

## Punteros y memoria dinámica

Facultad de Informática  
Universidad Complutense

Ana Gil Luezas

(Adaptadas del original de Luis Hernández Yáñez)



# Índice

---

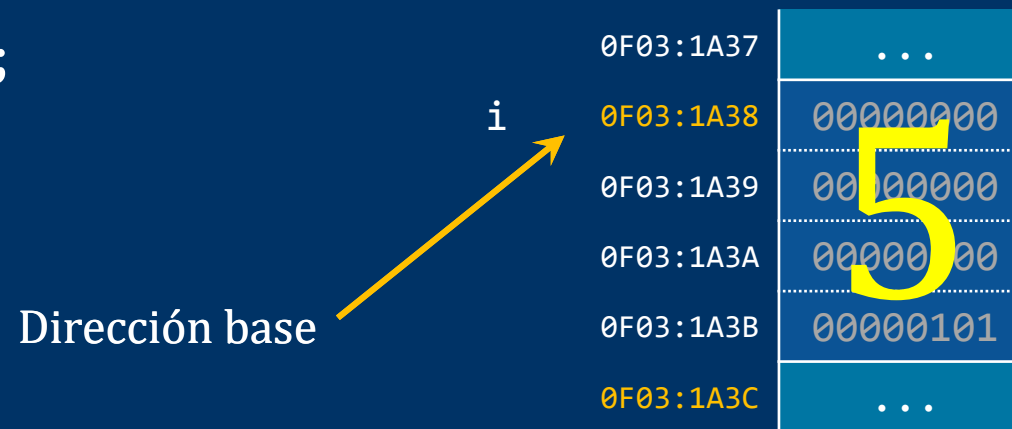
Direcciones de memoria y punteros	2
Operadores & y *	6
Inicialización de punteros (nullptr)	16
Asignación y comparación de punteros	19
Punteros a estructuras (operador ->)	23
Punteros a constantes y punteros constantes	26
Referencias	28
Punteros y paso de parámetros	31
Punteros y arrays	34
Punteros y funciones	36
Memoria y datos del programa	37
Memoria dinámica	42
Operadores new y delete	45
Gestión de la memoria	57
Errores comunes	61
Arrays y memoria dinámica	66
Arrays de punteros	67
Arrays dinámicos: new[] y delete[]	80



# Direcciones de memoria y punteros

Todo dato de un programa se almacena en la memoria, en unos cuantos bytes a partir de una dirección.

```
int i = 5;
```



Al dato se accede a partir de su *dirección base* (*i*: **0F03:1A38**), la dirección de la primera celda de memoria utilizada por ese dato.

El tipo de datos (**int**) indica cuántas celdas (**sizeof(int)** bytes) utiliza el dato (4 bytes): 00000000 00000000 00000000 00000101 → 5

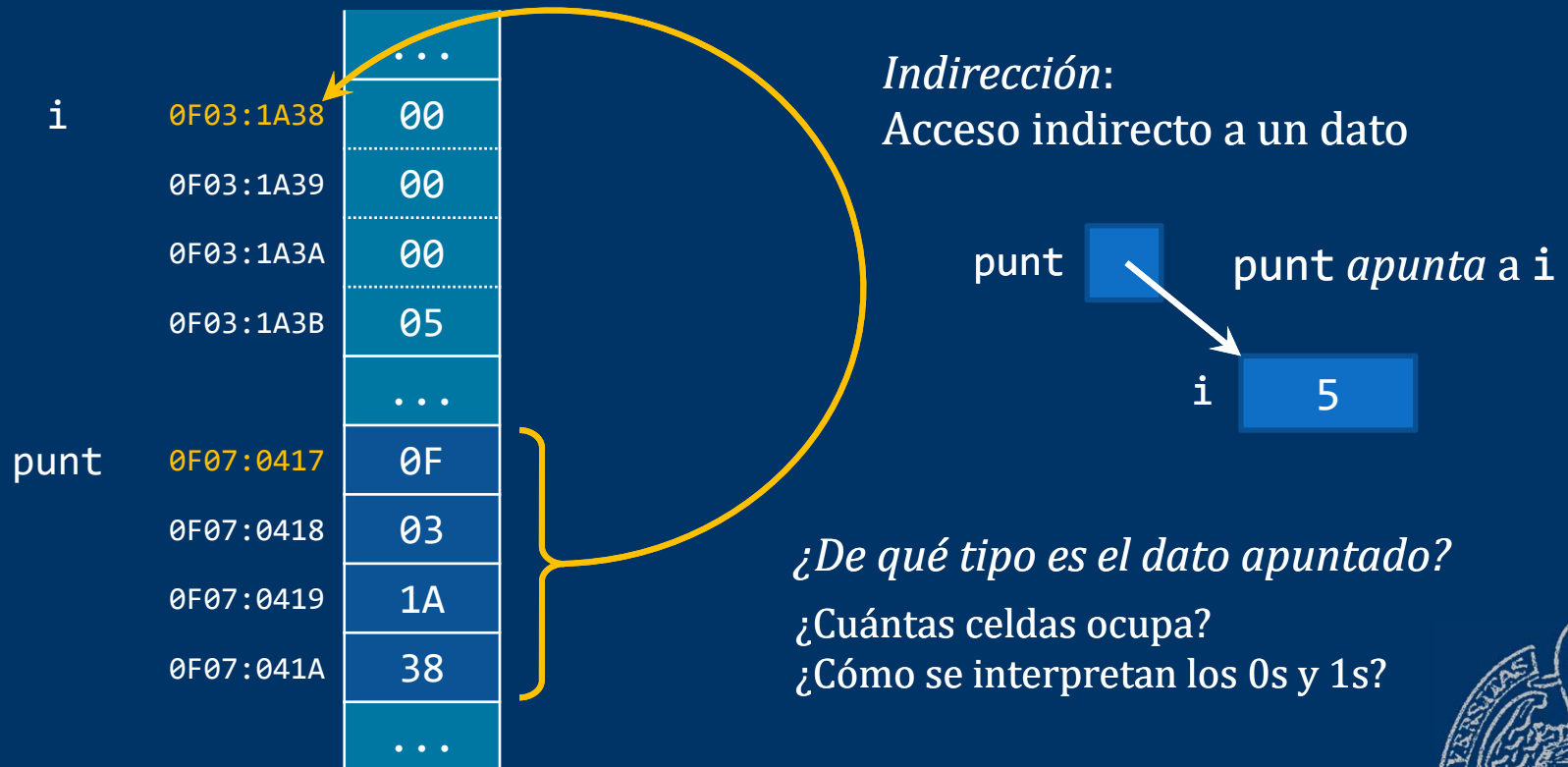
(La codificación de los datos puede ser diferente. Y la de las direcciones también.)



# Direcciones de memoria y punteros

Una *variable puntero* (o simplemente un *puntero*) sirve para acceder a través de ella a otra variable.

El valor del puntero será la *dirección de memoria* base de la otra variable.



# Direcciones de memoria y punteros

---

*Los punteros contienen direcciones de memoria*

La variable a la que apunta un puntero, como cualquier otra variable, será de un tipo concreto (¿cuánto ocupa? ¿cómo se interpreta?).

Este tipo se establece al declarar la variable puntero:

```
tipo * nombre;
```

El puntero *nombre* apuntará a una variable del *tipo* declarado (el tipo base del puntero).

El asterisco (\*) indica que es un puntero a datos de ese tipo.

```
int * punt; // punt inicialmente está indefinida,  
            // contiene una dirección que no es válida.
```

El puntero *punt* apuntará a una variable entera (*int*).

```
int i; // Dato entero vs. int * punt; // Puntero a entero
```



# Direcciones de memoria y punteros

---

Las variables puntero tampoco se inicializan automáticamente.  
Al declararlas sin inicializar contienen direcciones que no son válidas.

```
int * punt; // punt inicialmente está indefinida,  
            // contiene una dirección que no es válida.
```

Un puntero puede apuntar a cualquier variable del tipo base, o puede no apuntar a nada: valor (marca, centinela) `nullptr`

```
int * punt = nullptr; //inicialización, apunta a nada
```

*¿Para qué sirven los punteros?*

- ✓ Para compartir memoria (Paso de parámetros por referencia)
- ✓ Para gestionar variable dinámicas.  
(Variables que se crean y destruyen al solicitarlo el programa)
- ✓ Para implementar los arrays



# El Operador &

*Obtener la dirección de memoria de ...*

El operador monario & devuelve la dirección de memoria base de la variable a la que se aplica. Operador prefijo (precede).

```
int i;  
cout << &i; // Muestra la dirección de memoria de i
```

A un puntero se le puede asignar la dirección de una variable del mismo tipo que el tipo base del puntero:

```
int i = 5;  
int * punt = nullptr;  
punt = &i; // punt contiene la dirección base de i
```



Ahora, el puntero punt contiene una dirección de memoria válida.

punt *apunta a* (contiene la dirección base de) la variable entera i (**int**).



# El Operador &

*Obtener la dirección de memoria de ...*

```
int i, j;
```

```
...
```

```
int * punt;
```

	...	
i	0F03:1A38	
	0F03:1A39	
	0F03:1A3A	
	0F03:1A3B	
j	0F03:1A3C	
	0F03:1A3D	
	0F03:1A3E	
	0F03:1A3F	
	...	
punt	0F07:0417	
	0F07:0418	
	0F07:0419	
	0F07:041A	
	...	





# El Operador &

*Obtener la dirección de memoria de ...*

```
int i, j;
```

```
...
```

```
int * punt;
```

```
...
```

```
i = 5;
```

i

5

i

0F03:1A38

0F03:1A39

0F03:1A3A

0F03:1A3B

j

0F03:1A3C

0F03:1A3D

0F03:1A3E

0F03:1A3F

punt

0F07:0417

0F07:0418

0F07:0419

0F07:041A

...

...

...

...	
00	
00	
00	
05	



# El Operador &

*Obtener la dirección de memoria de ...*

```
int i, j;
```

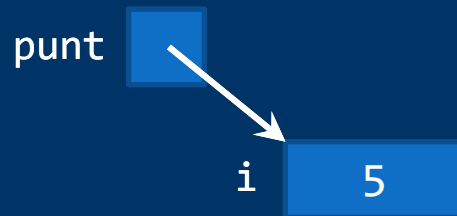
```
...
```

```
int * punt;
```

```
...
```

```
i = 5;
```

```
punt = &i;
```



...	...	
i	0F03:1A38	00
	0F03:1A39	00
	0F03:1A3A	00
	0F03:1A3B	05
j	0F03:1A3C	
	0F03:1A3D	
	0F03:1A3E	
	0F03:1A3F	
	...	
punt	0F07:0417	0F
	0F07:0418	03
	0F07:0419	1A
	0F07:041A	38
	...	

A yellow arrow originates from the memory address 0F03:1A38 (the address of variable 'i') and points to the memory address 0F07:0417 (the address of variable 'punt'). A yellow bracket on the right side of the memory dump groups the addresses from 0F07:0417 to 0F07:041A.



# El Operador \*

---

## *Acceso a la variable apuntada por ...*

El operador monario \* se aplica a un puntero para acceder a la variable apuntada por él. Operador prefijo (precede).

Una vez que un puntero contiene una dirección de memoria válida, se puede acceder a la variable a la que apunta con este operador.

```
punt = &i;      // punt a punta a i
cout << *punt; // Muestra lo que hay en i
```

\*punt: la variable apuntada por punt (la variable de la dirección que contiene el puntero punt).

*Acceso indirecto* a la variable i. i y \*punt son la misma variable.

```
punt = nullptr;
cout << *punt; // Error!! (punt a punta a nada)
```



# El Operador \*

*Acceso a la variable apuntada por ...*

```
int i, j;
```

```
...
```

```
int * punt;
```

```
...
```

```
i = 5;
```

```
punt = &i;
```

```
j = *punt;
```

```
...
```

```
*punt = 0;
```

punt:

i

0F03:1A38

00

0F03:1A39

00

0F03:1A3A

00

0F03:1A3B

05

j

0F03:1A3C

0F03:1A3D

0F03:1A3E

0F03:1A3F

...

→ punt

0F07:0417

0F

0F07:0418

03

0F07:0419

1A

0F07:041A

38

...



# El Operador \*

*Acceso a la variable apuntada por ...*

```
int i, j;
```

...

```
int * punt;
```

...

```
i = 5;
```

```
punt = &i;
```

```
j = *punt;
```

...

```
*punt = 0;
```

Direccionamiento  
indirecto  
(*indirección*).

Se accede al dato *i*  
de forma indirecta.

*\*punt*  $\equiv$  *i*



*i*

0F03:1A38

00

0F03:1A39

00

0F03:1A3A

00

0F03:1A3B

05

*j*

0F03:1A3C

0F03:1A3D

0F03:1A3E

0F03:1A3F

...

*punt*

0F07:0417

0F

0F07:0418

03

0F07:0419

1A

0F07:041A

38

...



# El Operador \*

*Acceso a la variable apuntada por...*

```
int i, j;
```

```
...
```

```
int * punt;
```

```
...
```

```
i = 5;
```

```
punt = &i;
```

```
j = *punt;
```

```
...
```

```
*punt = 0;
```

$*punt \equiv i$



	...	
i	0F03:1A38	00
	0F03:1A39	00
	0F03:1A3A	00
	0F03:1A3B	05
j	0F03:1A3C	00
	0F03:1A3D	00
	0F03:1A3E	00
	0F03:1A3F	05
	...	
punt	0F07:0417	0F
	0F07:0418	03
	0F07:0419	1A
	0F07:041A	38
	...	



# El Operador \*

*Acceso a la variable apuntada por...*

```
int i, j;
```

```
...
```

```
int * punt;
```

```
...
```

```
i = 5;
```

```
punt = &i;
```

```
j = *punt;
```

```
...
```

```
*punt = 0;
```

$*punt \equiv i$



i

0F03:1A38

00

0F03:1A39

00

0F03:1A3A

00

0F03:1A3B

00

j

0F03:1A3C

00

0F03:1A3D

00

0F03:1A3E

00

0F03:1A3F

05

...

punt

0F07:0417

0F

0F07:0418

03

0F07:0419

1A

0F07:041A

38

...



# Operadores & y \*

---

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

int main() {
    int i = 5;
    int j = 13;
    int * punt = nullptr;
    punt = &i;
    cout << *punt << endl; // Muestra el valor de i
    punt = &j;
    cout << *punt << endl; // Ahora muestra el valor de j
    int * otro = &i;
    cout << *otro + *punt << endl; // i + j
    int k = *punt;
    cout << k << endl; // Mismo valor que j

    return 0;
}
```





# Punteros y direcciones válidas

---

Un puntero NO contiene una dirección válida tras ser definido.

Un puntero obtiene una dirección válida:

- ✓ Al asignarle otro puntero (con el mismo tipo base) que ya contenga una dirección válida.
- ✓ Al asignarle la dirección de otra variable con el operador &.
- ✓ Al asignarle el valor `nullptr` (indica que se trata de un puntero nulo, un puntero que no apunta a nada).

```
int i;  
int * q; // q no tiene aún una dirección válida  
int * p = &i; // p toma una dirección válida  
q = nullptr; // ahora q ya tiene una dirección válida  
q = p; // otra dirección válida para q
```



# Punteros no inicializados

---

Un puntero no inicializado contiene una dirección desconocida.

```
int * punt; // no inicializado, PELIGRO!!  
*punt = 12;
```

*¿Dirección de la zona de datos del programa?*

¡Podemos estar modificando inadvertidamente un dato del programa!

→ El programa no obtendría los resultados esperados.

*¿Dirección de la zona de código del programa?*

¡Podemos estar modificando el propio código del programa!

→ Se podría ejecutar una instrucción incorrecta

*¿Dirección de la zona de código del sistema operativo?*

¡Podemos estar modificando el código del propio S.O.!

→ Consecuencias imprevisibles (*cuelgue*)



# Un valor seguro: `nullptr` (centinela)

Inicializando los punteros a `nullptr` podemos detectar errores:

```
int *punt = nullptr;
```

punt 

```
...
```

```
*punt = 13; // Error!! (punt a punta a nada)
```

punt contiene el valor `nullptr`: ¡Puntero nulo! ¿qué significa `*punt`?

→ ERROR: *¡Se intenta acceder a un dato a través de un puntero nulo!*

Se produce un error de ejecución, lo que ciertamente no es bueno.

Pero sabemos exactamente cuál ha sido el problema, lo que es mucho.

Sabemos por dónde empezar a investigar (depurar) y qué buscar.



# Asignación de punteros

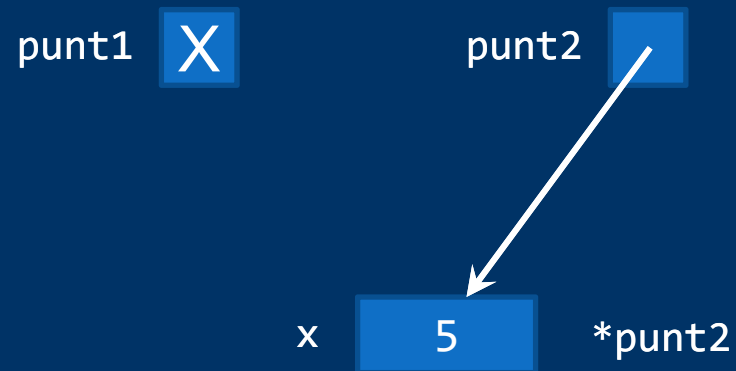
---

```
int x = 5;
```

```
int * punt1 = nullptr; // punt1 apunta a nada
```

```
int * punt2 = &x; // punt2 apunta a la variable x
```

\*punt2 y x son la misma variable

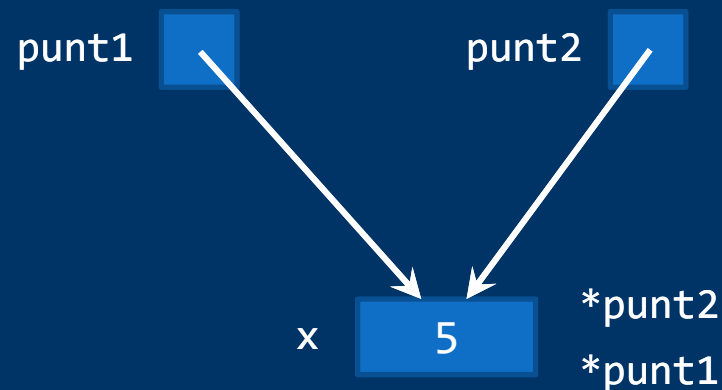


# Asignación de punteros

---

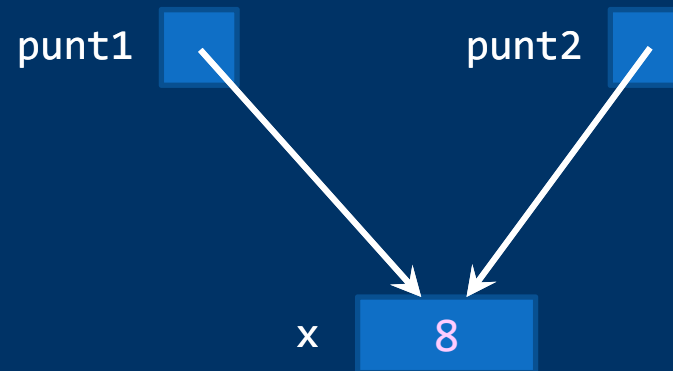
```
int x = 5;  
int * punt1 = nullptr; // punt1 apunta a nada  
int * punt2 = &x; // punt2 apunta a la variable x
```

```
punt1 = punt2; // ambos apuntan a la variable x  
*punt2, *punt1 y x son la misma variable
```



# Asignación de punteros

```
int x = 5;  
int * punt1 = nullptr; // punt1 apunta a nada  
int * punt2 = &x; // punt2 apunta a la variable x  
punt1 = punt2; // ambos apuntan a la variable x  
*punt1 = 8; // *punt1, *punt2 y x son la misma variable
```



A la variable x  
ahora se puede  
acceder de 3 formas:  
x   \*punt1   \*punt2



# Comparación de punteros

---

Los operadores relacionales `==` y `!=` nos permiten saber si dos punteros apuntan a una misma variable:

```
int x = 5;
int * punt1 = nullptr;
int * punt2 = &x;
...
if (punt1 == punt2)
    cout << "Apuntan al mismo dato" << endl;
else
    cout << "No apuntan al mismo dato" << endl;
Sólo tiene sentido comparar punteros con el mismo tipo base.
if (punt1 == nullptr)
    cout << "Puntero nulo (apunta a nada)" << endl;
```



# Tipos puntero

---

Declaración de tipos para los punteros con distintos tipos base:

```
typedef int * intPtr;  
typedef char * charPtr;  
typedef double * doublePtr;  
typedef tRegistro * tRegistroPtr;
```

```
int entero = 5;  
intPtr puntI = &entero;  
char character = 'C';  
charPtr puntC = &character;  
double real = 5.23;  
doublePtr puntD = &real;  
cout << *puntI << " " << *puntC << " " << *puntD << endl;
```

*\*puntero* es una variable del tipo base de *puntero*.

Con *\*puntero* podemos hacer lo que se pueda hacer con las variables del tipo base del *puntero*.





# Punteros a estructuras

---

Los punteros pueden apuntar a cualquier tipo de datos, también estructuras:

```
typedef struct {  
    int codigo;  
    string nombre;  
    double sueldo;  
} tRegistro;
```

```
tRegistro registro;  
tRegistro * puntero = &registro;
```

*Operador flecha (->)*: Permite acceder a los campos de una estructura a través de un puntero sin el operador de indirección (\*).

puntero->codigo      puntero->nombre      puntero->sueldo

puntero->...  $\equiv$  (\*puntero)...



# Punteros a estructuras

---

```
typedef struct {  
    int codigo;  
    string nombre;  
    double sueldo;  
} tRegistro;
```

```
tRegistro registro;  
tRegistro * puntero = &registro;  
registro.codigo = 12345;  
registro.nombre = "Javier";  
registro.sueldo = 95000;  
cout << puntero->codigo << " " << puntero->nombre  
      << " " << puntero->sueldo << endl;
```

$\text{puntero->codigo} \equiv (*\text{puntero}).\text{codigo} \neq \underbrace{* \text{puntero}.\text{codigo}}$

Se esperaría que puntero fuera una estructura con campo codigo de tipo puntero.



# Punteros y el modificador const

---

Cuando se declaran punteros con el modificador de acceso `const`, su efecto depende de dónde se coloque en la declaración:

<code>const tipo * puntero;</code>	Puntero a una constante
<code>tipo const* puntero;</code>	Puntero a una constante
<code>tipo * const puntero;</code>	Puntero constante

*Punteros a constantes:*

```
int entero1 = 5, entero2 = 13;
const int * punt_a_cte; // Puntero a dato constante
Punt_a_cte = &entero1;

(*punt_a_cte)++; // ERROR: ¡Dato constante no modificable!
punt_a_cte = &entero2; // Sin problema: el puntero no es cte.
```



# Punteros y el modificador const

---

## *Punteros constantes:*

```
int entero1 = 5, entero2 = 13;  
int * const punt_cte = &entero1; // Puntero constante
```

Al ser una constante hay que inicializarla en la definición

```
(*punt_cte)++; // Sin problema: el puntero no apunta a cte.  
punt_cte = &entero2; // ERROR: ¡Puntero constante!
```

## *Punteros constantes a constantes:*

```
const tipo * const puntero = &entero2;
```



# Referencias

---

Una referencia es una forma de dar otro nombre (*alias*) a una variable ya existente.

```
int x = 55;
int & rx = x; // rx es un alias de x
cout << "x: " << x << " rx: " << rx << endl; // x: 55 rx: 55
rx = rx + 1; // suma 1 a x
cout << "x: " << x << " rx: " << rx << endl; // x: 56 rx: 56
x = x + 1;    // suma 1 a x
cout << "x: " << x << " rx: " << rx << endl; // x: 57 rx: 57
```

Una referencia es una constante: hay que darle el valor en la definición y no se puede modificar.

Cuando se usa una variable de tipo referencia, en realidad se usa la variable a la que hace referencia (aquella de la que es *alias*)



# Referencias

---

```
typedef struct {  
    int codigo;  
    string nombre;  
    double sueldo;  
} tRegistro;  
  
tRegistro registro;  
tRegistro * puntero = &registro;  
tRegistro & referencia = registro;  
registro.codigo = 12345;  
puntero->nombre = "Javier";  
referencia.sueldo = 95000;  
cout << puntero->codigo << " " << referencia.nombre  
    << " " << registro.sueldo << endl;  
string & nomb = registro.nombre;  
nomb = "Pepe";
```



# Referencias y el modificador const

---

```
typedef struct {  
    int codigo;  
    string nombre;  
    double sueldo;  
} tRegistro;
```

```
tRegistro registro;  
string & nomb = registro.nombre;  
nomb = "Javier";  
const int & cod = registro.codigo; // Ref. a constante  
cod = 12345; // ERROR!!  
registro.codigo = 12345;  
const string * ps = &(registro.nombre);  
*ps = "Javier"; // ERROR!!
```



# Punteros y paso de parámetros

## *Paso de parámetros por referencia*

En el lenguaje C no existe el mecanismo de paso de parámetro por referencia (&). Sólo se pueden pasar parámetros por valor.

¿Cómo se implementa entonces el paso por referencia?

Por medio de punteros:

```
void incrementa(int * punt) {  
    (*punt)++;  
}  
...  
int entero = 5;  
incrementa(&entero);  
cout << entero << endl;
```

Paso por valor de una dirección de memoria:  
El argumento &entero no se puede modificar  
(punt contiene una copia del argumento)  
pero aquello a lo que apunta (la var. entero)  
se puede modificar a través de punt

Mostrará 6 en la consola.





# Punteros y paso de parámetros

```
int entero = 5;  
incrementa(&entero);
```

entero

5

punt recibe la dirección de entero

```
void incrementa(int * punt) {  
    (*punt)++;  
}
```

punt

entero

6

```
cout << entero << endl;
```

entero

6



# Punteros y paso de parámetros

---

*Paso de parámetros por referencia o con puntero*

```
void incrementa(int * punt)
{
    (*punt)++;
}
...
int entero = 5;
incrementa(&entero);
```

entero y (\*punt)  
son la misma variable

```
cout << entero << endl;
```

Mostrará 6 en la consola.

```
void incrementa(int & ent)
{
    ent++;
}
...
int entero = 5;
incrementa(entero);
```

entero y ent  
son la misma variable

```
cout << entero << endl;
```

Mostrará 6 en la consola.



# Punteros y arrays

---

*Identificador de variable array  $\equiv$   
Puntero constante al primer elemento del array*

Si tenemos:

```
int dias[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Entonces:

```
cout << *dias << endl;
```

Muestra 31 en la consola, el primer elemento del array.

`¡dias` es un puntero constante!

Siempre apunta al primer elemento. No se puede modificar su valor.

Al resto de los elementos del array, además de por índice, se puede acceder por medio de las *operaciones aritméticas de punteros*.

```
int * d = dias;  
cout << *d << endl; cout << *(&+d) << ' ' << *(dias+2) << endl;
```

Muestra: 31 28 31



# Punteros y paso de parámetros arrays

*¡Esto explica por qué no usamos & con los parámetros array!*

Como el identificador del array es un puntero, el array se comparte.

Declaraciones alternativas para parámetros array:

```
const int N = ...;
void cuadrado(int array[N]);
void cuadrado(int array[], int size); // Array no delimitado
void cuadrado(int * array, int size); // Puntero
```

Arrays no delimitados: No indicamos el tamaño, pudiendo aceptar cualquier array de ese tipo base (`int`).

Con arrays no delimitados y punteros se ha de proporcionar la dimensión, o usar un centinela, para poder recorrer el array .

Independientemente de cómo se declare el parámetro, se puede acceder a los elementos con índice (`array[i]`) o con puntero (`*(array+i)`).



# Punteros y funciones

*Podemos definir tipos para funciones*

```
typedef int (* tFunEnt) (int); // puntero a una función
```

*Una función es de tipo tFunEnt si tiene un parámetro int y devuelve como resultado un int*

```
int sig(int n) { return n+1; } // es de tipo tFunEnt
```

*Podemos definir variables de tipo tFunEnt*

```
tFunEnt fun = sig;
```

*Podemos definir funciones con parámetros de tipo tFunEnt*

```
void apliFunArray(int a[], int cont, tFunEnt F) {  
    for (int i=0; i<cont; ++i)  
        a[i] = F(a[i]); // (*F)(a[i])  
}
```

*Y utilizar funciones como argumento en las llamadas* (int aEnt[56] = {});

```
apliFunArray(aEnt, 56, sig); // ...
```



# Fundamentos de la programación

---

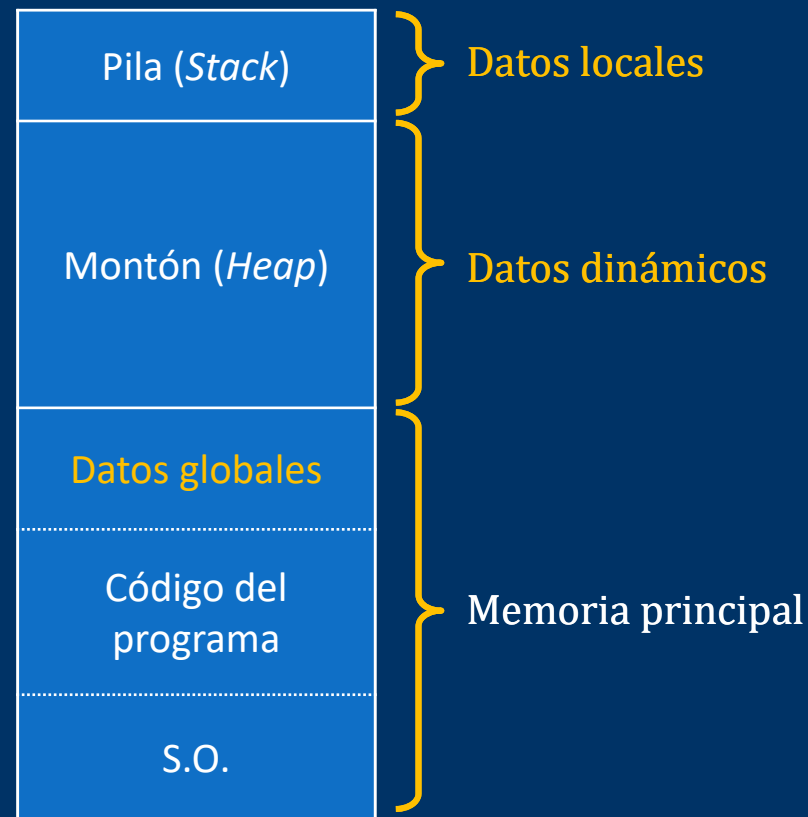
## Memoria y datos del programa



# Memoria y datos del programa

## *Regiones de la memoria*

El S.O. dispone en la memoria de la computadora varias regiones donde se almacenan distintas categorías de datos del programa:



# Memoria y datos del programa

## *La memoria principal*

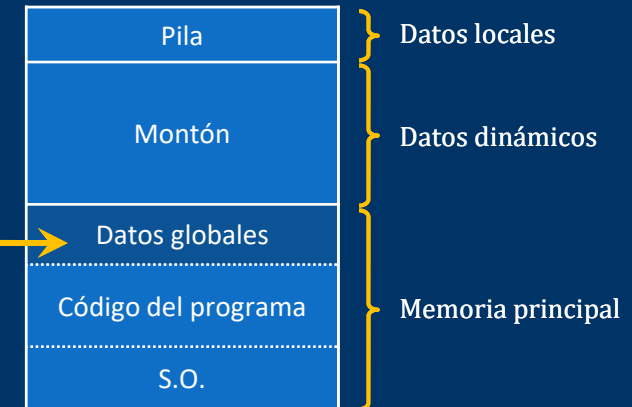
En la *memoria principal* se alojan los datos globales del programa: los que están declarados fuera de las funciones.

```
const int N = 1000;
```

```
typedef struct {  
    tRegistro registros[N];  
    int cont;  
} tTabla;
```

```
tTabla tabla;
```

```
int main() {  
    ...
```



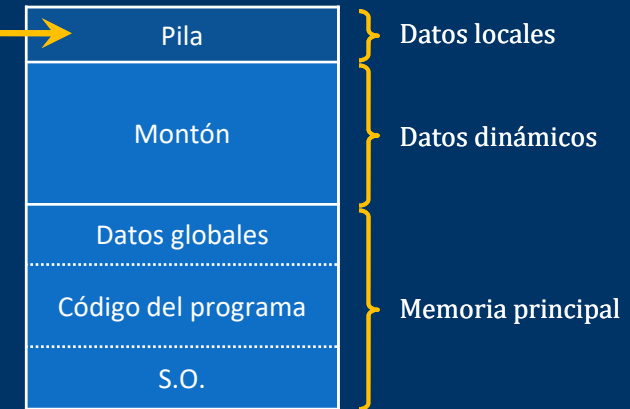


# Memoria y datos del programa

## *La pila (stack)*

En la *pila* se guardan los datos locales: parámetros y variables locales de las funciones.

```
void func(tTabla &tabla, double total)
{
    tTabla aux;
    int i;
    ...
}
```



func(tabla, resultado)

Los parámetros por valor requieren espacio para un dato del tipo declarado.

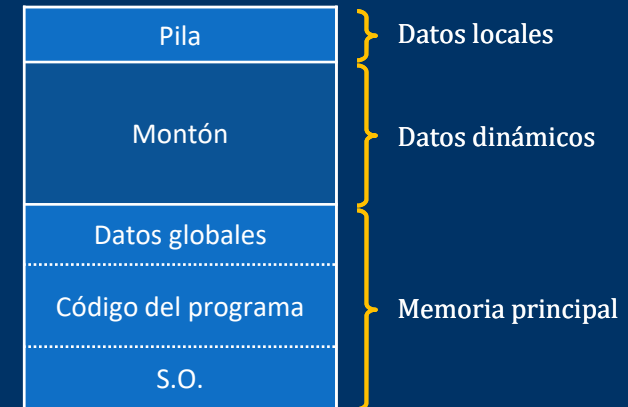
Los parámetros por referencia sólo para las direcciones de los argumentos.

Los datos locales (parámetros y variables de una función) se crean y destruyen automáticamente al ejecutarse una llamada a la función.



# Memoria y datos del programa

*El montón (heap) es una enorme zona de almacenamiento donde podemos alojar datos del programa que se creen y se destruyan a medida que se necesiten durante la ejecución del programa: Variables dinámicas*



Sistema de gestión de memoria dinámica (SGMD):

*Cuando se necesita memoria para una variable se solicita ésta al SGMD, quien reserva la cantidad adecuada para ese tipo de variable y devuelve la dirección de la primera celda de memoria de la zona reservada.*

*Cuando ya no se necesita más la variable, se libera la memoria que utilizaba indicando al SGMD que puede contar de nuevo con la memoria que se había reservado anteriormente.*



# Memoria dinámica

---

## *Variables dinámicas*

Se alojan en el montón al ejecutarse una solicitud al SGMD.



¿Por qué utilizar la memoria dinámica?

- ✓ Es un almacén de memoria muy grande: datos o listas de datos que no caben en la pila pueden ser alojados en el montón.
- ✓ El programa ajusta el uso de memoria a las necesidades de cada momento.
- ✓ El programa ajusta el tiempo de existencia de los datos: el momento de creación y destrucción lo determina el programa.



# Variables y asignación de memoria

---

*¿Cuándo se asigna memoria a las variables?*

- ✓ Variables **globales**:  
Se alojan en la memoria principal durante la carga del programa.  
Existen durante toda la ejecución del programa.
- ✓ Variables **locales** de una función (incluyendo parámetros):  
Se alojan en la pila del sistema durante la ejecución de una llamada a la función.  
Existen sólo durante la ejecución de esa llamada.
- ✓ Variables **dinámicas**:  
Se alojan en el montón (*heap*) cuando el programa lo solicita y se destruyen cuando el programa igualmente lo solicita.  
Existen *a voluntad* del programa.



# Variables dinámicas vs variables declaradas

---

## *Variables declaradas*

- ✓ Variables ( y constantes) declaradas con identificador y tipo:  
`int i;`
- ✓ A la variable se accede directamente a través del identificador:  
`cout << i; i = i + i;`

## *Variables dinámicas (anónimas)*

- ✓ Variables ( y constantes) no declaradas (sin nombre).
- ✓ Hay que acceder de forma indirecta, a través punteros.  
`int * p;`

Ya hemos visto que los datos estáticos también se pueden acceder a través de punteros (`int * p = &i;`).



# Operadores `new` y `delete`

---

Hasta ahora hemos trabajado con punteros que contienen direcciones de variables declaradas (variables globales o locales).

Sin embargo, los punteros también son la base sobre la que se apoya el sistema de gestión dinámica de memoria.

- ✓ Cuando queremos crear una variable dinámica de un tipo determinado, pedimos memoria del montón con el operador `new`.

El operador `new` reserva la memoria necesaria para ese tipo de datos y devuelve la dirección de la primera celda de memoria asignada a la variable; esa dirección hay que guardarla en un puntero.

- ✓ Cuando ya no necesitemos la variable, devolvemos la memoria que utiliza al montón mediante el operador `delete`.

Al operador se le pasa un puntero con la dirección de la primera celda de memoria (del montón) utilizada por la variable.



# Creación de variables dinámicas

---

## *El operador new*

`new tipo;` Reserva memoria del montón para una variable de ese *tipo* y devuelve la primera dirección de memoria utilizada, que debe ser asignada a un puntero.

```
tipo * puntero = new tipo;
```

```
int * p; // Todavía sin una dirección válida
```

```
p = new int; // Ya tiene una dirección válida ?
```

```
*p = 12;
```

La variable dinámica se accede exclusivamente a través de punteros; no hay ningún identificador asociado con ella que permita accederla.

```
int i; // i es una variable declarada
```

```
int * p1, * p2; // p1 y p2 son variables declaradas
```

```
p1 = &i; // Puntero que da acceso a la variable i
```

```
// (accesible con i o con *p1)
```

```
p2 = new int; // Puntero que da acceso a una variable
```

```
// dinámica (accesible sólo a través de *p2)
```

```
p1 = p2; // ahora también es accesible a través de *p1
```



# Eliminación de variables dinámicas

---

## *El operador delete*

`delete puntero;` Devuelve al montón la memoria utilizada por la variable dinámica apuntada por *puntero*.

```
int * p; // declaración de variable p (no es dinámica)
p = new int; // se crea una nueva variable dinámica
*p = 12;
...
delete p; // Ya no se necesita la variable apuntada por p
```

El puntero deja de contener una dirección válida y no se debe acceder a través de él hasta que no contenga nuevamente otra dirección válida.

Mientras tanto:

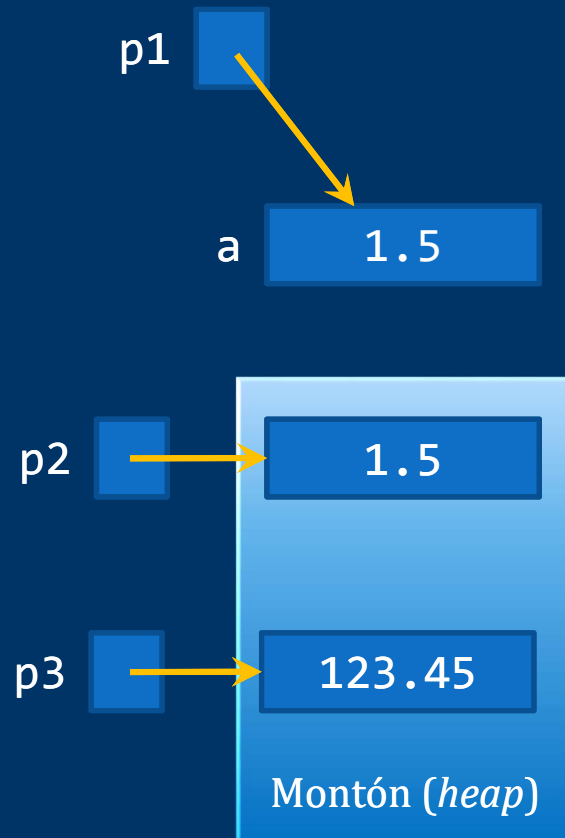
```
p = nullptr; // permite reconocer que ya no apunta a nada
```





# Un ejemplo

```
int main() {  
    double a = 1.5;  
    → double *p1, *p2, *p3;  
    p1 = &a;  
    p2 = new double;  
    *p2 = *p1;  
    p3 = new double;  
    *p3 = 123.45;  
    cout << *p1 << endl;  
    cout << *p2 << endl;  
    cout << *p3 << endl;  
    delete p2;  
    delete p3;  
  
    return 0;  
}
```



Identificadores:

4

(a, p1, p2, p3)

Variables:

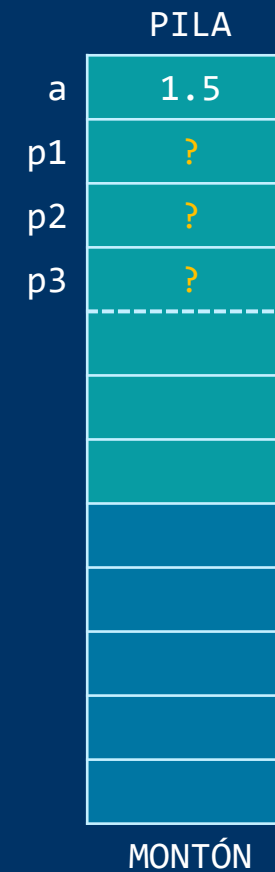
6

(4 + \*p2 y \*p3)



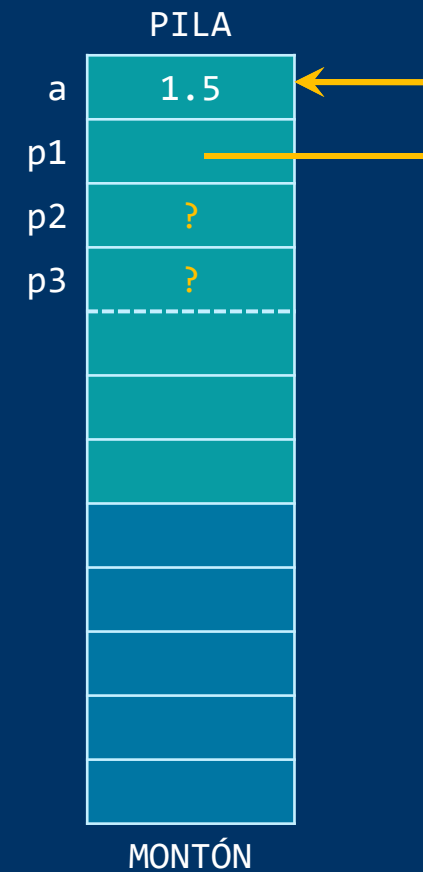
# Un ejemplo

```
int main() {  
    double a = 1.5;  
    double *p1, *p2, *p3;  
}
```



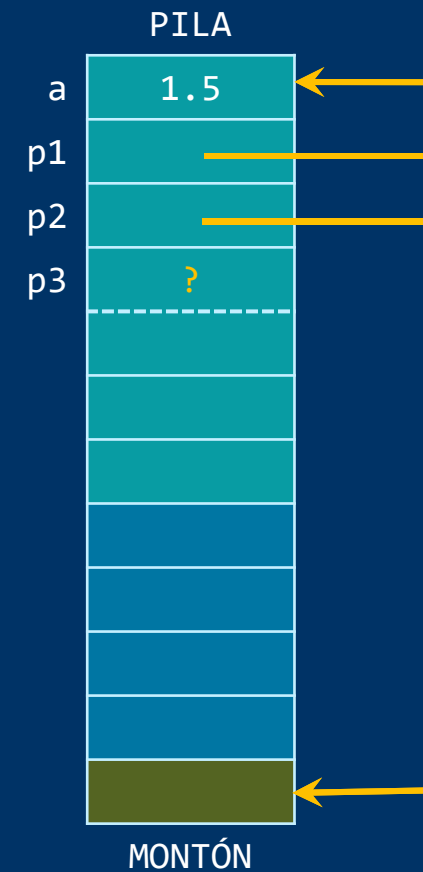
# Un ejemplo

```
int main() {  
    double a = 1.5;  
    double *p1, *p2, *p3;  
    p1 = &a;
```



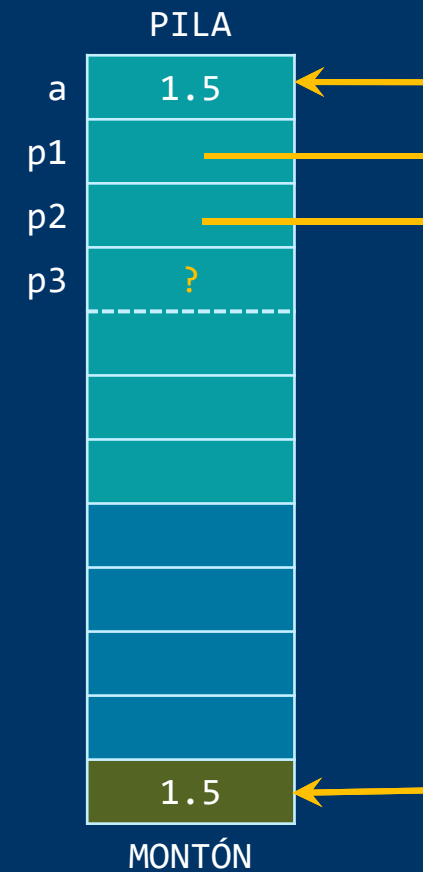
# Un ejemplo

```
int main() {  
    double a = 1.5;  
    double *p1, *p2, *p3;  
    p1 = &a;  
    p2 = new double;  
}
```



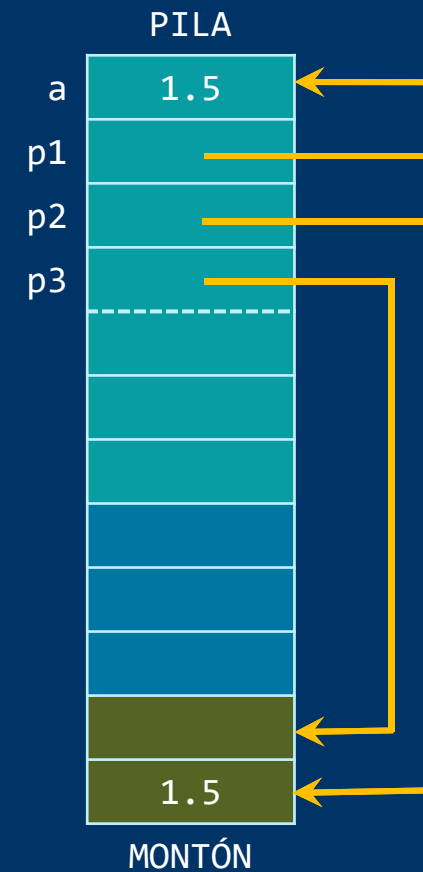
# Un ejemplo

```
int main() {  
    double a = 1.5;  
    double *p1, *p2, *p3;  
    p1 = &a;  
    p2 = new double;  
    *p2 = *p1;  
}
```



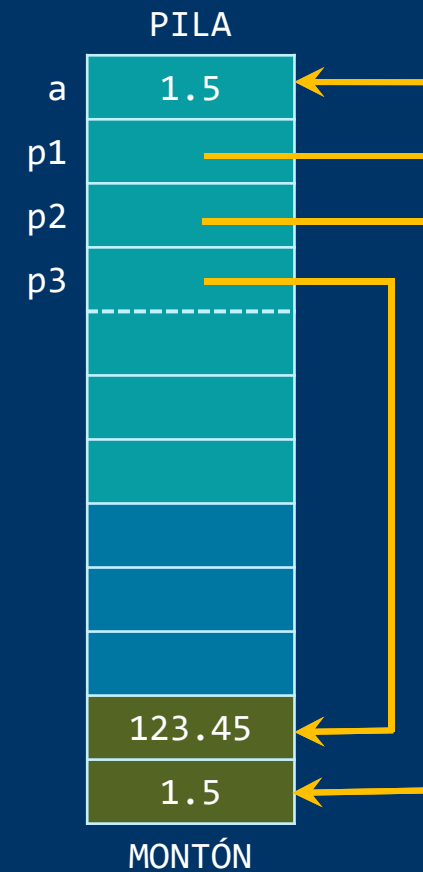
# Un ejemplo

```
int main() {  
    double a = 1.5;  
    double *p1, *p2, *p3;  
    p1 = &a;  
    p2 = new double;  
    *p2 = *p1;  
    p3 = new double;
```



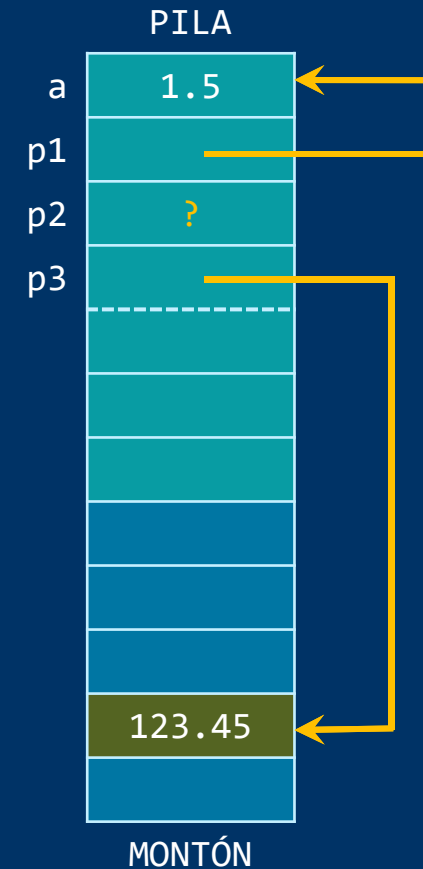
# Un ejemplo

```
int main() {  
    double a = 1.5;  
    double *p1, *p2, *p3;  
    p1 = &a;  
    p2 = new double;  
    *p2 = *p1;  
    p3 = new double;  
    *p3 = 123.45;  
}
```



# Un ejemplo

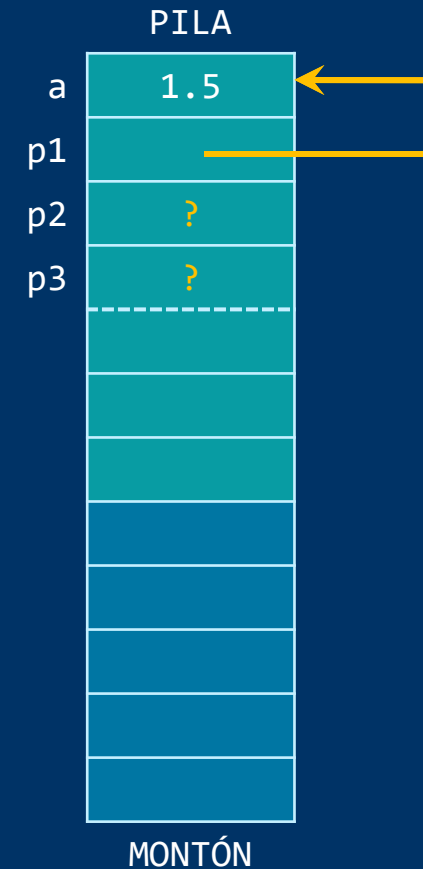
```
int main() {  
    double a = 1.5;  
    double *p1, *p2, *p3;  
    p1 = &a;  
    p2 = new double;  
    *p2 = *p1;  
    p3 = new double;  
    *p3 = 123.45;  
    cout << *p1 << endl;  
    cout << *p2 << endl;  
    cout << *p3 << endl;  
    delete p2;  
}
```





# Un ejemplo

```
int main() {  
    double a = 1.5;  
    double *p1, *p2, *p3;  
    p1 = &a;  
    p2 = new double;  
    *p2 = *p1;  
    p3 = new double;  
    *p3 = 123.45;  
    cout << *p1 << endl;  
    cout << *p2 << endl;  
    cout << *p3 << endl;  
    delete p2;  
    delete p3;  
}
```



# Fundamentos de la programación

---

## Gestión de la memoria



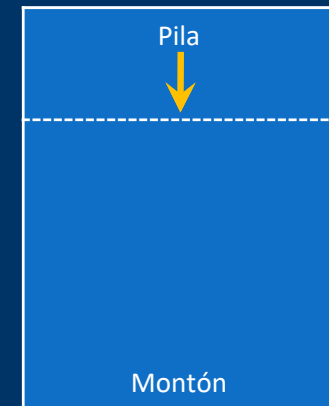
# Desbordamiento de la pila

---

## *Stack overflow*

La pila crece a medida que se llama a funciones, y decrece a medida que termina la ejecución de funciones.

La ocupación es contigua (todos los datos están juntos, comenzando en la dirección base de la pila).



Normalmente la pila tiene un tamaño máximo establecido que no puede sobrepasar aunque quede memoria en el montón. Cuando se agota se produce un *desbordamiento de la pila (stack overflow)*.

Evitar llamadas en cascada y parámetros por valor de tipos no básicos.

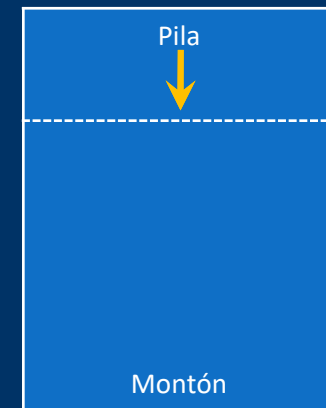


# Error de asignación de memoria

## *Heap (bad\_alloc)*

A medida que se crean datos dinámicos disminuye la cantidad de memoria libre en el montón.

Y a medida que se liberan aumenta.



Los datos no están contiguos, los huecos no tienen un tamaño concreto.

Al solicitar al SGMD (`new tipo`) un bloque de memoria (de tamaño `sizeof(tipo)`), busca y asigna un bloque contiguo del tamaño solicitado, devolviendo la dirección base del bloque.

`new tipo;`

falla si no queda suficiente memoria contigua del tamaño solicitado:  
devuelve una excepción `bad_alloc` (o `nullptr` para `new(nothrow)`)



# Gestión de la memoria dinámica

---

El Sistema de Gestión de Memoria Dinámica (SGMD) se encarga de localizar en el montón un bloque suficientemente grande para alojar la variable que se pida crear y sigue la pista de los bloques disponibles.

Pero no dispone de un *recolector de basura*, como Java o C#.

Es nuestra responsabilidad devolver al montón toda la memoria utilizada por nuestras variables dinámicas una vez que no se necesitan.

Los programas deben asegurarse de liberar, con el operador `delete`, todas las variables previamente creadas con el operador `new`.

Y siempre debe haber alguna forma (puntero) de acceder a cada dato dinámico. Es un grave error *perder* un dato en el montón.



# Errores comunes

## Olvido de inicialización de la variable dinámica

```
int * p;  
p = new int;
```



Se crea la variable dinámica, de tipo `int`, `p` queda inicializada a una dirección válida, pero `*p` NO se inicializa.

```
tRegistro * punt = new tRegistro;
```



Se crea la variable dinámica, de tipo `tRegistro`, `punt` queda inicializada a una dirección válida, pero `*punt` NO se inicializa.



# Errores comunes

---

Olvido de destrucción de un dato dinámico (*memory leaks*):

```
...  
int main() {  
    tRegistro *p;  
    p = new tRegistro;  
    leer(*p);  
    mostrar(*p);  
    return 0;    ← Falta delete p;  
}
```

El programa parecerá terminar correctamente, pero dejará memoria sin liberar.

El depurador de Visual C++:

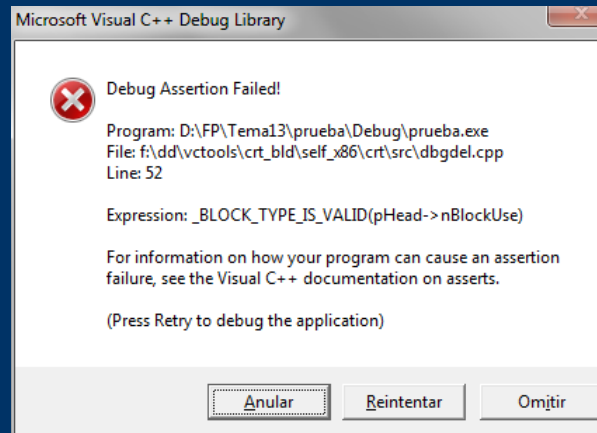
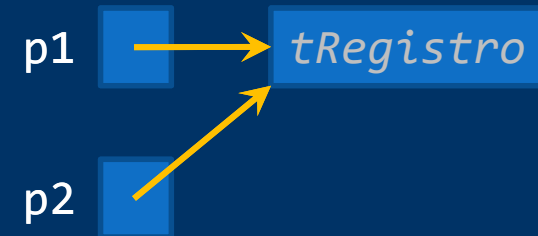
```
#ifdef _DEBUG  
#define DBG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )  
#define new DBG_NEW  
#endif  
_CrtDumpMemoryLeaks(); // para que muestre la basura
```



# Errores comunes

Intento de destrucción de un dato dinámico inexistente:

```
...  
int main() {  
    tRegistro * p1 = new tRegistro;  
    leer(*p1);  
    mostrar(*p1);  
    tRegistro *p2;  
    p2 = p1;  
    mostrar(*p2);  
    delete p1;  
    delete p2;  
  
    return 0;  
}
```



Sólo se ha creado  
una variable dinámica  
→ No se pueden destruir 2

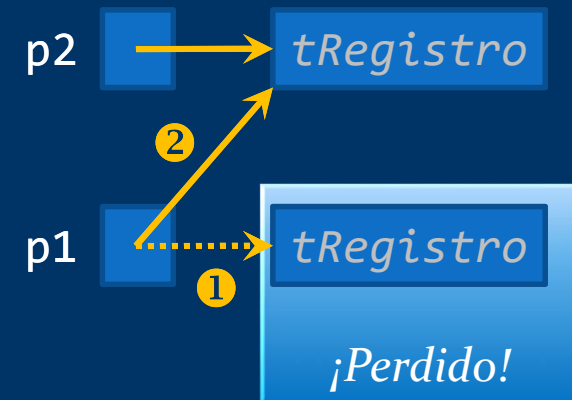




# Errores comunes

Pérdida de un dato dinámico (basura):

```
...  
int main() {  
    tRegistro *p1, *p2;  
    p1 = new tRegistro; ①  
    leer(*p1);  
    p2 = new tRegistro;  
    leer(*p2);  
    mostrar(*p1); ②  
    p1 = p2;  
    mostrar(*p1);  
  
    delete p1;  
    delete p2;  
  
    return 0;  
}
```



p1 deja de apuntar al dato dinámico  
que se creó primero  
→ Se pierde ese dato en el montón



# Errores comunes

---

Intento de acceso a un dato dinámico tras su eliminación:

```
...  
int main() {  
    tRegistro * p;  
    p = new tRegistro;  
    leer(*p);  
    mostrar(*p);  
    delete p;  
  
    ...  
    mostrar(*p);  
  
    return 0;  
}
```

p ha dejado de apuntar a una dirección válida  
→ Intento de acceso a memoria inexistente



# Fundamentos de la programación

---

## Arrays y memoria dinámica



# Arrays de punteros

---

```
typedef char tCadena[81];
typedef struct {
    int codigo;
    tCadena nombre;
    double valor;
} tRegistro;

const int TM = ...;

// Array de punteros a registros:

typedef struct {
    tRegistro * registros[TM];
    int cont;
} tLista;
```

Los punteros ocupan  
muy poco en memoria.  
Los datos a los que apunten  
se guardarán en el montón.



# Arrays de punteros a variables dinámicas

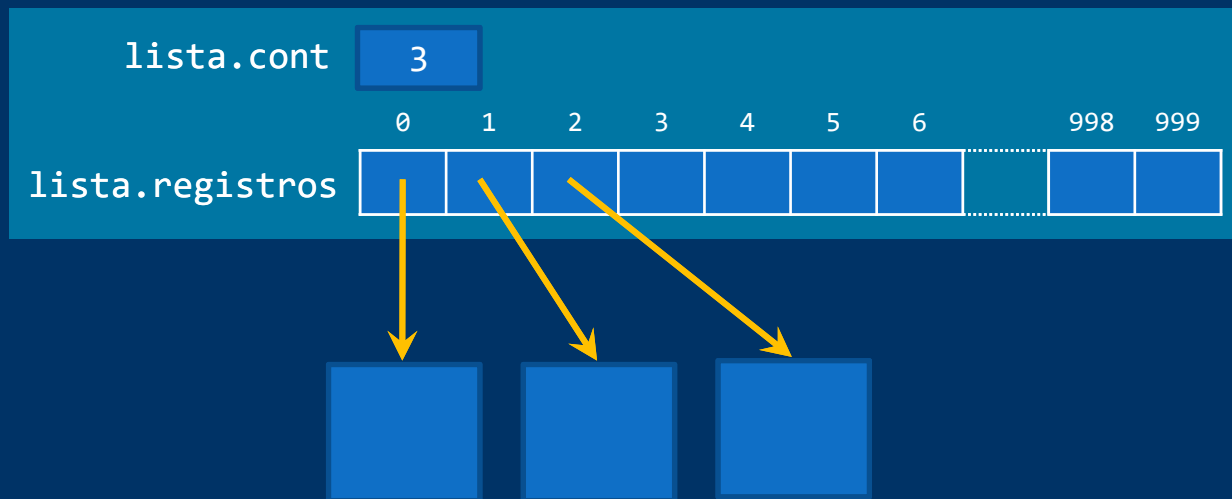
Los datos siempre están en la misma dirección de memoria.

El desplazamiento no cambia la ubicación en memoria de los datos.

El coste de desplazar los datos es independiente de su tamaño (tipo).

A los datos se accede a través de punteros (operador flecha):

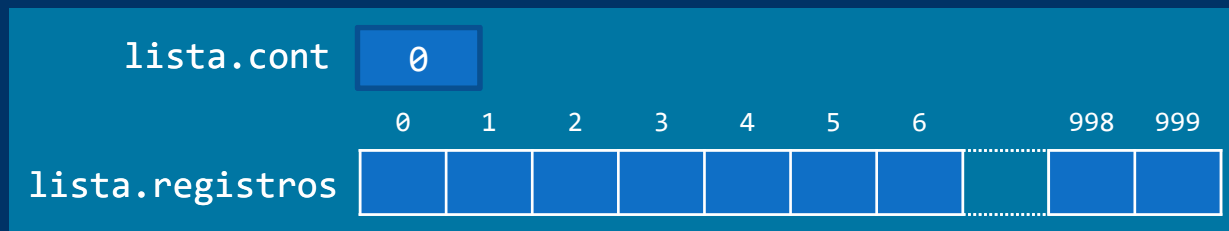
```
cout << lista.registros[0] -> nombre;
```



# Arrays de punteros a variables dinámicas

- Las variables se van creando a medida que se insertan datos

```
tLista lista;  
lista.cont = 0;
```

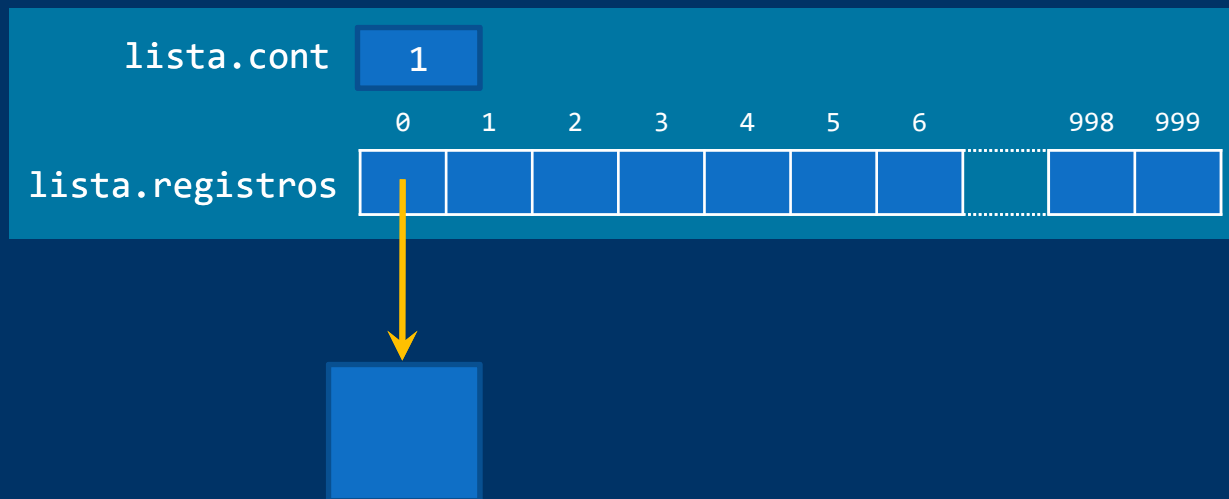


Y se van destruyendo a medida que los datos se eliminan de la lista



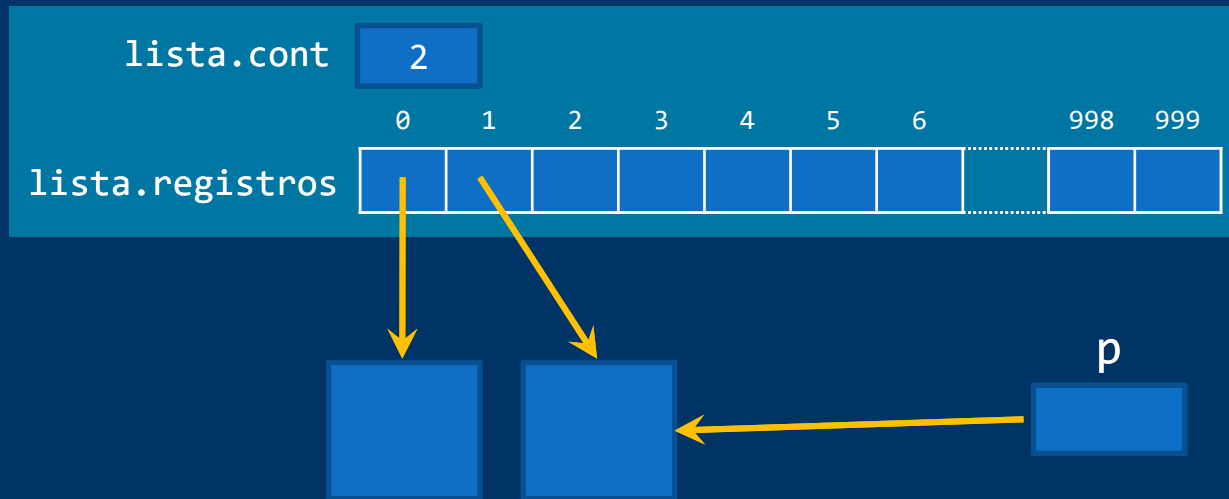
# Arrays de punteros a variables dinámicas

```
tLista lista;  
lista.cont = 0;  
lista.registros[lista.cont] = new tRegistro;  
leer(*(lista.registro[lista.cont]));  
++lista.cont;
```



# Arrays de punteros a variables dinámicas

```
tLista lista;  
lista.cont = 0;  
lista.registros[lista.cont] = new tRegistro;  
leer(lista.registro[lista.cont]); ++lista.cont;  
tRegistro * p = new tRegistro; leer(*p);  
lista.registros[lista.cont] = p; ++lista.cont;
```

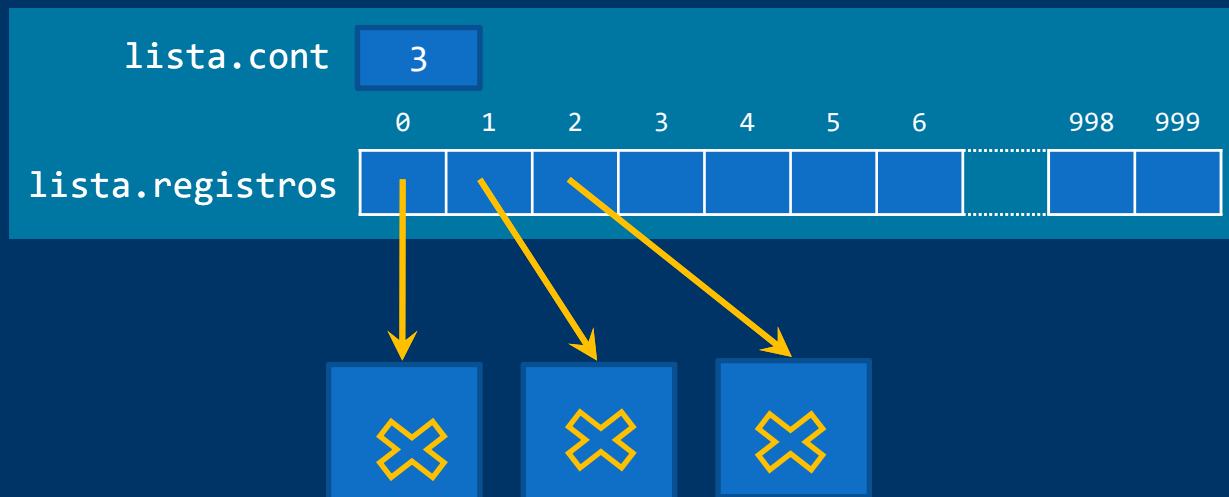




# Arrays de punteros a variables dinámicas

No hay que olvidarse de devolver la memoria al montón:

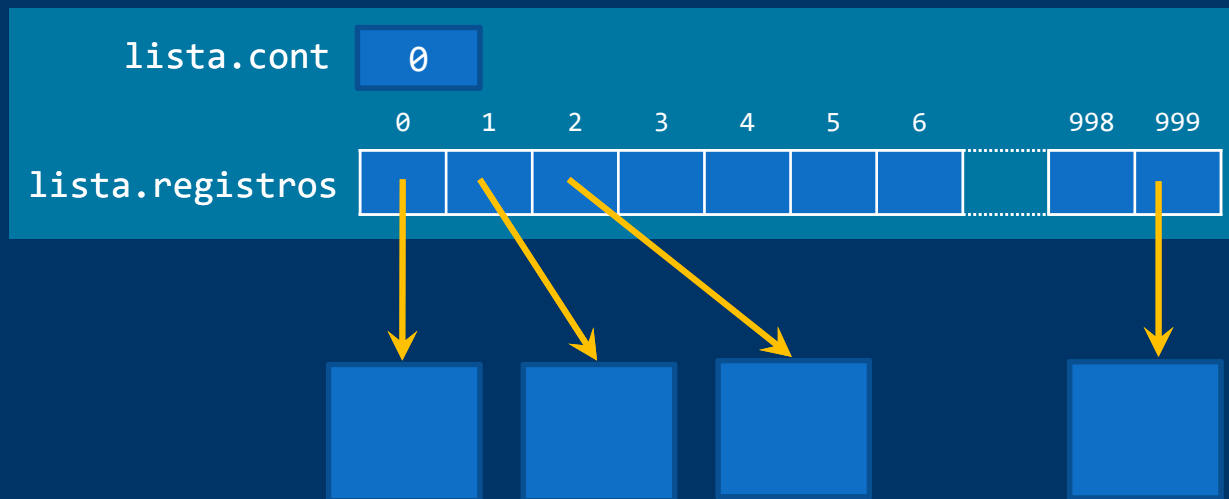
```
for (int i = 0; i < lista.cont; ++i)  
    delete lista.registros[i];
```



# Arrays de punteros a variables dinámicas

- Se crean todas las variables al iniciar la lista y se destruyen todas cuando la lista deje de utilizarse

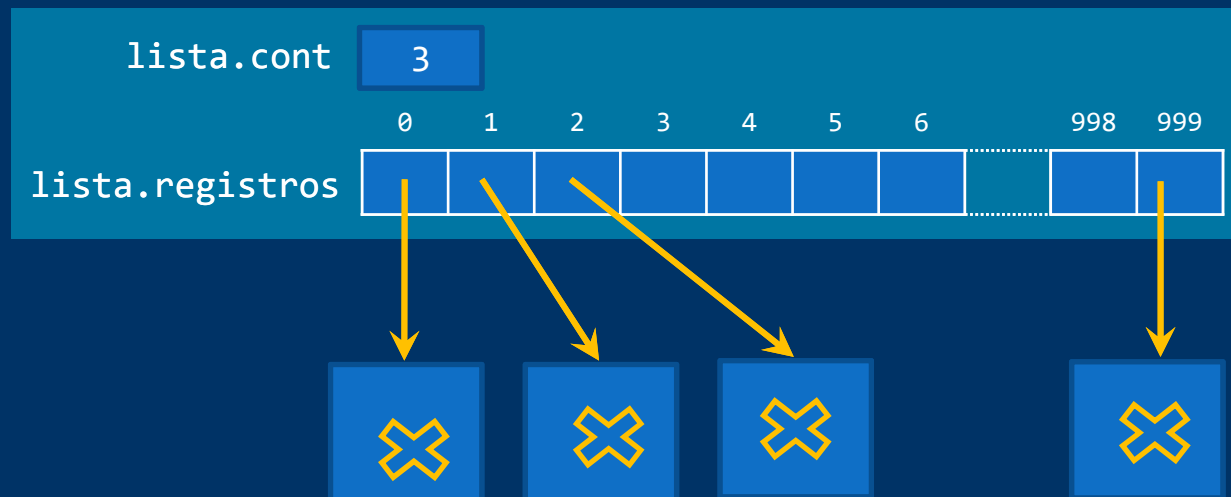
```
tLista lista;  
lista.cont = 0;  
for (int c = 0; c < TM; ++c)  
    lista.registros[lista.cont] = new tRegistro;
```



# Arrays de punteros a variables dinámicas

No hay que olvidarse de devolver la memoria al montón:

```
for (int i = 0; i < TM; ++i)  
    delete lista.registros[i];
```



# Ejemplo: Una lista con variables dinámicas

```
#ifndef LISTADD_H
#define LISTADD_H
#include "registro.h"

const int TM = ...;

typedef struct {
    tRegistro * registros[TM]; ←
    int cont;
} tLista;

void iniciar(tLista & lista);
void mostrar(const tLista & lista);
bool insertar(tLista & lista, const tRegistro & registro);
bool eliminar(tLista & lista, int code);
bool buscar(const tLista & lista, int code, int & pos);
bool cargar(tLista & lista);
void guardar(const tLista & lista);
void destruir(tLista & lista);

#endif
```



# Ejemplo: Una lista con variables dinámicas

```
void iniciar(tLista & lista) {  
    lista.cont = 0;  
    for (int c = 0; c < TM; ++c)  
        lista.registros[lista.cont] = new tRegistro;
```

Creamos todos al iniciar

```
}  
bool insertar(tLista & lista, const tRegistro & registro) {  
    bool ok = true;  
    if (lista.cont == TM) ok = false;  
    else {  
        *(lista.registros[lista.cont]) = registro;  
        ++lista.cont;  
    }  
    return ok;  
}
```

No usamos `new`,  
se crearon  
todos los registros  
al iniciar

```
bool eliminar(tLista & lista, int code) {  
    int pos; bool ok = true;  
    if (buscar(lista, code, pos)) {  
        tRegistro * aux = lista.registros[pos];  
        desplazarIzq(lista, pos);  
        --lista.cont;  
        lista.registros[cont] = aux;  
    } else ok = false;  
    return ok;  
}
```

No usamos `delete`,  
se destruirán  
todos los registros  
al final



# Ejemplo: Una lista con variables dinámicas

```
void iniciar(tLista & lista) {  
    lista.cont = 0;  
}
```

Al iniciar no creamos ninguno

```
bool insertar(tLista & lista, const tRegistro & registro) {  
    bool ok = true;  
    if (lista.cont == TM) ok = false;  
    else {  
        lista.registros[lista.cont] = new tRegistro;  
        *(lista.registros[lista.cont]) = registro;  
        ++lista.cont;  
    }  
    return ok;  
}
```

Usamos `new`,  
a medida que se insertan

```
bool eliminar(tLista & lista, int code) {  
    int pos; bool ok = true;  
    if (buscar(lista, code, pos)) {  
        delete lista.registros[pos];  
        desplazarIzq(lista, pos);  
        --lista.cont;  
    } else ok = false;  
    return ok;  
}
```

Usamos `delete`,  
a medida que se eliminan



# Ejemplo: Una lista con variables dinámicas

```
void destruir(tLista & lista) {  
    for (int i = 0; i < TM; ++i)  
        delete lista.registros[i];  
    lista.cont = 0;  
}
```

Destruimos todos

-----

Destruimos los que queden

```
void destruir(tLista & lista) {  
    for (int i = 0; i < lista.cont; ++i)  
        delete lista.registros[i];  
    lista.cont = 0;  
}
```



# Ejemplo: Una lista con variables dinámicas

Suponiendo que la lista sea ordenada

```
bool buscar(const tLista & lista, int code, int & pos) {
    pos = 0; int ini = 0, fin = lista.cont - 1, mitad;
    bool encontrado = false;
    while ((ini <= fin) && !encontrado) {
        mitad = (ini + fin) / 2;
        if (code < lista.registros[pos]->codigo) ini = mitad - 1;
        else if (lista.registros[pos]->codigo < code) fin = mitad + 1;
        else encontrado = true;
    }
    if (encontrado) pos = mitad; else pos = ini;
    return encontrado;
}

void mostrar(const tLista & lista) {
    cout << endl << "Elementos de la lista:" << endl
         << "-----" << endl;
    for (int i = 0; i < lista.cont; i++)
        mostrar(*(lista.registros[i]));
}
```





# Ejemplo: Una lista con variables dinámicas

```
#include <iostream>
using namespace std;
#include "registro.h"
#include "listaDD.h"

int main() {
    tLista lista;
    iniciar(lista);
    if (cargar(lista)) {
        mostrar(lista);
        ...
    }
    destruir(lista);
    return 0;
}
```

```
D:\FP\Tema9>listadinamica
```

```
Elementos de la lista:
```

```
-----
12345 - Disco duro - 123.59 euros
324356 - Placa base core i7 - 234.50 euros
2121 - Multipuerto USB - 15.00 euros
54354 - Disco externo 500 Gb - 95.00 euros
112341 - Procesador AMD - 132.95 euros
66678325 - Marco digital 2 Gb - 78.99 euros
600673 - Monitor 22" Nisu - 154.50 euros
```



# Arrays dinámicos: `new[]` y `delete[]`

Un array dinámico es un array que se mantiene en el montón.

*Creación: el tamaño no tiene que ser constante.*

`tipoBase *ad = new tipoBase[n];` -> tamaño `sizeof(tipoBase)*n`

Se crea un array de `n` elementos de tipo `tipoBase` en el montón

Se utiliza como los arrays estáticos: con `ad[i]`.

```
int main() {  
    int n; ...  
    int *p = new int[n];  
    for (int i = 0; i < n; i++) p[i] = i;  
    for (int i = 0; i < n; i++) cout << p[i] << endl;  
    delete [] p;  
    return 0;  
}
```

Creación del array dinámico  
de `n` elementos de tipo `int`

Destrucción del array dinámico



# Arrays dinámicos: ejemplo

---

El coste del desplazamiento de los datos depende de su tamaño (tipo).

El desplazamiento cambia la ubicación en memoria.

```
...  
#include "registro.h"  
  
const int TM = ...;  
  
// Lista: array dinámico y contador  
typedef struct {  
    tRegistro * registros;  
    int cont;  
    int tamaño; ←  
} tLista;  
  
...
```



# Arrays dinámicos: ejemplo

---

```
#include <iostream>
using namespace std;
#include "registro.h"
#include "listaAD.h"

int main() {
    tLista lista;
    iniciar(lista);

    if (cargar(lista)) {
        ...
        mostrar(lista);
    }

    destruir(lista);
    return 0;
}
```



# Arrays dinámicos: ejemplo

---

```
void iniciar(tLista & lista){  
    lista.registros = new registros[TM];  
    lista.cont = 0;  
    lista.tamaño = TM;  
}
```

Una vez creado el array  
se comporta como uno estático  
(salvo su ubicación)

Hasta que se destruye

```
void destruir(tLista &lista){  
    delete[] lista.registros;  
    lista.registros = nullptr;  
    lista.cont = 0;  
    lista.tamaño = 0;  
}
```



# Arrays dinámicos: ejemplo

```
bool insertar(tLista & lista, const tRegistro & registro) {  
    bool ok = true;  
    if (lista.cont == lista.tamaño) ←  
        ok = false;  
    else {  
        lista.registros[lista.cont] = registro;    // al final!  
        lista.cont++;  
    }  
    return ok;  
}
```

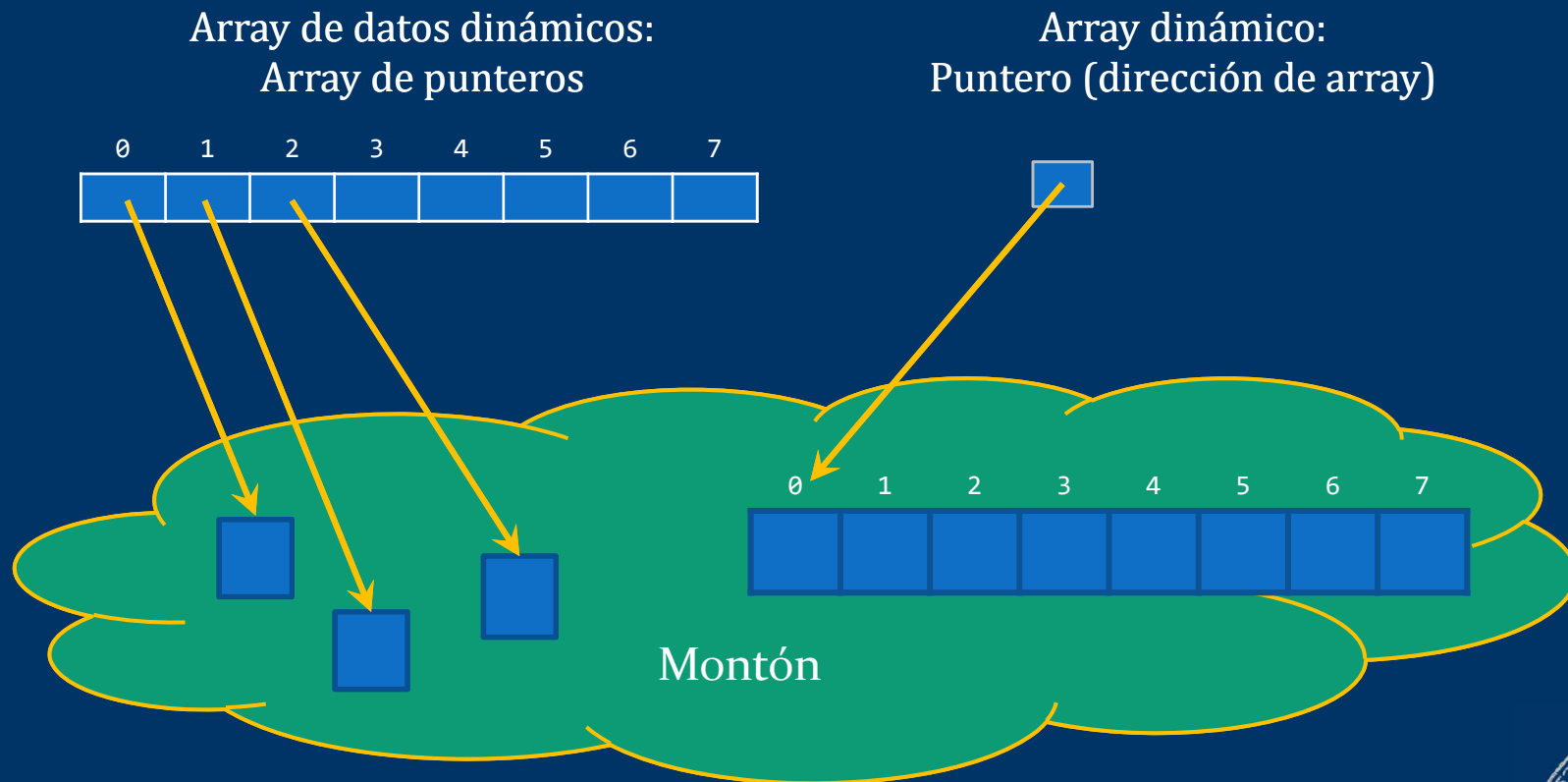
```
bool eliminar(tLista & lista, int code) {  
    int pos; bool ok = true;  
    if (buscar(lista, code, pos)) {  
        desplazarIzq(lista, pos);  
        lista.cont--;  
    } else ok = false;  
    return ok;  
}
```

El desplazamiento es poco eficiente  
y cambia la ubicación de los datos



# Arrays dinámicos vs. arrays de dinámicas

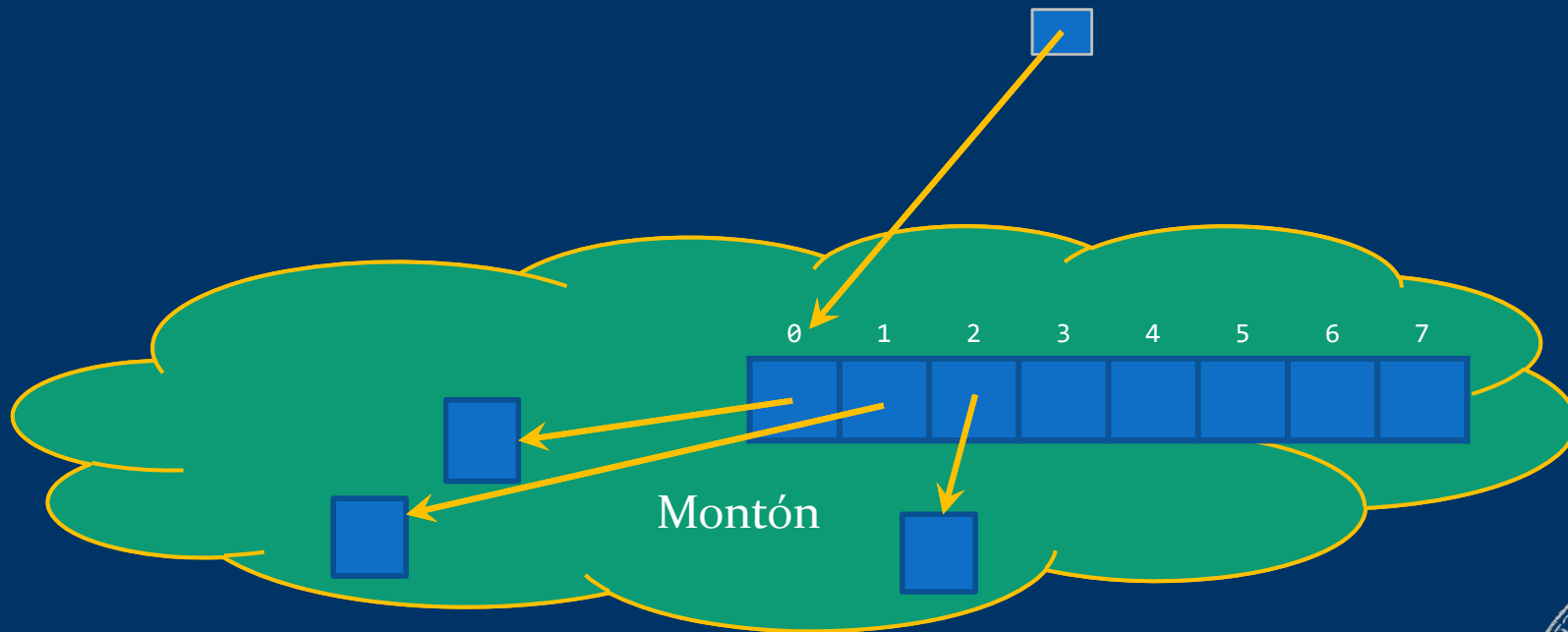
El array de punteros puede estar en la pila y los datos en el montón.  
El array dinámico se crea entero en el montón.



# Arrays dinámicos de variables dinámicas

```
tipoBase * * p = new tipoBase*[n];  
p[0] = new tipoBase;  
p[1] = new tipoBase;  
p[2] = new tipoBase;
```

Array dinámico de punteros:  
puntero p (dirección de array)








# Acerca de *Creative Commons*



## *Licencia CC (Creative Commons)*

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

-  Reconocimiento (*Attribution*):  
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
-  No comercial (*Non commercial*):  
La explotación de la obra queda limitada a usos no comerciales.
-  Compartir igual (*Share alike*):  
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Pulsa en la imagen de arriba a la derecha para saber más.

