

4

## Punteros y Memoria Dinámica

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

Luis Hernández Yáñez  
Facultad de Informática  
Universidad Complutense



# Índice

---

Direcciones de memoria y punteros	849
Operadores de punteros	854
Punteros y direcciones válidas	864
Punteros no inicializados	866
Un valor seguro: NULL	867
Copia y comparación de punteros	868
Tipos puntero	873
Punteros a estructuras	875
Punteros a constantes y punteros constantes	877
Punteros y paso de parámetros	879
Punteros y arrays	883
Memoria y datos del programa	886
Memoria dinámica	891
Punteros y datos dinámicos	895
Gestión de la memoria	907
Errores comunes	911
Arrays de datos dinámicos	916
Arrays dinámicos	928



## Direcciones de memoria y punteros



# Direcciones de memoria

## *Los datos en la memoria*

Todo dato se almacena en memoria:

Varios bytes a partir de una dirección

```
int i = 5;
```



El dato (*i*) se accede a partir de su *dirección base* (0F03:1A38)

Dirección de la primera celda de memoria utilizada por el dato

El tipo del dato (*int*) indica cuántos bytes (4) requiere el dato:

00000000 00000000 00000000 00000101 → 5

(La codificación de los datos puede ser diferente; y la de las direcciones también)

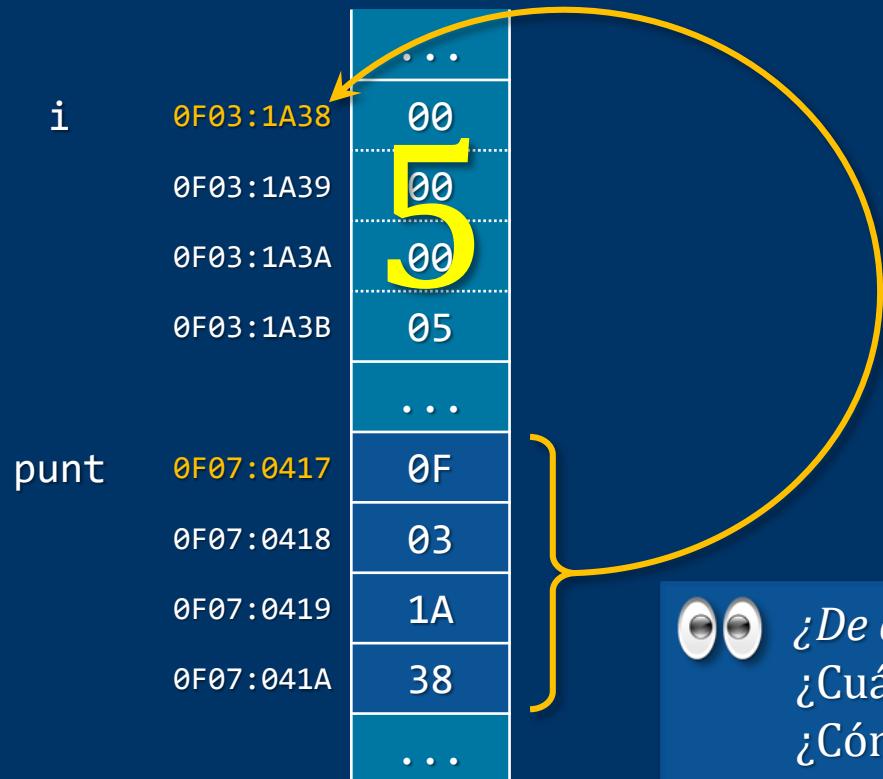


# Variables punteros

*Los punteros contienen direcciones de memoria*

Un *puntero* sirve para acceder a través de él a otro dato

El valor del puntero es la dirección de memoria base de otro dato



## *Indirección:* Acceso indirecto a un dato

punt apunta a i

*¿De qué tipo es el dato apuntado?  
¿Cuántas celdas ocupa?  
¿Cómo se interpretan los 0/1?*



## *Los punteros contienen direcciones de memoria*

¿De qué tipo es el dato apuntado?

La variable a la que apunta un puntero será de un tipo concreto

¿Cuánto ocupa? ¿Cómo se interpreta?

El tipo de variable apuntado se establece al declarar el puntero:

*tipo \*nombre;*

El puntero *nombre* apuntará a una variable del *tipo* indicado

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

*int \*punt; // punt inicialmente contiene una dirección  
// que no es válida (no apunta a nada)*

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

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



## *Los punteros contienen direcciones de memoria*

Las variables puntero tampoco se inicializan automáticamente

Al declararlas sin inicializador contienen direcciones no válidas

```
int *punt; // punt inicialmente contiene una dirección  
           // que no es válida (no apunta a nada)
```

Un puntero puede apuntar a cualquier dato de su tipo base

Un puntero no tiene por qué apuntar necesariamente a un dato  
(puede no apuntar a nada: valor **NULL**)

### *¿Para qué sirven los punteros?*

- ✓ Para implementar el paso de parámetros por referencia
- ✓ Para manejar datos dinámicos  
(Datos que se crean y destruyen durante la ejecución)
- ✓ Para implementar los arrays



## Operadores de punteros



# Operadores de punteros

&

*Obtener la dirección de memoria de ...*

Operador monario y prefijo

& devuelve la dirección de memoria base del dato al que precede

```
int i;
```

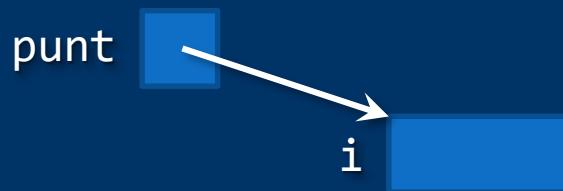
```
cout << &i; // Muestra la dirección de memoria de i
```

Un puntero puede recibir la dirección de datos de su tipo base

```
int i;
```

```
int *punt;
```

```
punt = &i; // punt contiene la dirección de i
```



Ahora **punt** ya contiene una dirección de memoria válida

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

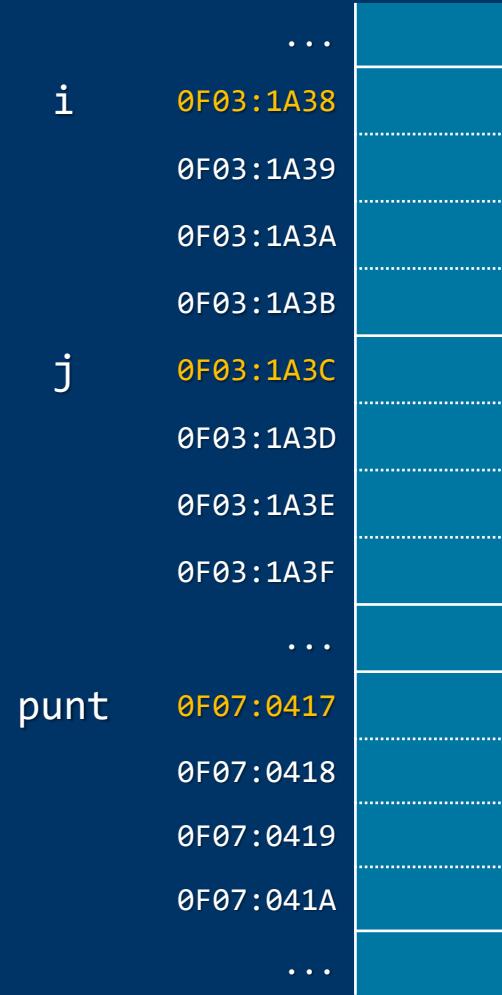


# Operadores de punteros

&

*Obtener la dirección de memoria de ...*

```
int i, j;  
...  
int *punt;
```



# Operadores de punteros

&

*Obtener la dirección de memoria de ...*

```
int i, j;  
...  
int *punt;  
...  
i = 5;
```

i 5

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	
	0F07:0418	
	0F07:0419	
	0F07:041A	
	...	

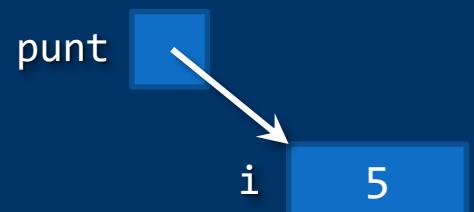


# Operadores de punteros

&

*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
	...	



# Operadores de punteros

---

\*

*Obtener lo que hay en la dirección ...*

Operador monario y prefijo

\* accede a lo que hay en la dirección de memoria a la que precede

Permite acceder a un dato a través un puntero que lo apunte:

punt = &i;

cout << \*punt; // Muestra lo que hay en la dirección punt

\*punt: lo que hay en la dirección que contiene el puntero punt

punt contiene la dirección de memoria de la variable i

\*punt accede al contenido de esa variable i

*Acceso indirecto* al valor de i



# Operadores de punteros

\*

*Obtener lo que hay en la dirección ...*

```
int i, j;  
...  
int *punt;  
...  
i = 5;  
punt = &i;  
j = *punt;
```

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
	...	...	



# Operadores de punteros

\*

*Obtener lo que hay en la dirección ...*

```
int i, j;  
...  
int *punt;  
...  
i = 5;  
punt = &i;  
j = *punt;
```

\*punt:

Direccionamiento  
indirecto  
(*dirección*)  
Se accede al dato **i**  
de forma indirecta

...	...	...
	→ 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
		...



# Operadores de punteros

\*

*Obtener lo que hay en la dirección ...*

```
int i, j;  
...  
int *punt;  
...  
i = 5;  
punt = &i;  
j = *punt;
```



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



## Ejemplo de uso de punteros

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

int main() {
    int i = 5;
    int j = 13;
    int *punt;
    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;
}
```

5  
13  
18  
13



## Punteros y direcciones válidas



# Punteros y direcciones válidas

---

*Todo puntero ha de tener una dirección válida*

Un puntero sólo debe ser utilizado si tiene una dirección válida

Un puntero NO contiene una dirección válida tras ser definido

Un puntero obtiene una dirección válida:

- ✓ Asignando la dirección de otro dato (operador &)
- ✓ Asignando otro puntero (mismo tipo base) que ya sea válido
- ✓ Asignando el valor **NULL** (puntero nulo, no apunta a nada)

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



# Punteros no inicializados

---

*Punteros que apuntan a saber qué...*

Un puntero no inicializado contiene una dirección desconocida

```
int *punt; // No inicializado  
*punt = 12; // ¿A qué dato se está asignando el valor?
```

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

¡Podemos modificar inadvertidamente un dato del programa!

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

¡Podemos modificar el código del propio programa!

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

¡Podemos modificar el código del propio S.O.!

→ Consecuencias imprevisibles (*cuelgue*)

(Los S.O. modernos protegen bien la memoria)

¡SALVAJES!



# Un valor seguro: NULL

---

*Punteros que no apuntan a nada*

Inicializando los punteros a **NULL** podemos detectar errores:

```
int *punt = NULL;  
...  
*punt = 13;
```

punt X

punt ha sido inicializado a **NULL**: ¡No apunta a nada!

Si no apunta a nada, ¿¿qué significa **\*punt**??? No tiene sentido

→ ERROR: *¡Acceso a un dato a través de un puntero nulo!*

Error de ejecución, lo que ciertamente no es bueno

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

Sabemos dónde y qué buscar para depurar



## Copia y comparación de punteros

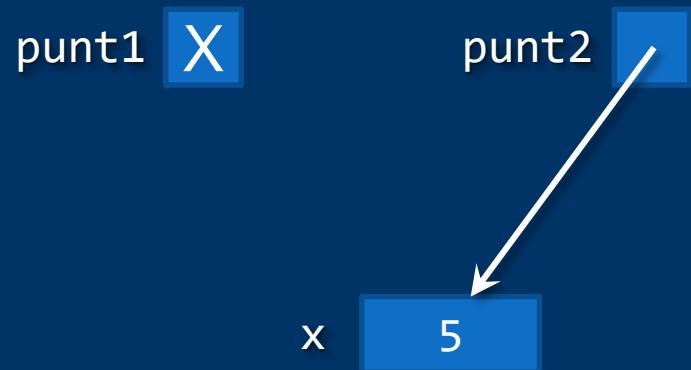


# Copia de punteros

## *Apuntando al mismo dato*

Al copiar un puntero en otro, ambos apuntarán al mismo dato:

```
int x = 5;  
int *punt1 = NULL; // punt1 no apunta a nada  
int *punt2 = &x; // punt2 apunta a la variable x
```



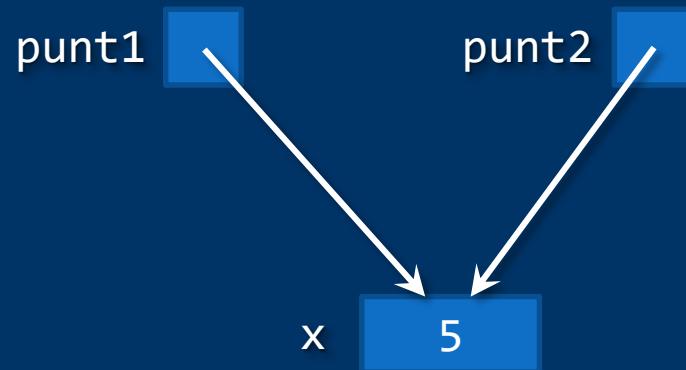
# Copia de punteros

---

## *Apuntando al mismo dato*

Al copiar un puntero en otro, ambos apuntarán al mismo dato:

```
int x = 5;  
int *punt1 = NULL; // punt1 no apunta a nada  
int *punt2 = &x; // punt2 apunta a la variable x  
punt1 = punt2; // ambos apuntan a la variable x
```

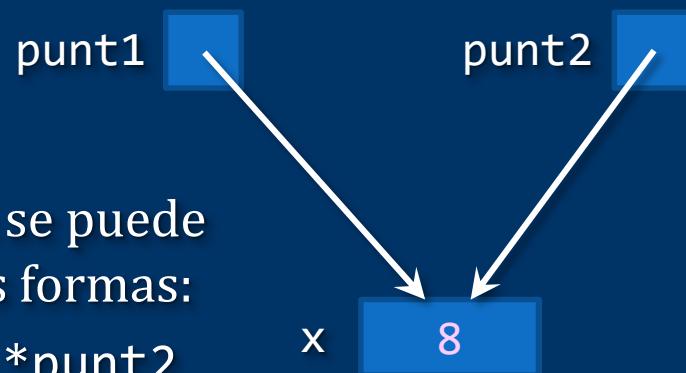


# Copia de punteros

## *Apuntando al mismo dato*

Al copiar un puntero en otro, ambos apuntarán al mismo dato:

```
int x = 5;  
int *punt1 = NULL; // punt1 no apunta a nada  
int *punt2 = &x; // punt2 apunta a la variable x  
punt1 = punt2; // ambos apuntan a la variable x  
*punt1 = 8;
```



Al dato x ahora se puede acceder de tres formas:

x \*punt1 \*punt2



# Comparación de punteros

---

*¿Apuntan al mismo dato?*

Operadores relacionales == y !=:

```
int x = 5;
int *punt1 = NULL;
int *punt2 = &x;

...
if (punt1 == punt2) {
    cout << "Apuntan al mismo dato" << endl;
}
else {
    cout << "No apuntan al mismo dato" << endl;
}
```



Sólo se pueden comparar punteros con el mismo tipo base



## Tipos puntero



## *Declaración de tipos puntero*

Declaramos tipos para los punteros con distintos tipos base:

```
typedef int *tIntPtr;
typedef char *tCharPtr;
typedef double *tDoublePtr;
int entero = 5;
tIntPtr puntI = &entero;
char caracter = 'C';
tCharPtr puntC = &caracter;
double real = 5.23;
tDoublePtr puntD = &real;
cout << *puntI << " " << *puntC << " " << *puntD << endl;
```

Con *\*puntero* podemos hacer lo que con otros datos del tipo base



# Punteros a estructuras

---

## *Acceso a estructuras a través de punteros*

Los punteros pueden apuntar también a estructuras:

```
typedef struct {
    int codigo;
    string nombre;
    double sueldo;
} tRegistro;
tRegistro registro;
typedef tRegistro *tRegistroPtr;
tRegistroPtr puntero = &registro;
```

Operador flecha (->):

Acceso a los campos a través de un puntero sin usar el operador \*

puntero->codigo      puntero->nombre      puntero->sueldo  
puntero->... ≡ (\*puntero)....



## Acceso a estructuras a través de punteros

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

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

puntero sería una estructura con campo codigo de tipo puntero



# Punteros y el modificador `const`

---

## *Punteros a constantes y punteros constantes*

El efecto del modificador de acceso `const` depende de su sitio:

`const tipo *puntero;` Puntero a una constante

`tipo *const puntero;` Puntero constante

Punteros a constantes:

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

```
(*punt_a_cte)++; // ERROR: ¡Dato no modificable!
punt_a_cte = &entero2; // OK: El puntero no es cte.
```



## *Punteros a constantes y punteros constantes*

El efecto del modificador de acceso `const` depende de su sitio:

`const tipo *puntero;` Puntero a una constante

`tipo *const puntero;` Puntero constante

Punteros constantes:

```
typedef int *const tIntPtrCte; // Puntero constante
```

```
int entero1 = 5, entero2 = 13;
```

```
tIntPtrCte punt_cte = &entero1;
```

```
(*punt_cte)++; // OK: El puntero no apunta a cte.
```

```
punt_cte = &entero2; // ERROR: ¡Puntero constante!
```



## Punteros y paso de parámetros



## *Paso de parámetros por referencia o variable*

En el lenguaje C no hay mecanismo de paso por referencia (&)

Sólo se pueden pasar parámetros por valor

¿Cómo se simula el paso por referencia? Por medio de punteros:

```
void incrementa(int *punt);
```

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

Mostrará 6 en la consola

Paso por valor:  
El argumento (el puntero) no cambia  
Aquellos a lo que apunta (el entero)  
SÍ puede cambiar



# Punteros y paso de parámetros

*Paso de parámetros por referencia o variable*

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

entero 5

punt recibe la dirección de entero

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

punt 6  
entero 6

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

entero 6



# Punteros y paso de parámetros

---

*Paso de parámetros por referencia o variable*

¿Cuál es el equivalente en C a este prototipo de C++?

```
void foo(int &param1, double &param2, char &param3);
```

Prototipo equivalente:

```
void foo(int *param1, double *param2, char *param3);
```

```
void foo(int *param1, double *param2, char *param3) {  
    // Al primer argumento se accede con *param1  
    // Al segundo argumento se accede con *param2  
    // Al tercer argumento se accede con *param3  
}
```

¿Cómo se llamaría?

```
int entero; double real; char caracter;  
//...  
foo(&entero, &real, &caracter);
```



## Punteros y arrays



# Punteros y arrays

---

## *Una íntima relación*

Variable array  $\equiv$  Puntero al primer elemento del array

Así, 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



¡Un nombre de array es un puntero constante!

Siempre apunta al primer elemento (no se puede modificar)

Acceso a los elementos del array:

Por índice o con aritmética de punteros (Anexo)



# Punteros y paso de parámetros arrays

---

## *Paso de arrays a subprogramas*

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

El nombre del array es un puntero: ya es un paso por referencia

Prototipos equivalentes para parámetros array:

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

Arrays no delimitados y punteros: se necesita la dimensión

Elementos: se acceden con índice (`arr[i]`) o con puntero (`*arr`)

Una función sólo puede devolver un array en forma de puntero:

```
intPtr inicializar();
```



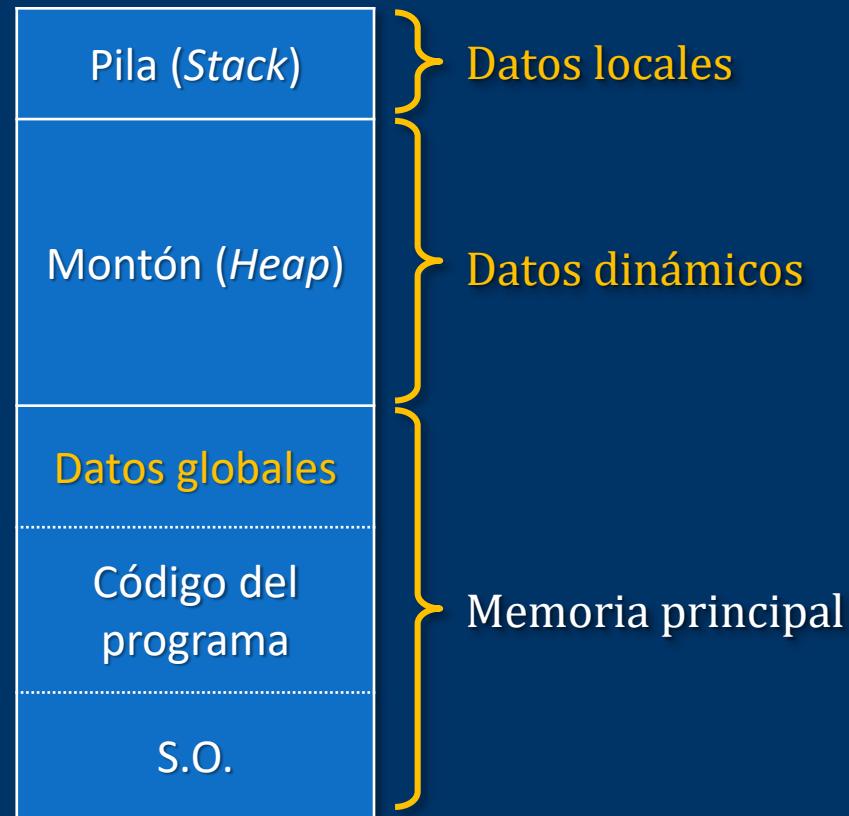
## Memoria y datos del programa



# Memoria y datos del programa

## *Regiones de la memoria*

El sistema operativo distingue varias regiones en la memoria:

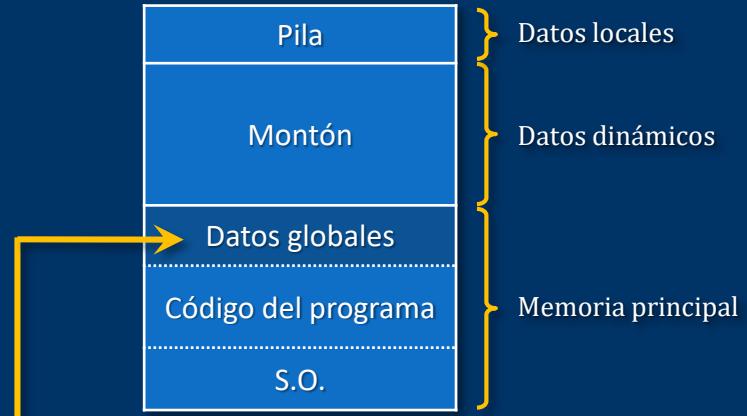


# Memoria y datos del programa

## *Memoria principal*

Datos globales del programa:  
Declarados fuera  
de los subprogramas

```
typedef struct {  
    ...  
} tRegistro;  
const int N = 1000;  
typedef tRegistro tArray[N];  
typedef struct {  
    tArray registros;  
    int cont;  
} tLista;  
  
int main() {  
    ...
```



# Memoria y datos del programa

## *La pila (stack)*

Datos locales de subprogramas:  
Parámetros por valor  
y variables locales

```
void func(tLista lista, double &total)
{
    tLista aux
    int i
    ...
}
```



Y los punteros temporales  
que apuntan a los argumentos  
de los parámetros por referencia

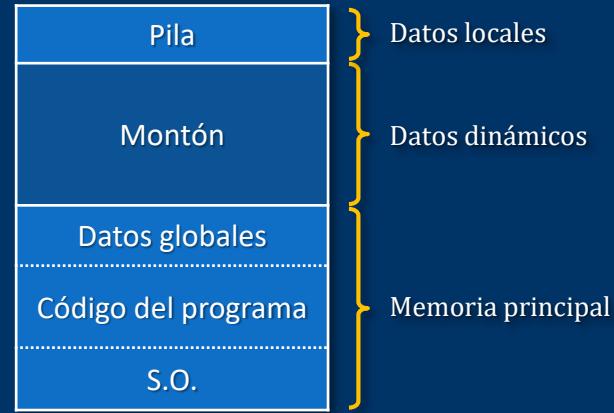


# Memoria y datos del programa

## *El montón (heap)*

### *Datos dinámicos*

Datos que se crean y se destruyen durante la ejecución del programa, a medida que se necesita



### Sistema de gestión de memoria dinámica (SGMD)

Cuando se necesita memoria para una variable se solicita

El SGMD reserva espacio y devuelve la dirección base

Cuando ya no se necesita más la variable, se destruye

Se libera la memoria y el SGMD cuenta de nuevo con ella



## Memoria dinámica



# Memoria dinámica

## Datos dinámicos

Se crean y se destruyen durante la ejecución del programa

Se les asigna memoria del montón



¿Por qué utilizar memoria dinámica?

- ✓ Almacén de memoria muy grande: datos o listas de datos que no caben en memoria principal pueden caber en el montón
- ✓ El programa ajusta el uso de la memoria a las necesidades de cada momento: ni le falta ni la desperdicia



# Datos y asignación de memoria

---

## *¿Cuándo se asigna memoria a los datos?*

- ✓ Datos **globales**

En memoria principal al comenzar la ejecución del programa

Existen durante toda la ejecución del programa

- ✓ Datos **locales** de un subprograma

En la pila al ejecutarse el subprograma

Existen sólo durante la ejecución de su subprograma

- ✓ Datos **dinámicos**

En el montón cuando el programa lo solicita

Existen *a voluntad* del programa



# Datos estáticos frente a datos dinámicos

---

## *Datos estáticos*

- ✓ Datos declarados como de un tipo concreto:  
`int i;`
- ✓ Se acceden directamente a través del identificador:  
`cout << i;`

## *Datos dinámicos*

- ✓ Datos accedidos a través de su dirección de memoria  
Esa dirección de memoria debe estar en algún puntero  
Los punteros son la base del SGMD

Los datos estáticos también se pueden acceder a través de punteros

`int *p = &i;`



## Punteros y datos dinámicos



# Creación de datos dinámicos

---

## *El operador new*

Devuelve **NULL** si no queda memoria suficiente

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

```
int *p; // Todavía sin una dirección válida  
p = new int; // Ya tiene una dirección válida  
*p = 12;
```

La variable dinámica se accede exclusivamente por punteros

No tiene identificador asociado

```
int i; // i es una variable estática  
int *p1, *p2;  
p1 = &i; // Puntero que da acceso a la variable  
        // estática i (accesible con i o con *p1)  
p2 = new int; // Puntero que da acceso a una variable  
              // dinámica (accesible sólo a través de p2)
```



## *Inicialización con el operador new*

El operador `new` admite un valor inicial para el dato creado:

```
int *p;  
p = new int(12);
```

Se crea la variable, de tipo `int`, y se inicializa con el valor 12

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

```
int main() {  
    tRegistro reg;  
    reg = nuevo();  
    tRegistro *punt = new tRegistro(reg);  
    mostrar(*punt);  
    ...  
}
```



# Eliminación de datos dinámicos

---

## *El operador delete*

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

```
int *p;  
p = new int;  
*p = 12;  
...  
delete p; // Ya no se necesita el entero apuntado por p
```



¡El puntero deja de contener una dirección válida!



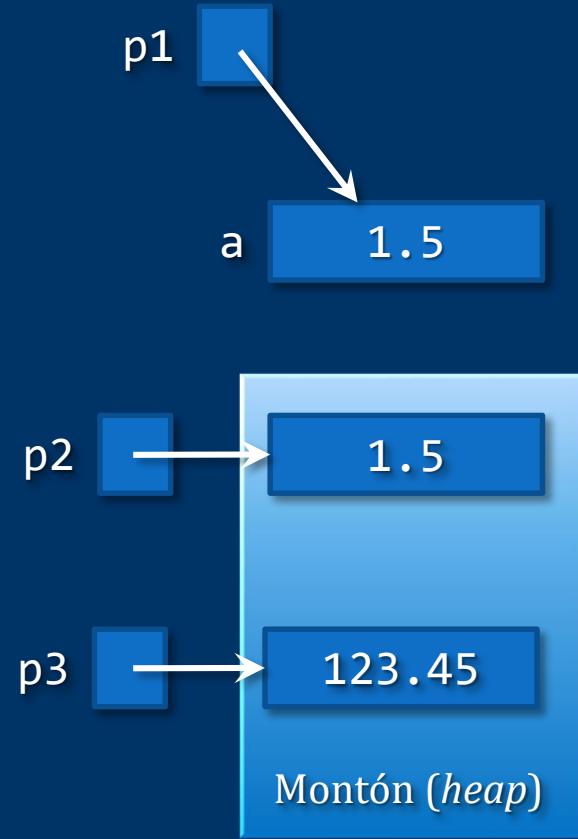
# Ejemplo de variables dinámicas

dinamicas.cpp

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

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

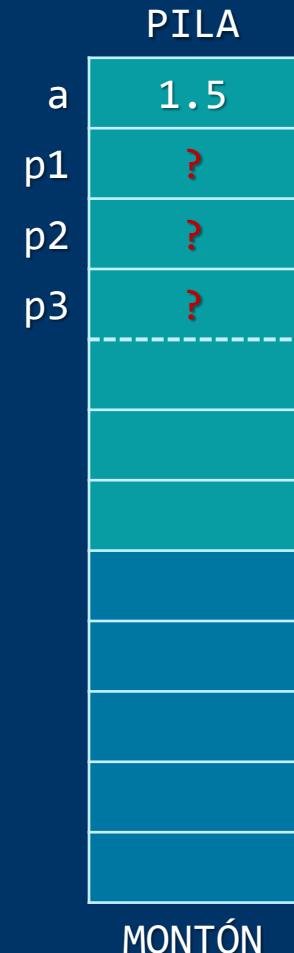
(+ \*p2 y \*p3)



# Ejemplo de variables dinámicas

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

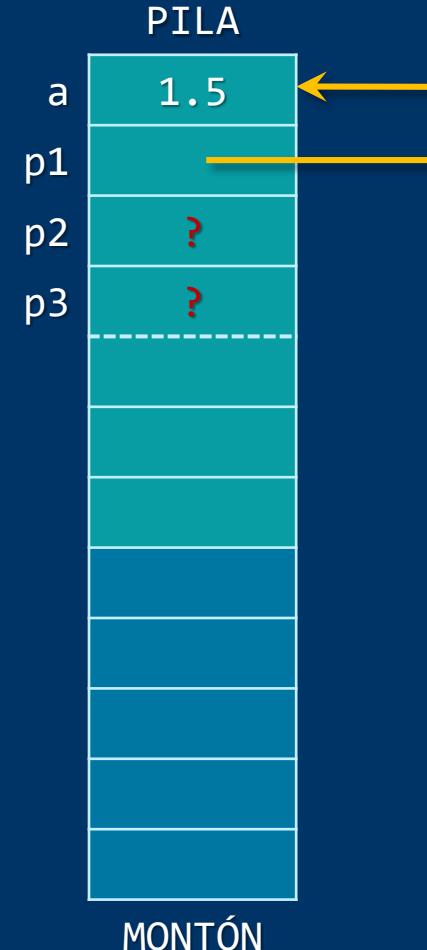
int main() {
    double a = 1.5;
    double *p1, *p2, *p3;
```



# Ejemplo de variables dinámicas

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

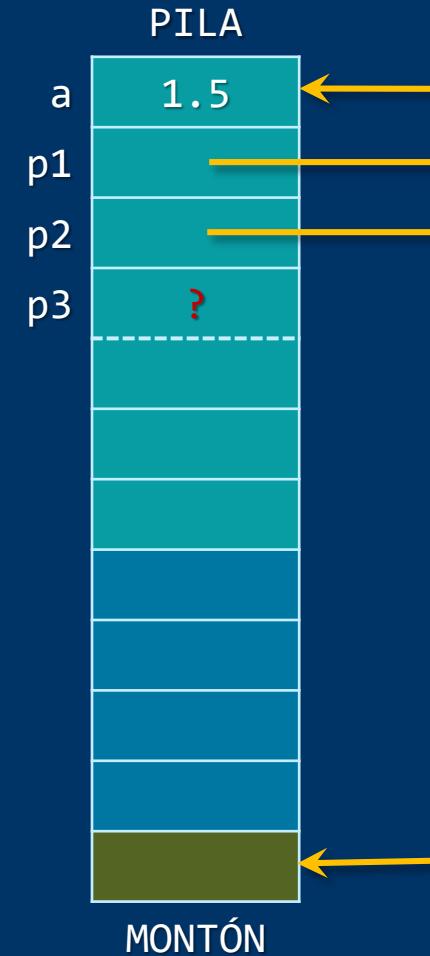
int main() {
    double a = 1.5;
    double *p1, *p2, *p3;
    p1 = &a;
```



# Ejemplo de variables dinámicas

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

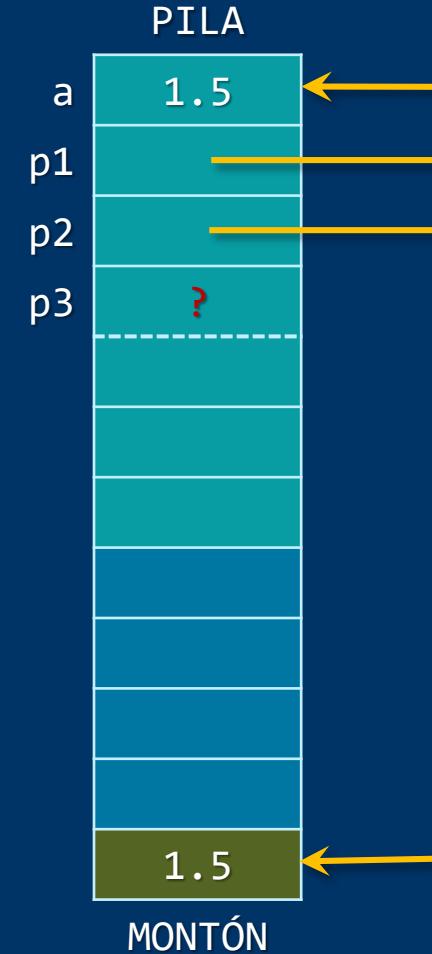
int main() {
    double a = 1.5;
    double *p1, *p2, *p3;
    p1 = &a;
    p2 = new double;
```



# Ejemplo de variables dinámicas

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

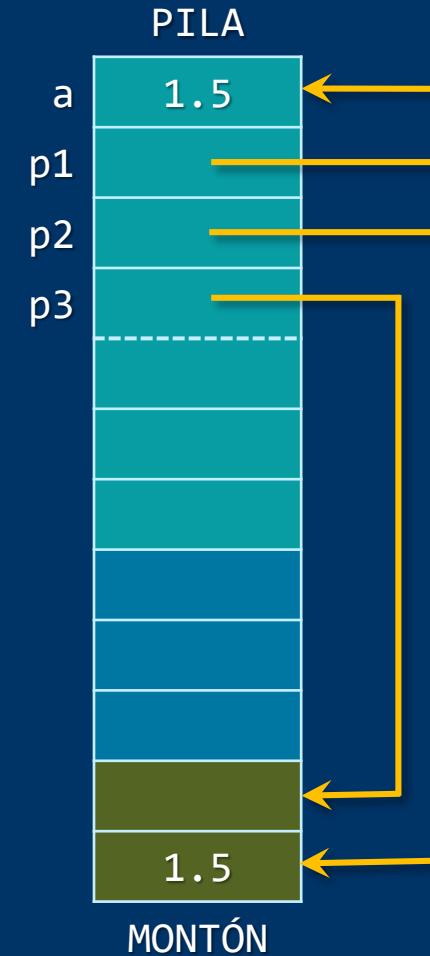
int main() {
    double a = 1.5;
    double *p1, *p2, *p3;
    p1 = &a;
    p2 = new double;
    *p2 = *p1;
```



# Ejemplo de variables dinámicas

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

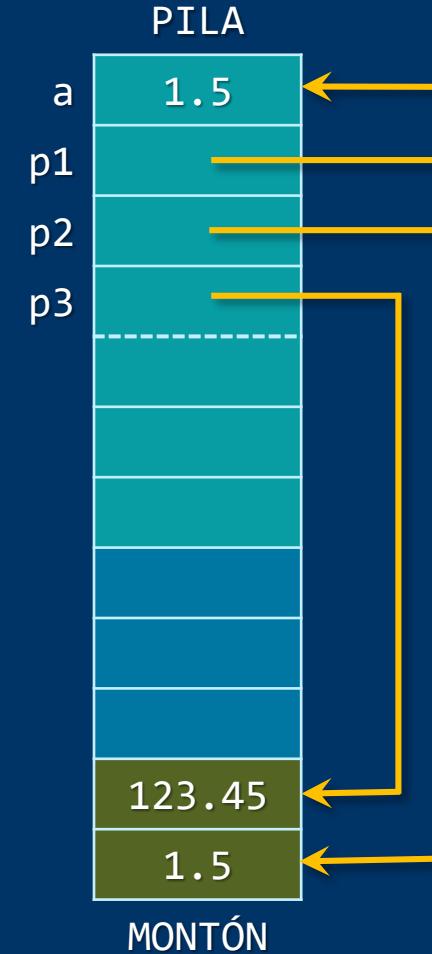
int main() {
    double a = 1.5;
    double *p1, *p2, *p3;
    p1 = &a;
    p2 = new double;
    *p2 = *p1;
    p3 = new double;
```



# Ejemplo de variables dinámicas

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

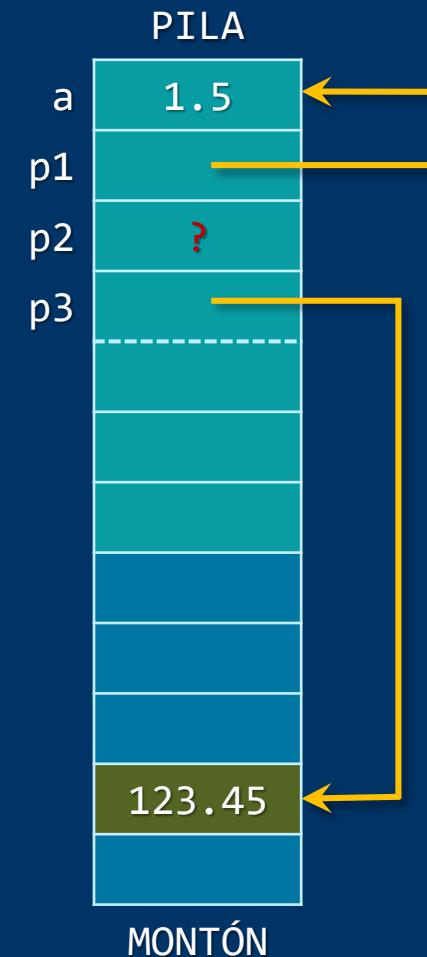
int main() {
    double a = 1.5;
    double *p1, *p2, *p3;
    p1 = &a;
    p2 = new double;
    *p2 = *p1;
    p3 = new double;
    *p3 = 123.45;
```



# Ejemplo de variables dinámicas

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

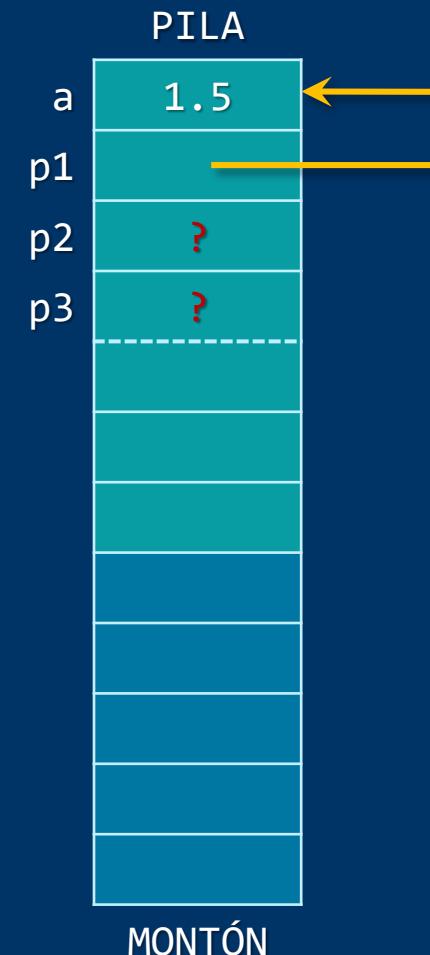
int main() {
    double a = 1.5;
    double *p1, *p2, *p3;
    p1 = &a;
    p2 = new double;
    *p2 = *p1;
    p3 = new double;
    *p3 = 123.45;
    cout << *p1 << endl;
    cout << *p2 << endl;
    cout << *p3 << endl;
    delete p2;
```



# Ejemplo de variables dinámicas

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

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;
```



## Gestión de la memoria



# Errores de asignación de memoria

La memoria se reparte entre la pila y el montón  
Crecen en direcciones opuestas

Al llamar a subprogramas la pila crece

Al crear datos dinámicos el montón crece



## *Colisión pila-montón*

Los límites de ambas regiones se encuentran

Se agota la memoria

## *Desbordamiento de la pila*

La pila suele tener un tamaño máximo establecido

Si se sobrepasa se agota la pila



# Gestión de la memoria dinámica

---

## *Gestión del montón*

Sistema de Gestión de Memoria Dinámica (SGMD)

Gestionar la asignación de memoria a los datos dinámicos

Localiza secciones adecuadas y sigue la pista de lo disponible

No dispone de un *recolector de basura*, como el lenguaje Java

*¡Hay que devolver toda la memoria solicitada!*

Deben ejecutarse tantos `delete` como `new` se hayan ejecutado

La memoria disponible en el montón debe ser exactamente la misma antes y después de la ejecución del programa

Y todo dato dinámico debe tener algún acceso (puntero)

Es un grave error *perder* un dato en el montón



## Errores comunes



# Mal uso de la memoria dinámica I

## *Olvido de destrucción de un dato dinámico*

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

 Falta `delete p;`

G++ no indicará ningún error y el programa parecerá terminar correctamente, pero dejará memoria desperdiciada

Visual C++ sí comprueba el uso de la memoria dinámica y nos avisa si dejamos memoria sin liberar



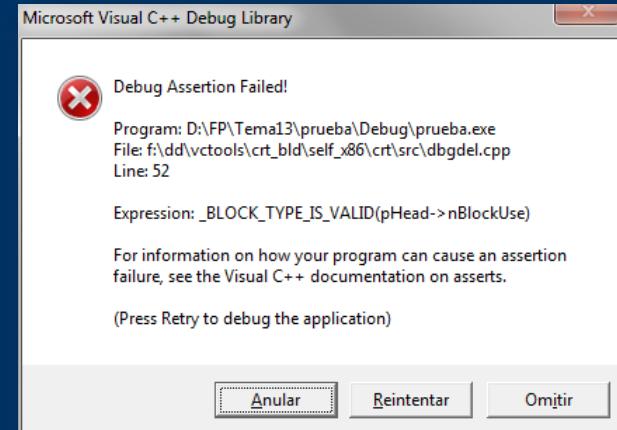
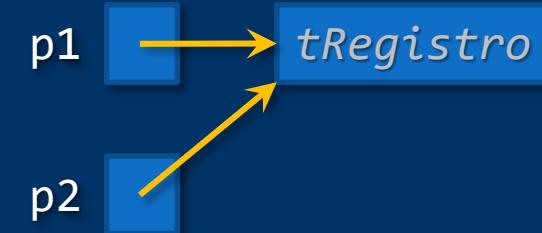
# Mal uso de la memoria dinámica II

*Intento de destrucción de un dato inexistente*

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



Sólo se ha creado  
una variable



# Mal uso de la memoria dinámica III

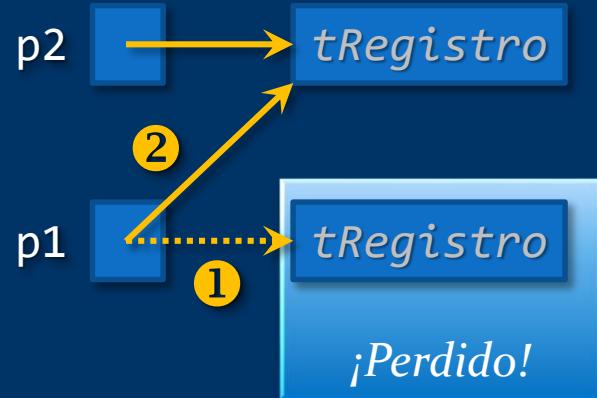
## Pérdida de un dato dinámico

```
...
int main() {
    tRegistro *p1, *p2;
    p1 = new tRegistro(nuevo());
    p2 = new tRegistro(nuevo()); 1

    mostrar(*p1);
    p1 = p2;
    mostrar(*p1); 2

    delete p1;
    delete p2;

    return 0;
}
```



Se pierde un dato en el montón  
Se intenta eliminar un dato ya eliminado



# Mal uso de la memoria dinámica IV

*Intento de acceso a un dato tras su eliminación*

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

 p ha dejado de apuntar  
al dato dinámico destruido  
→ Acceso a memoria inexistente



## Arrays de datos dinámicos



# Arrays de datos dinámicos

---

## *Arrays de punteros a datos dinámicos*

```
typedef struct {  
    int codigo;  
    string nombre;  
    double valor;  
} tRegistro;  
typedef tRegistro *tRegPtr;
```

Los punteros ocupan  
muy poco en memoria

Los datos a los que apunten  
estarán en el montón

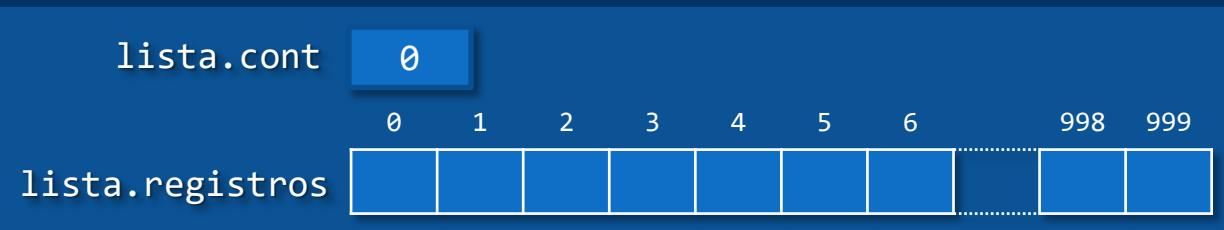
```
const int N = 1000;  
// Array de punteros a registros:  
typedef tRegPtr tArray[N];  
typedef struct {  
    tArray registros;  
    int cont;  
} tLista;
```

Se crean a medida que se insertan  
Se destruyen a medida que se eliminan



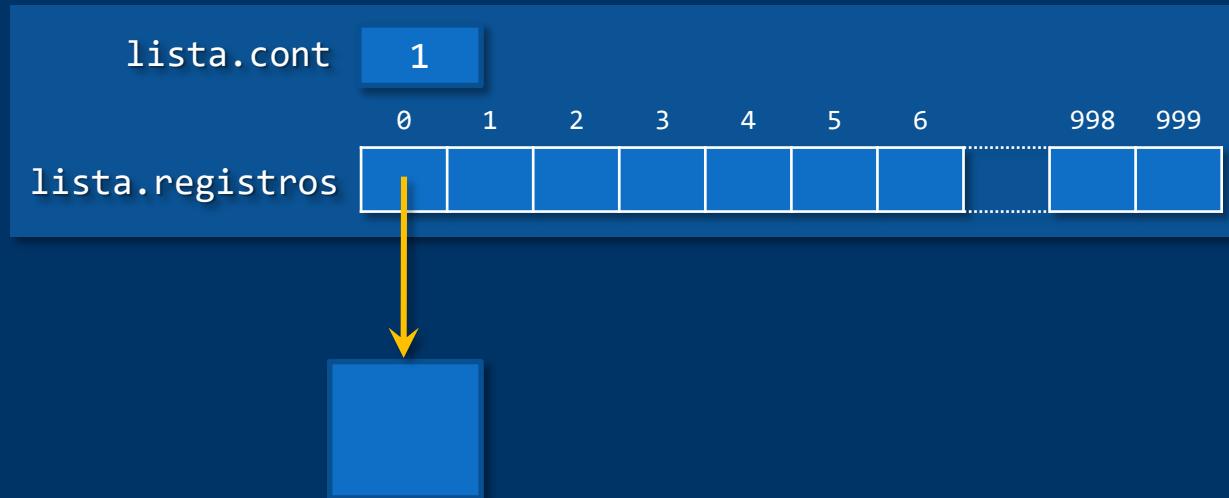
# Arrays de datos dinámicos

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



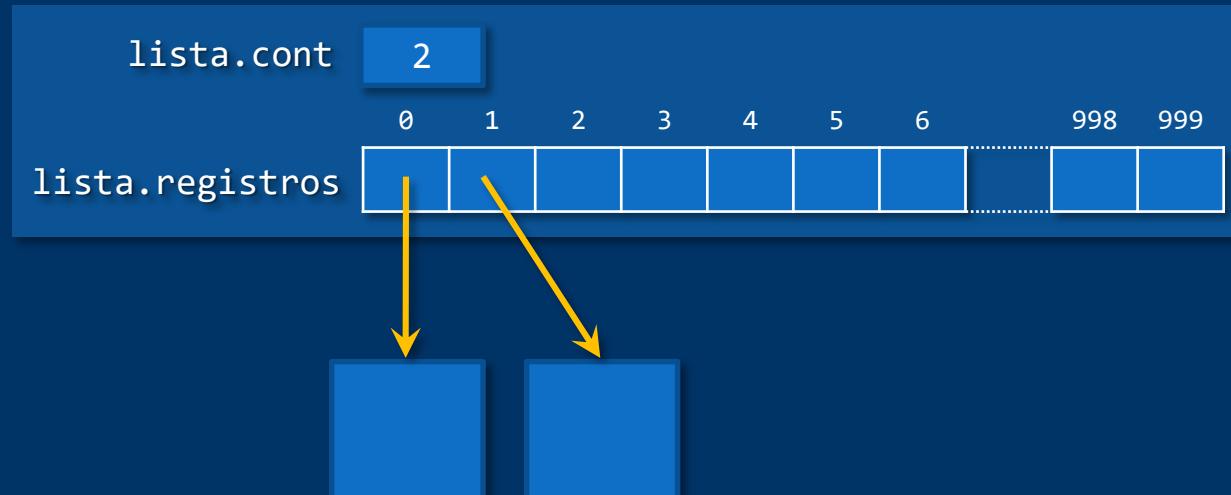
# Arrays de datos dinámicos

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



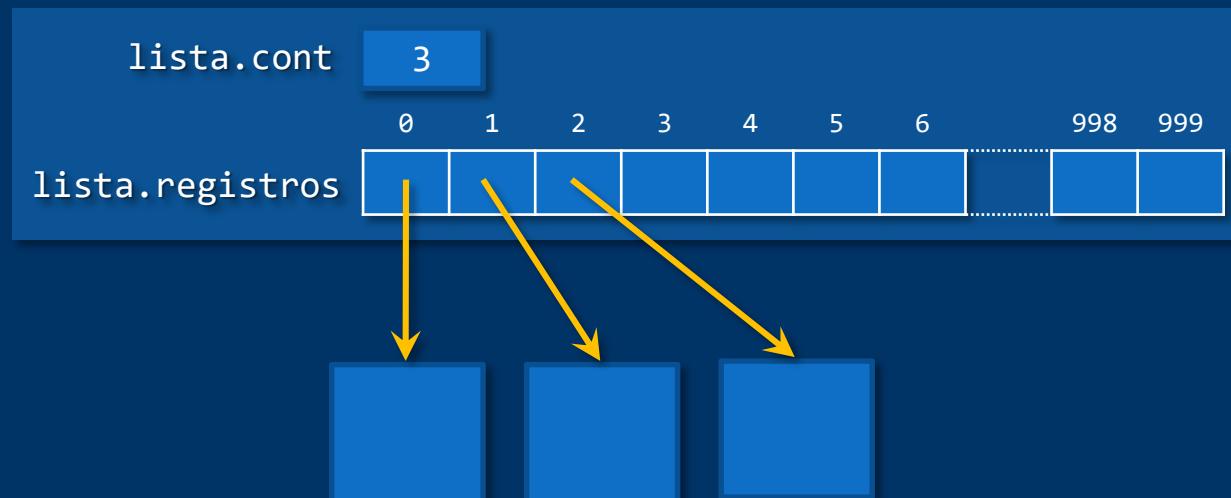
# Arrays de datos dinámicos

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



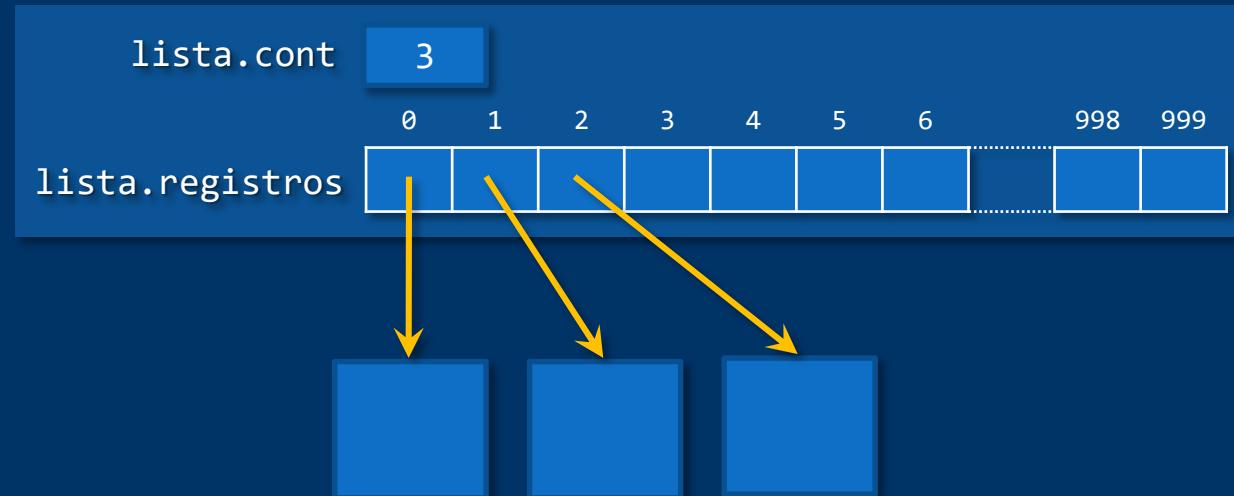
# Arrays de datos dinámicos

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



# Arrays de datos dinámicos

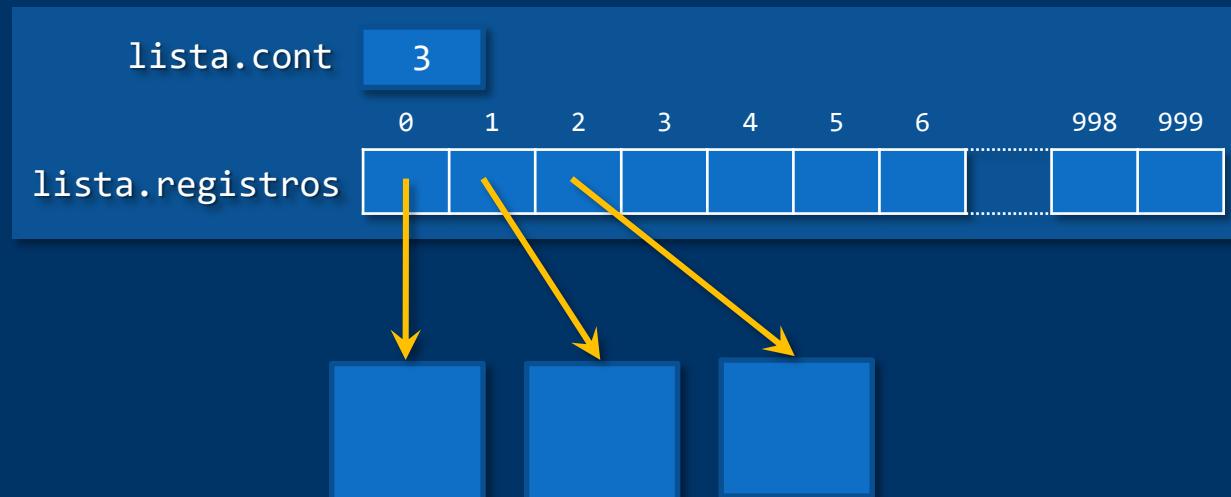
Los registros se acceden a través de los punteros (operador `->`):  
`cout << lista.registros[0]->nombre;`



# Arrays de datos dinámicos

No hay que olvidarse de devolver la memoria al montón:

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



# Arrays de datos dinámicos

lista.h

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

const int N = 1000;
const string BD = "bd.dat";
typedef tRegPtr tArray[N];
typedef struct {
    tArray registros;
    int cont;
} tLista;

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

#endif
```

registro.h con el tipo puntero:

```
typedef tRegistro *tRegPtr;
```



# Arrays de datos dinámicos

lista.cpp

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

void eliminar(tLista &lista, int code, bool &ok) {
    ok = true;
    int ind = buscar(lista, code);
    if (ind == -1) {
        ok = false;
    }
    else {
        delete lista.registros[ind];
        for (int i = ind + 1; i < lista.cont; i++) {
            lista.registros[i - 1] = lista.registros[i];
        }
        lista.cont--;
    }
}
```



# Arrays de datos dinámicos

```
int buscar(const tLista &lista, int code) {
    // Devuelve el índice o -1 si no se ha encontrado
    int ind = 0;
    bool encontrado = false;
    while ((ind < lista.cont) && !encontrado) {
        if (lista.registros[ind]->codigo == code) {
            encontrado = true;
        }
        else {
            ind++;
        }
    }
    if (!encontrado) {
        ind = -1;
    }
    return ind;
}

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

```



# Arrays de datos dinámicos

listadinamica.cpp

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

int main() {
    tLista lista;
    bool ok;
    cargar(lista, ok);
    if (ok) {
        mostrar(lista);
        destruir(lista);
    }

    return 0;
}
```

Elementos de la lista:

-----		
12345	- Disco duro	- 123.59 euros
324356	- Placa base core i7	- 234.50 euros
2121	- Multpuerto 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



# Arrays dinámicos

---

## *Creación y destrucción de arrays dinámicos*

Array dinámico: array que se ubica en la memoria dinámica

Creación de un array dinámico:

```
tipo *puntero = new tipo[dimensión];  
int *p = new int[10];
```

Crea un array de 10 **int** en memoria dinámica

Los elementos se acceden a través del puntero:  $p[i]$

Destrucción del array:

```
delete [] p;
```



# Arrays dinámicos

```
#include <iostream>
using namespace std;
const int N = 10;

int main() {
    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;
}
```



¡No olvides destruir el array dinámico!



# Ejemplo de array dinámico

listaAD.h

```
...
#include "registro.h"

const int N = 1000;

// Lista: array dinámico (puntero) y contador
typedef struct {
    tRegPtr registros;
    int cont;
} tLista;

...
```



# Ejemplo de array dinámico

listaAD.cpp

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

void eliminar(tLista &lista, int code, bool &ok) {
    ok = true;
    int ind = buscar(lista, code);
    if (ind == -1) {
        ok = false;
    }
    else {
        for (int i = ind + 1; i < lista.cont; i++) {
            lista.registros[i - 1] = lista.registros[i];
        }
        lista.cont--;
    }
} ...
```

No usamos `new`  
Se han creado todo el array al cargar

No usamos `delete`  
Se destruye todo el array al final



# Ejemplo de array dinámico

---

```
int buscar(tLista lista, int code) {
    int ind = 0;
    bool encontrado = false;
    while ((ind < lista.cont) && !encontrado) {
        if (lista.registros[ind].codigo == code) {
            encontrado = true;
        }
        else {
            ind++;
        }
    }
    if (!encontrado) {
        ind = -1;
    }
    return ind;
}

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



# Ejemplo de array dinámico

```
void cargar(tLista &lista, bool &ok) {
    ifstream archivo;
    char aux;
    ok = true;
    archivo.open(BD.c_str());
    if (!archivo.is_open()) {
        ok = false;
    }
    else {
        tRegistro registro;
        lista.cont = 0;
        lista.registros = new tRegistro[N];
        archivo >> registro.codigo;
        while ((registro.codigo != -1) && (lista.cont < N)) {
            archivo >> registro.valor;
            archivo.get(aux); // Saltamos el espacio
            getline(archivo, registro.nombre);
            lista.registros[lista.cont] = registro;
            lista.cont++;
            archivo >> registro.codigo;
        }
        archivo.close();
    }
}
```

Se crean todos a la vez



# Ejemplo de array dinámico

ejemploAD.cpp

Mismo programa principal que el del array de datos dinámicos  
Pero incluyendo listaAD.h, en lugar de lista.h

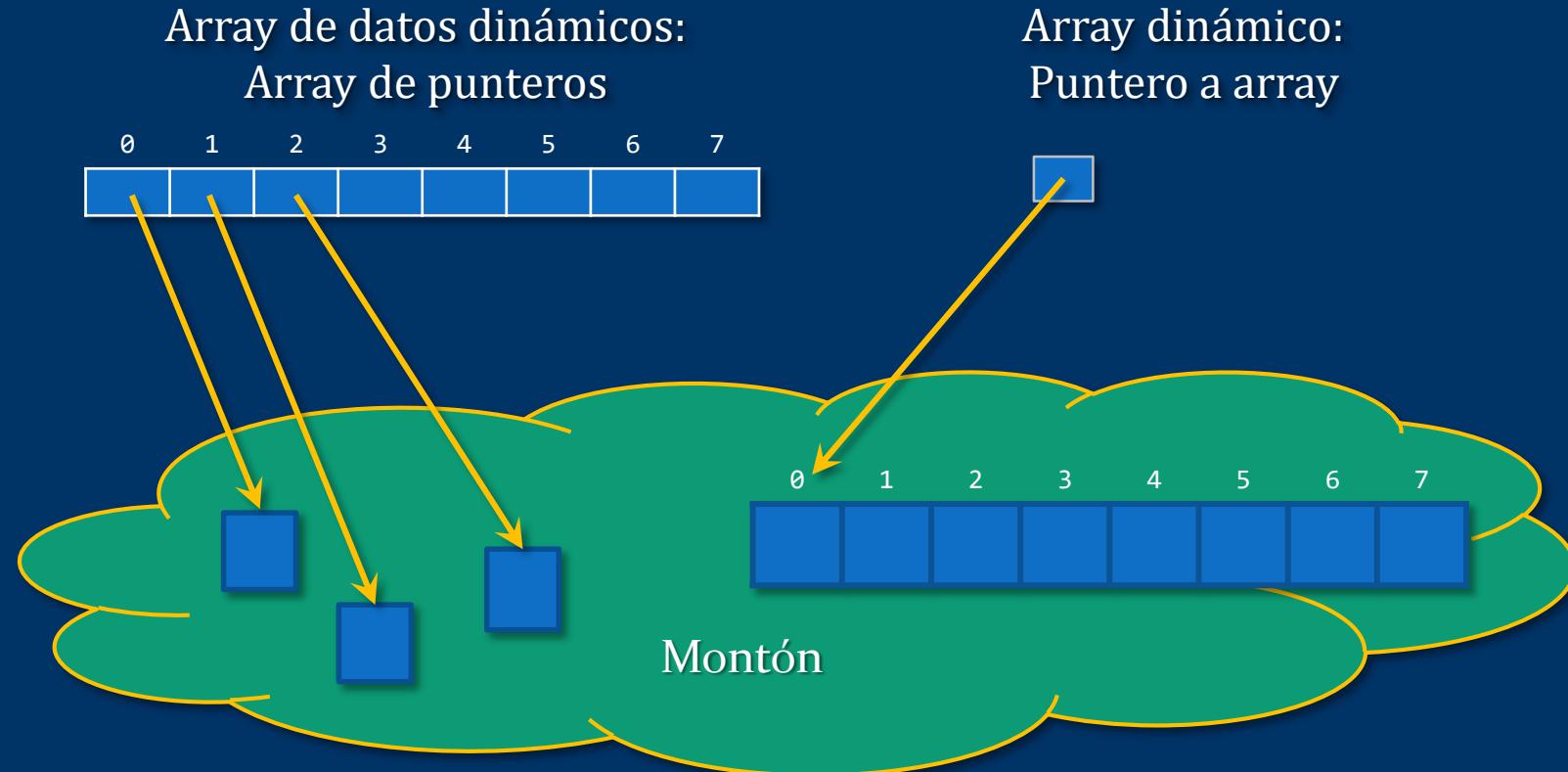
Elementos de la lista:		
-----		
12345	- Disco duro	- 123.59 euros
324356	- Placa base core i7	- 234.50 euros
2121	- Multpuerto USB	- 15.00 euros
54354	- Disco externo 500 Gb	- 95.00 euros
112341	- Procesador AMD	- 132.95 euros
66678325	- Marco digital 2 Gb	- 78.99 euros
600673	- Monitor 22" Nisu	- 154.50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



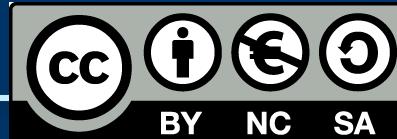
# Arrays dinámicos vs. arrays de dinámicos

Array de datos dinámicos: Array de punteros a datos dinámicos

Array dinámico: Puntero a array en memoria dinámica



# 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.

