Lanzillotta  
AC. Marina Rodríguez  
AC. David Fernández  
Prog. Trinidad Picco



**Atribución - No Comercial** (*by-nc*): Se permite la generación de obras derivadas siempre que no se haga con fines comerciales. Tampoco se puede utilizar la obra original con fines comerciales. Esta licencia no es una licencia libre.