

# Metodología de la Programación

## Tema 2. Punteros y memoria dinámica

Departamento de Ciencias de la Computación e I.A.



DECSAI  
Universidad de Granada



ETSIIT Universidad de Granada

Curso 2015-16

## Parte II

### Gestión Dinámica de Memoria

# Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

# Estructura de la memoria asociada a un programa

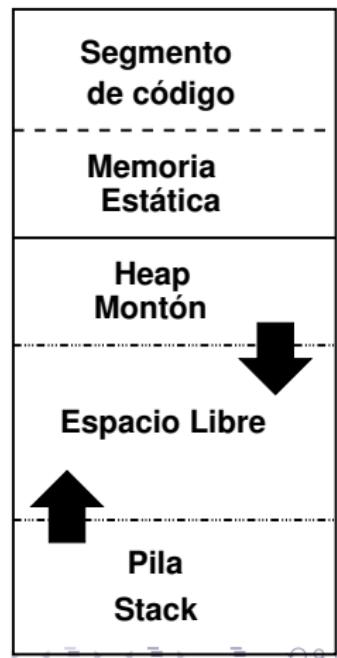
Gracias a la gestión de memoria del Sistema Operativo, los programas tienen una visión más simplificada del uso de la memoria, la cual ofrece una serie de componentes bien definidos.

## Segmento de código

Es la parte de la memoria asociada a un programa que contiene las instrucciones ejecutables del mismo.

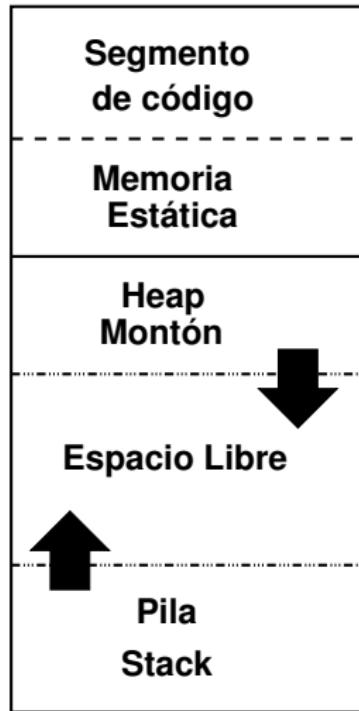
## Memoria estática

- Reserva antes de la ejecución del programa
- Permanece fija
- No requiere gestión durante la ejecución
- El sistema operativo se encarga de la reserva, recuperación y reutilización.
- Variables globales y static.



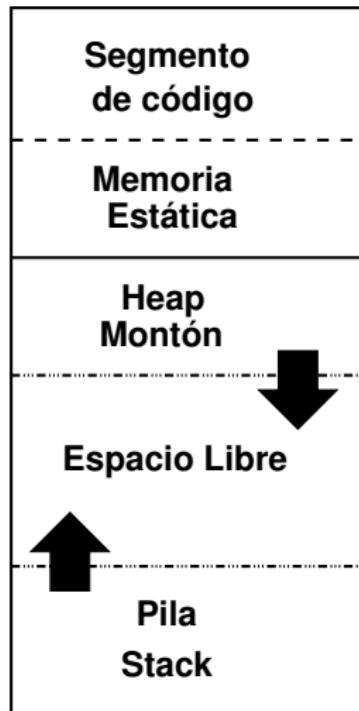
## La pila (Stack)

- Es una zona de memoria que gestiona las llamadas a funciones durante la ejecución de un programa.
- Cada vez que se realiza una llamada a una función en el programa, se crea un **entorno de programa**, que se libera cuando acaba su ejecución.
- La reserva y liberación de la memoria la realiza el S.O. de forma automática durante la ejecución del programa.
- Las variables locales no son variables estáticas. Son un tipo especial de variables dinámicas, conocidas como **variables automáticas**.



## El montón (Heap)

- Es una zona de memoria donde se reservan y se liberan “trozos” durante la ejecución de los programas según sus propias necesidades.
- Esta memoria surge de la necesidad de los programas de “crear nuevas variables” en tiempo de ejecución con el fin de optimizar el almacenamiento de datos.



## Ejemplo

Supongamos que se desea realizar un programa que permita trabajar con una lista de datos relativos a una persona.

```
struct Persona{  
    char nombre[80];  
    int DNI;  
    image foto;  
};
```

¿Qué inconvenientes tiene la definición Persona arrayPersona[100]?

- Si el número de posiciones usadas es mucho menor que 100, tenemos reservada memoria que no vamos a utilizar.
- Si el número de posiciones usadas es mayor que 100, el programa no funcionará correctamente.

"Solución": Ampliar la dimensión del array y volver a compilar.

## Consideraciones:

- La utilización de variables estáticas o automáticas para almacenar información cuyo tamaño no es conocido a priori (sólo se conoce exactamente en tiempo de ejecución) resta generalidad al programa.
- La alternativa válida para solucionar estos problemas consiste en la posibilidad de reservar la memoria justa que se precise (y liberarla cuando deje de ser útil), **en tiempo de ejecución**.
- Esta memoria se reserva en el Heap y, habitualmente, se habla de **variables dinámicas** para referirse a los bloques de memoria del Heap que se reservan y liberan en tiempo de ejecución.

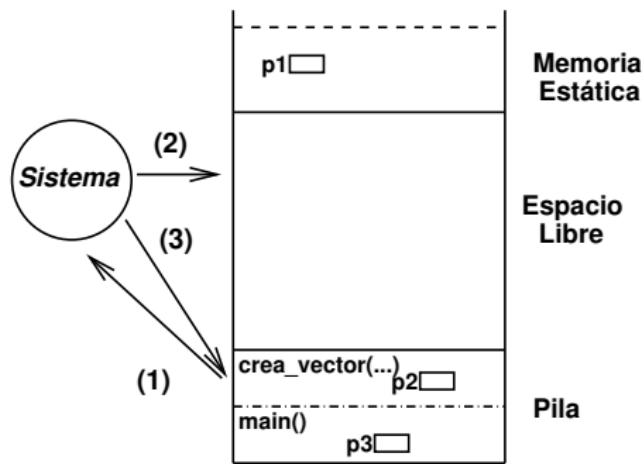
# Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria**
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

# Gestión dinámica de la memoria

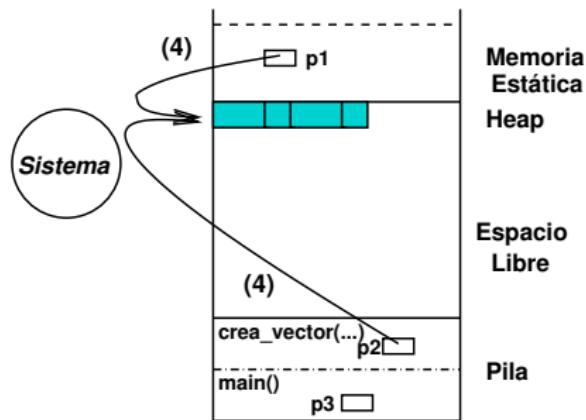
El sistema operativo es el encargado de controlar la memoria que queda libre en el sistema.

- (1) Petición al S.O. (tamaño)
- (2) El S.O. comprueba si hay suficiente espacio libre.
- (3) Si hay espacio suficiente, devuelve la ubicación donde se encuentra la memoria reservada, y marca dicha zona como memoria ocupada.

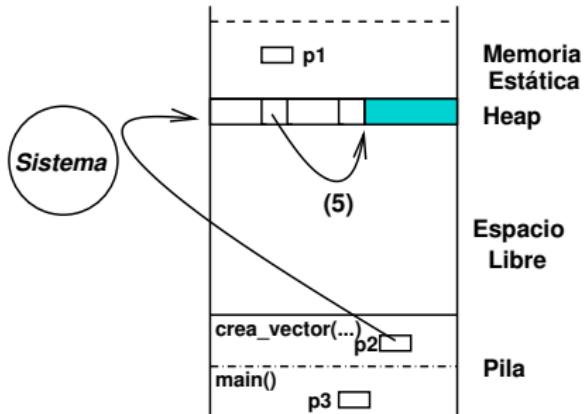


# Reserva de memoria

- (4) La ubicación de la zona de memoria se almacena en una variable estática (**p1**) o en una variable automática (**p2**). Por tanto, si la petición devuelve una dirección de memoria, **p1** y **p2** deben ser variables de tipo *puntero* al tipo de dato que se ha reservado.

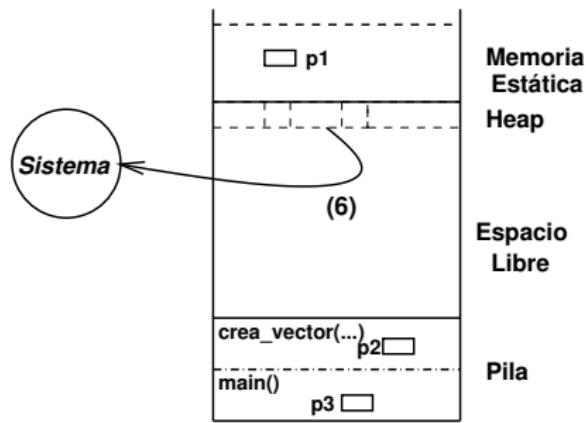


- 5 A su vez, es posible que las nuevas variables dinámicas creadas puedan almacenar la dirección de nuevas peticiones de reserva de memoria.



# Liberación de memoria

- 6 Finalmente, una vez que se han utilizado las variables dinámicas y ya no se van a necesitar más, es necesario liberar la memoria que se está utilizando e informar al S.O. que esta zona de memoria vuelve a estar libre para su utilización.



## ¡ RECORDAR LA METODOLOGÍA !

- ① Reservar memoria.
- ② Utilizar memoria reservada.
- ③ Liberar memoria reservada.

# Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples**
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

# El operador new

```
<tipo> *p;  
p = new <tipo>;
```

- **new** reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (`sizeof(tipo)` bytes), devolviendo la dirección de memoria dónde empieza la zona reservada.
- Si `new` no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

## Otra opción (no recomendable)

```
<tipo> *p;  
p = new (nothrow) <tipo>;
```

En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (0).

## Ejemplo

```
int main(){
    int *p;

    p = new int;
    *p = 10;
}
```

### Notas:

- Observar que **p** se declara como un puntero más.
- Se pide memoria en el Heap para guardar un dato **int**. Si hay espacio para satisfacer la petición, **p** apuntará al principio de la zona reservada por **new**. Asumiremos que siempre hay memoria libre para asignar.
- Se trabaja, como ya sabemos, con el objeto referenciado por **p**.

# El operador delete

```
delete puntero;
```

**delete** permite liberar la memoria del Heap que previamente se había reservado y que se encuentra referenciada por un puntero.

## Ejemplo

```
int main(){
    int *p, q=10;

    p = new int;
    *p = q;
    .....
    delete p;
}
```

## Notas:

- El objeto referenciado por **p** deja de ser “operativo” y la memoria que ocupaba está disponible para nuevas peticiones con **new**.

# Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos**
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

## Objetos dinámicos compuestos

Para el caso de objetos compuestos (p.e. struct) la metodología a seguir es la misma, aunque teniendo en cuenta las especificidades de los tipos compuestos.

En el caso de los **struct**, la instrucción **new** reserva la memoria necesaria para almacenar todos y cada uno de los campos de la estructura.

```
int main(){
    Persona *yo;
    struct Persona{
        char nombre[80];
        char DNI[10];
    };
    yo = new Persona;
    lee_linea((*yo).nombre,80);
    lee_linea((*yo).DNI,10);
    .....
    delete yo;
}
```

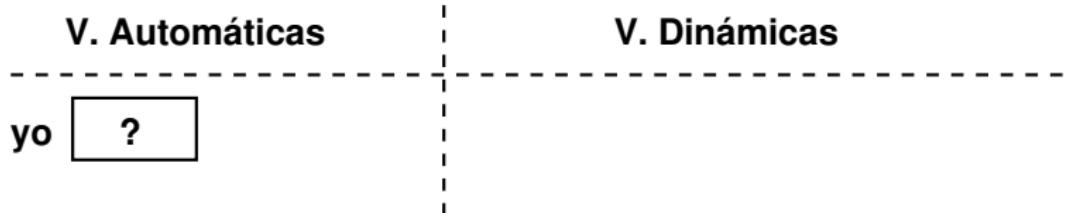
# Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados**
- 17 Arrays dinámicos
- 18 Matrices dinámicas

## Ejemplo: Objetos dinámicos autoreferenciados

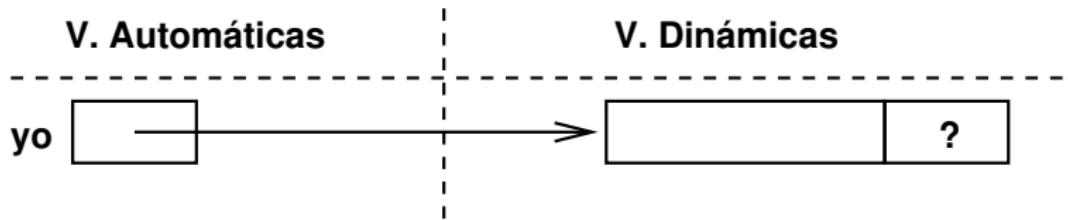
Dada la definición del siguiente tipo de dato Persona y declaración de variable

```
struct Persona{  
    char nombre[80];  
    Persona *amigos;  
};  
  
Persona *yo;
```



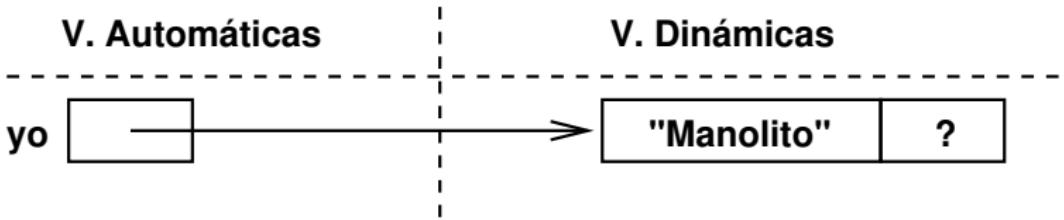
¿Qué realiza la siguiente secuencia de instrucciones?

1. `yo = new Persona;`



Reserva memoria para almacenar (en el Heap) un dato de tipo Persona. Como es un tipo compuesto, realmente se reserva espacio para cada uno de los campos que componen la estructura, en este caso, un array de 80 posiciones y un *puntero*.

2. `strcpy(yo->nombre, "Manolito");`

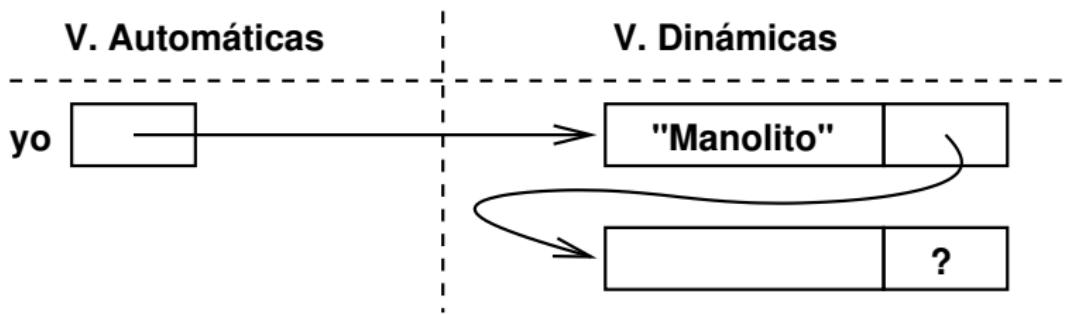


Asigna un valor al campo nombre del nuevo objeto dinámico creado.

Como la referencia a la variable se realiza mediante un puntero, puede utilizarse el operador flecha (`->`) para el acceso a los campos de un registro.

3. `yo->amigos = new Persona;`

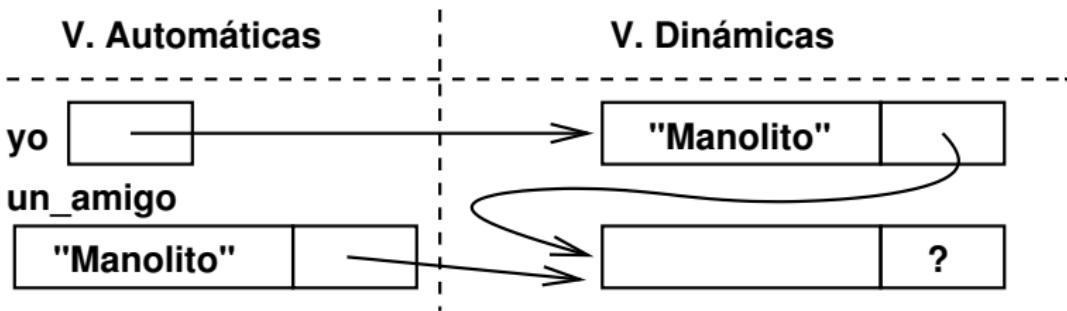
Reserva memoria para almacenar (en el Heap) otro dato de tipo Persona, que es referenciada por el campo amigos de la variable apuntada por yo (creada anteriormente).



Por tanto, a partir de una variable dinámica se pueden definir nuevas variables dinámicas siguiendo una filosofía semejante a la propuesta en el ejemplo.

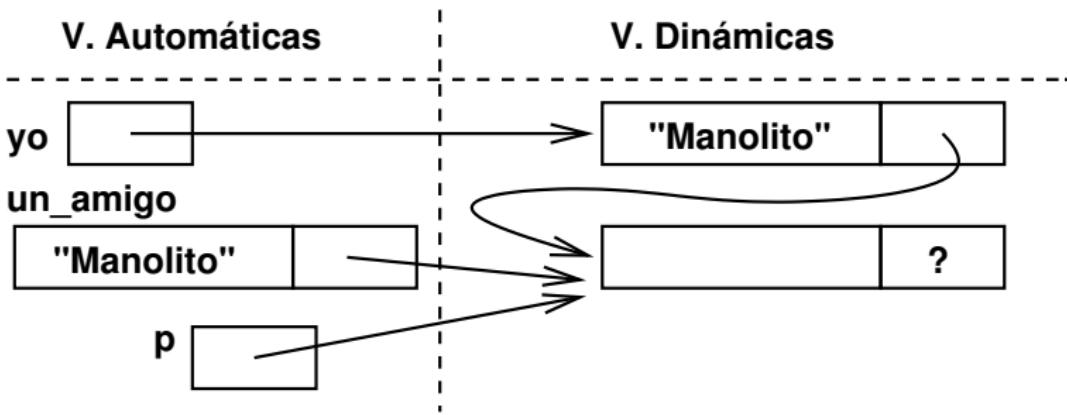
4. Persona un\_amigo = \*yo;

Se crea la variable automática `un_amigo` y se realiza una copia de la variable que es apuntada por `yo`.



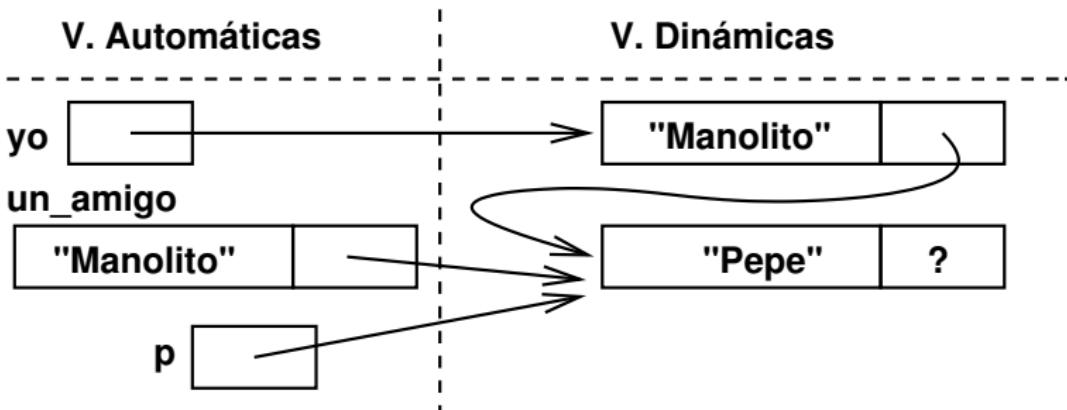
5. Persona \*p = yo->amigos;

La variable p almacena la misma dirección de memoria que el campo amigos de la variable apuntada por yo.

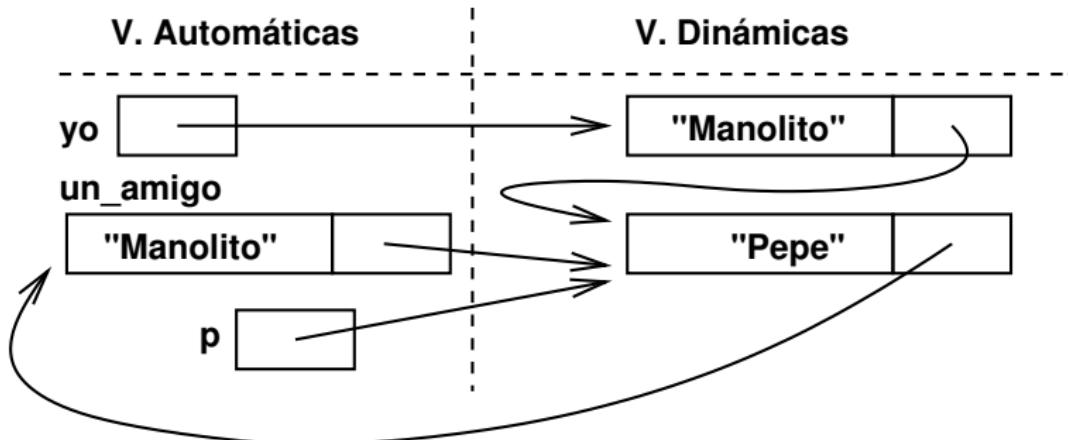


6. `strcpy(p->nombre, "Pepe");`

Usando la variable `p` (apunta al último dato creado) damos valor al campo `nombre`.

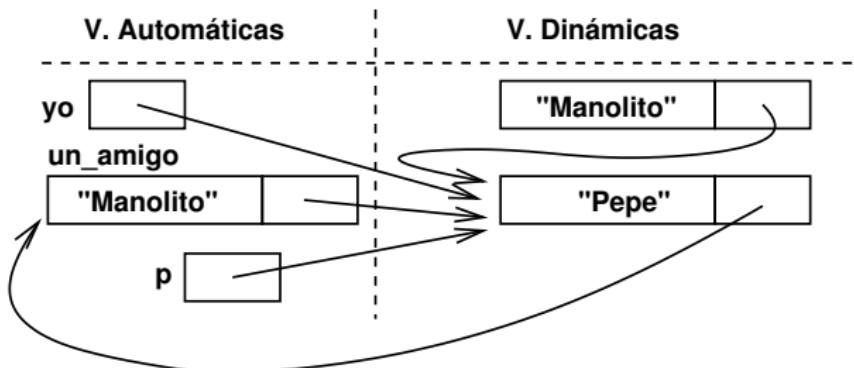


7. `p->amigos = &un_amigo;`



Es posible hacer que una variable dinámica apunte a una variable automática o estática usando el operador `&`.

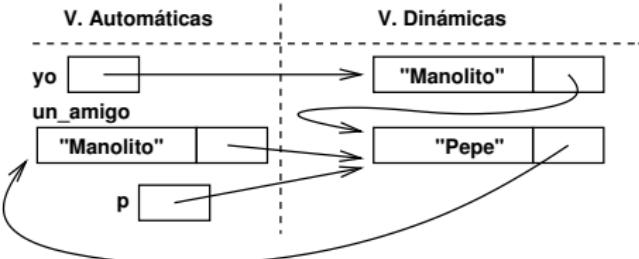
8.  $\text{yo} = \text{p};$



Con esta orden se pierde el acceso a uno de los objetos dinámicos creados, siendo imposible su recuperación. Por tanto, antes de realizar una operación de este tipo, hay que asegurar:

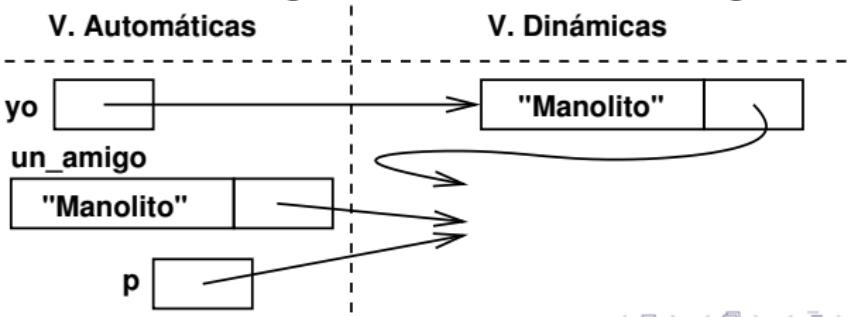
- que no perdemos la referencia a ese objeto (existe otro puntero que lo referencia).
- Si la variable ya no es útil para el programa, debemos liberar antes la memoria (indicando al sistema que esa zona puede ser utilizada para almacenar otros datos).

Volvamos a la situación anterior



9. `delete un_amigo.amigos;`

Esta sentencia libera la memoria cuya dirección de memoria se encuentra almacenada en el campo amigos de la variable `un_amigo`.



- La liberación implica que la zona de memoria queda disponible para que otro programa (o él mismo) pudieran volver a reservarla. Sin embargo, la dirección que almacenaba el puntero usado para la liberación (y el resto de punteros) se mantiene tras la liberación.
- Por consiguiente, **hay que tener cuidado y no usar la dirección almacenada en un puntero que ha liberado la memoria**. Por ejemplo:

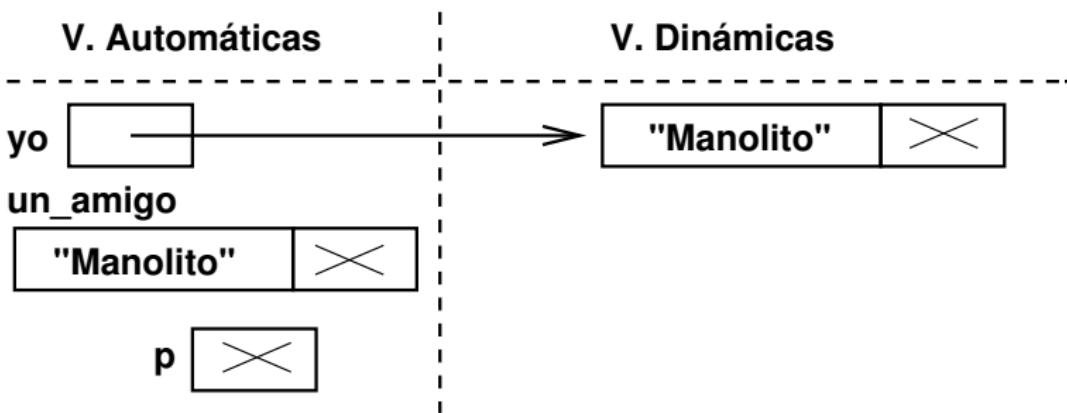
```
strcpy(un_amigo.amigos->nombre, "Alex");
```

- De igual forma, hay que tener cuidado con todos aquellos apuntadores que mantenían la dirección de una zona liberada, ya que se encuentran con el mismo problema.

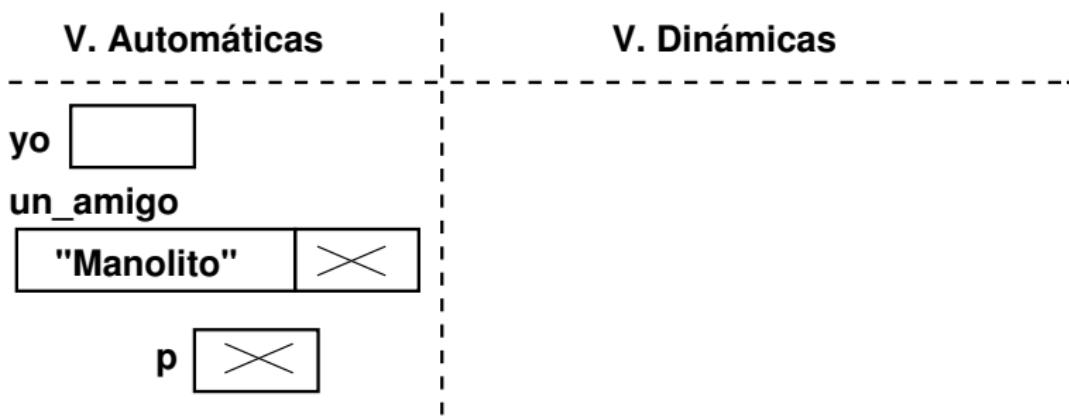
```
strcpy(yo->amigos->nombre, "Alex");
```

Una forma de advertir esta situación es asignar la dirección nula a todos aquellos punteros que apunten a zonas de memoria que ya no existen.

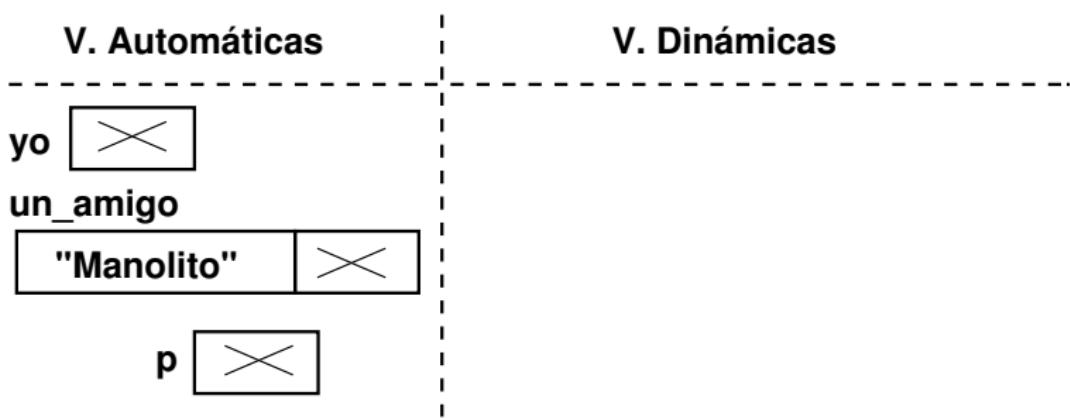
10. `yo->amigos = un_amigo.amigos = p = 0;`



11. `delete yo;`



12.    `yo = 0;`



# Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 **Arrays dinámicos**
- 18 Matrices dinámicas

# Arrays dinámicos

- Hasta ahora sólo podíamos crear un array conociendo *a priori* el número máximo de elementos que podría llegar a tener. P.e.  
`int vector[20];`
- Esa memoria está ocupada durante la ejecución del módulo en el que se realiza la declaración.
- Para reservar la memoria estrictamente necesaria:

## El operador new □

```
<tipo> *p;  
p = new <tipo> [num];
```

- Reserva una zona de memoria en el Heap para almacenar `num` datos de tipo `<tipo>`, devolviendo la dirección de memoria inicial.  
`num` es un entero estrictamente mayor que 0.

La liberación se realiza con

El operador delete □

```
delete □ puntero;
```

libera (pone como disponible) la zona de memoria **previamente reservada** por una orden new □, zona referenciada por puntero.

Con la utilización de esta forma de reserva dinámica podemos crear arrays que tengan justo el tamaño necesario. Podemos, además, crearlo justo en el momento en el que lo necesitamos y destruirlo cuando deje de ser útil.

# Ejemplo I

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int *v=0, n;
6
7     cout << "Numero de casillas: ";
8     cin >> n;
9     // Reserva de memoria
10    v = new int [n];
11
12    // Procesamiento del vector dinamico:
13    //      lectura y escritura de su contenido
14    for (int i= 0; i<n; i++) {
15        cout << "Valor en casilla "<<i<< ":";
```

## Ejemplo II

```
16     cin >> v[i];
17 }
18 cout << endl;
19
20
21 for (int i= 0; i<n; i++)
22     cout << "En la casilla " << i
23             << " guardo: " << v[i] << endl;
24
25 // Liberar memoria
26 delete [] v;
27 v = 0;
28
29 }
```

## Ejemplo

Una función que devuelve una copia en un array dinámico de un array automático.

```
1 #include <iostream>
2 using namespace std;
3
4 int *copia_vector(const int v[], int n){
5     int *copia = new int[n];
6     for (int i=0; i<n; i++)
7         copia[i]=v[i];
8     return copia;
9 }
10 int main(){
11     int v1[30], *v2=0, m;
12     cout << "Numero de casillas: ";
13     cin >> m;
```

```
14  for (int i=0; i<m; i++) { // Rellenar el vector
15      cout << "Valor en casilla "<<i<< ":" ;
16      cin >> v1[i];
17  }
18  cout << endl;
19
20 // Copiar en v2 (dinámico) el vector v1
21 v2 = copia_vector(v1,m);
22
23 // Escribir vector v2
24 for (int i=0; i<m; i++)
25     cout << "En la casilla " << i
26             << " guardo: "<< v2[i] << endl;
27 // Liberar memoria
28 delete [] v2;
29 v2 = 0;
```

30 }

## ¡Cuidado!

Un **error** muy común a la hora de construir una función que copie un array es el siguiente:

```
int *copia_vector(const int v[], int n){  
    int copia[100];  
    for (int i=0; i<n; i++)  
        copia[i]=v[i];  
    return copia;  
}
```

## ¡Cuidado!

Al ser copia una variable local no puede ser usada fuera del ámbito de la función en la que está definida.

# Ejemplo:

Ampliación del espacio ocupado por un array dinámico (Ampliar)

```
void ampliar (int *&v, int old_tama, int new_tama){  
    if (new_tama > old_tama){  
        int *v_ampliado = new int[new_tama];  
  
        for (int i=0; i<old_tama; i++)  
            v_ampliado[i] = v[i];  
  
        delete []v;  
        v = v_ampliado;  
    }  
}
```

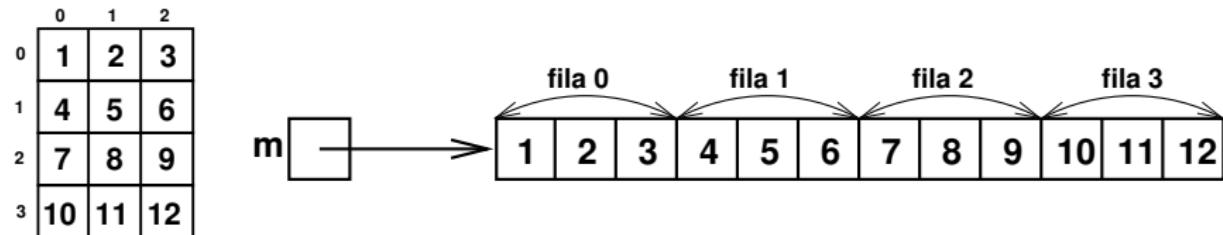
Cuestiones a tener en cuenta:

- *v* se pasa por referencia porque se va a modificar.
- Es necesario liberar *v* antes de asignarle el valor de *v\_ampliado*.

# Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 **Matrices dinámicas**

# Matriz 2D usando un array 1D



- Creación de la matriz:

```
int *m;
int nfil, ncol;
m = new int[nfil*ncol];
```

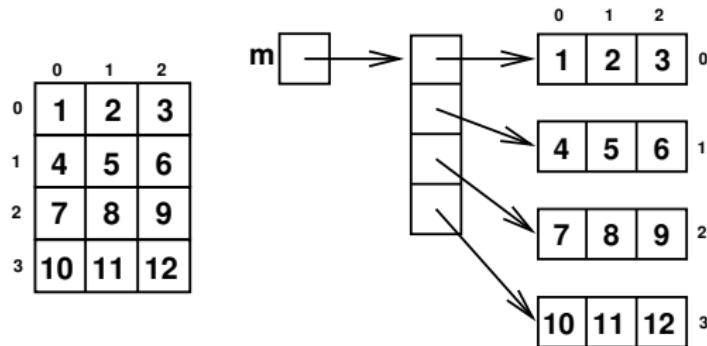
- Acceso al elemento f,c:

```
int a;
a = m[f*ncol+c];
```

- Liberación de la matriz:

```
delete[] m;
```

# Matriz 2D usando un array 1D de punteros a arrays 1D



- Creación de la matriz:

```
int **m;
int nfil, ncol;
m = new int*[nfil];
for (int i=0; i<nfil; ++i)
    m[i] = new int[ncol];
```

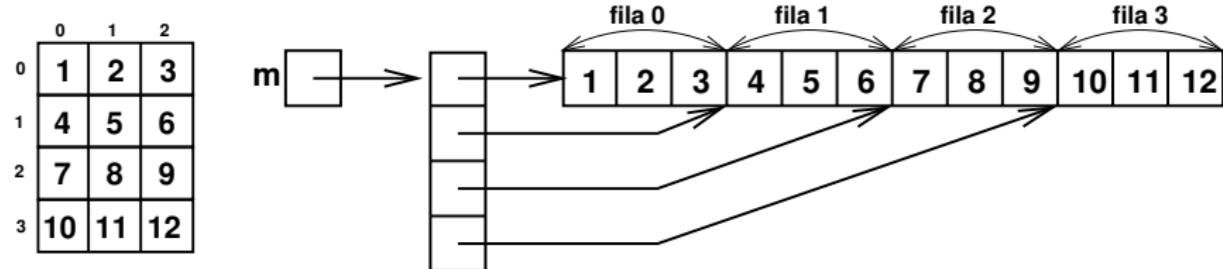
- Acceso al elemento f,c:

```
int a;
a = m[f][c];
```

- Liberación de la matriz:

```
for(int i=0;i<nfil; ++i)
    delete[] m[i];
delete[] m;
```

# Matriz 2D usando un array 1D de punteros a un único array



- Creación de la matriz:

```

int **m;
int nfil, ncol;
m = new int*[nfil];
m[0] = new int[nfil*ncol];
for (int i=1; i<nfil; ++i)
    m[i] = m[i-1]+ncol;
    
```

- Acceso al elemento f,c:

```

int a;
a = m[f][c];
    
```

- Liberación de la matriz:

```

delete[] m[0];
delete[] m;
    
```