

9

Punteros y memoria dinámica

Grados en Ingeniería Informática, Ingeniería
del Software e Ingeniería de Computadores

Miguel Gómez-Zamalloa Gil
(adaptadas del original de Luis Hernández Yáñez y Ana Gil)

Facultad de Informática
Universidad Complutense



Índice

Direcciones de memoria y punteros	2
Operadores de punteros	7
Punteros y direcciones válidas	18
Punteros no inicializados	20
Un valor seguro: nullptr	21
Copia y comparación de punteros	22
Tipos de punteros	27
Punteros a estructuras	29
Punteros a constantes y punteros constantes	31
Punteros y paso de parámetros	33
Punteros y arrays	38
Memoria y datos del programa	41
Memoria dinámica	46
Punteros y datos dinámicos	50
Gestión de la memoria	63
Inicialización de datos dinámicos	67
Errores comunes	69
Arrays de datos dinámicos	74
Arrays dinámicos	86



Direcciones de memoria y punteros

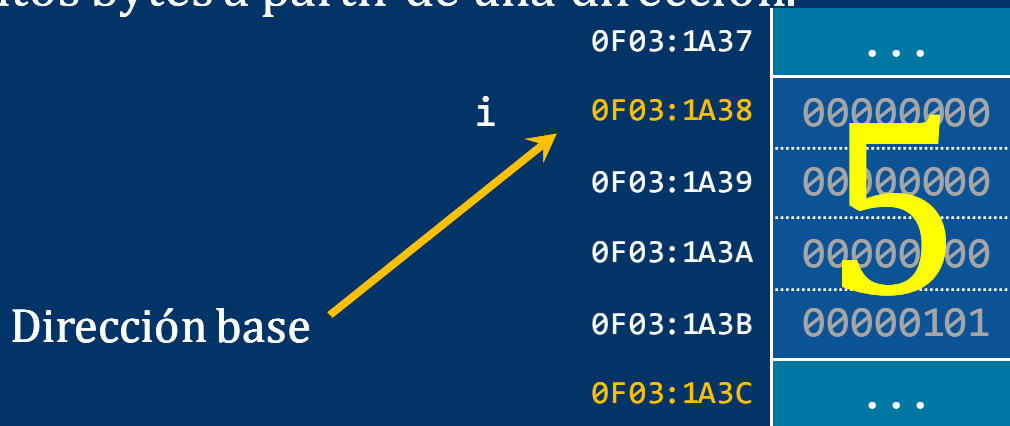


Direcciones de memoria y punteros

Los datos en la memoria

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

```
int i = 5;
```



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

El tipo del dato (**int**) indica cuántas celdas (`sizeof(int)` bytes) utiliza
ese dato (4): 00000000 00000000 00000000 00000101 → 5

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

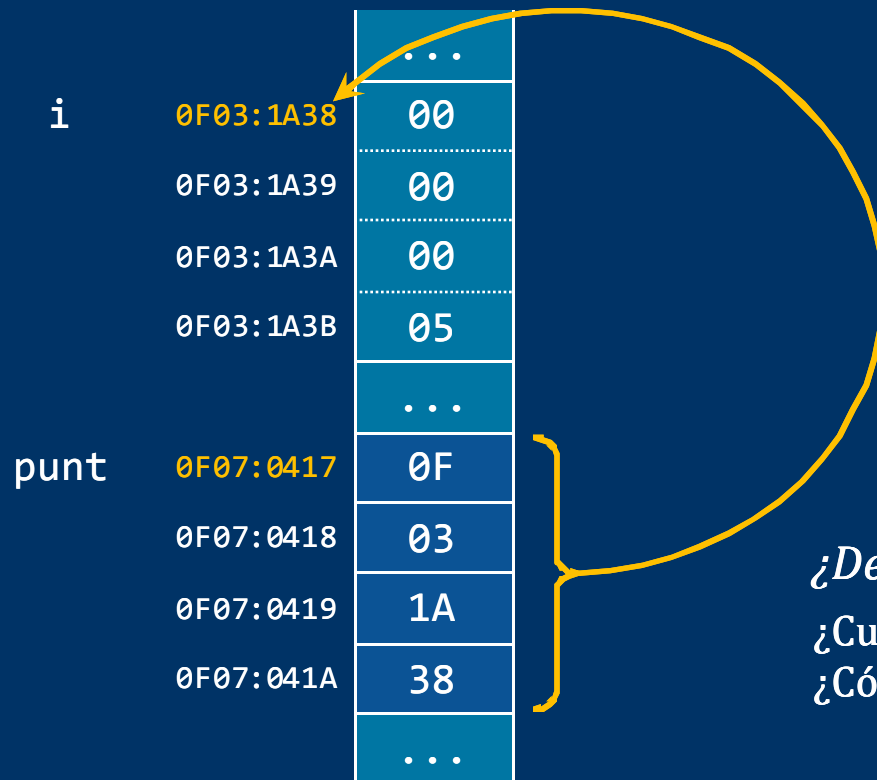


Direcciones de memoria y punteros

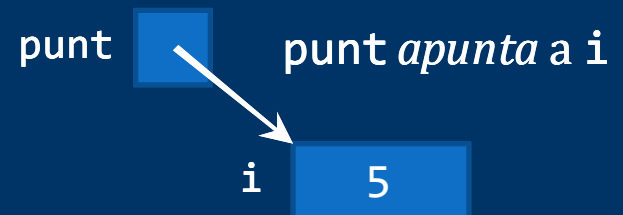
Los punteros contienen direcciones de memoria

Una *variable puntero* (o simplemente *puntero*) sirve para acceder a través de ella a otro dato del programa.

El valor del puntero será la dirección de memoria base de otro dato.



Indirección:
Acceso indirecto a un dato



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



Direcciones de memoria y punteros

Los punteros contienen direcciones de memoria

¿De qué tipo es el dato apuntado?

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

El tipo de variable a la que apunta un puntero se establece al declarar la variable puntero:

tipo **nombre*;

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

Los punteros contienen direcciones de memoria

Las variables puntero tampoco se inicializan automáticamente.
Al declararlas sin inicializador 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 dato 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 datos (Paso de parámetros por referencia).
- ✓ Para gestionar datos dinámicos.
(Datos que se crean y destruyen durante la ejecución.)
- ✓ Para implementar los arrays.



Operadores de punteros



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 base de cualquier dato 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).



Operadores de punteros



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	
	...	



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	
	...	

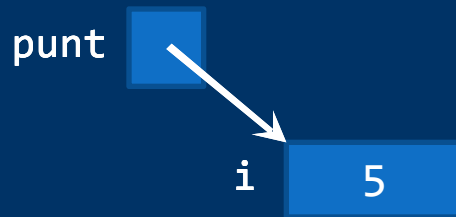


Operadores de punteros



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



Acceso a la variable de la dirección ...

El operador monario * accede a la memoria de la dirección a la que se aplica el operador (un puntero). 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)
```



Operadores de punteros



Obtener lo que hay en la dirección ...

```
int i, j;  
...  
int *punt;  
...  
i = 5;  
punt = &i;  
j = *punt;  
...  
*punt = 0;
```

punt:

→ punt

	...	
i	0F03:1A38	00
	0F03:1A39	00
	0F03:1A3A	00
	0F03:1A3B	05
j	0F03:1A3C	
	0F03:1A3D	
	0F03:1A3E	
	0F03:1A3F	
	...	
ount	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 = 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
j	0F03:1A3B	05
	0F03:1A3C	
	0F03:1A3D	
	0F03:1A3E	
punt	0F03:1A3F	
	...	
	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 = 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
		...	



Operadores de punteros



Obtener lo que hay en la dirección ...

```
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 de punteros

Ejemplo de uso de punteros

```
#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



Punteros y direcciones válidas

Todo puntero ha de tener una dirección válida

Un puntero sólo debe ser utilizado, para acceder al dato al que apunte, si se está seguro de que contiene una dirección válida.

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 otro dato 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

Punteros que apuntan a saber qué...

Una 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`

Punteros que no apuntan a nada

Inicializando los punteros a `nullptr` podemos detectar errores:

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

```
...
```

punt 

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

punt ha sido inicializado a `nullptr`: ¡No apunta a nada!

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

→ 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.



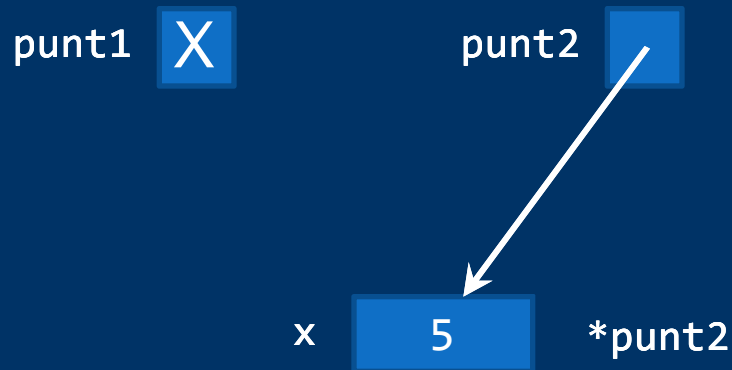
Copia y comparación de punteros



Copia de punteros

Compartiendo

```
int x = 5;  
int *punt1 = nullptr; // punt1 no apunta a nada  
  
int *punt2 = &x; // punt2 apunta a la variable x  
*punt2 y x son la misma variable
```

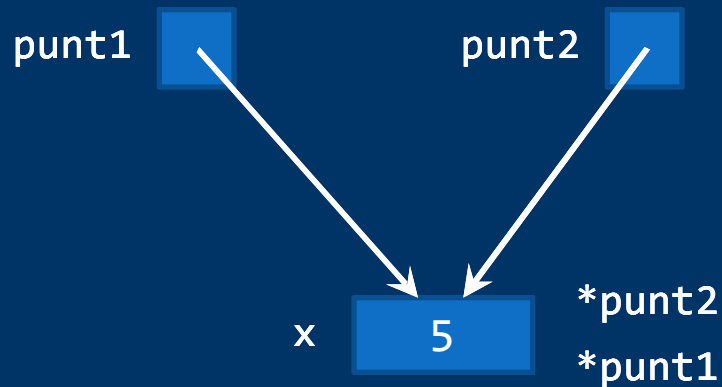


Copia de punteros

Asignación de punteros

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

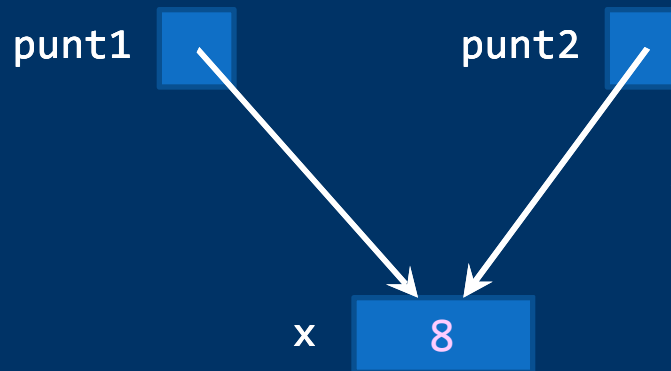
```
punt1 = punt2; // ambos apuntan a la variable x  
*punt2, *punt1 y x son la misma variable
```



Copia de punteros

Compartiendo

```
int x = 5;  
int *punt1 = nullptr; // punt1 no 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

¿Apuntan al mismo dato?

Los operadores relacionales `==` y `!=` nos permiten saber si dos punteros apuntan a un mismo dato:

```
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 de punteros



Tipos puntero

Declaración de tipos de punteros

Declaramos tipos para los punteros con distintos tipos base:

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

Acceso a estructuras a través de punteros

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

```
typedef struct {  
    int codigo;  
    string nombre;  
    double sueldo;  
} tRegistro;  
tRegistro registro;  
typedef tRegistro *tRegistroPtr;  
tRegistroPtr 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->... ≡ (*puntero)...



Punteros a estructuras

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;
```

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

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



Punteros y el modificador const

Punteros a constantes y punteros constantes

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

`const tipo *puntero;` Puntero a una constante

`tipo *const puntero;` Puntero constante

Punteros a constantes:

```
typedef const int *intCtePtr; // Puntero a dato constante
```

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

```
intCtePtr 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 a constantes y punteros constantes

Punteros constantes:

```
typedef int *const intPtrCte; // Puntero constante
int entero1 = 5, entero2 = 13;
intPtrCte punt_cte = &entero1;
(*punt_cte)++; // Sin problema: el puntero no apunta a cte.
punt_cte = &entero2; // ERROR: ¡Puntero constante!
```



Punteros y paso de parámetros



Punteros y paso de parámetros

Paso de parámetros por referencia o variable

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;
```

Mostrará 6 en la consola.

Paso por valor:

El argumento (una dirección de memoria) no se puede modificar (punt contiene una copia del argumento).

Pero aquello a lo que apunta SÍ (entero se modifica a través de punt).



Punteros y paso de parámetros

Paso de parámetros con puntero

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

entero 5

punt recibe la dirección de entero

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



```
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 paso de parámetros

Paso de parámetros por referencia o variable

¿Cuál es el equivalente con punteros a este prototipo? ¿Cómo se llama?

```
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  
}
```

Llamada:

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



Punteros y arrays



Punteros y arrays

Una íntima relación

Identificador de 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.

¡El nombre (identificador) del array es un puntero constante!

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

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



Punteros y paso de parámetros arrays

Paso de arrays a funciones

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

Como el nombre del array es un puntero, ya es un paso por referencia.

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 para poder recorrer el array (o un centinela).

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



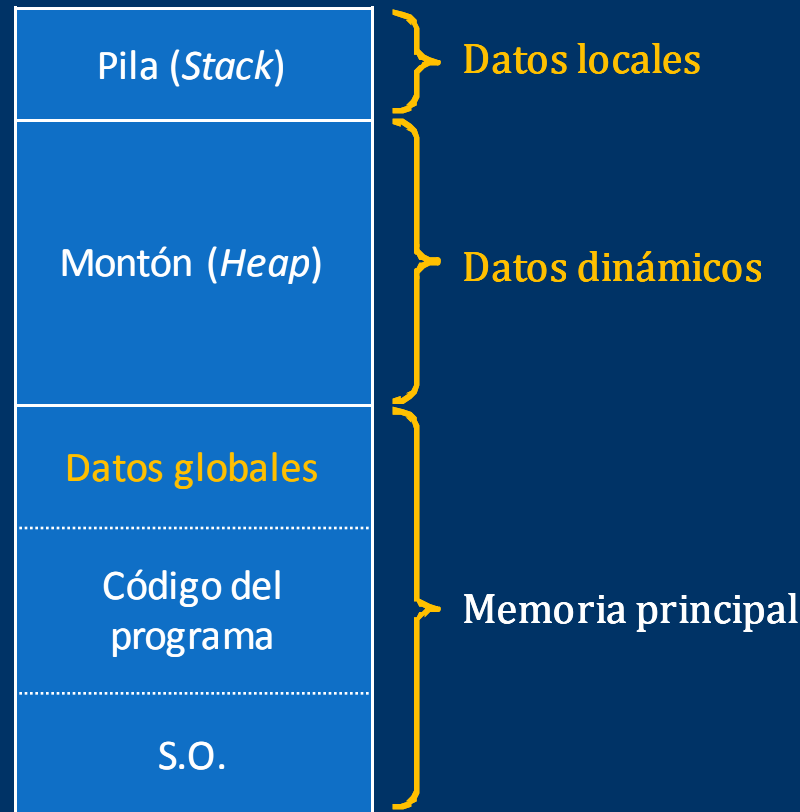
Memoria y datos del programa



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

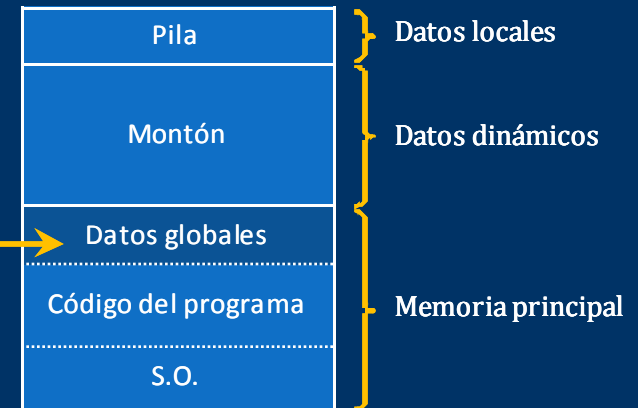
La memoria principal

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

```
const int N = 1000;
typedef tRegistro tLista[N];
typedef struct {
    tLista registros;
    int cont;
} tTabla;
```

```
tTabla tabla;
```

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

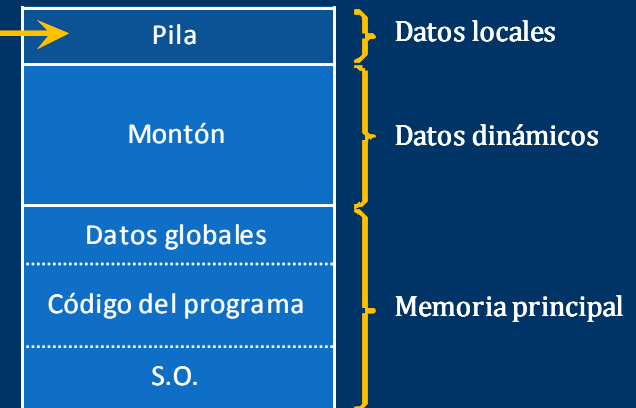


Memoria y datos del programa

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 (punteros).

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

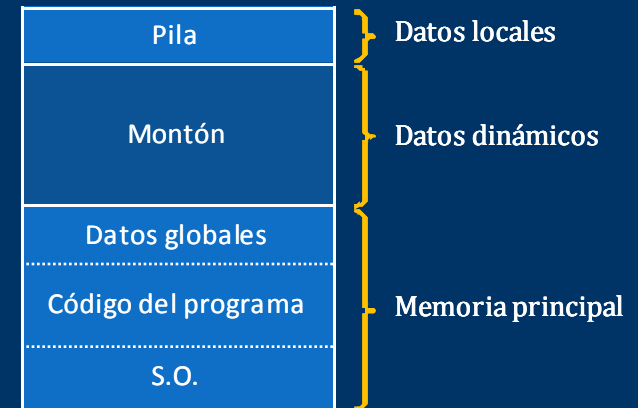
El montón (heap)

El *montón* 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: *Datos dinámicos*

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



Memoria dinámica

Datos dinámicos

Datos que se crean y se destruyen durante la ejecución del programa, al ejecutarse una solicitud al SGMD. Se les asigna memoria del montón.



¿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 la 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.



Datos y asignación de memoria

¿Cuándo se asigna memoria a los datos?

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



Datos dinámicos frente a datos declarados

Datos declarados

- ✓ 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;`

Datos dinámicos (anónimos)

- ✓ Variables (y constantes) no declaradas (sin nombre). Hay que acceder a través de su dirección de memoria.
- ✓ Se necesita tener guardada esa dirección de memoria en algún sitio: Puntero.

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



Punteros y datos dinámicos



Punteros y datos dinámicos

Operadores `new` y `delete`

Hasta ahora hemos trabajado con punteros que contienen direcciones de datos declarados (variables globales o en la pila).

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 variable y devuelve la dirección de la primera celda de memoria asignada a la variable; esa dirección se guarda 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 datos dinámicos

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.

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



Eliminación de datos dinámicos

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
```



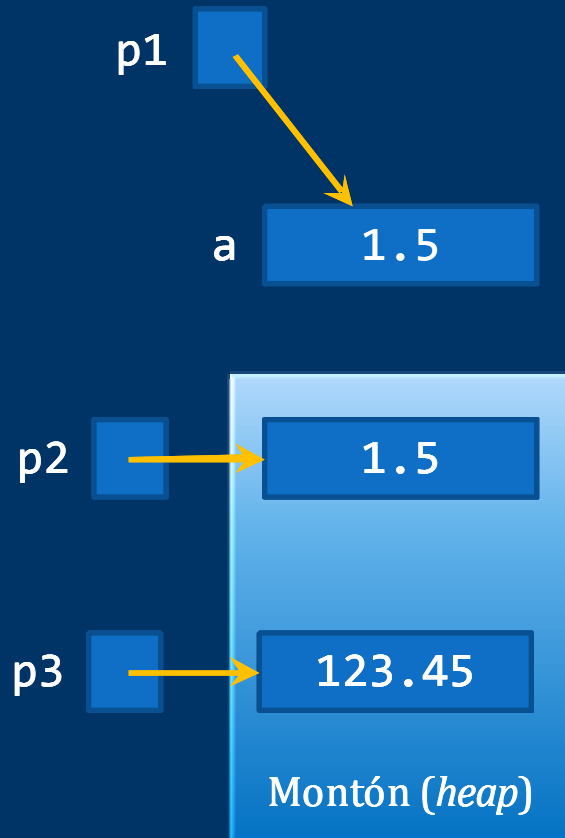
Un ejemplo

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;

    return 0;
}
```



Identificadores:

4

(a, p1, p2, p3)

Variables:

6

(4 + *p2 y *p3)

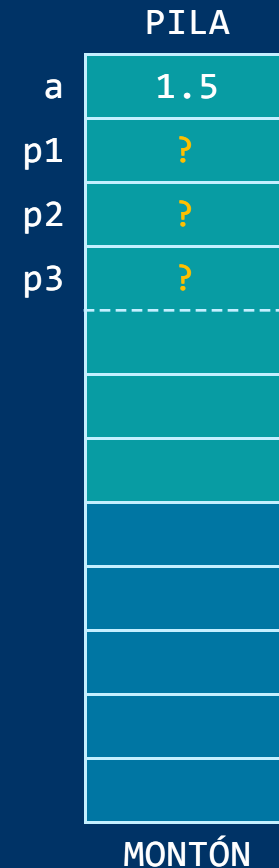


Un ejemplo

Ejemplo de variables dinámicas

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

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

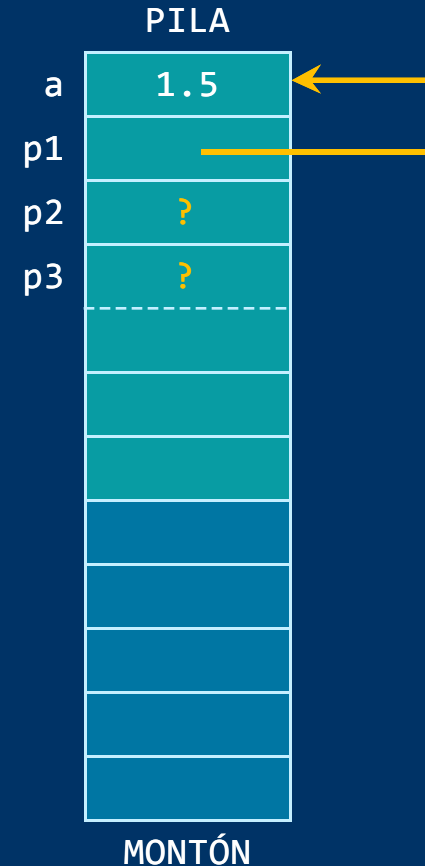


Un ejemplo

Ejemplo de variables dinámicas

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

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

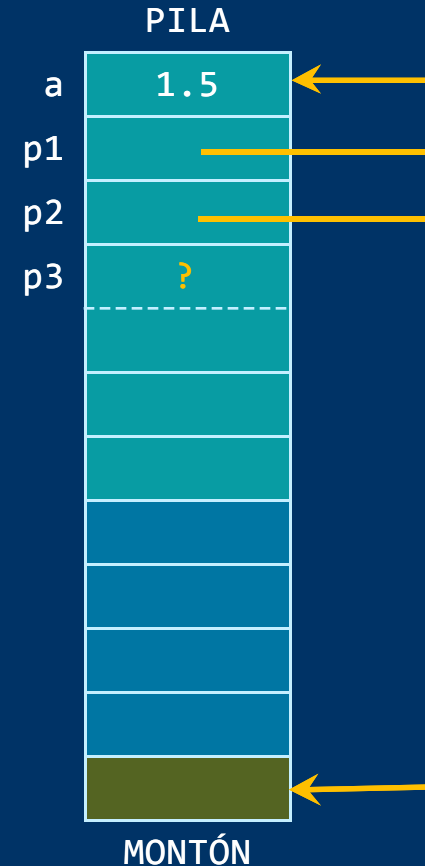


Un ejemplo

Ejemplo de variables dinámicas

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

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

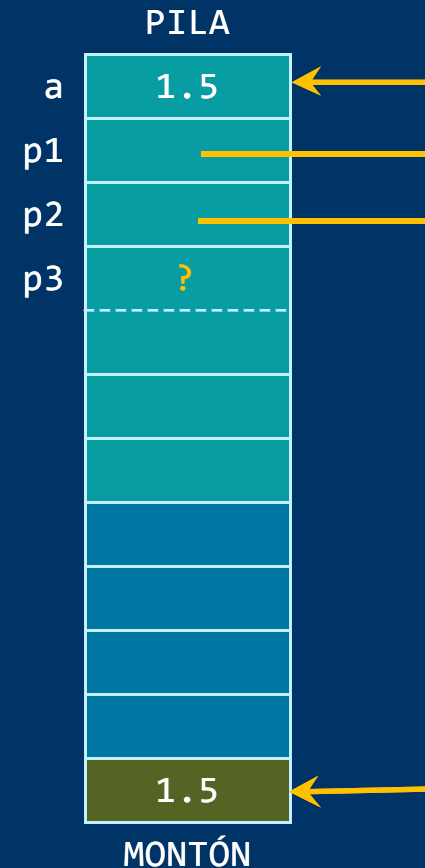


Un ejemplo

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;
```

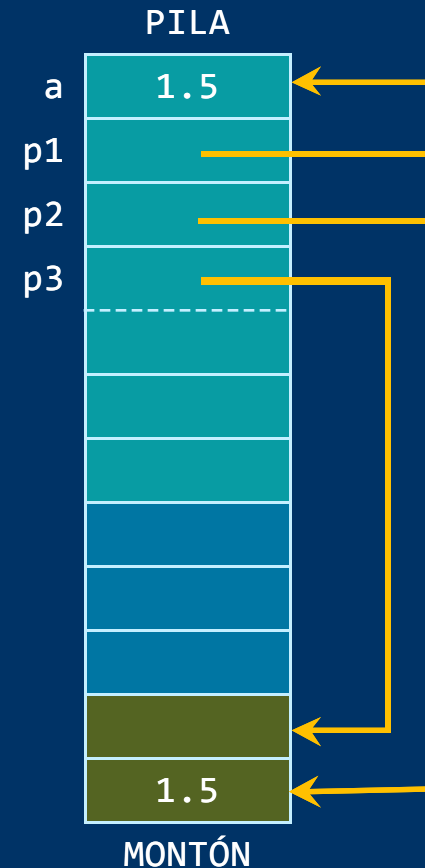


Un ejemplo

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;
```

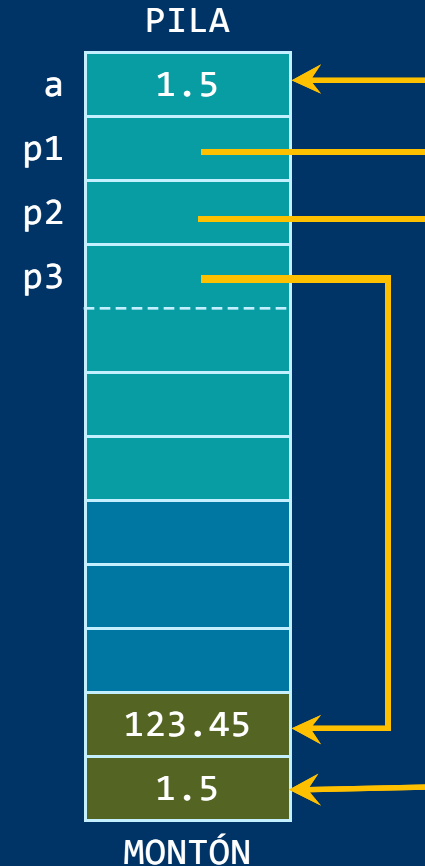


Un ejemplo

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;
```

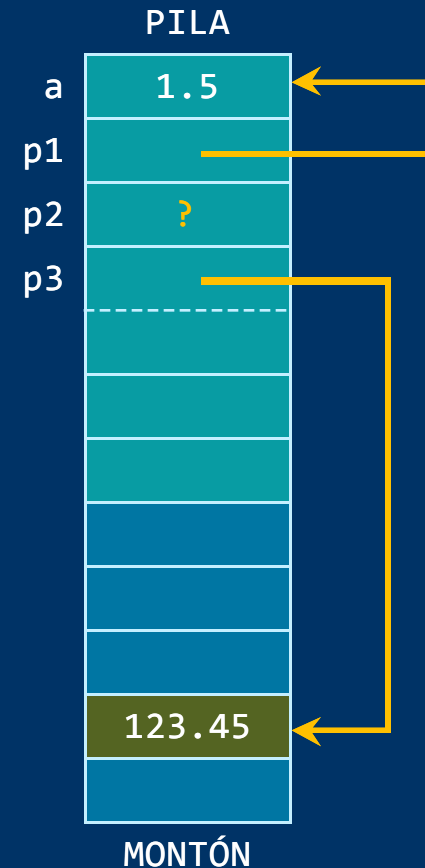


Un ejemplo

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;
}
```

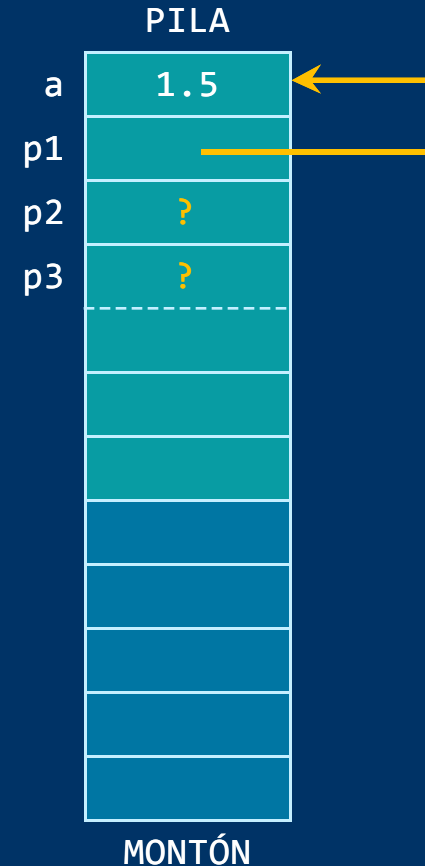


Un ejemplo

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

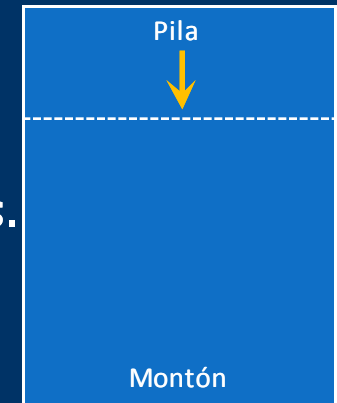


Agotamiento de la memoria

Errores de asignación de memoria (stack)

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. Si lo sobrepasa lo que se produce es un *desbordamiento de la pila (stack overflow)*.

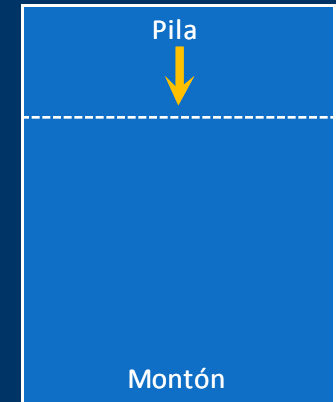
Evitar llamadas en cascada y parámetros por valor de tipos no básicos.



Agotamiento de la memoria

Errores de asignación de memoria (heap)

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 se queda suficiente memoria contigua del tamaño solicitado.



Gestión de la memoria dinámica

Gestión del montón

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 el lenguaje Java.

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 destruir, con el operador `delete`, todas las variables previamente creadas con el operador `new`.

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

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



Inicialización de datos dinámicos



Inicialización de datos dinámicos

Inicialización con el operador new


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

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

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

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

```
int main() { // tRegistro *punt = new tRegistro; leer(*punt);  
    tRegistro reg;  
    leer(reg);  
    tRegistro *punt = new tRegistro(reg);  
    mostrar(*punt);  
    delete punt;  
    return 0;  
}
```



Errores comunes



Errores comunes

Mal uso de la memoria dinámica I

Olvido de destrucción de un dato dinámico:

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

← Falta `delete p;`

G++ no dará ninguna indicación del 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.

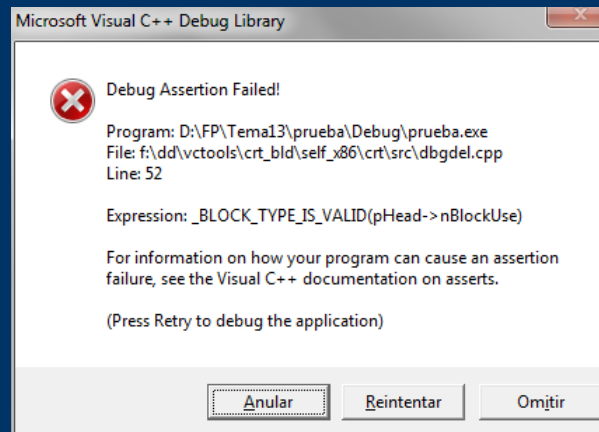
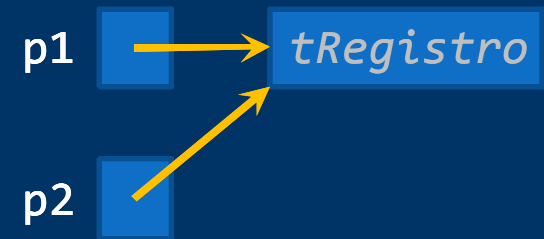


Errores comunes

Mal uso de la memoria dinámica II

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

Mal uso de la memoria dinámica III

Pérdida de un dato dinámico:

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

①



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



Errores comunes

Mal uso de la memoria dinámica IV

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

    return 0;
}
```



Arrays de datos dinámicos



Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

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

const int TM = ...;

// Array de punteros a registros:
typedef tRegPtr tArrayPtr[TM];
typedef struct {
    tArrayPtr registros;
    int cont;
} tLista;
```

Los punteros ocupan muy poco en memoria.

Los datos a los que apunten se guardarán en el montón.

Se crearán a medida que se inserten en la lista.

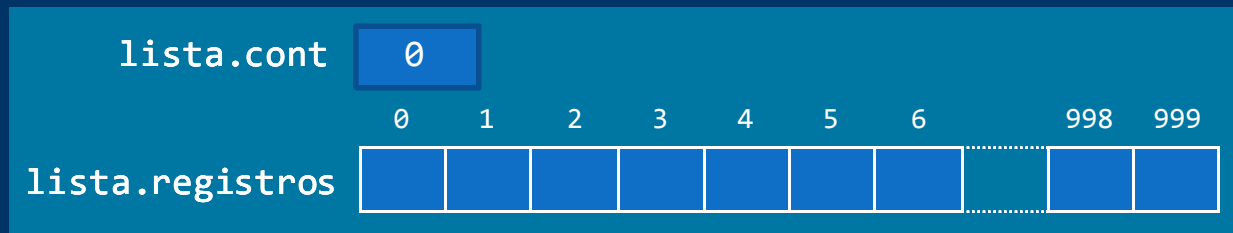
Se destruirán a medida que se eliminen de la lista.



Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

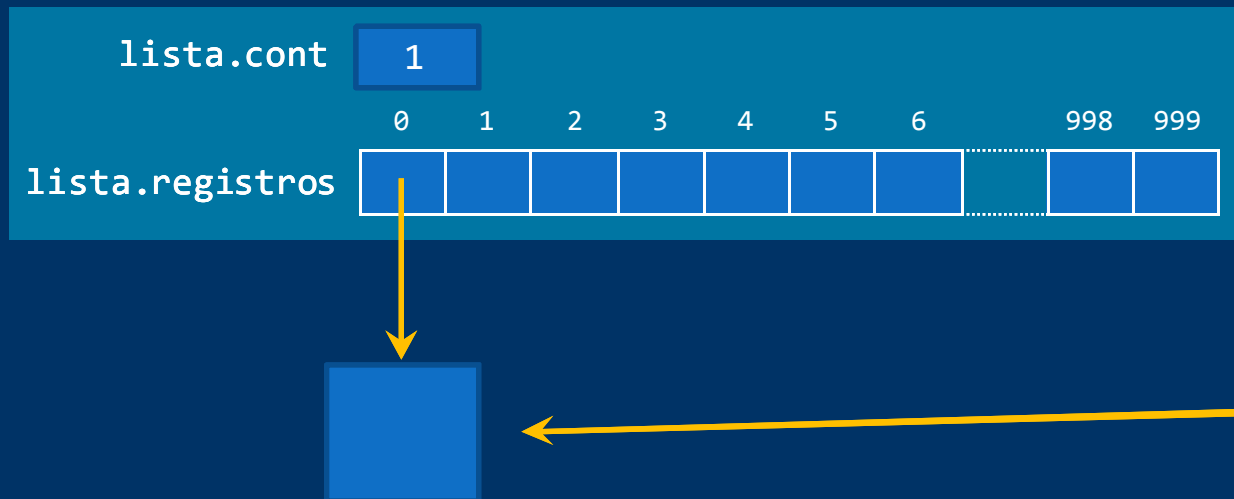
```
tLista lista;  
lista.cont = 0;
```



Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

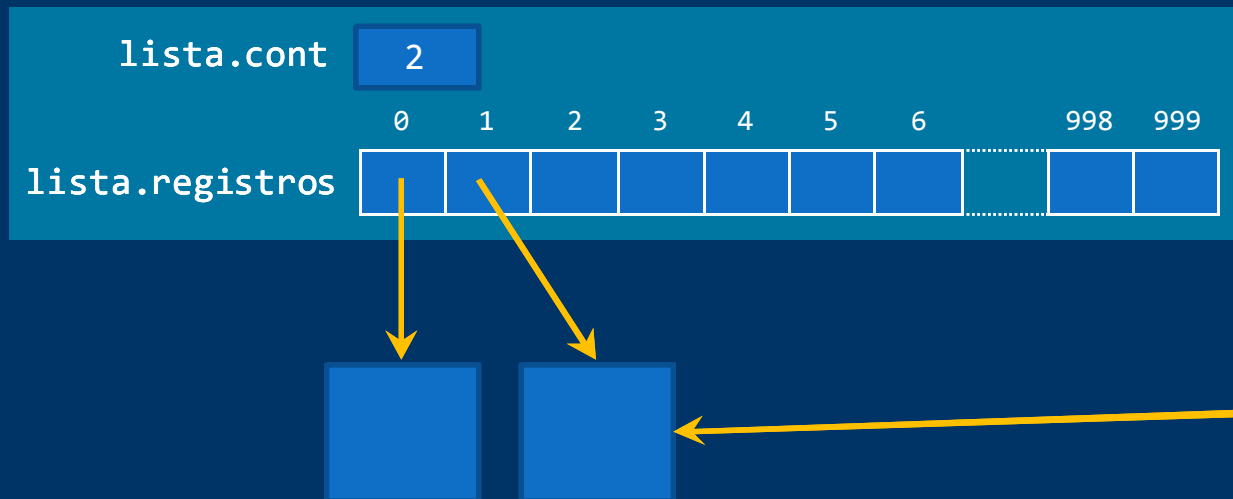
```
tLista lista;  
lista.cont = 0;  
tRegPtr p = new tRegistro; leer(*p);  
lista.registros[lista.cont] = p; lista.cont++;
```



Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

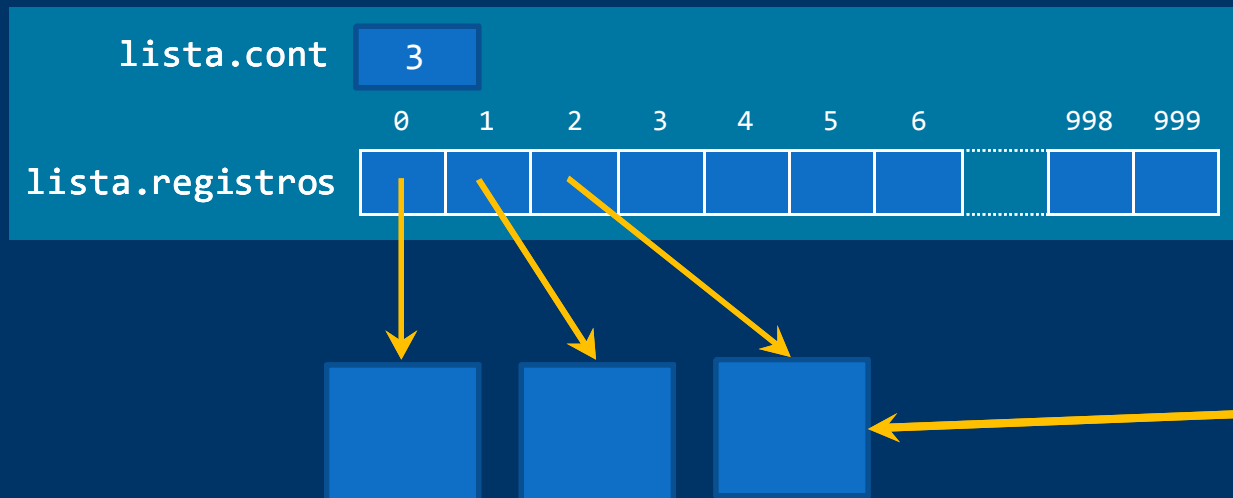
```
tLista lista;  
lista.cont = 0;  
tRegPtr p = new tRegistro; leer(*p);  
lista.registros[lista.cont] = p; lista.cont++;  
p = new tRegistro; leer(*p);  
lista.registros[lista.cont] = p; lista.cont++;
```



Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

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

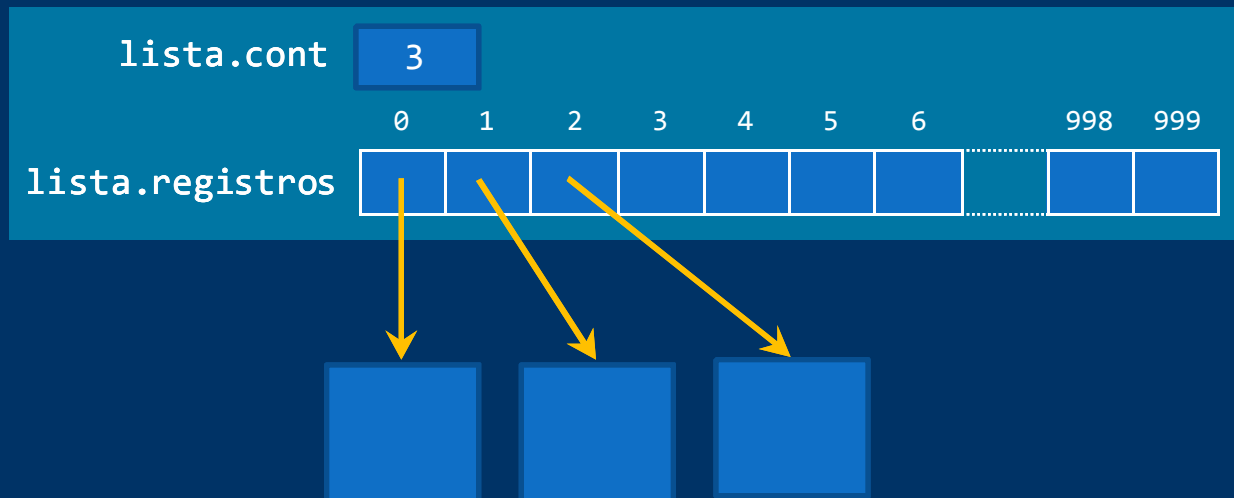


Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

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

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

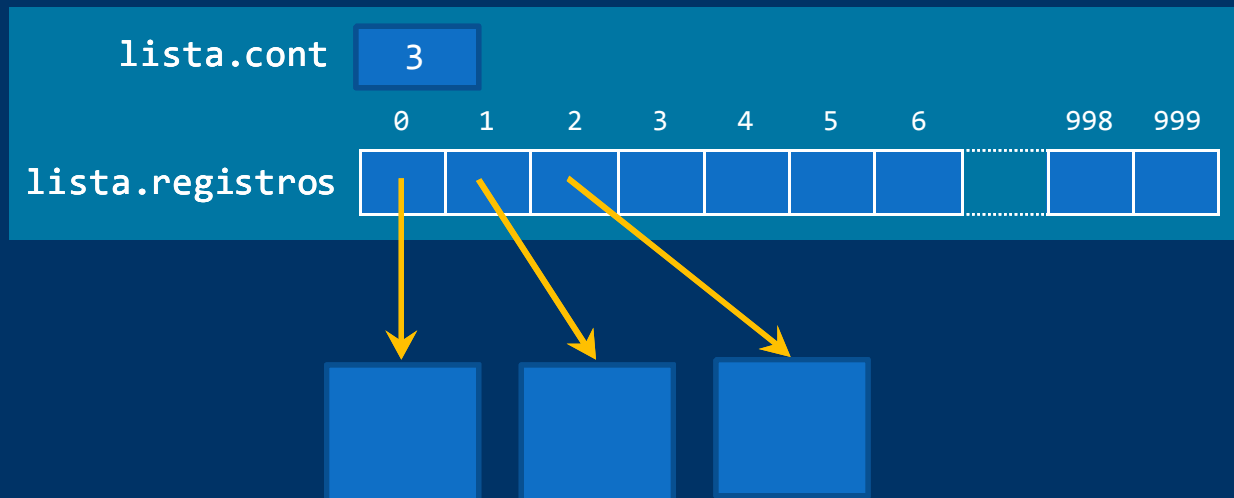


Arrays de datos dinámicos

Arrays de punteros a 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

Implementación de la lista de datos dinámicos

listaDD.h

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

const int TM = ...;
typedef tRegPtr tArrayPtr[TM];
typedef struct {
    tArrayPtr registros;
    int cont;
} tLista;

const char NombreBD[] = "datos.dat";

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
```



Arrays de datos dinámicos

Implementación de la lista de datos dinámicos

listaDD.cpp

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

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

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



Arrays de datos dinámicos

Implementación de la lista de datos dinámicos

```
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]);
}
```

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



Arrays de datos dinámicos

Implementación de la lista de datos dinámicos

datosD.cpp

```
#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 - Multupuerto 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

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

```
int *p = new int[10];
```

Se crea un array de 10 `int` en el montón.

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

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

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

```
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;
}
```

Destrucción del array dinámico



Arrays dinámicos

Ejemplo de array dinámico

listaAD.h

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

const int TM = ...;

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

...
```



Arrays dinámicos

Ejemplo de array dinámico

datosAD

```
#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 de array dinámico

listaAD.cpp

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

Se crean todos los
registros a la vez

```
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`,
pues se han creado
todos los registros
anteriormente



Arrays dinámicos

Ejemplo de array dinámico

listaAD.cpp

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

No usamos `delete`,
pues se destruyen
todos los registros
al final

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

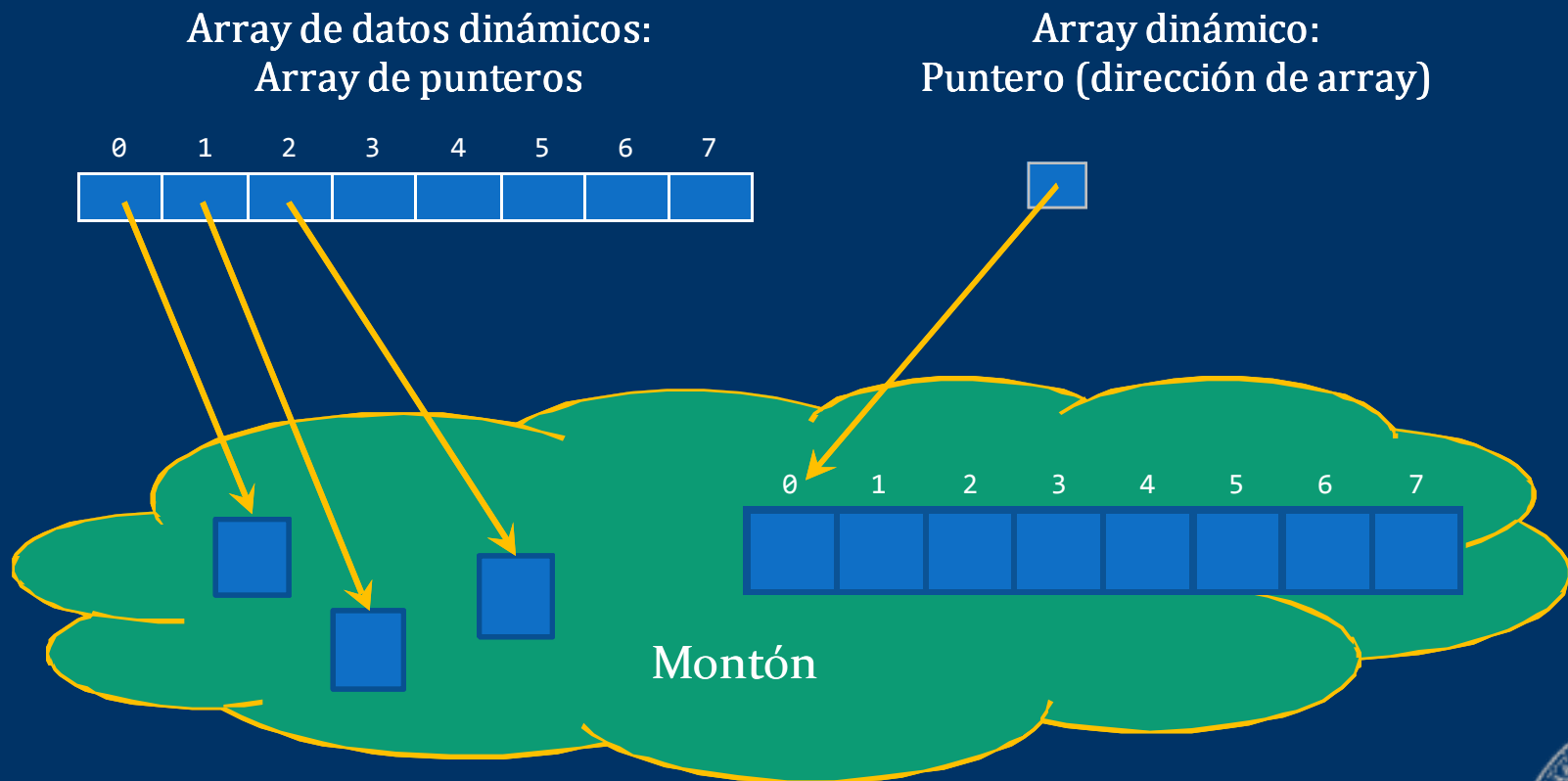
Se destruyen todos
los registros a la vez

...



Arrays dinámicos vs. arrays de dinámicos

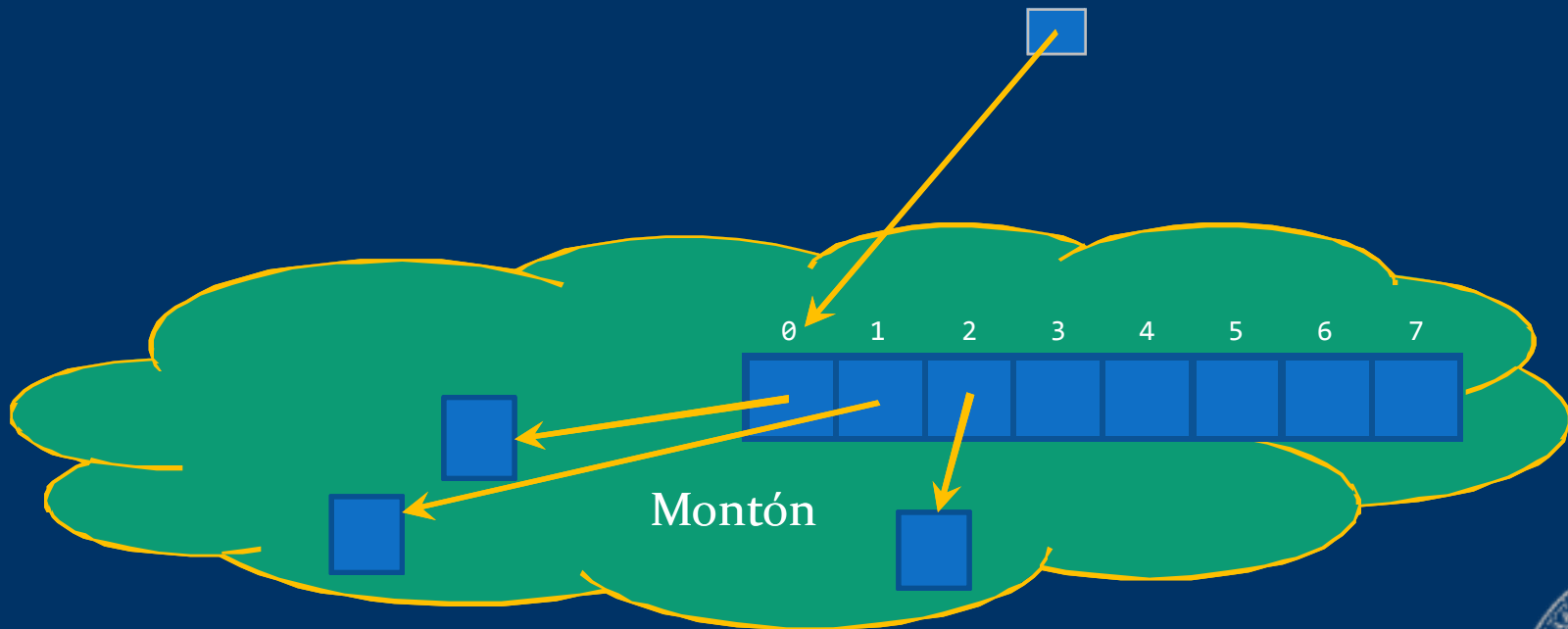
Los arrays de datos dinámicos van tomando del montón memoria a medida que la necesitan, mientras que el array dinámico se crea entero en el montón (`new tipoBase[N];` -> tamaño `sizeof(tipoBase) * N`):



Arrays dinámicos de datos dinámicos

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

Array dinámico de datos dinámicos:
p (dirección de array)



Referencias bibliográficas



- ✓ *C++: An Introduction to Computing* (2ª edición)
J. Adams, S. Leestma, L. Nyhoff. Prentice Hall, 1998
- ✓ *El lenguaje de programación C++* (Edición especial)
B. Stroustrup. Addison-Wesley, 2002
- ✓ *Programación en C++ para ingenieros*
F. Xhafa et al. Thomson, 2006








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.

