Metodología de la Programación I

Punteros
y
Gestión Dinámica de Memoria

Objetivos

- Conocer y manejar el tipo de dato puntero.
- Conocer la estructura y la gestión de la memoria de un ordenador desde el punto de vista de un programador de C++.
- Usar apropiadamente los operadores de reserva y liberación de memoria dinámica.
- Iniciar a los alumnos en el desarrollo de tipos de datos que requieren manejo de memoria dinámica.

2.1 El tipo de dato puntero

- Un puntero es un tipo de dato que contiene la dirección de memoria de un dato, incluyendo una dirección *especial* llamada *dirección nula*, representada en C++ por el valor `0`. En C la dirección nula se suele representar por la constante `NULL` (definida en `<stdlib.h>` y en `<stddef.h>` entre otros).

- Declaración de datos de tipo puntero.

  Un dato de tipo puntero se debe declarar como cualquier otro dato.

  El formato es el siguiente:

  `<tipo> *<nombre>`

  donde:

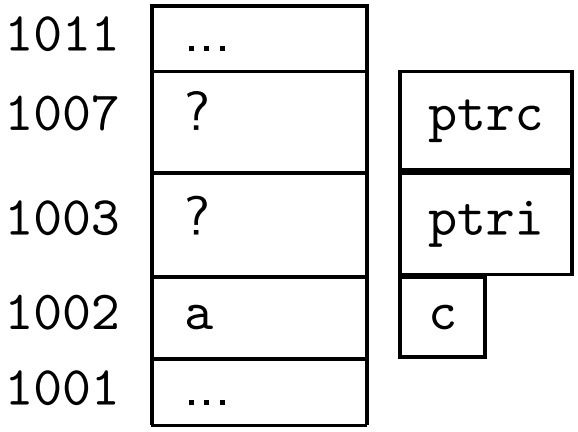
  - `<nombre>` es el nombre de la variable puntero.
  - `<tipo>` es el tipo del objeto cuya dirección de memoria contiene `<nombre>`.

Ejemplo: Declaración de punteros

Cdigo

```
char c = 'a';
char *ptrc;
int  *ptri;
```

Representación gráfica de la memoria



declara

- `c` como un variable de tipo caracter cuyo valor es `'a'`,
- `ptri` como una variable de tipo puntero que puede contener direcciones de memoria de objetos de tipo `int`
- `ptrc` como una variable puntero que puede contener direcciones de memoria de objetos de tipo `char`.

Se dice que

- `ptri` es un *puntero a enteros*
- `ptrc` es un *puntero a caracteres*.

Nota 1: Cuando se declara un puntero se reserva memoria para albergar la dirección de memoria de un dato, no el dato en sí.

Nota 2: El tamaño de memoria reservado para albergar un puntero es el mismo (usualmente 32 bits) independientemente del tipo de dato al que 'apunte'.

2.1.1 Operaciones básicas con punteros

Los operadores básicos para trabajar con punteros son dos:

- Operador de dirección `&`

  `&<variable>` devuelve la dirección de memoria donde **empieza** la variable `<variable>`.

  El operador `&` se utiliza habitualmente para asignar valores a datos de tipo puntero.

```
int i, *ptri;
ptri = &i;
```

  `i` es una variable de tipo entero, por lo que la expresión `&i` es la dirección de memoria donde comienza un entero y, por tanto, puede ser asignada al puntero `ptri`.

  Se dice que `ptri` *apunta* o *referencia* a `i`.

- Operador de indirección *

  *<puntero> devuelve el contenido del objeto referenciado por <puntero>.

  Esta operación se usa para acceder al objeto *referenciado o apuntado* por el puntero.

```
char c, *ptrc;

ptrc = &c;
*ptrc = 'A'; // equivalente a c = 'A'
```

  ptrc es un puntero a caracter que contiene la dirección de c, por tanto, la expresión *ptrc es el objeto apuntado por el puntero, es decir, c.

- Un puntero contiene una dirección de memoria y se puede interpretar como un número entero aunque un puntero no es un número entero. No obstante existen un conjunto de operadores que se pueden realizar sobre un puntero: +,-,++,--, !=, ==

Ejemplo: Ejemplo de uso de punteros

```
int main() {
  char y = 5, z = 3;
  char *nptr;
  char *mptr;

  nptr = &y;

  z = *nptr;

  *nptr = 7;

  mptr = nptr;

  mptr = &z;

  *mptr = *nptr;

  y = (*mptr) + 1;
}
```

- Existe una gran relación entre punteros y vectores.

  Al declarar un vector

  `<tipo> <identif>[<n_elem>]`

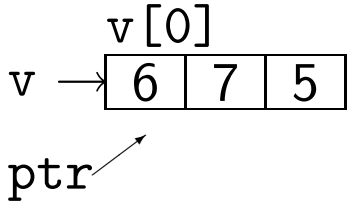
  realmente:

1. Se reserva memoria para almacenar `<n_elem>` elementos de tipo `<tipo>`.
2. Se crea un puntero CONSTANTE llamado `<identif>` que apunta a la primera posición de la memoria reservada.

   Por tanto, el *identificador* de un vector, es un *puntero* CONSTANTE a la dirección de memoria que contiene el primer elemento. Es decir, `v` es igual a `&v[0]`.

Podemos, pues, hacer:

```
int v[3];
int *ptr;
...
ptr = &v[0];     // ptr = v
```



`v[0]=6` es equivalente a `*v=6` y a `*(&v[0])=6`

- Por la misma razón, a los punteros se les pueden poner subíndices y utilizarlos como si fuesen vectores.

  `v[i]` es equivalente a `ptr[i]`

2.1.2 Errores más comunes

- Asignar punteros de distinto tipo:

```
int a = 10, *ptri;
float b = 5.0, *ptrf;

ptri = &a;
ptrf = &b;
ptrf = ptri; // ERROR
```

- Uso de punteros no inicializados:

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

- Asignación de valores al puntero y no a la variable a la que apunta:

```
char y = 5, *nptr = &y;
nptr = 9; // ERROR
```

2.1.3 Punteros a punteros

Un puntero a puntero es un puntero que contiene la dirección de memoria de otro puntero.

Ejemplo: Punteros a punteros

```
int a = 5;
int *p;
int **q;

p = &a;

q = &p;
```

En este caso, para acceder al valor de la variable a tenemos tres opciones:

`a` (la forma usual)

`*p` (a través del puntero `p`)

`**q` (a través del puntero a puntero `q`) equivalente a `*(*q)`

2.2 Punteros y cadenas-C

En C estándar, una **cadena de caracteres** se define como un vector de tipo carácter. Para controlar el final de la cadena, se utiliza el carácter *especial* (`'\0'`) denominado caracter fin de cadena o carácter nulo.

En C++, se puede usar el tipo *string* para manejar cadenas. Para distinguirlos, a las primeras las denominamos *cadenas-C* o *cstring*

Una cadena es un vector de tipo *char*. Por ejemplo,

```
char cad1[10];
```

donde

- *cad1* es un vector de caracteres que puede almacenar una cadena de hasta 9 caracteres (el último debe ser el terminador de cadena).

Así una nueva forma de iniciar los valores de un vector de *char* es

```
char cad2[]="Pepe", cad3[20]="lluvia";
```

- *cad2* es un vector de caracteres de tamaño 5 que almacena la cadena *Pepe* y
- *cad3* es un vector de tamaño 20, que tiene la cadena *lluvia*, el carácter terminador de la cadena y 12 posiciones no utilizadas.
- Las cadenas *"Pepe"*, *"lluvia"* son de tipos *const char[5]* y *const char[7]*, respectivamente.

A causa de la fuerte relación entre vectores y punteros, podemos manejar las cadenas con el tipo *"char *"*. De hecho, en la prática se suele utilizar más esta notación.

2.2.1 Escritura y lectura de cadenas-C

Para escribir y leer una cadena de caracteres, podemos usar *cin* y *cout* como con cualquier otro tipo de dato básico. Por ejemplo

```
char cs1[80];

cout << "Introduce un nombre: ";
cin >> cs1;
cout << cs1;
```

La sentencia de lectura lee caracteres hasta encontrar un separador (espacio, tabulador, salto de línea).

Para leer una línea completa (todos los caracteres hasta el siguiente salto de línea), se puede usar

```
char cs1[80];

do{
  cin.getline(cs1,80);
}while (*csi!='\0');
```

donde los parámetros corresponden a un vector de caracteres y el tamaño mínimo reservado, respectivamente. Se introduce dentro de un `do-while` para evitar introducir líneas vacías.

2.2.2 Funciones para cadenas-C

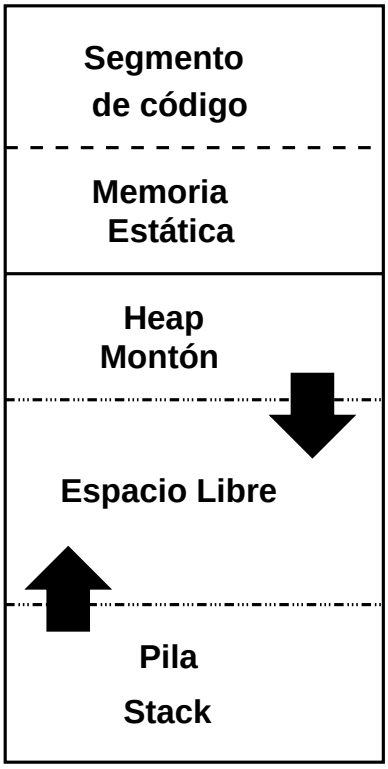
En lenguaje C se dispone de un conjunto de cadenas para facilitar el uso de las cadenas-C. En C++, se debe incluir <cstring>. Algunas son

- Longitud de una cadena.
  ```
  int strlen (const char *s)
  ```
  Aunque indicamos *int*, en realidad es *size_t*.
- Asignar una cadena.
  ```
  char *strcpy (char *dst, const char *src)
  ```
  concatena *src* en *dst* y devuelve *dst*.
- comparar dos cadenas.
  ```
  int strcmp (const char *c1, const char *c2
  ```
  que devuelve positivo si la primera mayor, negativo si la segunda mayor y cero si iguales.

2.3 Estructura de la memoria

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`.



2.3.1 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.
- El entorno de programa almacena, entre otros:
  - La siguiente instrucción que se ejecutará cuando se resuelva la llamada a la función (dirección de retorno).
  - Los argumentos y variables locales definidos en la función.
- 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**.

2.3.2 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 1: Supongamos que se desea realizar un programa que permita trabajar con una lista de datos relativos a una persona.

```
struct Persona{
   string nombre;
   int DNI;
   image foto;
};
```

A priori, es necesario definir un tamaño apropiado para almacenar un vector de `Persona`. Supongamos que realizamos la siguiente definición:

```
Persona VectorPersona[100];
```

¿Qué inconvenientes tiene esta definición?

En cada ejecución se puede requerir un número diferente de posiciones del vector.

- 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, ya que se usan posiciones de memorias que pueden estar usando otras variables del programa.
  "Solución": Ampliar la dimensión del vector y volver a compilar.

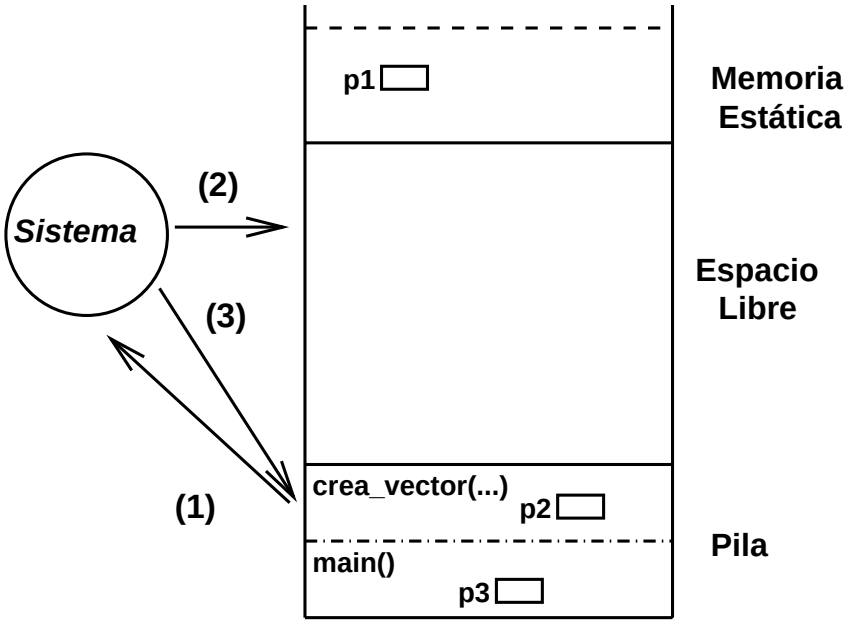
La utilización de variables estáticas o automáticas para almacenar información cuyo tamaño no es conocido a priori (solo se conoce exactamente en tiempo de ejecución) resta generalidad al programa.

La alternativa válida para solucionar estos problemas consiste en la posibilidad de, **en tiempo de ejecución**: pedir la memoria necesaria para almacenar la información y de liberarla cuando ya no sea necesaria.

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.

2.4 Gestión dinámica de la memoria

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

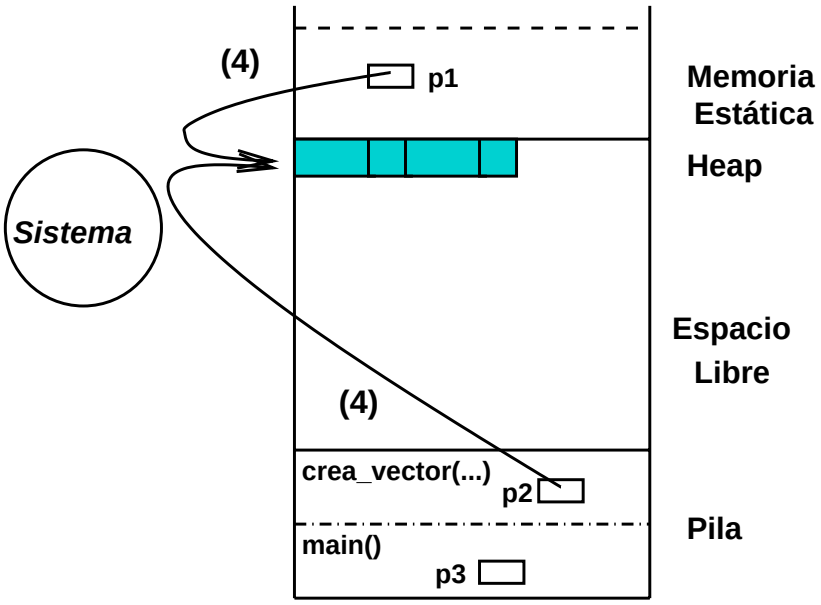


(1) Petición al S.O. (tamaño)

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

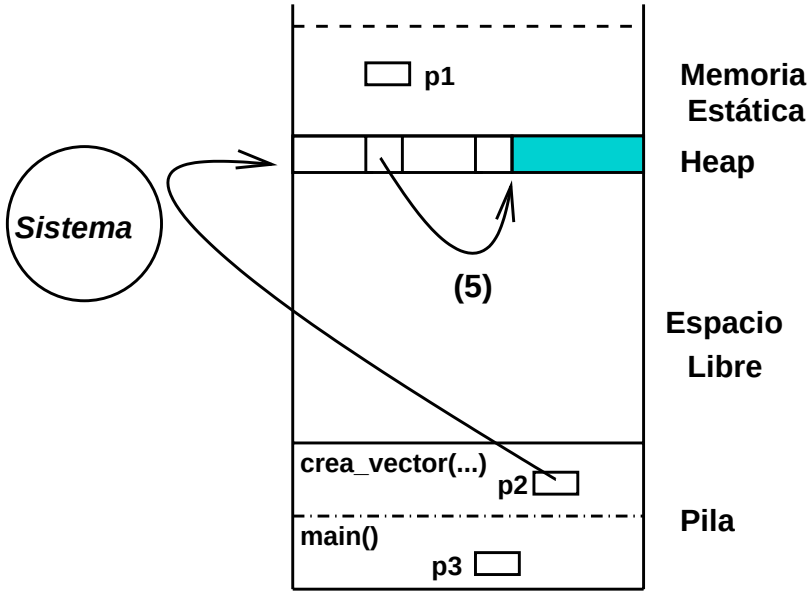
(3) Si hay espacio suficiente, devuelve la ubicación donde se encuentra la memoria reservada, y marca dicha zona como memoria ocupada.

2.4.1 Reserva de memoria



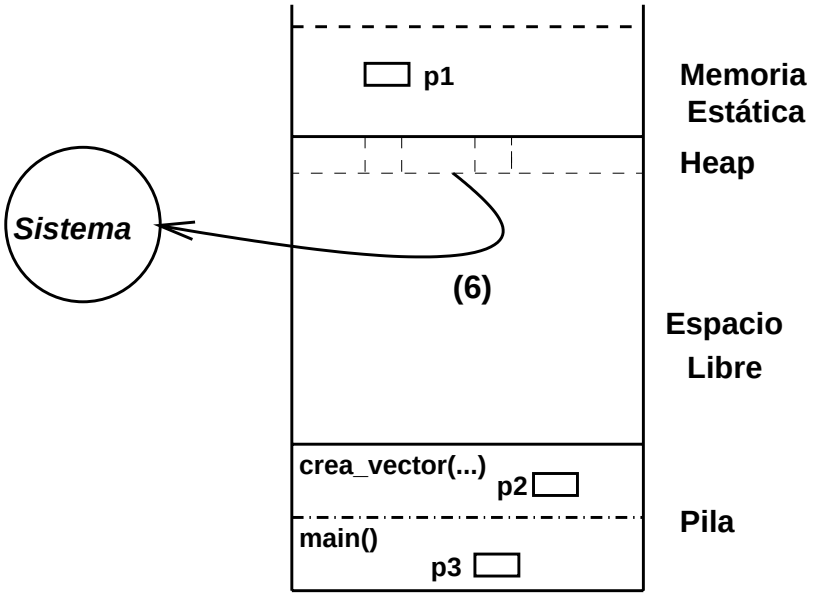
(4) La ubicación de la zona de memoria se almacena en una variable estática (**p1**) o en una variable automática (**p2**).

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.

2.4.2 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.

2.4.3 Objetos dinámicos simples

El operador `new`

- `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 donde 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. El manejo de excepciones se verá en otras asignaturas.

Ejemplo:

```
(1)   int *p,
          q = 10;

(2)   p = new int;
(3)   *p = q;
```

Notas:

1. Observar que `p` se declara como un puntero más.
2. 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.
3. 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:

```
(1)   int *p,
          q = 10;

(2)   p = new int;
(3)   *p = q;
      .......
(4)   delete p;
```

Notas:

4. El objeto referenciado por `p` deja de ser "operativo" y la memoria que ocupaba está disponible para nuevas peticiones con `new`.

2.4.4 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.

```
struct Persona{
   string nombre;
   string DNI;
};
Persona *yo;

yo = new Persona;

cin >> (*yo).nombre;
cin >> (*yo).DNI;
......
delete yo;
```

El operador -> de registros

En el tema anterior se vió que existían dos operadores para el acceso a los campos de un registro: el operador punto (.) y el operador flecha (->).

El operador flecha se utiliza para el acceso a los campos de un registro que se encuentra apuntado por un puntero.

Veamos el siguiente ejemplo:

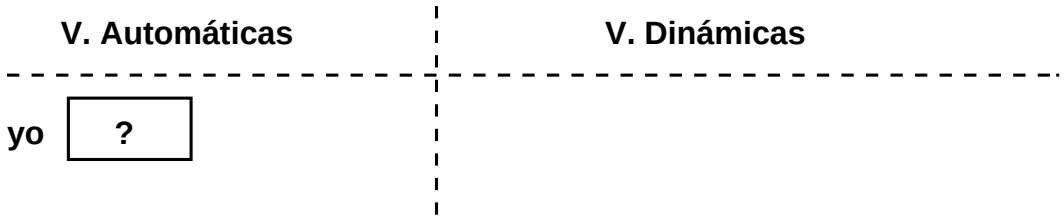
```
struct Persona{
 string nombre;
 string DNI;
};
Persona yo,    // variable a tipo Persona.
        *clon; // puntero a tipo Persona.
clon = &yo;

yo->nombre="Pepe"       // Error
clon->nombre = "Pepe"   // Correcto
(*clon).nombre="Pepe"   // Equivalente
```

Para usar correctamente el operador flecha, la variable que se encuentra a la izquierda de la -> debe ser de tipo puntero a la estructura.

Un ejemplo más completo

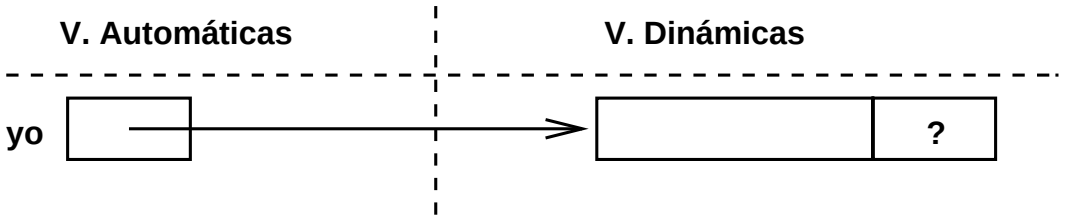
Dada la definición del siguiente tipo de dato `Persona` y declaración de variable

```
struct Persona {
    string nombre;
    Persona *amigos;
};

Persona *yo;
```



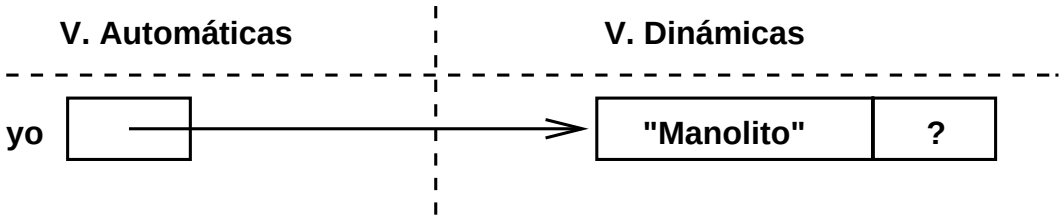
¿Qué realiza la siguiente secuencia de instrucciones?

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 `string` y un *puntero*.

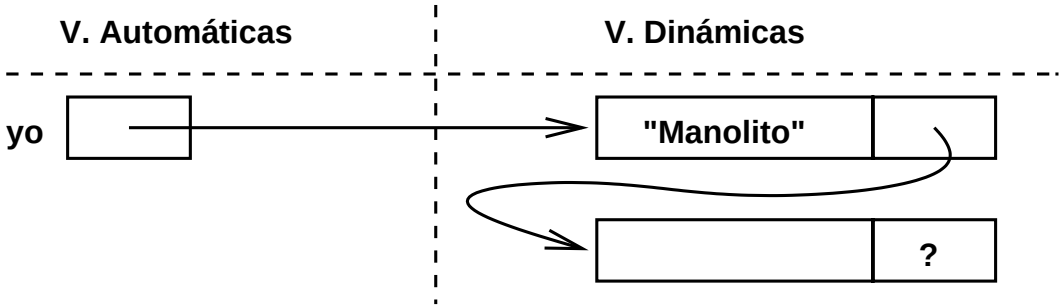
2. `yo->nombre = "Manolito";`



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.

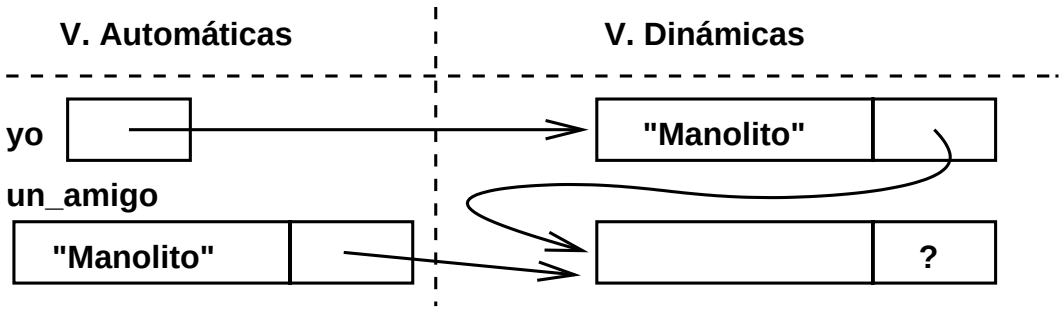
3. `yo->amigos = new Persona;`

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).

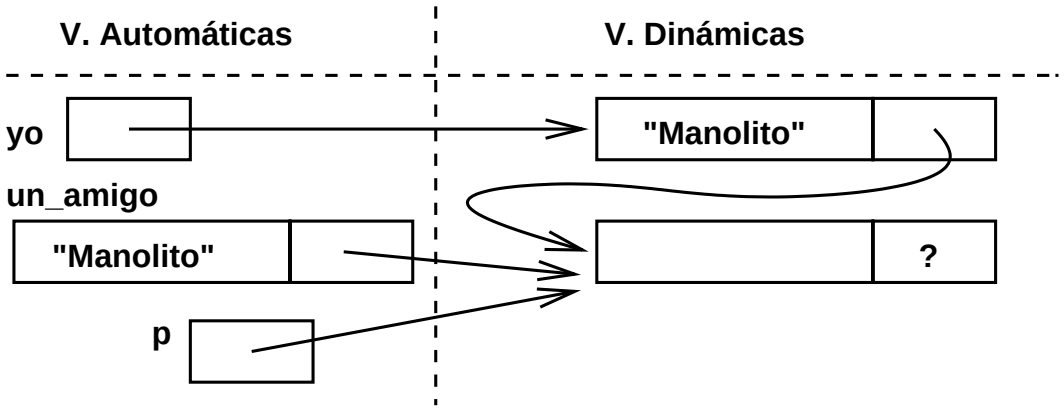


Por tanto, a partir de una variable dinámica se pueden definir nuevas variables dinámicas siguiendo una filosofía semejante a la propuesta en el ejemplo.

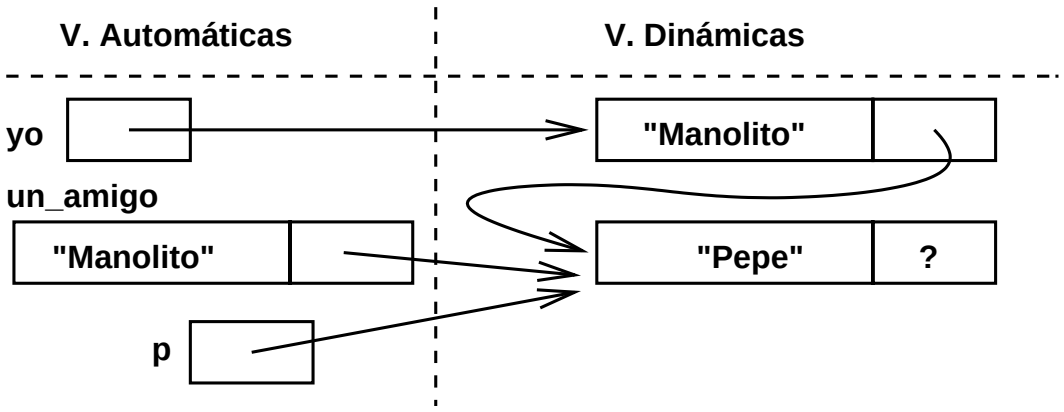
4. `Persona un_amigo = *yo;`
   Se crea la variable automática `un_amigo` y se realiza una copia de la variable que es apuntada por `yo`.



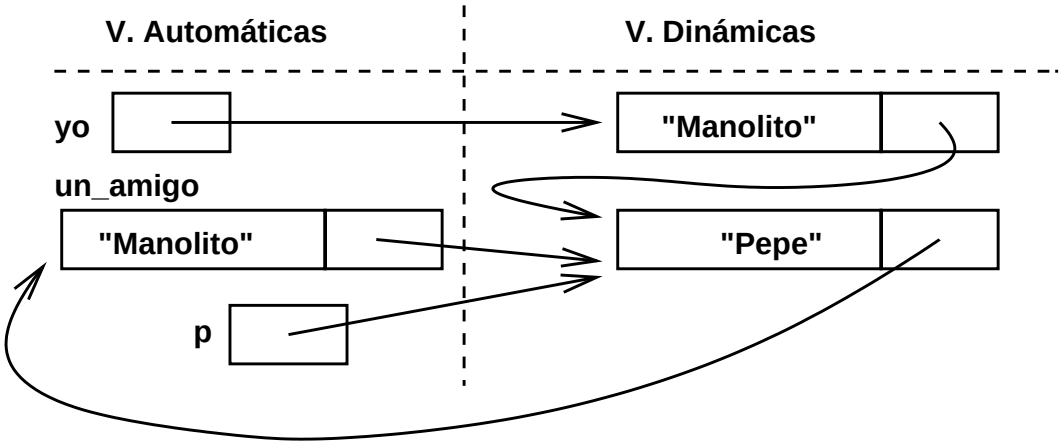
5. `Persona *p = yo->amigos;`
   La variable `p` almacena la misma dirección de memoria que el campo `amigos` de la variable apuntada por `yo`.



6. `p->nombre = "Pepe";`
   Usando la variable `p` (apunta al último dato creado) damos valor al campo `nombre`.

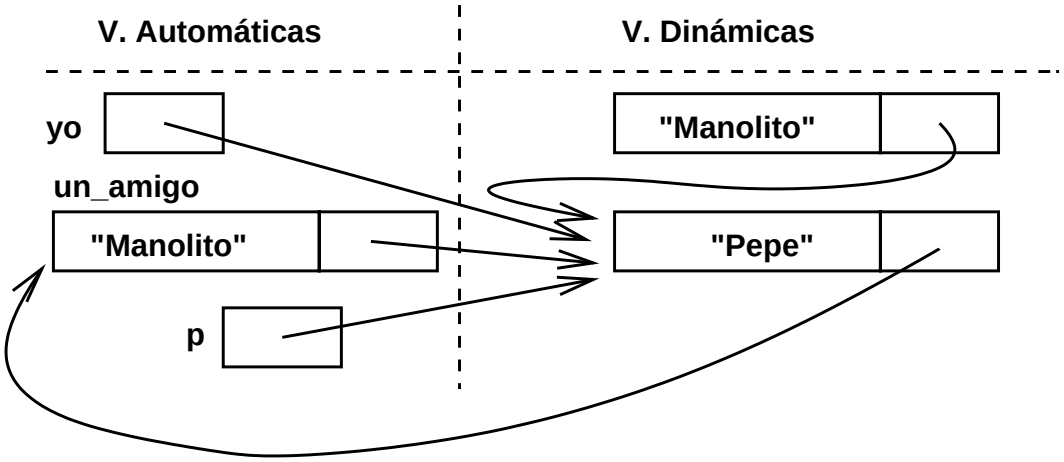


7. `p->amigos = &un_amigo;`



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

8. `yo = p;`

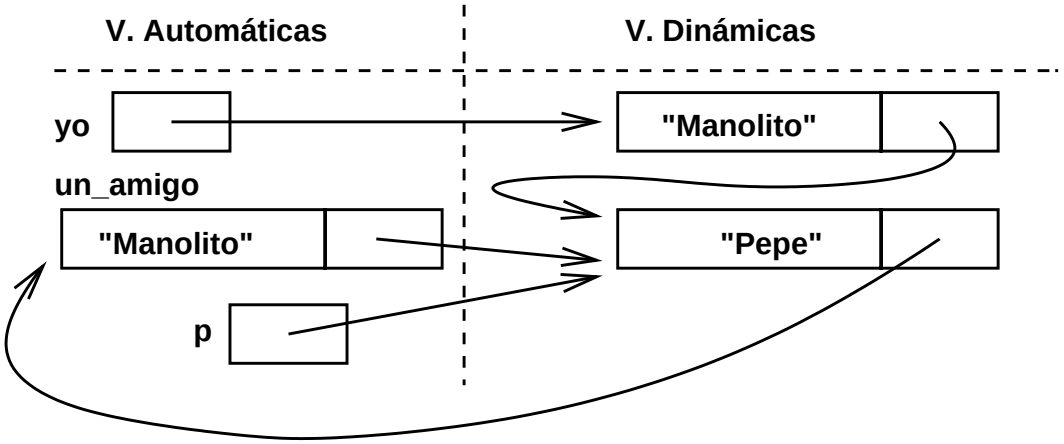


Con este orden se pierde el acceso a uno de los objetos dinámicos creados, siendo imposible su recuperación. Por tanto, antes de realizar una operación de este tipo, hay que asegurar:

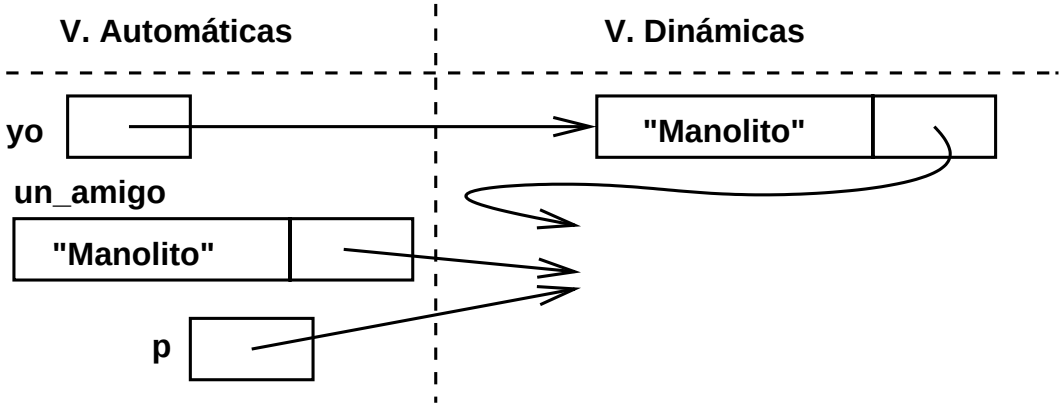
a) que no perdemos la referencia a ese objeto (existe otro puntero que lo referencia). Esto se debe considerar cuando el objeto dinámico sigue siendo útil para el programa.

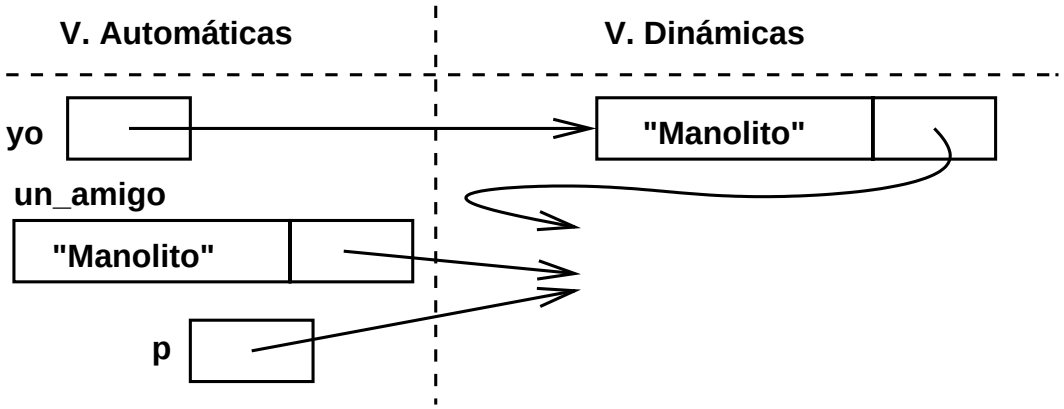
b) Si la variable ya no es útil para el programa, debemos liberar antes la memoria (indicando al sistema que esa zona puede ser utilizada para almacenar otros datos).

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:

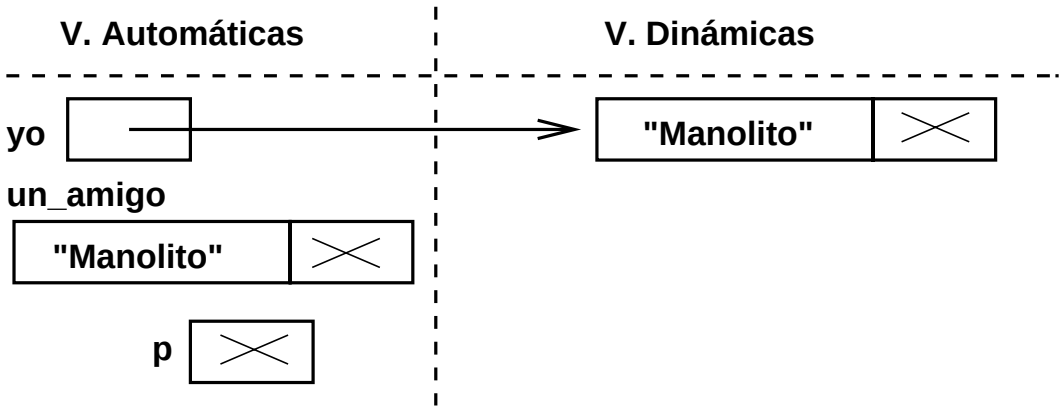
```
un_amigo.amigos->nombre = "Alex";
```

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

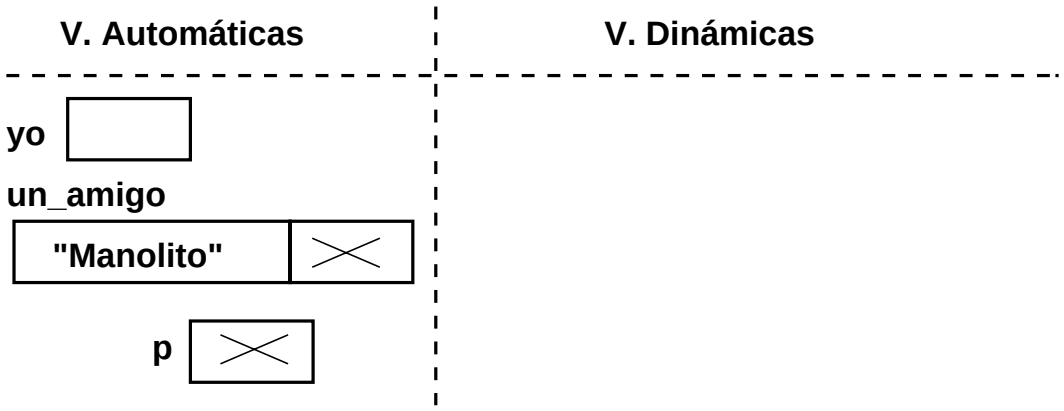
```
yo->amigos->nombre = "Alex";
```

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;`



2.4.5 Vectores dinámicos

- Hasta ahora sólo podemos crear un vector conociendo *a priori* el número mínimo de elementos que podrá tener. P.e. `int vector[22];`
- 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.

- La liberación se realiza con `delete []`
  `delete []` *puntero*`;`
  que libera (pone como disponible) la zona de memoria **previamente reservada** por una orden `new []` y que está referencia por un punetro.
- Con la utilización de de esta forma de reserva dinámica podemos crear vectores 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:

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

int main()
{
   int *v, n;

   cout << "Numero de casillas: ";
   cin >> n;
   // Reserva de memoria
   v = new  int [n];
```

```
    // Procesamiento del vector dinamico:
    //    lectura y escritura de su contenido

    for (int i= 0; i<n; i++) {
       cout << "Valor en casilla "<<i<< ": ";
       cin >> v[i];
    }
    cout << endl;

    for (int i= 0; i<n; i++)
       cout << "En la casilla " << i
            << " guardo: "<< v[i] << endl;

    // Liberar memoria
    delete [] v;

}
```

Una función puede devolver un vector dinámico como copia de otro que recibe como argumento.

```
int * copia_vector1 (int v[], int n)
{
   int * copia = new  int [n];

   for (int i= 0; i<n; i++)
     copia[i] = v[i];
   }
   return (copia);
}
```

El vector devuelto tendrá que ser, cuando deje de ser necesario, liberado con `delete []`:

```
   int * v1_copia;
   ......
   v1_copia = copia_vector1 (v1, 500);
   // Usamos v1_copia
   ...
   delete [] v1_copia;
```

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

```
int *copia_vector1(const int v[], int n)
{
   int copia[100]; // o cualquier otro valor
                   // mayor que n

   for (int i= 0; i<n; i++)
      copia[i] = v[i];

  return (copia);
}
```

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 vector dinámico.

```
void AmpliarVector (int * &v, int nelems,
                    int nadicional)
{
   int v_ampliado = new int[nelems + nadicional];

   for (int i= 0; i<nelems; 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 el lugar al que apunta.
- Es necesario liberar `v` antes de asignarle el mismo valor de `v_copia` ya que si no perderíamos cualquier referencia a ese espacio de memoria.
- Poco eficiente. Probar con `memcpy()`.

Ejemplo: Registros y cadenas de caracteres

```
struct Persona{
 char *nombre;
 char NIF[10]; //8digit+letra+'\0'
};

Persona yo;
char aux[120];

// Leer datos
cout << "Introduzca nombre: ";
do {
 cin.getline(aux,120);
}while (strlen(aux) == 0);
yo.nombre = new char[strlen(aux)];
strcpy(yo.nombre,aux);

cout << "Introduzca NIF: ";
do{
 cin.getline(yo.NIF);
}while (strlen(yo.NIF) == 0);
```

2.4.6 Matrices dinámicas

Problema:

Gestionar matrices 2D de forma dinámica, en tiempo de ejecución.

Motivación:

El lenguaje proporciona matrices estáticas para las que debe conocerse el tamaño en tiempo de compilación. En el caso de matrices 2D, el compilador debe conocer el número de filas y columnas.

Necesitamos poder crear y trabajar con matrices 2D cuyo tamaño (filas y columnas) sea exactamente el que requiera el problema a resolver.

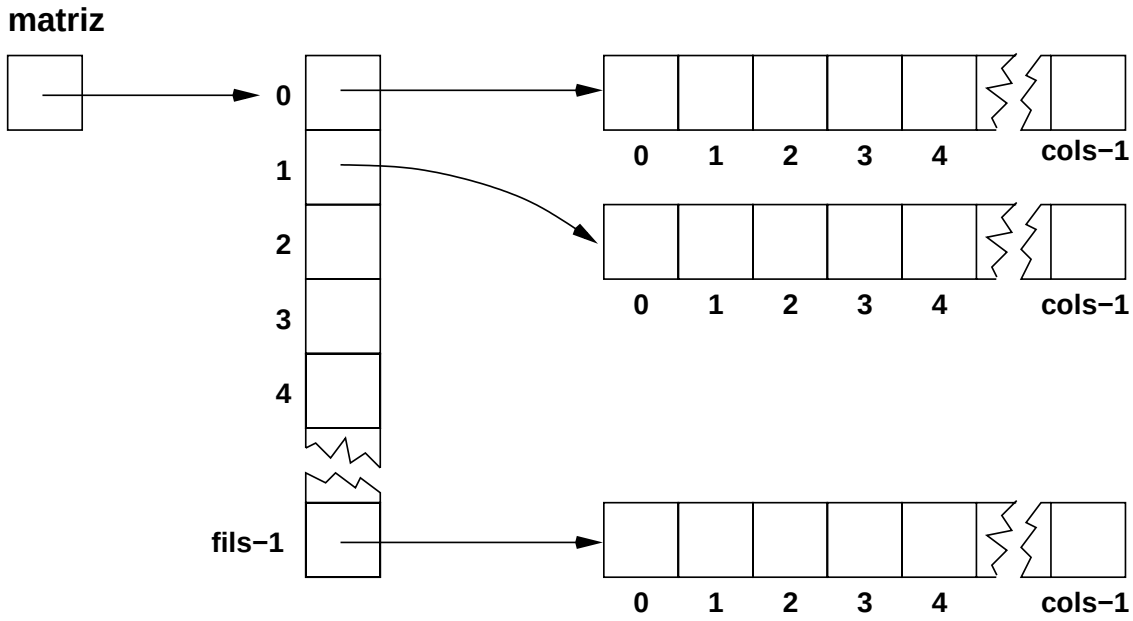
En particular, posibilitaremos:

- Creación de matrices dinámicas.
- Destrucción de matrices dinámicas.
- Acceso mediante índices.

Aproximaciones:

1. Datos guardados en filas independientes.
2. Datos guardados en una única fila.

Solución 1




```
int ** Matriz2D_1 (int fils, int cols);

void LiberaMatriz2D_1 (int **matriz,
                       int fils, int cols);
```

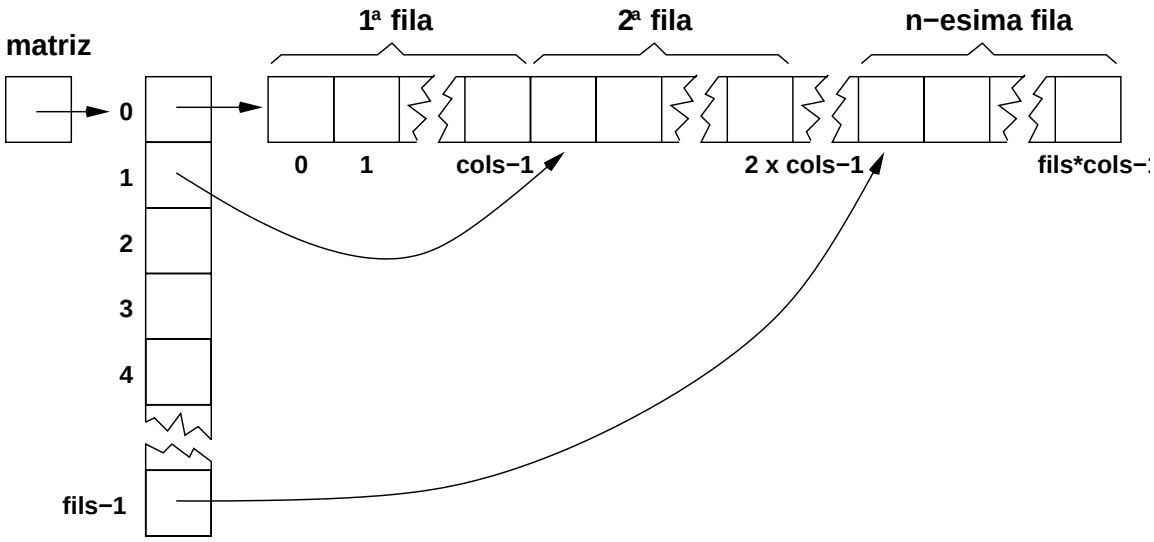
Creación (1)

```
int ** Matriz2D_1 (int fils, int cols)
{
   bool error = false;
   int ** matriz;
   int f, i;

   // "matriz" apunta a un  vect. de punt. a filas

   matriz = new  int * [fils];

   for (f=0; f<fils; f++)
   // "matriz[f]" apuntara a un vector de int*
    matriz[f] = new  int [cols];

  return (matriz);
}
```

Liberación (1)

```
void LiberaMatriz2D_1 (int **matriz,int fils,int cols)
{
   for (int f=0; f<fils; f++)
      delete [] matriz[f];

   delete [] matriz;
}
```

Solución 2



Las funciones específicas para esta representación tendrán el mismo interfaz que las de la solución anterior:

```
int ** Matriz2D_2 (int fils, int cols);

void LiberaMatriz2D_2 (int **matriz,
                       int fils, int cols);
```

Creación (2)

```
int ** Matriz2D_2 (int fils, int cols)
{
   int ** matriz;
   int f;
   // "matriz" apunta a un  vect. de punt.
   // que apuntaran al inicio de cada fila

   matriz = new  int * [fils];

   matriz[0] = new  int [fils*cols];
   for(f=1; f<fils ; f++)
     matriz[f] = matriz[f-1] + cols;
   }
   return (matriz);
}
```

Liberación (2)

```
void LiberaMatriz2D_2 (int **matriz,
                       int fils, int cols)
{
   delete [] matriz[0];
   delete [] matriz;
}
```

Ejemplo de utilización

```
#include <iostream>
#include <iomanip>

using namespace std;

void   LeeDimensiones (int &num_filas, int &num_cols);
int ** Matriz2D_1 (int fils, int cols);
int ** Matriz2D_2 (int fils, int cols);
void   PintaMatriz2D (int **matriz, int fils, int cols)
void   LiberaMatriz2D_1 (int **matriz,
                         int fils, int cols);
void   LiberaMatriz2D_2 (int **matriz, int fils,
                         int cols);
/*********************************************/
```

```
/*************************************************/

int main ()
{
   int ** m1; // "m1" y "m2" seran matrices
   int ** m2; // dinamicas 2D.

   int filas, cols;
   int f, c;

   // Leer num. de filas y columnas

   LeeDimensiones (filas, cols);

   // Crear matrices dinamicas

   cout << endl << "Creando Matriz 1 (";
   cout << filas << "X"<< cols << ")" << endl;

   m1 = Matriz2D_1 (filas, cols);

   cout << "Creando Matriz 2 (";
   cout << filas << "X"<< cols << ")" << endl;

   m2 = Matriz2D_2 (filas, cols);

   // Rellenarlas (observar el acceso por indices)

   cout << endl << "Rellenando matrices" << endl;
```

```
   for (f=0; f<filas; f++)
      for (c=0; c<cols; c++) {
         m1[f][c] = ((f+1)*10)+c+1;
         m2[f][c] = ((f+1)*10)+c+1;
      }

   // Mostrar su contenido

   cout << endl << "Matriz 1:" << endl;
   PintaMatriz2D (m1, filas, cols);

   cout << endl << "Matriz 2:" << endl;
   PintaMatriz2D (m2, filas, cols);

   // Liberar la memoria ocupada

   cout << endl << "Liberando matriz 1" << endl;
   LiberaMatriz2D_1 (m1, filas, cols);

   cout << "Liberando matriz 2" << endl << endl;
   LiberaMatriz2D_2 (m2, filas, cols);

   return (0);
}
/*************************************************/
```

```
void  LeeDimensiones (int &num_filas, int &num_cols)
{
   cout << "Numero de filas : ";
   cin >> num_filas;
   cout << "Numero de columnas : ";
   cin >> num_cols;
}
/*****************************************************/

void PintaMatriz2D (int **matriz, fils, int cols)
{
   for (int f=0; f<fils; f++) {
      for (int c=0; c<cols; c++)
         cout << setw(4) << matriz[f][c]; // Indices
      cout << endl;
   }
}
/*****************************************************/
// Resto de funciones: Matriz2D_1(), Matriz2D_2(),
// LiberaMatriz2D_1 y LiberaMatriz2D_2()
```