Metodología de la Programación

Tema 2. Punteros y memoria dinámica

Parte 2. Memoria dinámica

Sylvia Acid (acid@decsai.ugr.es)
derivado de la obra de Andrés Cano
Departamento de Ciencias de la Computación e I.A.

Curso 2025-26

Contenido del Tema 2

Parte 1: Punteros
Parte 2: Gestión Dinámica de Memoria

Indice

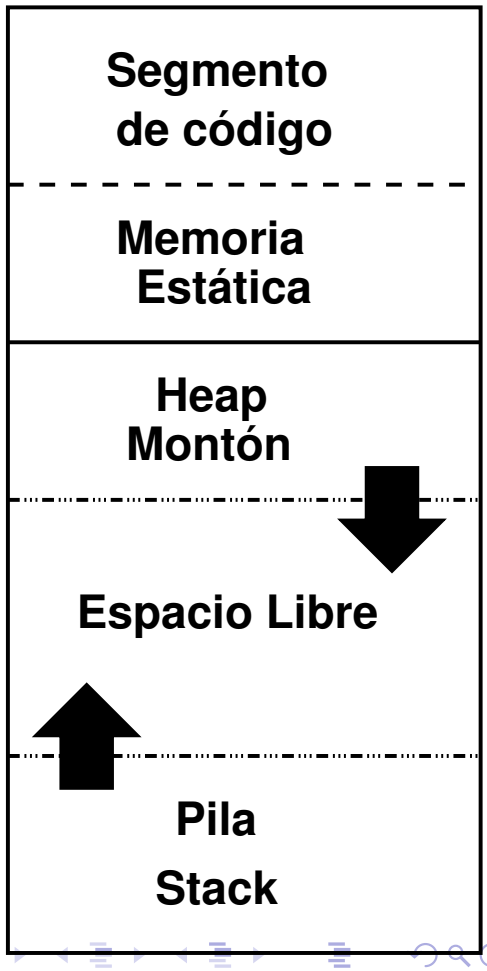
Las limitaciones

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.

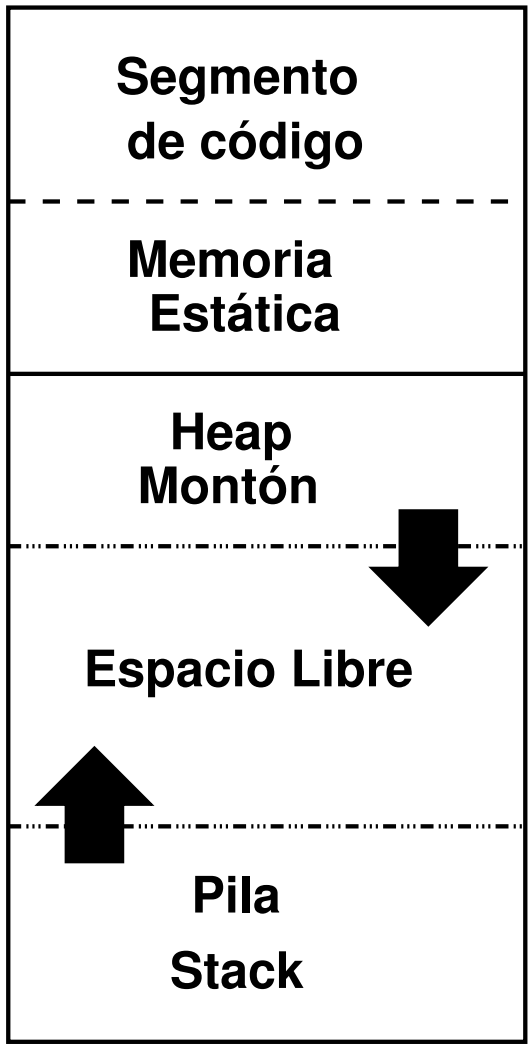
Memoria estática

- 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**.



Motivación

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

```
struct Persona{
  char nombre[80];
  char DNI[8];
  Imagen foto;
};
```

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

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

"Solución" a 2: Ampliar la dimensión del array y volver a compilar.

Características

Los arrays estáticos deben conocer su tamaño máximo en **tiempo de compilación**.
Su tamaño no puede depender de la entrada del usuario
Su tamaño no puede variar durante en tiempo de ejecución

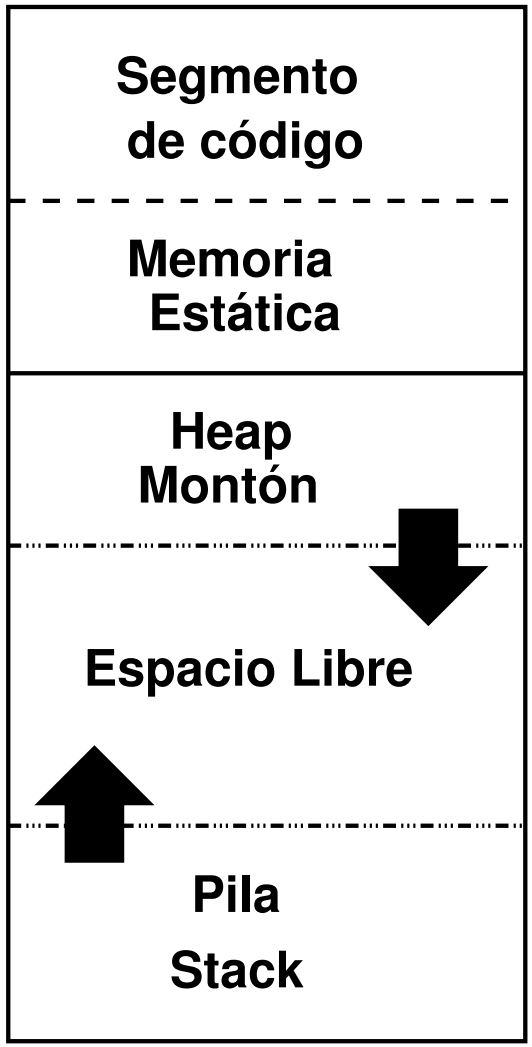
Consideraciones:

- La utilización de variables estáticas o automáticas para almacenar información cuyo tamaño no es conocido a priori, resta generalidad al programa.
- La alternativa para solucionar estos problemas consiste en 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. Se habla de **variables dinámicas** para referirse a los bloques de memoria del Heap que se reservan y liberan en **tiempo de ejecución**.

Un nuevo espacio de memoria y un nuevo tiempo

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.



Variables Dinámicas

Variables que refieren a bloques de memoria del Heap que se reservan y liberan en tiempo de ejecución. Son **referenciadas por una variable de tipo puntero a** Tipo.

Consideraciones:

La gestión de memoria es una causa frecuente de errores, por lo que se aconseja el uso de **clases bien diseñadas** que **garanticen accesos seguros**.

Indice

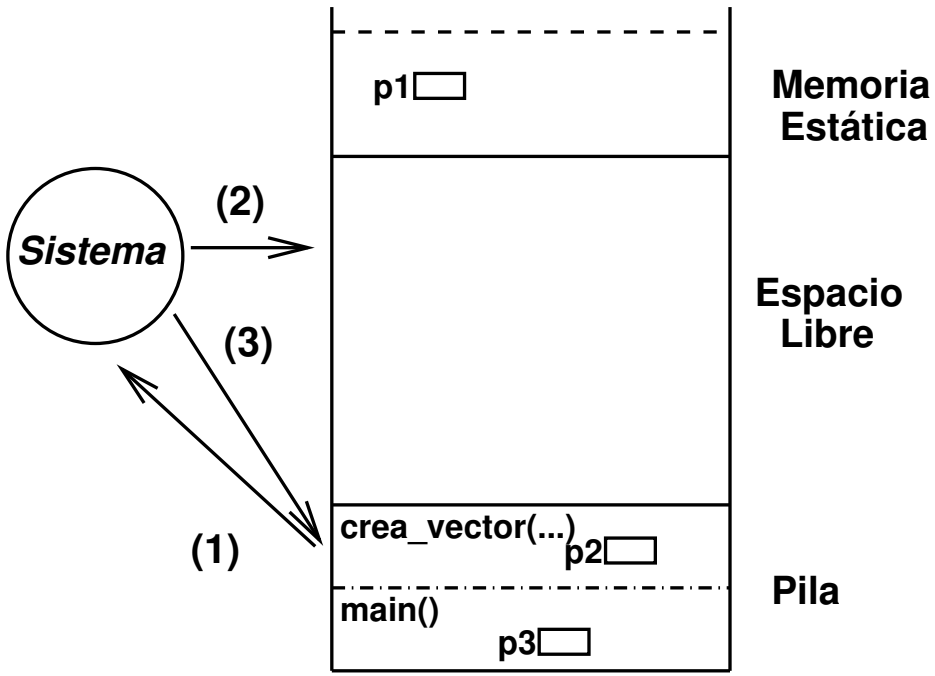
Gestión dinámica de la memoria

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

(1) Petición al S.O. (tamaño)

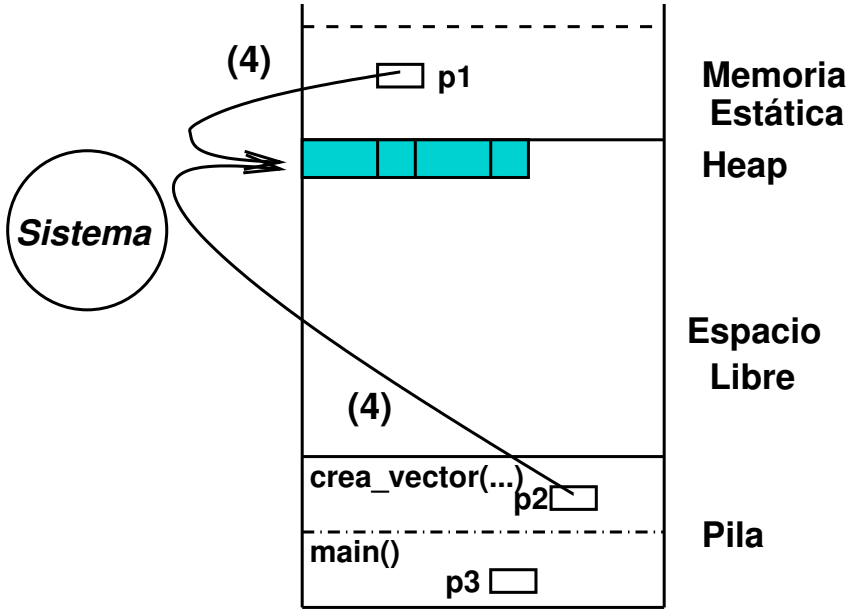
(2) El S.O. comprueba si hay suficiente espacio libre.

(3) Si hay espacio suficiente, **devuelve la ubicación** de comienzo de la memoria reservada, y **marca el espacio** de memoria como ocupada.

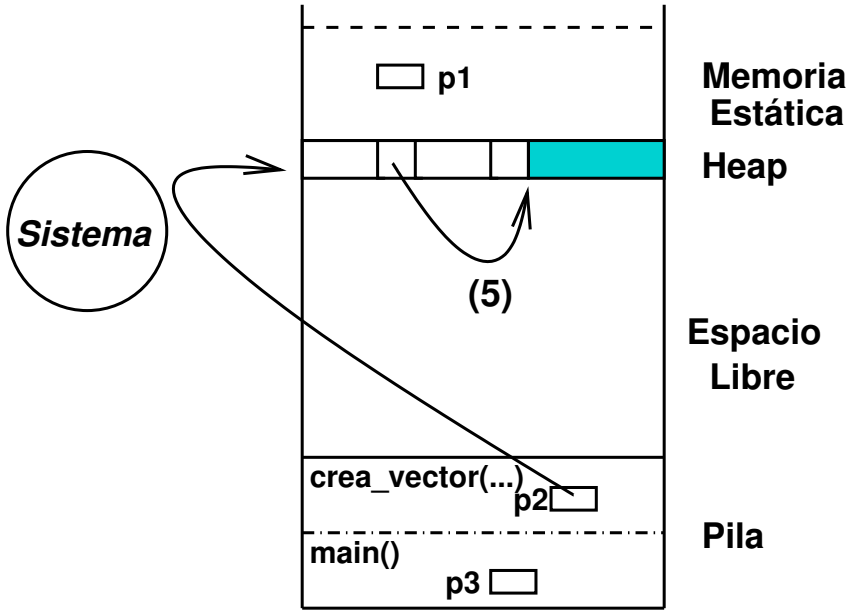


Reserva de memoria

(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.

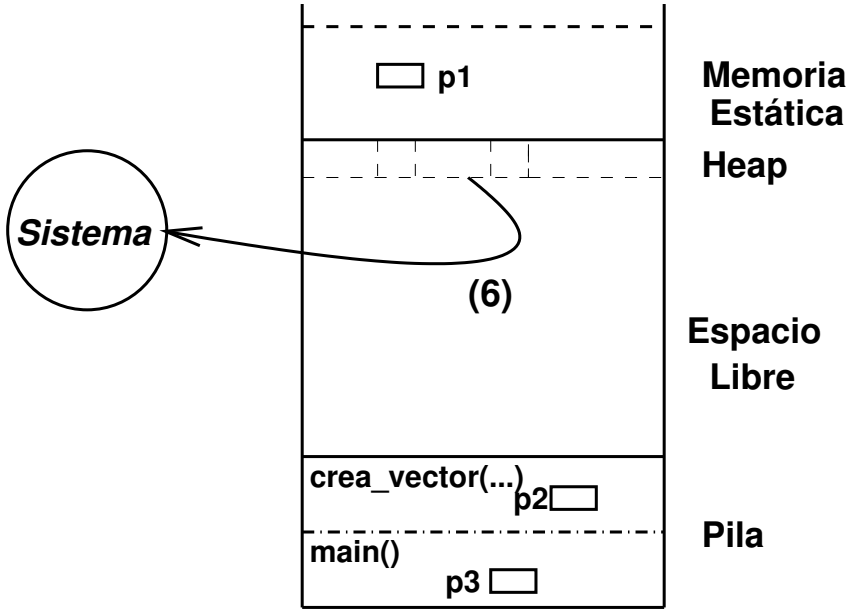


5 A su vez, es posible que las nuevas variables dinámicas creadas puedan almacenar la dirección de nuevas peticiones de reserva de memoria.



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.



La gestión de la memoria dinámica (en el heap), es responsabilidad del programador.

Debe seguir esta metodología

1. Reservar memoria. Uso de **new**
2. Utilizar memoria reservada.
3. Liberar memoria reservada. Uso de **delete**

Indice

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.
- Supondremos que siempre habrá suficiente memoria.

Ejemplo

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

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

Notas:

- `p` se declara como un puntero a int cualquiera.
- Se pide memoria en el Heap para guardar un dato `int`. Si hay espacio para satisfacer la petición, `p` apuntará a la zona reservada por `new`.
- El uso de ese espacio es, como ya sabemos, a través de un puntero `p` al objeto referenciado.

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+1;
  .......
  delete p;
}
```

- Trás delete, el objeto referenciado por `p` deja de ser "operativo" y la memoria queda disponible para nuevas peticiones con `new`.
- El operador `delete` **no** cambia el valor de `p`.

Indice

Objetos dinámicos, struct

Para el caso de objetos compuestos como `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.

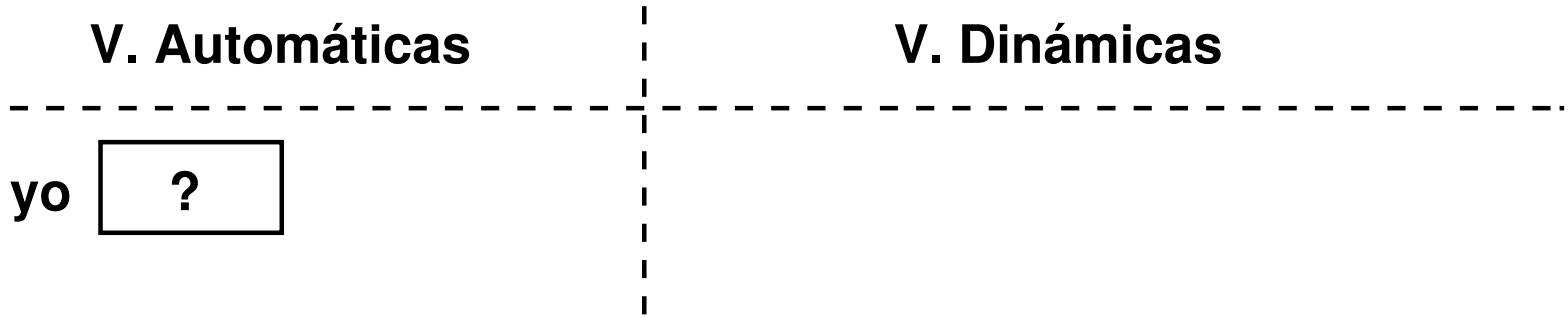
```
struct Persona{
  char nombre[80];
  char DNI[8];
};
```

```
int main(){
    Persona *yo;

    yo = new Persona;
    lee_linea((*yo).nombre,80);
    lee_linea((*yo).DNI,8);
    .........
    delete yo;
}
```

Ejemplo: registros dinámicos autoreferenciados

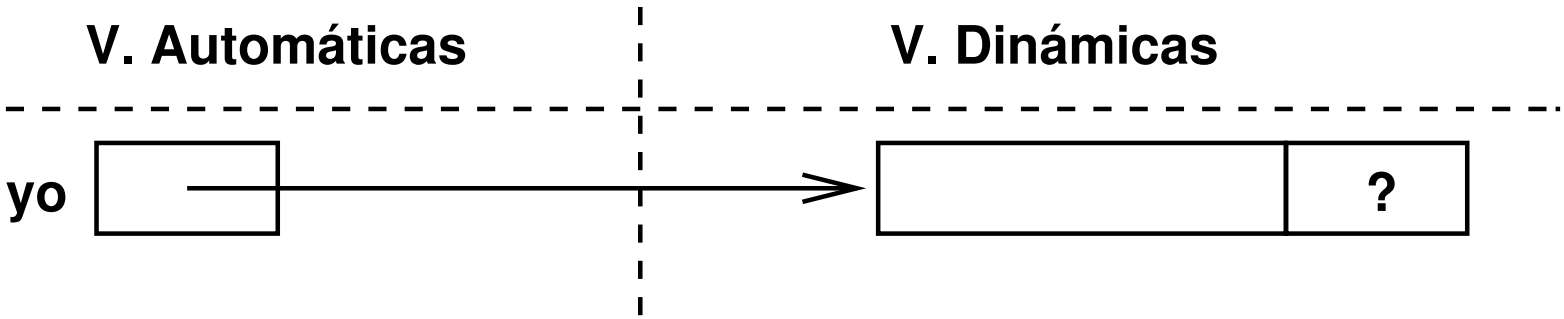
Dada la definición del siguiente tipo de dato Persona y declaración de variable

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

Persona *yo;
```



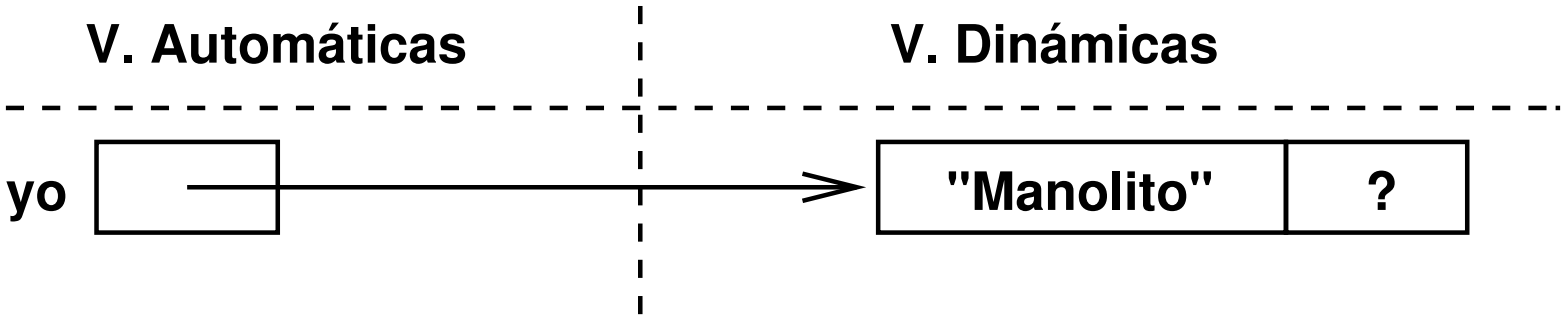
¿Qué realiza la siguiente secuencia de instrucciones?

1. `yo = new Persona;`



Reserva memoria para almacenar (en el Heap) un dato de tipo `Persona`.
Como es un tipo compuesto, se reserva espacio para cada uno de los campos que componen la estructura.
Espacio $= \sum$ 80 char $+$ espacio de un *puntero*.

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



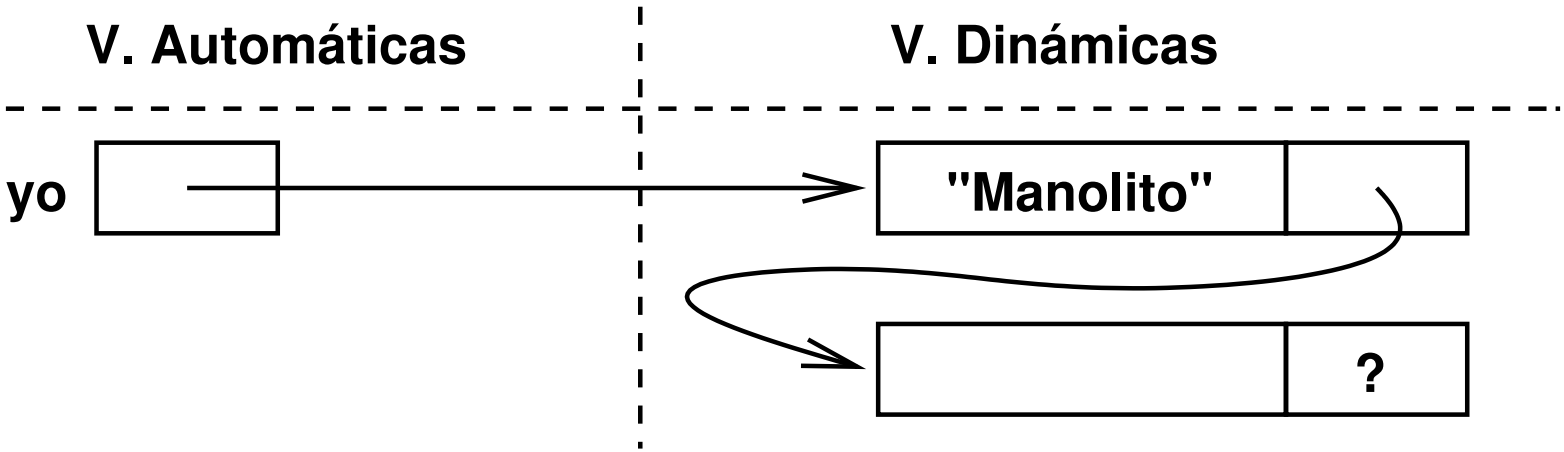
Explotación y uso del espacio del heap.

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 (->) para el acceso a los campos de un registro.

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

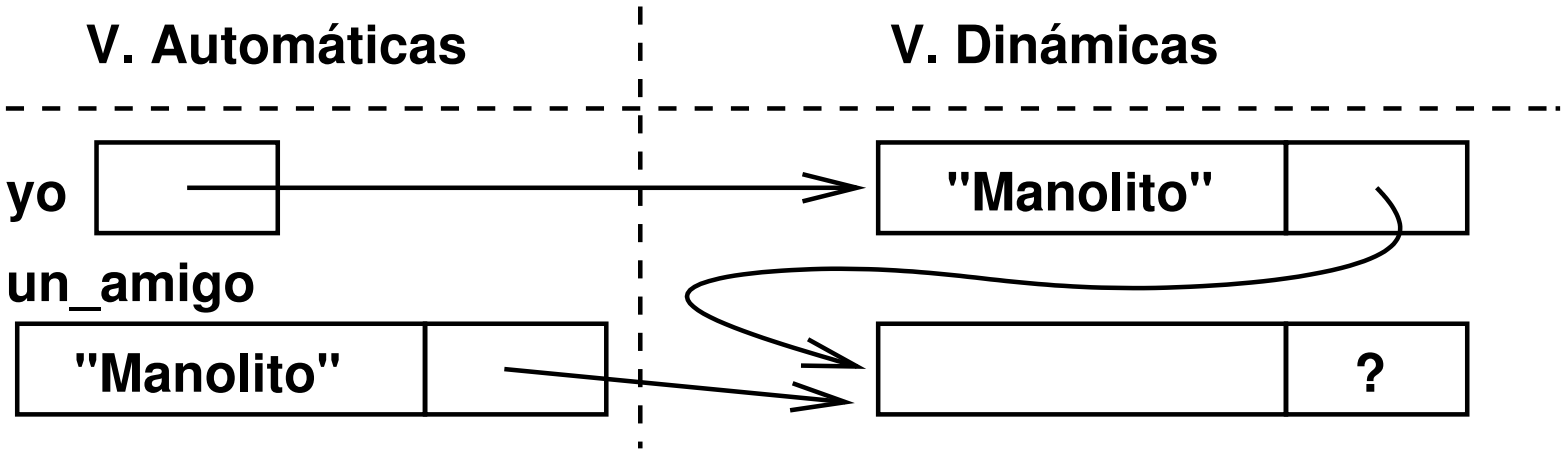
Reserva memoria para almacenar (en el Heap) otro dato de tipo `Persona`, que es referenciada por el campo `sig` de la variable apuntada por `yo` (creada anteriormente).



Por tanto, a partir de una variable dinámica se pueden definir nuevas variables dinámicas siguiendo el mismo procedimiento de la explotación del puntero dinámico.

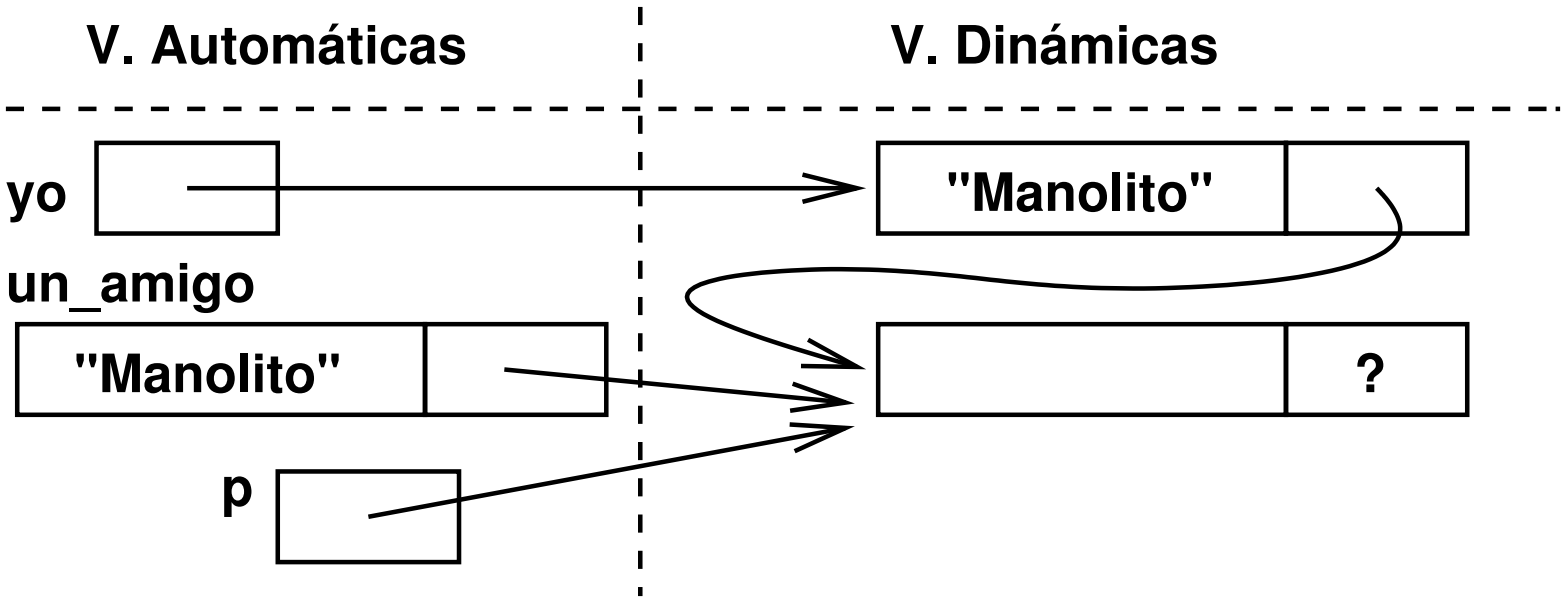
4. `Persona un_amigo = *yo;`

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



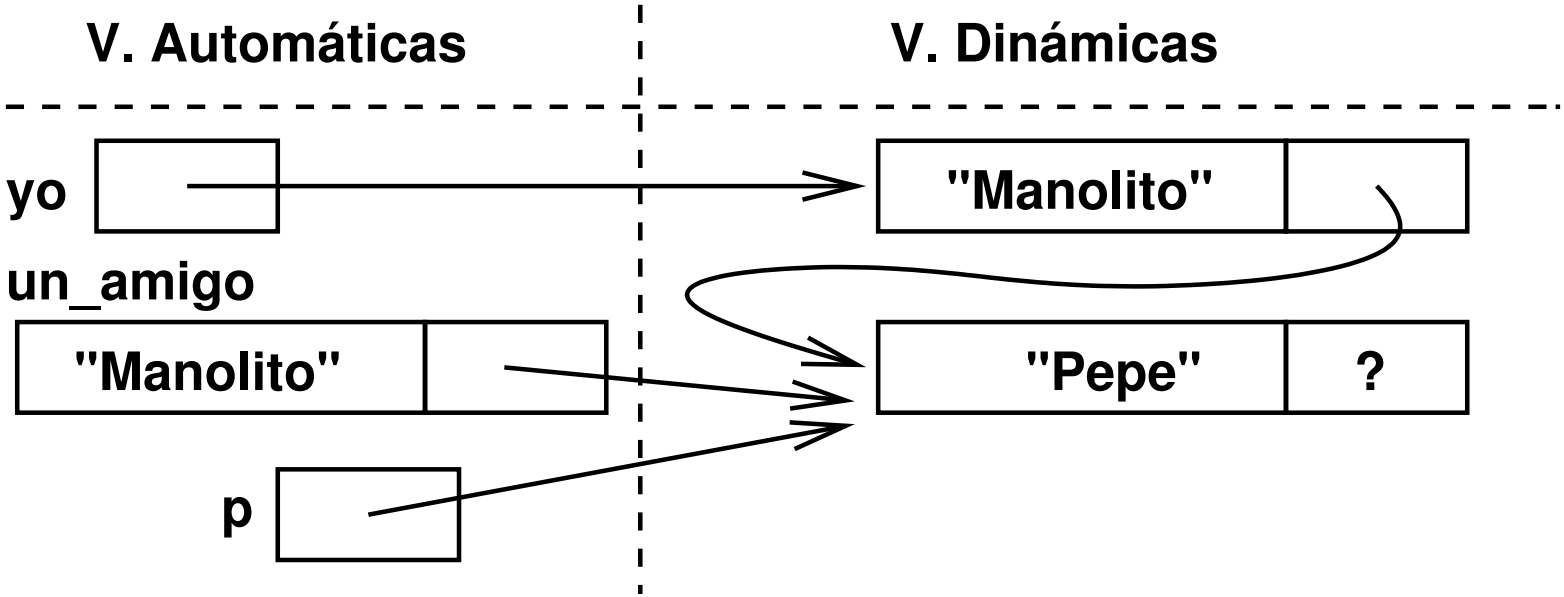
5. `Persona *p = yo->sig;`

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

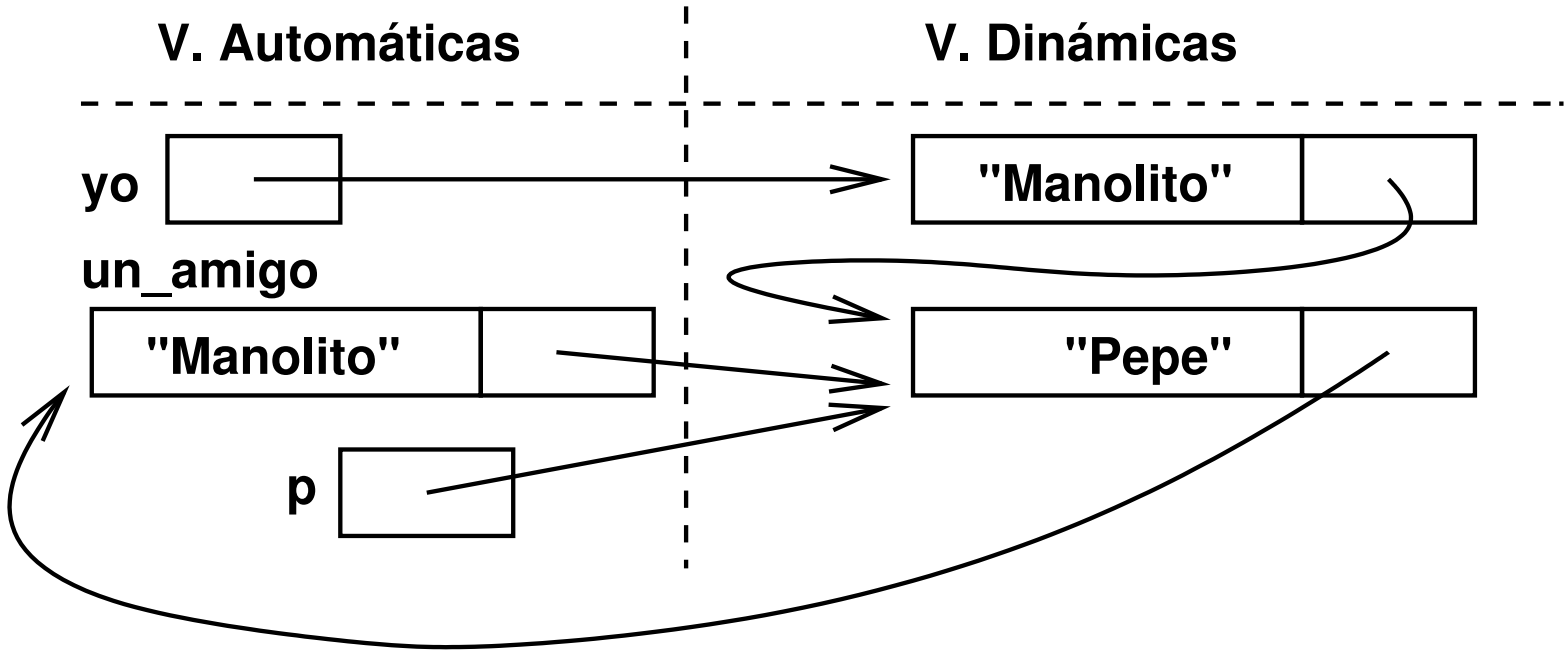


```
6.    strcpy(p->nombre,"Pepe");
```

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

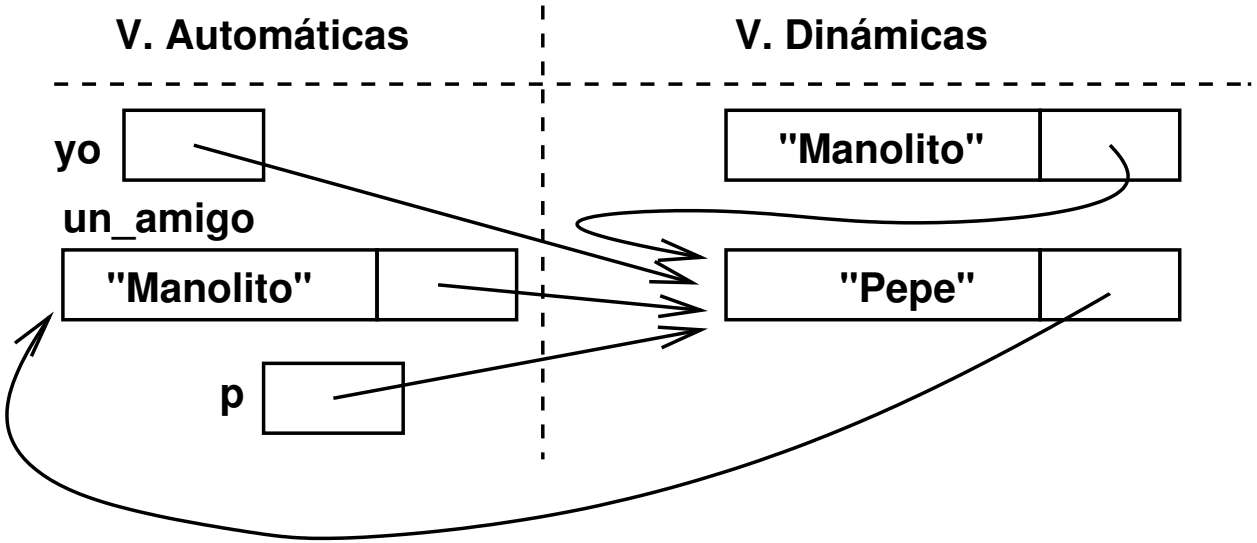


7. `p->sig = &un_amigo;`



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

8. `yo = p;`

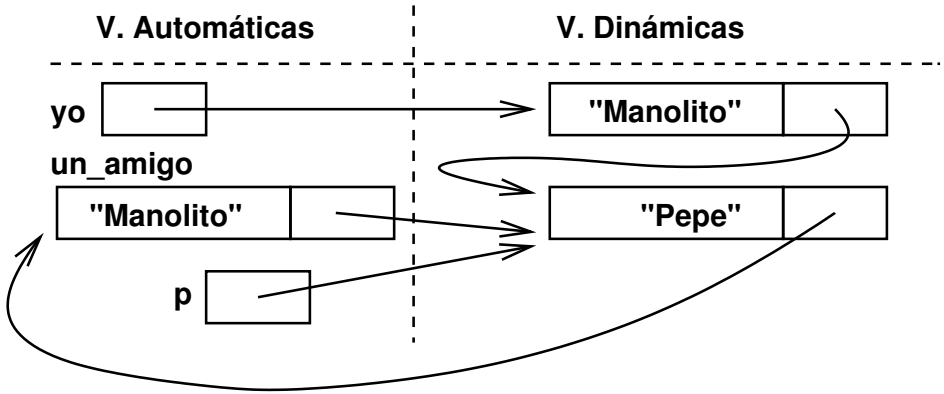


Con esta orden se pierde el acceso a uno de los objetos dinámicos creados, siendo **imposible su recuperación**. Por tanto, antes de realizar una operación de este tipo, hay que asegurarse que:

a) no perdemos la referencia a ese objeto (existe otro puntero que lo referencia).

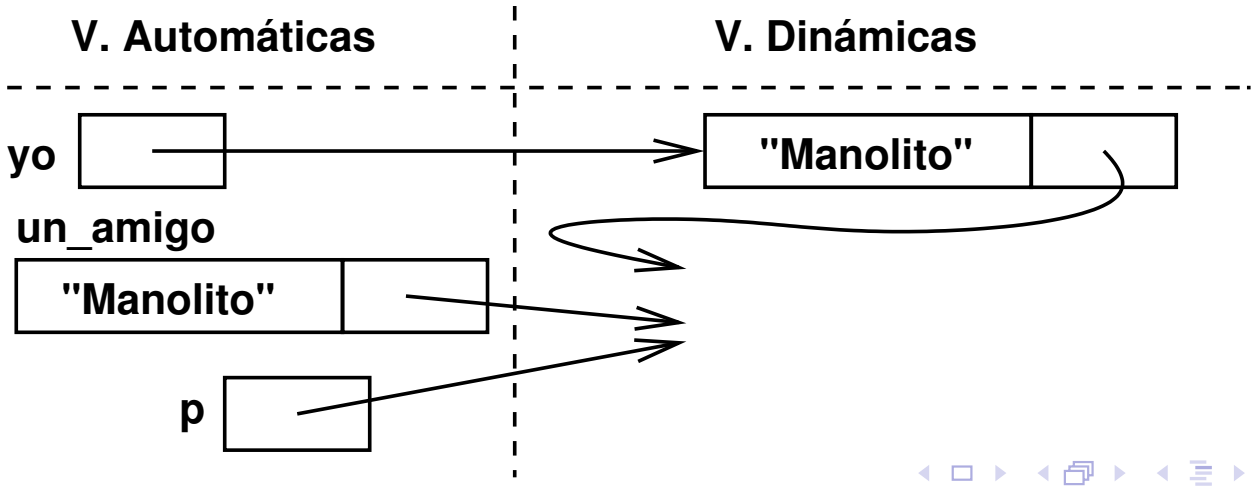
b) si la variable ya no es útil para el programa, antes de la asignación, debemos liberar la memoria (indicando al sistema que esa zona queda libre y puede ser reutilizada).

Volvamos a la situación anterior



9. **delete** `un_amigo.sig;`

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



- La liberación implica que la zona de memoria queda libre para que el programa (u otro programa) 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. Los punteros se han quedado 'colgados' (Dangling).
- Por consiguiente, **hay que tener cuidado y no usar la dirección almacenada en un puntero que ha liberado la memoria**. Por ejemplo:

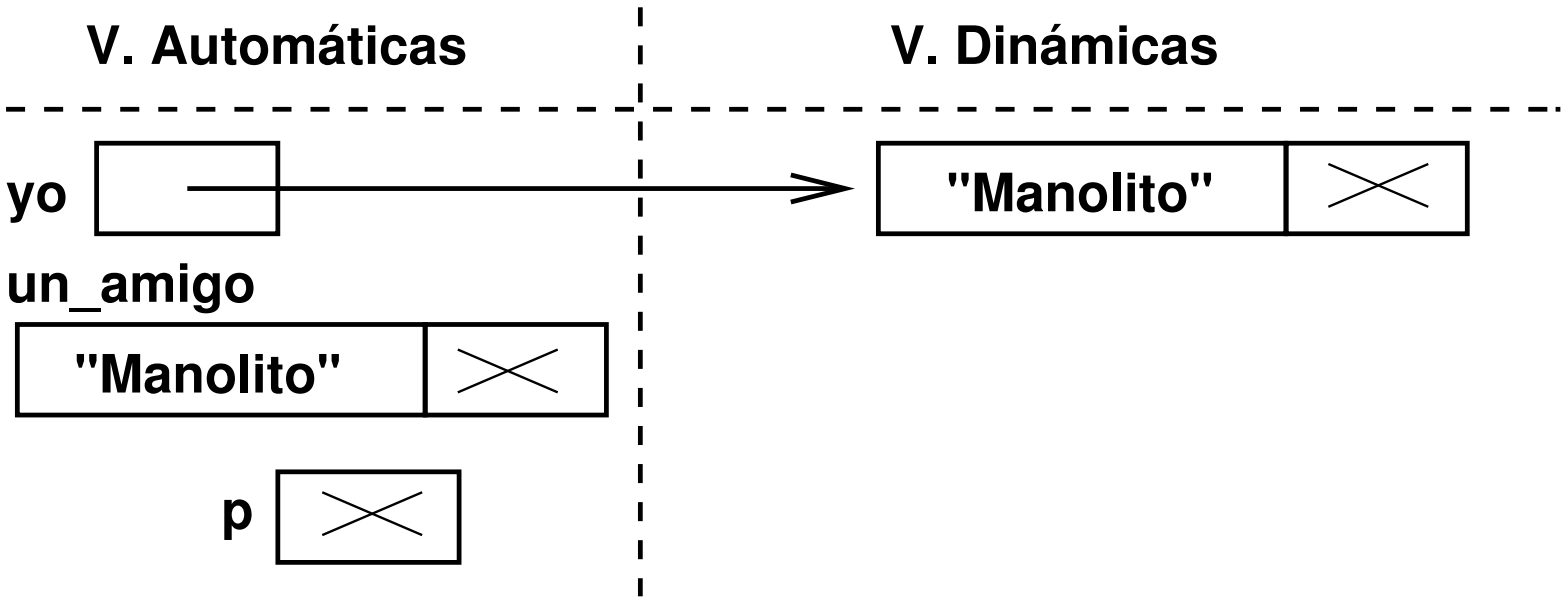
```
strcpy(un_amigo.sig->nombre,"Alex");
```

- 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.

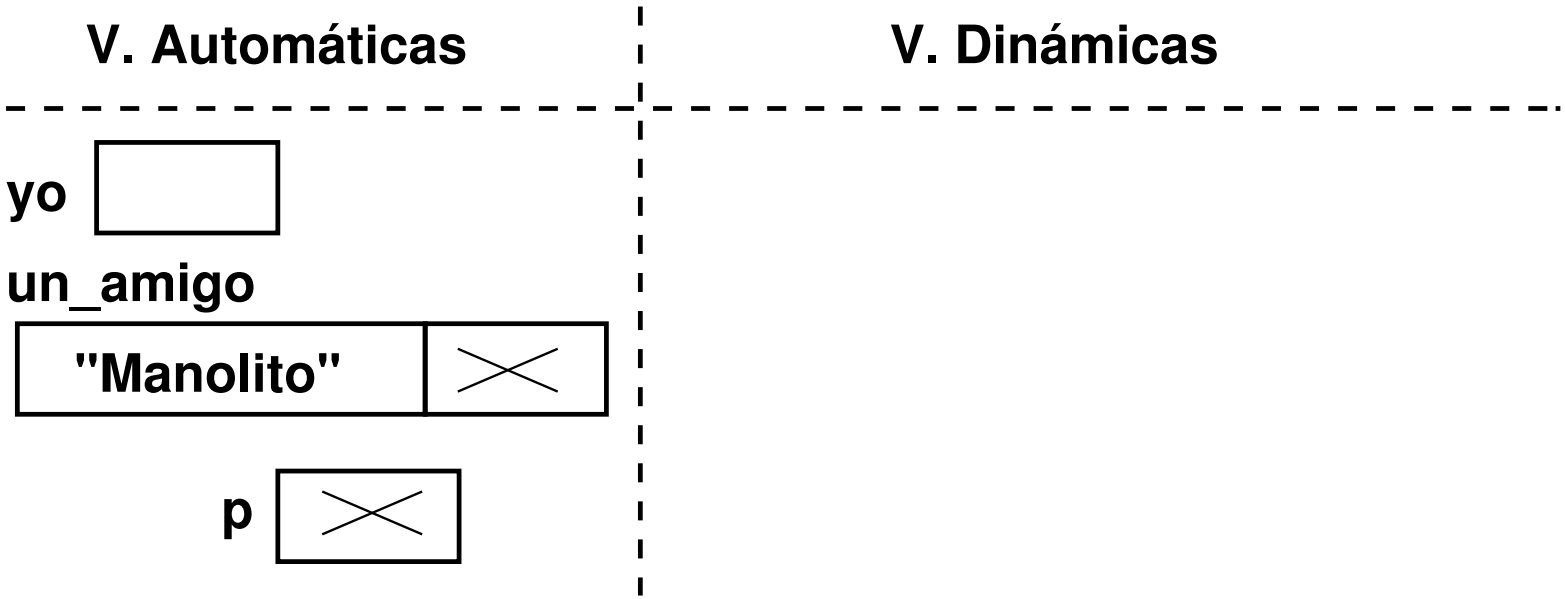
```
strcpy(yo->sig->nombre,"Alex");
```

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.

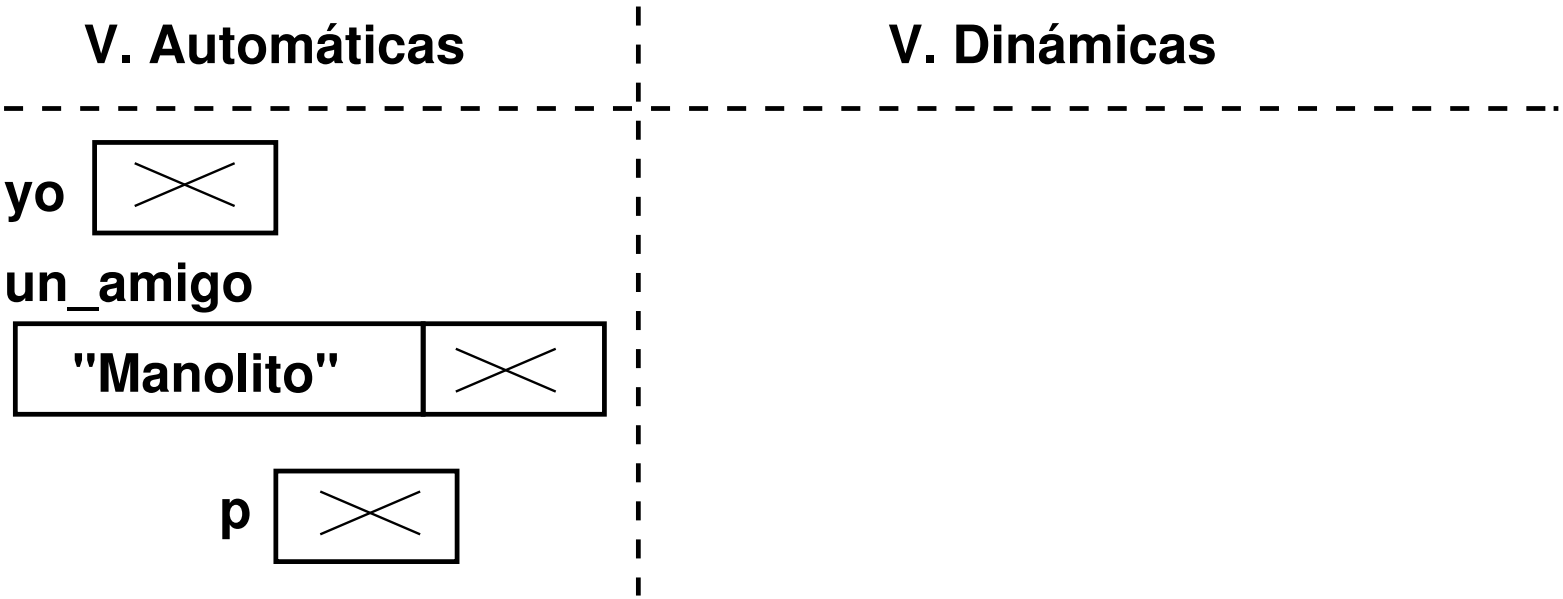
10. `yo->sig = un_amigo.sig = p = nullptr;`



11. `delete yo;`



12. `yo = nullptr;`

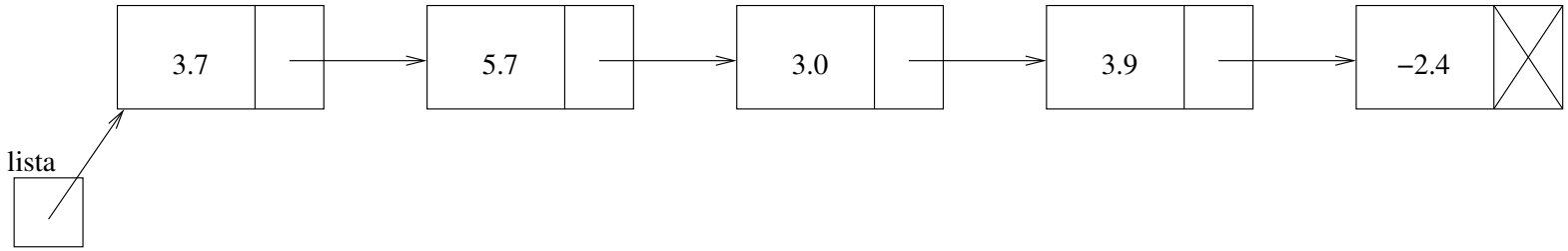


Lista de celdas enlazadas

Lista de celdas enlazadas

Es una **estructura de datos lineal** que nos permite guardar un conjunto de elementos del mismo tipo usando celdas enlazadas.

- Cada celda se alojará en el Heap.
- Usaremos punteros para enlazar una celda con la siguiente.




```
struct Celda{
   double dato;
   Celda* sig;
}
```

Lista de celdas enlazadas

```
#include <iostream>
using namespace std;
struct Celda{
    double dato;
    Celda* sig;
};

int main(){
    Celda* lista;
    double valor;

    lista = nullptr;
    cin >> valor;
    while(valor != 0.0){ // Creación de las celdas de la lista
        Celda* aux = new Celda;
        aux->dato = valor;
        aux->sig = lista;
        lista = aux;
        cin >> valor;
    }
```

Lista de celdas enlazadas

```
    // Mostrar la lista en salida estandar
    aux = lista;
    while(aux != nullptr){
        cout << aux -> dato << " ";
        aux = aux->sig;
    }
    cout << endl;

    while (lista != nullptr) { // Destrucción de la lista
        Celda* aux = lista;
        lista = aux->sig;
        delete aux;
    }
}
```

Lista de celdas enlazadas

Función para insertar al principio de la lista

```
void insertarPrincipioLista(Celda* &lista, double valor){
    Celda* aux = new Celda;
    aux->dato = valor;
    aux->sig = lista;
    lista = aux;
}
```

Lista de celdas enlazadas

Función para insertar al principio de la lista

```
void insertarPrincipioLista(Celda* &lista, double valor){
    Celda* aux = new Celda;
    aux->dato = valor;
    aux->sig = lista;
    lista = aux;
}
```

Función para mostrar el contenido de la lista

```
void mostrarLista(Celda* lista){
    Celda* aux = lista;
    while(aux != nullptr){
        cout << aux -> dato << " ";
        aux = aux->sig;
    }
    cout << endl;
}
```

Lista de celdas enlazadas

Función para destruir la lista

```
void destruirLista(Celda* &lista){
    while (lista != nullptr) {
        Celda* aux = lista;
        lista = aux->sig;
        delete aux;
    }
}
```

Lista de celdas enlazadas

Función para insertar al final de la lista

- Si la lista está vacía, inserto al principio.
- Si la lista no esta vacía
  - Busco puntero p a última celda.
  - Inserto después de posición p.

Lista de celdas enlazadas

Función para insertar después de una celda apuntada por un puntero p

- Hacer que `aux` (puntero auxiliar) apunte a nueva celda.
- Asignar a `aux->dato`, el nuevo dato.
- Asignar a `aux->sig`, el valor de `p->sig`.
- Asignar a `p->sig` el valor de `aux`.

Lista de celdas enlazadas

Función para insertar antes de una celda apuntada por un puntero p

- Si se quiere insertar al principio o la lista está vacía, insertar al principio.
- En caso contrario:
  - Buscar un puntero `aux` que apunte a celda anterior a la apuntada por `p`
  - Hacer que `aux2` (puntero auxiliar) apunte a nueva celda.
  - Asignar a `aux2->dato`, el nuevo dato.
  - Asignar a `aux2->sig`, el valor de `p`.
  - Asignar a `aux->sig`, el valor de `aux2`.

Lista de celdas enlazadas

Función para insertar antes de una celda apuntada por un puntero p

- Si se quiere insertar al principio o la lista está vacía, insertar al principio.
- En caso contrario:
  - Buscar un puntero `aux` que apunte a celda anterior a la apuntada por `p`
  - Hacer que `aux2` (puntero auxiliar) apunte a nueva celda.
  - Asignar a `aux2->dato`, el nuevo dato.
  - Asignar a `aux2->sig`, el valor de `p`.
  - Asignar a `aux->sig`, el valor de `aux2`.

Función para borrar la celda apuntada por un puntero p

Indice

Objetos dinámicos I

Al igual que hicieramos con tipos primitivos y con `struct` vamos a usar el espacio en el heap para almacenar objetos tipo `class` en tiempo de ejecución.

La metodología a seguir es:

1. Reservar memoria. Uso de **new**
2. Utilizar memoria reservada.
3. Liberar memoria reservada. Uso de **delete**

Objetos dinámicos II

Operador `new`

- **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
- Y llama al **constructor** de la clase para inicializar los datos del objeto.

Operador `delete`

- Llama al **destructor** de la clase. (lo veremos en el próximo tema)
- Y después **libera la memoria** de todos y cada uno de los campos del objeto.

Objetos dinámicos compuestos

Ejemplo con class

```
class Estudiante {
    static const TOPE = 15
    string nombre;
    int nAsignaturasMatricula;
    unsigned codigosAsignaturasMatricula[TOPE];
public:
    Estudiante();
    Estudiante(string name);

    void setNombre(string nuevoNombre);
    string getNombre() const;
    void insertaAsignatura(int codigo);
    int getNumeroAsignaturas() const;
    int getCodigoAsignatura(int index) const;
    ...
};
```

Objetos dinámicos compuestos

```
int main() {
    Estudiante *armando;
    armando=new Estudiante("Armando Bronca");
    armando->insertaAsignatura(302);
    armando->insertaAsignatura(307);
    armando->insertaAsignatura(205);
    ...
    delete armando;

    Estudiante ramon("Ramón Rodríguez Ramírez");
    ramon.insertaAsignatura(307);
    ramon.insertaAsignatura(205);
    ...
}
```

Determina en qué zona de memoria está ubicado cada campo del objeto...

Indice

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 (marca 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 se **ajusten** al tamaño necesario. Podemos, además, crearlo 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 = nullptr, n;
6
7      cout << "Numero de casillas: ";
8      cin >> n;
9      // Reserva de memoria
10     v = new int [n];
```

Ejemplo I

```
1     for (int i= 0; i<n; i++) {     //  Lectura del vector dinamico
2         cout << "Valor en casilla "<<i<< ": ";
3         cin >> v[i];
4     }
5     cout << endl;
6
7     for (int i= 0; i<n; i++)  //  Escritura del vector dinamico
8         cout << "En la casilla " << i
9              << " guardo: "<< v[i] << endl;
10
11    delete [] v;  // Liberar memoria
12    v = nullptr;
13 }
```

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=nullptr, 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      for (int i=0; i<m; i++)   // Escribir vector v2
24          cout << "En la casilla " << i
25               << " guardo: "<< v2[i] << endl;
26
27      delete [] v2; // Liberar memoria
28      v2 = nullptr;
29  }
```

¡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;
}
```

¡Error!

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;
  }
}
```

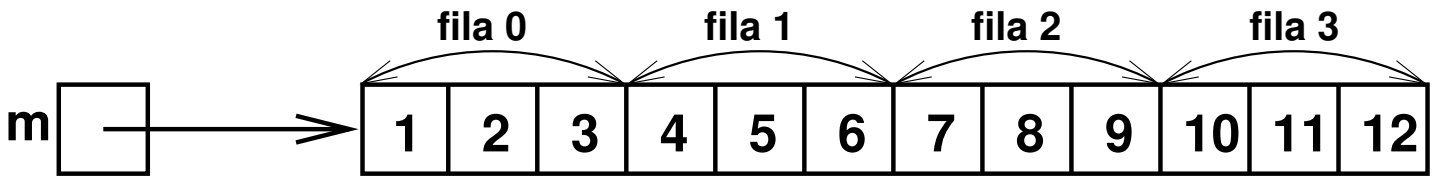
Cuestiones a tener en cuenta:

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

Indice

Matriz 2D usando un array 1D

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12



- **Creación de la matriz**:

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

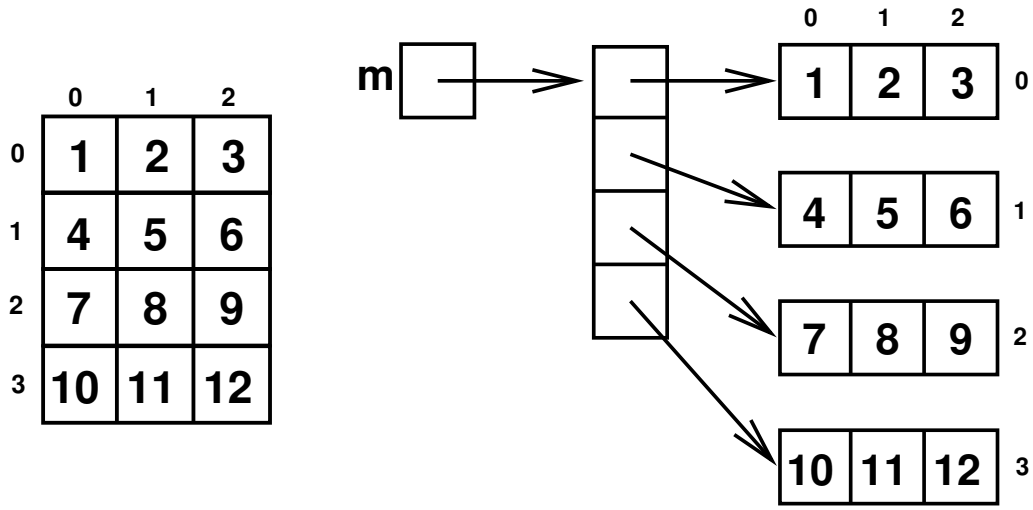
- **Acceso al elemento** `f,c`:

```
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];
```

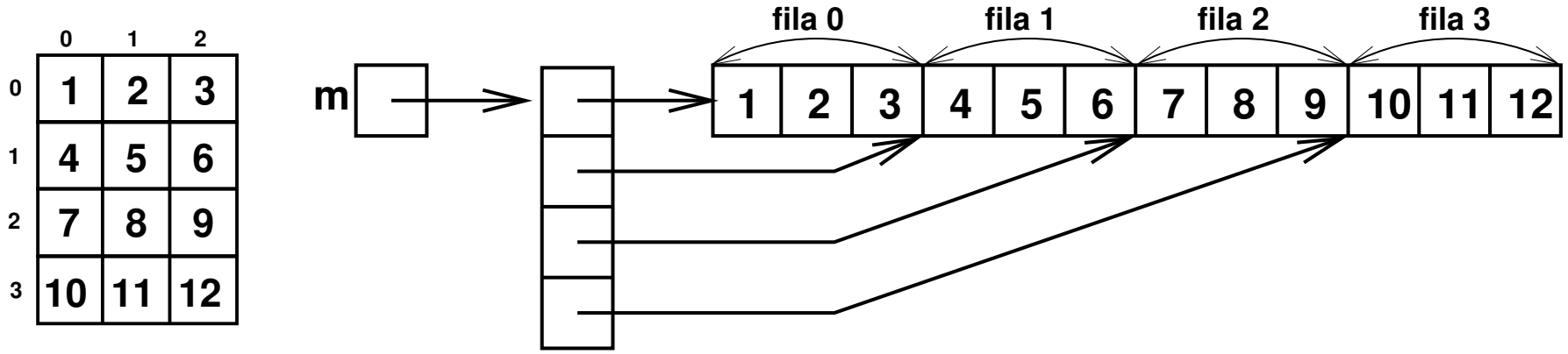
- **Acceso al elemento `f,c`:**

```
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;
```

Indice

Algunos errores comunes

- Uso de punteros no inicializados

```
char y=5, *nptr;
*nptr=5;// ERROR
```

- Acceso fuera del espacio reservado

```
int array[CAPACIDAD], i,j;
for (int i = 0; i < util; i++)
     if (array[i] < array[i+1]
        ...
```

Algunos errores comunes

- Uso de punteros no inicializados

```
char y=5, *nptr;
*nptr=5;// ERROR
```

- Acceso fuera del espacio reservado

```
int array[CAPACIDAD], i,j;
for (int i = 0; i < util; i++)
     if (array[i] < array[i+1]
        ...
```

Y ahora con punteros !!!!

```
int array[CAPACIDAD], i,j;
for (int i = 0; i < util; i++)
     if (*(array+i) < *(array+i+1)
        ...
```

Cuestiones abiertas

- Cuántos operadores de reserva de memoria conoces?
- Qué operadores de liberación de memoria conoces?
- Dado el código:

```
int main()
{
    const int DIM_ARRAY_SALIDA=20;

    double arraySalida[DIM_ARRAY_SALIDA], arrayL1[DIM_ARRAY_SALIDA/2], arrayL
    int utilArraySalida;
    int utilArrayL1, utilArrayL2;

    leer(arrayL1,utilArrayL1);
    leer(arrayL2,utilArrayL2);
    concatenar(arrayL1,utilArrayL1,arrayL2,utilArrayL2, arraySalida, utilArra
    imprimirArray(arraySalida, utilArraySalida);

}
```

Desarrollar un nuevo main() para trabajar con vectores dinámicos.


creative
commons


BY NC SA