

9

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	2
Operadores de punteros	7
Punteros y direcciones válidas	17
Punteros no inicializados	19
Un valor seguro: NULL	20
Copia y comparación de punteros	21
Tipos de punteros	26
Punteros a estructuras	28
Punteros a constantes y punteros constantes	30
Punteros y paso de parámetros	32
Punteros y arrays	36
Memoria y datos del programa	39
Memoria dinámica	44
Punteros y datos dinámicos	48
Gestión de la memoria	61
Inicialización de datos dinámicos	64
Errores comunes	66
Arrays de datos dinámicos	71
Arrays dinámicos	83



Fundamentos de la programación

Direcciones de memoria y punteros

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 2

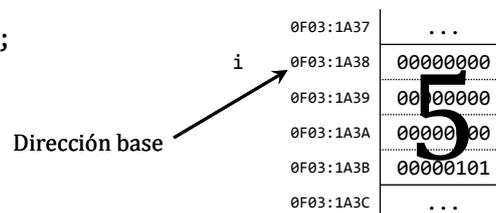


Direcciones de memoria

Los datos en la memoria

Todo dato (variable o constante) de un programa se almacena en la memoria: en unos cuantos bytes a partir de una dirección.

```
int i = 5;
```



El 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 (bytes) utiliza ese dato (4):

00000000 00000000 00000000 00000101 → 5

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

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 3

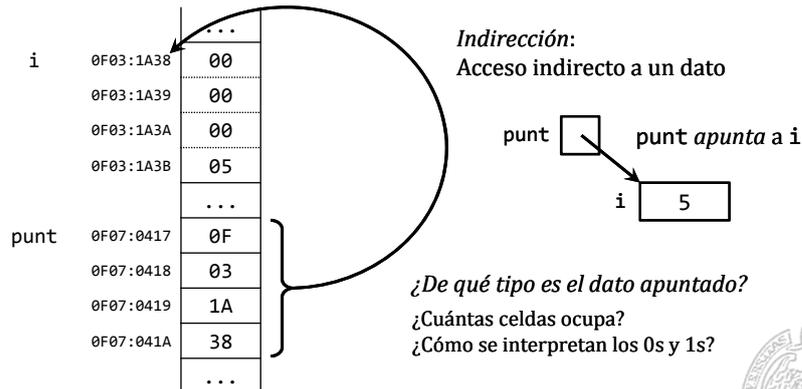


Punteros

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.



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 4

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 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
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 5

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 contiene una dirección
           // que no es válida (no apunta a nada).
```

Un puntero puede apuntar a cualquier dato del 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.)

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 6



Fundamentos de la programación

Operadores de punteros

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 7



Operadores de punteros

&

Obtener la dirección de memoria de ...

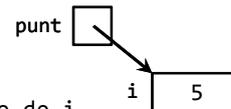
El operador monario & devuelve la dirección de memoria base del dato al que se aplica el operador. Operador prefijo (precede).

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

Ahora, el puntero `punt` ya contiene una dirección de memoria válida. `punt` apunta a (contiene la dirección base de) la variable entera `i` (`int`).



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 8



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	
	...	

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 9

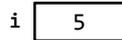


Operadores de punteros

&

Obtener la dirección de memoria de ...

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

Luis Hernández Yáñez

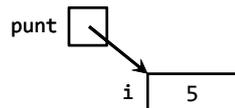


Operadores de punteros

&

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
...	

Luis Hernández Yáñez



Operadores de punteros *

Obtener lo que hay en la dirección ...

El operador monario * accede a lo que hay en la dirección de memoria 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 al dato al que apunta con este operador.

```
punt = &i;
```

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

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

Como el puntero punt contiene la dirección de memoria de la variable i, *punt accede al contenido de esa variable i.

Acceso indirecto al valor de i.

Se obtienen, a partir de la dirección de memoria base que contiene punt, tantos bytes como correspondan al tipo base (int) (4) y se interpretan como un dato de ese tipo base (int).

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 12



Operadores de punteros *

Obtener la dirección de memoria de ...

```
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
	...	

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 13



Operadores de punteros *

Obtener la dirección de memoria de ...

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

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

*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
...		

Luis Hernández Yáñez



Operadores de punteros *

Obtener la dirección de memoria de ...

```
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
...		

Luis Hernández Yáñez



Operadores de punteros

Ejemplo de uso de punteros

punteros.cpp

```
#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;
}
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 16



Fundamentos de la programación

Punteros y direcciones válidas

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 17



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 copiarle 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 NULL (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 = NULL; // ahora q ya tiene una dirección válida
q = p; // otra dirección válida para q
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 18



Punteros no inicializados

Punteros que apuntan a saber qué...

Una puntero no inicializado contiene una dirección desconocida.

```
int *punt; // no inicializado
*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*)

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 19



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: *¡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.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 20



Fundamentos de la programación

Copia y comparación de punteros

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 21

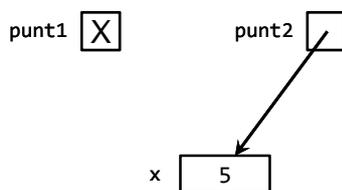


Copia de punteros

Apuntando al mismo dato

Cuando copiamos un puntero sobre otro, ambos apuntan al mismo dato:

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



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 22

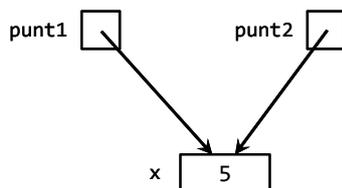


Copia de punteros

Apuntando al mismo dato

Cuando copiamos un puntero sobre otro, ambos apuntan 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
```



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 23

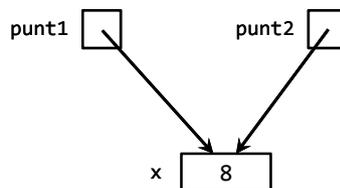


Copia de punteros

Apuntando al mismo dato

Cuando copiamos un puntero sobre otro, ambos apuntan 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;
```



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

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 24



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 = NULL;
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.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 25



Fundamentos de la programación

Tipos de punteros

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 26



Tipos puntero

Declaración de tipos de punteros

tipos.cpp

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 caracter = 'C';
charPtr puntC = &caracter;
double real = 5.23;
doublePtr puntD = &real;
cout << *puntI << " " << *puntC << " " << *puntD << endl;
```

Con **puntero* podemos hacer lo que se pueda hacer con los datos del tipo base del *puntero*.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 27



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)....
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 28



Punteros a estructuras

Acceso a estructuras a través de punteros

structPtr.cpp

```
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;
```

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

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

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 29



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.
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 30



Punteros y el modificador const

Punteros a constantes y punteros constantes

constPtr.cpp

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!
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 31



Fundamentos de la programación

Punteros y paso de parámetros

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 32



Punteros y paso de parámetros

Paso de parámetros por referencia o variable

param.cpp

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);
```

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

```
...
int entero = 5;
incrementa(&entero);
cout << entero << endl;
```

Mostrará 6 en la consola.

Paso por valor:

El argumento no se modifica tras la ejecución (el puntero no cambia).

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

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 33



Punteros y paso de parámetros

Paso de parámetros por referencia o variable

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

entero

punt recibe la dirección de entero

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

punt

entero

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

entero

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 34



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);
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 35



Fundamentos de la programación

Punteros y arrays

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 36



Punteros y arrays

Una íntima relación

Nombre 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 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.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 37



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
```

Las tres declaraciones del parámetro array son equivalentes.

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.

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

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 38



Fundamentos de la programación

Memoria y datos del programa

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

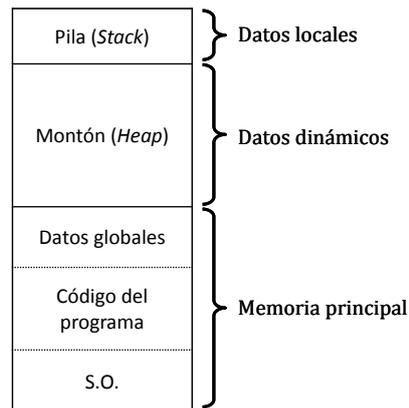
Página 39



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:



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 40



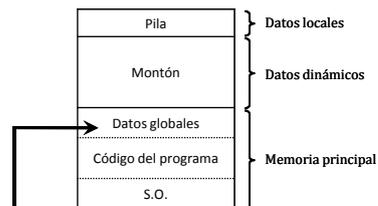
Memoria y datos del programa

La memoria principal

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

```
typedef char tCadena[80];
typedef struct {
    ...
} tRegistro;
const int N = 1000;
typedef tRegistro tLista[N];
typedef struct {
    tLista registros;
    int cont;
} tTabla;

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



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 41



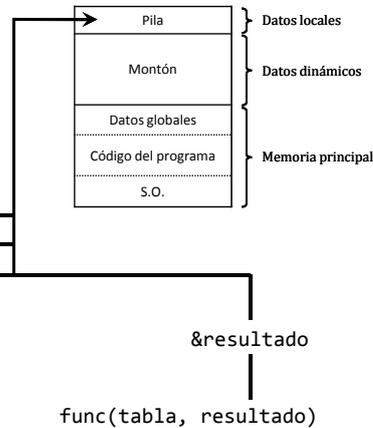
Memoria y datos del programa

La pila (stack)

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

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

También se guardan los punteros que se manejan internamente para apuntar a los argumentos de los parámetros por referencia.



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 42



Memoria y datos del programa

El montón (heap)

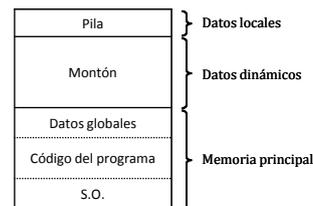
El *montón* es una enorme zona de almacenamiento donde podemos alojar temporalmente datos del programa que se creen y se destruyan a medida que se necesiten durante la ejecución del programa.

Datos creados 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.



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 43



Fundamentos de la programación

Memoria dinámica

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

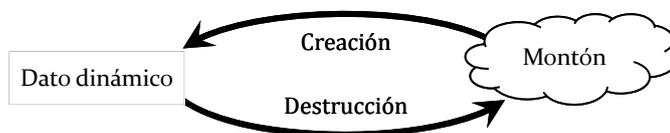
Página 44



Memoria dinámica

Datos dinámicos

Datos que se crean y se destruyen durante la ejecución del programa.
Se les asigna memoria del montón.



¿Por qué utilizar la memoria dinámica?

- ✓ Es un almacén de memoria muy grande: aquellos datos o listas de datos que no quepan en memoria principal, o que consuman demasiado espacio en la pila, pueden ser alojados en el montón.
- ✓ El programa ajusta el uso de la memoria a las necesidades de cada momento: ni le falta ni la desperdicia.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 45



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 guardan en la pila del sistema durante la ejecución de esa función.
Existen sólo durante la ejecución de esa función.
- ✓ Datos dinámicos:
Se crean en el montón (*heap*) cuando el programa los solicita
y se destruyen cuando el programa igualmente lo solicita.
Existen *a voluntad* del programa.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 46



Datos estáticos frente a datos dinámicos

Datos estáticos

- ✓ Constantes y variables declaradas como de un tipo concreto:
`int i;`
- ✓ Su información se accede directamente a través del identificador:
`cout << i;`

Datos dinámicos

- ✓ Constantes y variables accedidas 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;`).

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 47



Fundamentos de la programación

Punteros y datos dinámicos

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 48



Punteros y datos dinámicos

Operadores new y delete

Hasta ahora hemos trabajado con punteros que contienen direcciones de datos estáticos (variables en memoria principal 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.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 49



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 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)
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 50



Eliminación de datos dinámicos

El operador delete

`delete puntero;` Devuelve al montón la memoria utilizada 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 y no se debe acceder a través de él hasta que no contenga nuevamente otra dirección válida.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 51



Un ejemplo

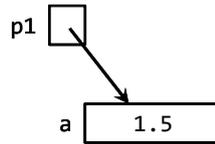
Ejemplo de variables dinámicas

dinamicas.cpp

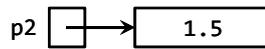
```
#include <iostream>
using namespace std;

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;

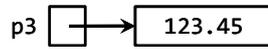
    return 0;
}
```



Identificadores:
4
(a, p1, p2, p3)



Variables:
6
(+ *p2 y *p3)



Montón (heap)

Luis Hernández Yáñez

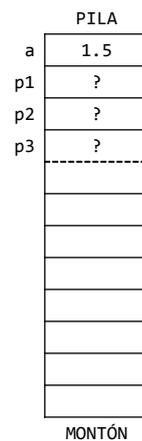


Un ejemplo

Ejemplo de variables dinámicas

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

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



Luis Hernández Yáñez

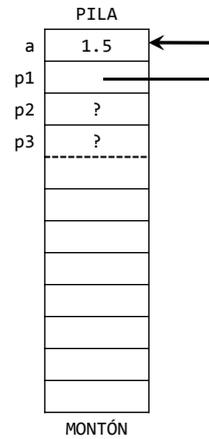


Un ejemplo

Ejemplo de variables dinámicas

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

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



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 54

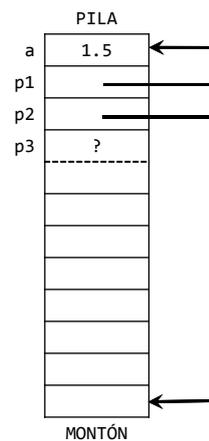


Un ejemplo

Ejemplo de variables dinámicas

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

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



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 55

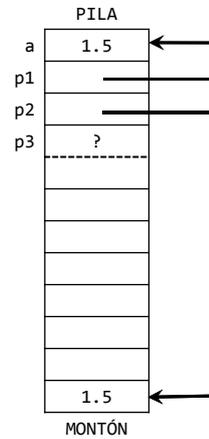


Un ejemplo

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;
```



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 56

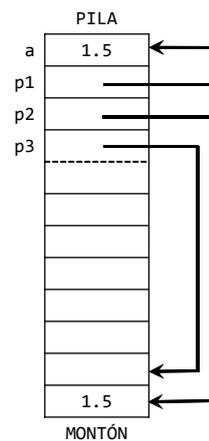


Un ejemplo

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;
```



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 57

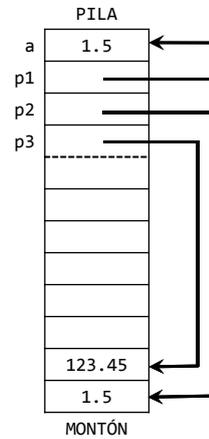


Un ejemplo

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;
```



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 58

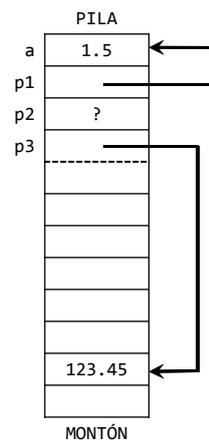


Un ejemplo

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;
```



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 59



Agotamiento de la memoria

Errores de asignación de memoria

Para aprovechar bien la memoria que deja libre la memoria principal, la pila y el montón crecen en direcciones opuestas.

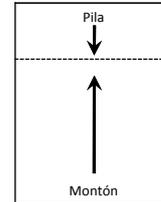
A medida que se llama a funciones la pila crece.

A medida que se crean datos dinámicos el montón crece.

Si los límites de ambas regiones de memoria se encuentran, se produce una *colisión pila-montón*.

El programa falla porque no se pueden crear más datos dinámicos ni se pueden realizar más llamadas a funciones.

Normalmente la pila suele tener un tamaño máximo establecido que no puede sobrepasar aunque el montón no esté utilizando el resto. Si lo sobrepasa lo que se produce es un *desbordamiento de la pila*.



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.



Fundamentos de la programación

Inicialización de datos dinámicos

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 64



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);
```

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

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

registros.cpp

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



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 65



Fundamentos de la programación

Errores comunes

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 66



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);
    ← Falta delete p;
    return 0;
}

```

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.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 67



Errores comunes

Mal uso de la memoria dinámica II

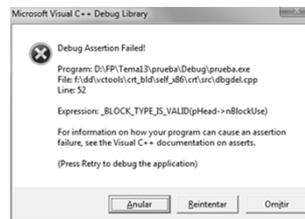
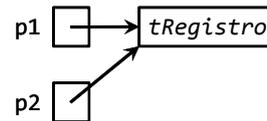
Intento de destrucción de un dato dinámico inexistente:

```

...
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 dinámica
→ No se pueden destruir 2

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 68



Errores comunes

Mal uso de la memoria dinámica III

Pérdida de un dato dinámico:

```

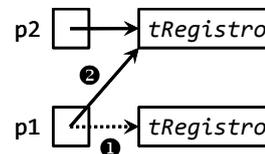
...
int main() {
    tRegistro *p1, *p2;
    p1 = new tRegistro(nuevo()); ①
    p2 = new tRegistro(nuevo());

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

    delete p1;
    delete p2;

    return 0;
}

```



¡Perdido!

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

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 69



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(nuevo());

    mostrar(*p);
    delete p;
    ...
    mostrar(*p);    p ha dejado de apuntar al dato dinámico destruido
                    → Intento de acceso a memoria inexistente

    return 0;
}

```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 70



Fundamentos de la programación

Arrays de datos dinámicos

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 71



Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

```
typedef char tCadena[80];
typedef struct {
    int codigo;
    tCadena nombre;
    double valor;
} tRegistro;
typedef tRegistro *tRegPtr;
```

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

```
// Array de punteros a registros:
typedef tRegPtr tArray[N];
typedef struct {
    tArray 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.

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

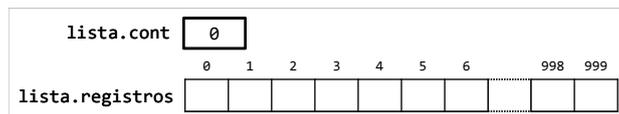
Página 72



Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

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



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

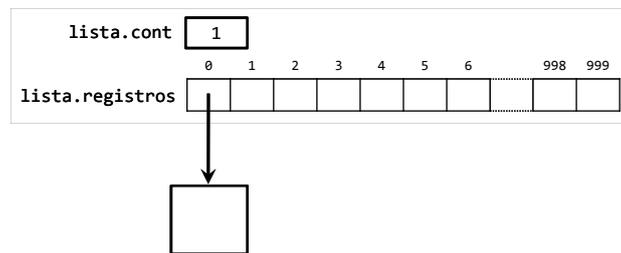
Página 73



Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

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



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

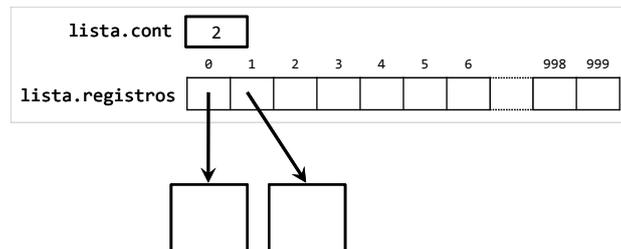
Página 74



Arrays de datos dinámicos

Arrays de punteros a 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++;
```



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

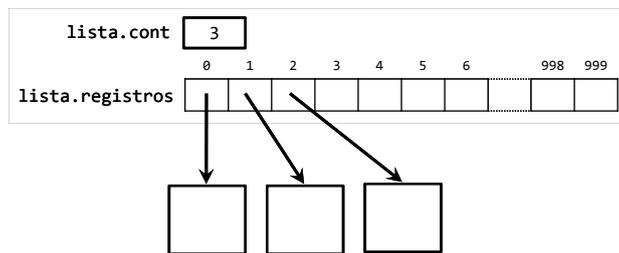
Página 75



Arrays de datos dinámicos

Arrays de punteros a 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++;
```



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 76

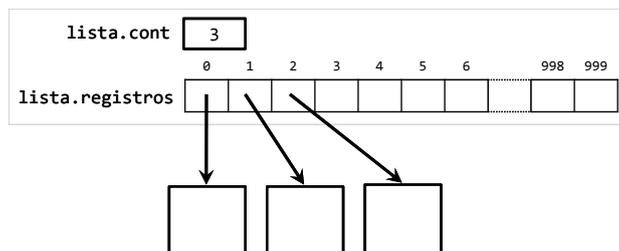


Arrays de datos dinámicos

Arrays de punteros a datos dinámicos

Los registros se acceden a través de punteros (operador flecha):

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



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 77

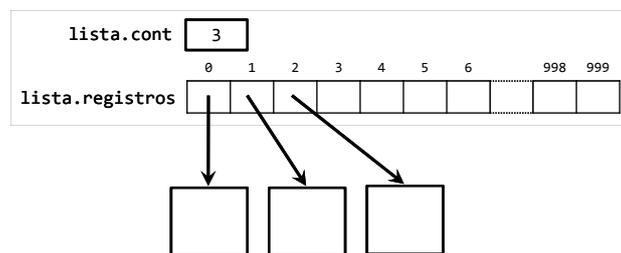


Arrays de datos dinámicos

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];
```



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 78



Arrays de datos dinámicos

Implementación de la lista dinámica

lista.h

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

const int N = 1000;
typedef tRegPtr tArray[N]; ←
typedef struct {
    tArray registros;
    int cont;
} tlista;

const char BD[] = "bd.dat";

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

#endif
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 79



Arrays de datos dinámicos

Implementación de la lista dinámica

lista.cpp

```

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

bool eliminar(tLista &lista, int code) {
    bool 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--;
    }
    return ok;
} ...

```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 80



Arrays de datos dinámicos

Implementación de la lista dinámica

```

int buscar(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 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;
} ...

```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 81



Arrays de datos dinámicos

Implementación de la lista dinámica

listadinamica.cpp

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

int main() {
    tLista 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
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 82



Fundamentos de la programación

Arrays dinámicos

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 83



Arrays dinámicos

Creación y destrucción de arrays dinámicos

Un array dinámico es un array de datos dinámicos para el que se crean todos los datos dinámicos automáticamente al declararlo:

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

Se crean las 10 variables dinámicas, de tipo int. Se acceden con `p[i]`.

```
#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;
}
```

← Destrucción del array dinámico

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 84



Arrays dinámicos

Ejemplo de array dinámico

listaAD.h

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

const int N = 1000;

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

...
```

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 85



Arrays dinámicos

Ejemplo de array dinámico

listaAD.cpp

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

bool eliminar(tLista &lista, int code) {
    bool 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--;
    }
    return ok;
} ...
```

No usamos new,
pues se han creado
todos los registros
anteriormente

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

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 86



Arrays dinámicos

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 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) {
    delete [] lista.registros;
    lista.cont = 0;
} ...
```

Usamos punto en lugar de ->

Acceso como array

Se destruyen todos a la vez

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 87



Arrays dinámicos

Ejemplo de array dinámico

```
bool cargar(tLista &lista) {
    bool ok = true;
    ifstream archivo;
    archivo.open(BD);
    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.getline(registro.nombre, 80);
            lista.registros[lista.cont] = registro;
            lista.cont++;
            archivo >> registro.codigo;
        }
        archivo.close();
    } ...
}
```

Se crean todos a la vez

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 88



Arrays dinámicos

Ejemplo de array dinámico

```
void guardar(tLista lista) {
    ofstream archivo;
    archivo.open(BD);
    for (int i = 0; i < lista.cont; i++) {
        archivo << lista.registros[i].codigo << " ";
        archivo << lista.registros[i].valor;
        archivo << lista.registros[i].nombre << endl;
    }
    archivo.close();
}
```

ejemploAD.cpp

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

Luis Hernández Yáñez



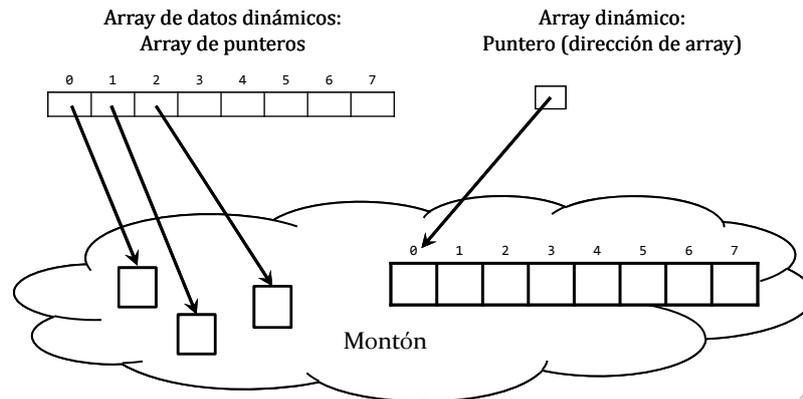
Fundamentos de la programación: Punteros y memoria dinámica

Página 89



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:



Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 90



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

Luis Hernández Yáñez



Fundamentos de la programación: Punteros y memoria dinámica

Página 91



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.

