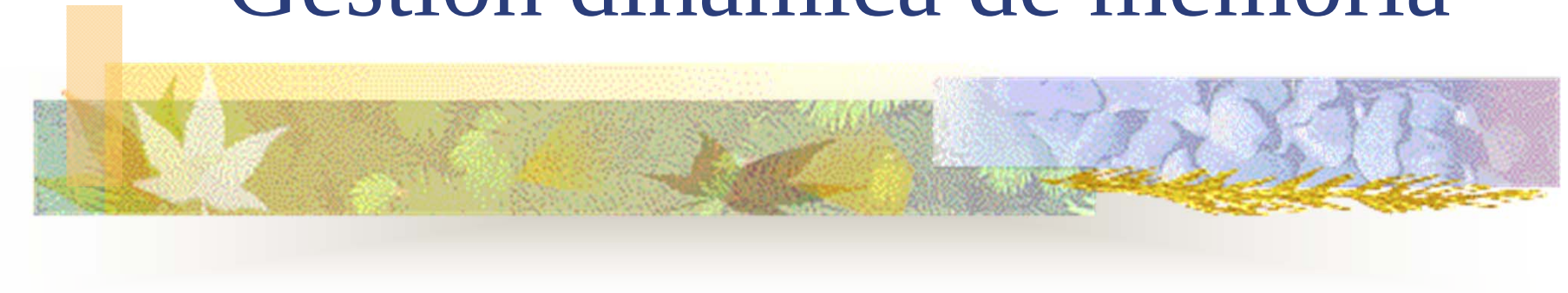
Gestión dinámica de memoria

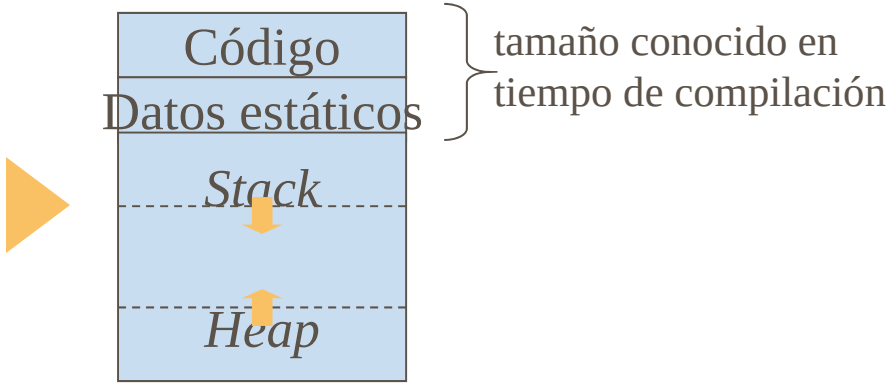


Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

Introducción

Organización de la memoria en tiempo de ejecución



- Datos
  - Datos estáticos (variables globales y estáticas)
  - Datos Locales (*stack*)
  - Objetos dinámicos (*heap*)

Objetos dinámicos

- Se crean y destruyen a voluntad, según la necesidad, y en tiempo de ejecución
- El número de objetos dinámicos es desconocido a priori y puede variar durante la ejecución del programa
- Se alojan en el *heap*

Gestión dinámica de memoria en C

- Reserva de memoria:
  - ***calloc***
  - ***malloc***
  - ***realloc***
- Liberación de memoria:
  - ***free***
- Todas estas funciones están en **stdlib.h**

Gestión dinámica de memoria en C

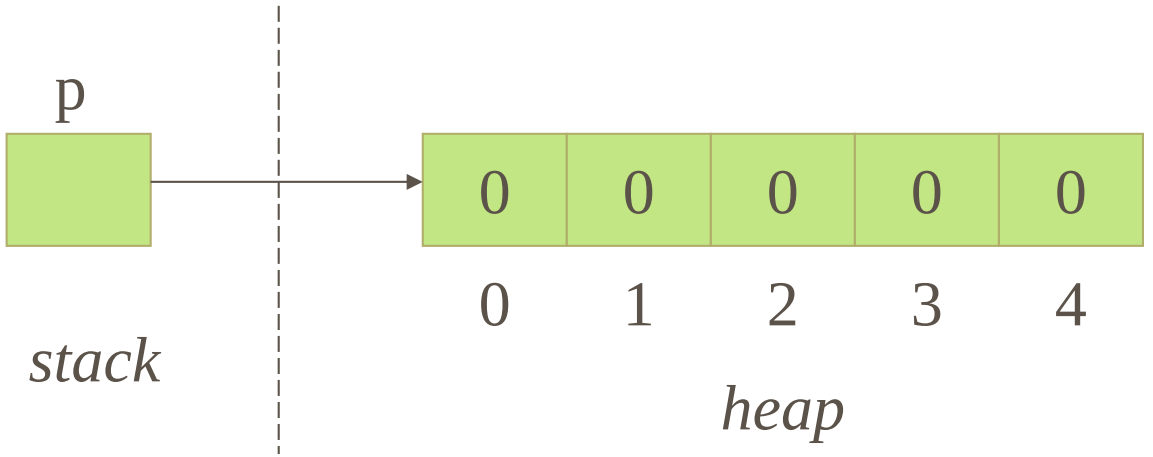
void *calloc(size_t nelem, size_t size)

- Devuelve un puntero a un vector de *nelem* de tamañao *size* cada uno
  - La reserva de memoria se hace en el *heap*
  - **Inicializa con ceros** todos los elementos de la zona de memoria reservada
  - Esta zona queda reservada, no pudiéndose asignar en otra reserva posterior, salvo que se libere previamente
  - Se puede hacer un *casting* seguro del valor devuelto a un puntero de cualquier tipo de dato de tamaño menor o igual a *size*
- En caso de no poder satisfacerse la petición, devuelve NULL

Procedimiento habitual de reserva

```
int* p;
if((p=(int*)calloc(n,sizeof(int)))==NULL)
{
  printf("Error: no pudo asignarse memoria");
  exit(-1);
}
```

No olvidar el casting, porque la función devuelve un void *



Comprobar si el resultado es NULL ¡¡Hacerlo siempre así!!

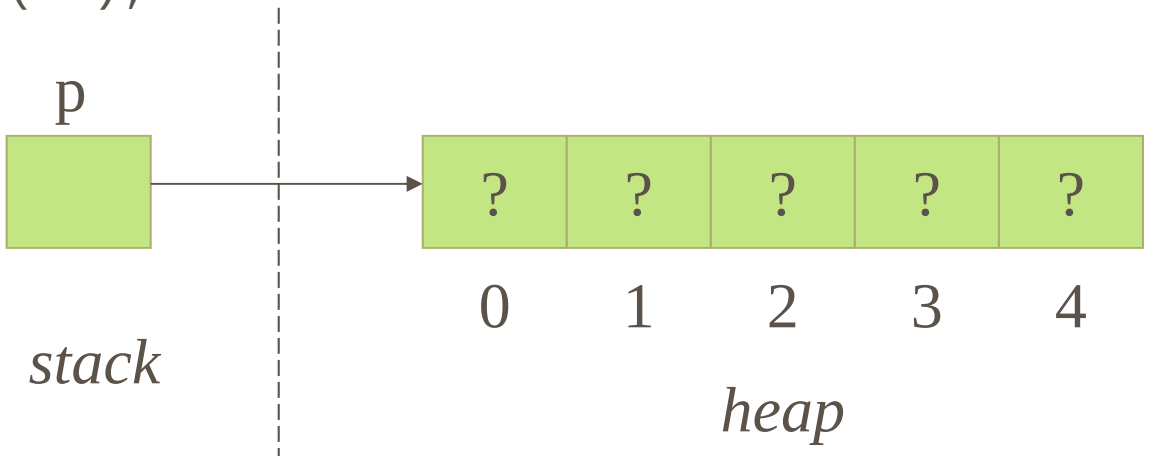
Gestión dinámica de memoria en C

void *malloc(size_t size)

- Devuelve un puntero a un vector de *nelem* de tamañao *size* cada uno
  - La reserva de memoria se hace en el *heap*
  - **NO inicializa** con ningún valor la zona de memoria reservada
  - Esta zona queda reservada, no pudiéndose asignar en otra reserva posterior, salvo que se libere previamente
  - Se puede hacer un *casting* seguro del valor devuelto a un puntero de cualquier tipo de dato de tamaño menor o igual a *size*
- En caso de no poder satisfacerse la petición, devuelve NULL

Procedimiento habitual de reserva

```
int* p;
if((p=(int*)malloc(n*sizeof(int)))==NULL)
{
  printf("Error: no pudo asignarse memoria");
  exit(-1);
}
```



Comprobar si el resultado es NULL ¡¡Hacerlo siempre así!!

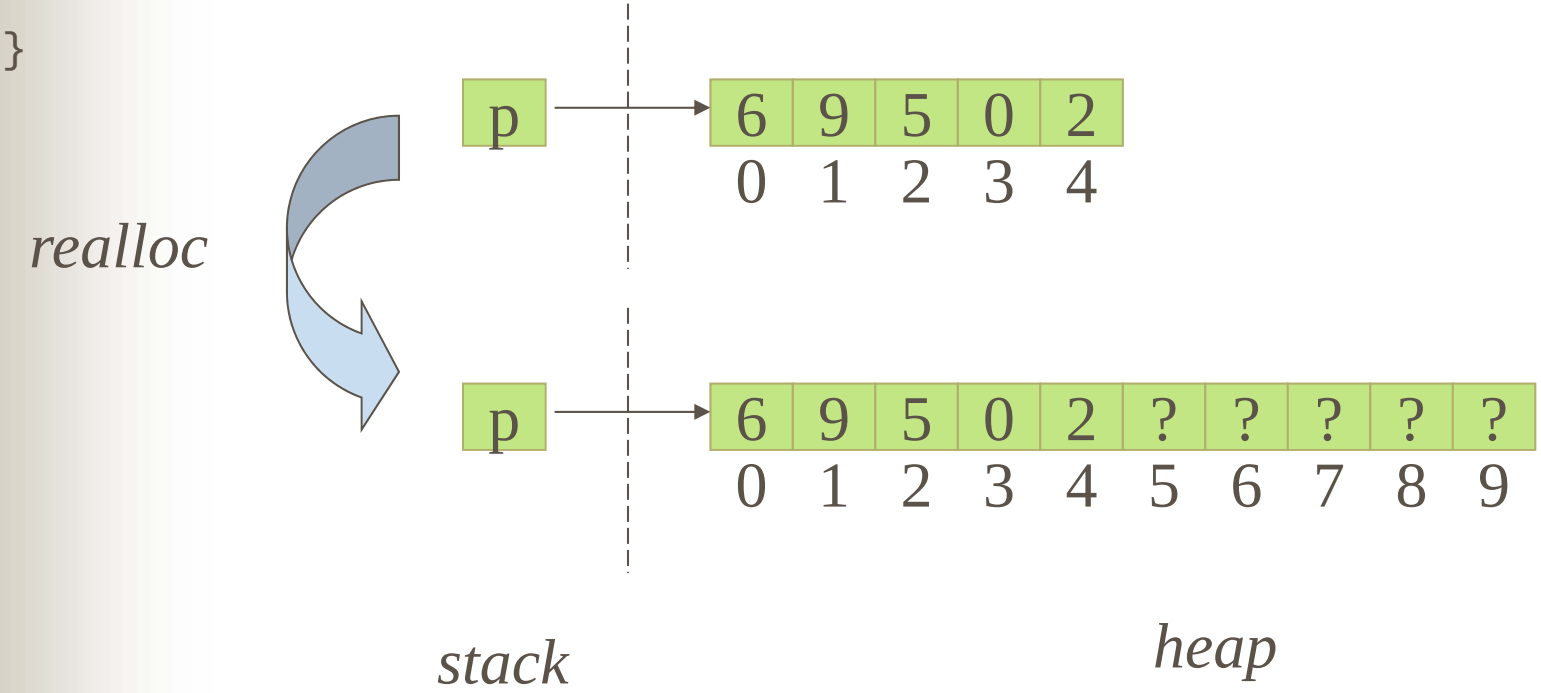
Gestión dinámica de memoria en C

void *realloc(void *ptr, size_t size)

- Reserva memoria para un objeto de tamaño *size*
- Devuelve la dirección del nuevo objeto, o NULL en caso de error
  - Si *ptr* es NULL, la función no almacena valores iniciales en el nuevo objeto creado
  - En otro caso, *ptr* debe ser la dirección de un objeto reservado previamente con *calloc*, *malloc* o *realloc*
    - Si el objeto previo es menor que el nuevo, *realloc* copia el objeto previo al inicio del nuevo objeto reservado (el contenido del resto de los elementos que quedan por rellenar es desconocido)
    - En otro caso, copia solamente la parte inicial del objeto previo que cabe dentro del objeto reservado
  - Si la petición no tiene éxito (devuelve NULL) el contenido del puntero sobre el que se hace la petición no se modifica (no se pierde nada)
  - Si *realloc* consigue reservar memoria para el nuevo objeto, desreserva la memoria asignada al objeto previo. En otro caso el objeto previo no cambia

Reasignación de espacio

```
if((p=(int*)realloc(p, 2*n*sizeof(int)))==NULL)
{
  printf("Error: no pudo asignarse memoria");
  exit(-1);
}
```



Gestión dinámica de memoria en C

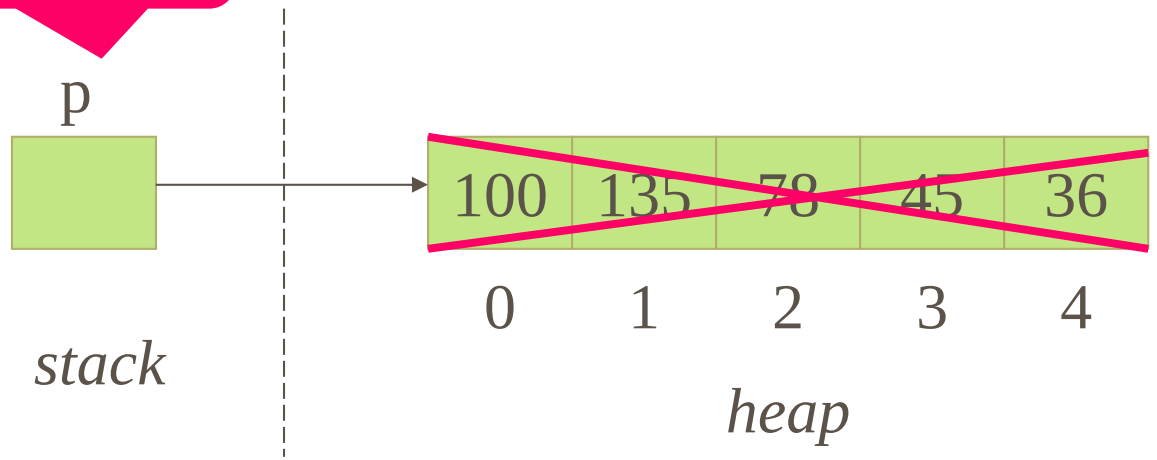
void free(void *ptr)

- Libera la zona de memoria del *heap* referenciada por *ptr*
  - *ptr* sigue conservando su valor (apuntando a la misma dirección de memoria), free no lo actualiza a NULL
- *ptr* debe ser un puntero devuelto por *calloc*, *malloc*, o *realloc*
- Si *ptr* es NULL no tiene efecto

Liberación de memoria

free(p)

Ojo! free no pone el valor de p a NULL



No olvidar liberar la memoria¡¡Hacerlo siempre así!!

Ejemplo

```
int* p;
p = (int*)malloc(5*sizeof(int)); //NO inicializa
free(p);
```

```
int* p;
p= (int*)calloc (5, sizeof(int));// SI inicializa
free(p);
```

Efecto:

1. Reserva espacio en el *heap* para *n* objetos de tamaño de un entero (reserva *n*sizeof(int) bytes*)
2. Marca como reservada esa zona en el *heap*
3. Hace la conversión apropiada para que el tipo devuelto (*void**) sea convierta a *int**
4. El puntero *p* contiene la dirección inicial de la zona de memoria asignada si todo va bien, o NULL si hay algún problema

El destino de un puntero

- Un puntero puede tener dos posibles caminos durante su existencia
  - Puede “apuntar” a un espacio de memoria de otra variable

```
int *a, b = 10;
a = &b;
```

  - Puede “apuntar” a un espacio de memoria propio. Ejemplo

```
int *a, b = 10;
a = malloc (sizeof (int));
a = b;
```

Algunas consideraciones importantes

Sobre la declaración de un puntero:

- Su valor inicial es basura
- No implica reserva de memoria en el *heap*, es una variable más (que almacena una dirección de memoria) y como tal se aloja en la zona de datos globales o en la pila
- Para referenciar a datos dinámicos, se le deberá asignar el resultado de *malloc()* o *calloc()*
- Podrá referenciar datos globales, locales o dinámicos

Algunas consideraciones importantes

Sobre la petición de memoria con *malloc*, *calloc* o *realloc*:

- Comprobar si el valor devuelto es NULL
- Si la petición con *realloc* no tiene éxito (devuelve NULL) el contenido del puntero sobre el que se hace la petición no se modifica (no se pierde nada)
- Devuelven *void** por lo que se requiere hacer un *casting* al tipo del puntero
- El operador *sizeof* permite calcular el tamaño del tipo de objeto para el que se reserva memoria.
- Se pueden utilizar para reservar memoria para objetos como estructuras, vectores y matrices de estructuras, etc.

Algunas consideraciones importantes

Sobre la zona de memoria asignada:

- Queda reservada y no se asignará a otra petición con *calloc*, *malloc*, *realloc*
- Puede accederse a ella de la manera habitual
- No se garantiza que el programa no pueda acceder fuera de los límites reservados. En este caso, el efecto será un error en tiempo de ejecución

Algunas consideraciones importantes

Sobre la liberación con *free*:

- El objetivo es que esa zona de memoria pueda reutilizarse
- La zona liberada queda, simplemente, marcada como libre. No se borra su contenido
- Después de liberarla puede accederse a ella (*free* no pone el puntero a NULL), aunque este procedimiento es totalmente desaconsejable
- Debe haber un free por cada *malloc* y *calloc*

Ejemplo básico

```
int* reservaVector(int nElementos)
{ int* p;
  if((p=(int*)calloc(nElementos,sizeof(int)))==NULL)
  { printf("\nError en reserva de memoria\n");
    exit(-1);
  }
  return(p);
}

void main()
{ int* p; //Puntero a enteros
  int noVale[5];
  int i, nElementos=5;
  //1.RESERVA NO VALIDA
  //no podemos asignar memoria dinamica a un vector declarado estaticamente
  //noVale = (int*) malloc (nElementos*sizeof(int));

  //2.RESERVA DE MEMORIA
   p=reservaVector(nElementos);
  //IMPORTANTE: no olvidar liberar memoria cuando ya no vaya a ser utilizada
  free(p);
}
```

vectDin.c

Ejemplo básico

```
void reservaVectorReferencia(int** Vector, int nElementos)
{
  if((*Vector=(int*)calloc(nElementos,sizeof(int)))==NULL)
  {
    printf("\nError en reserva de memoria\n");
    exit(-1);
  }
}
void main()
{ int* p; //Puntero a enteros
  int i, nElementos=5;
  reservaVectorReferencia(&p, nElementos);
  free(p);
}
```

Otros ejemplos

- Con cadenas de caracteres
  - ***duplicad.c***: Ejemplo de copia de cadenas estáticas a dinámicas
  - ***copycad.c***: Ejemplo de utilización de la función *realloc*. Lee una cadena e intenta copiarla en otra que inicialmente es demasiado pequeña. Se solicita más memoria hasta que se dispone del espacio suficiente para hacer la copia

duplicad.c

Otros ejemplos

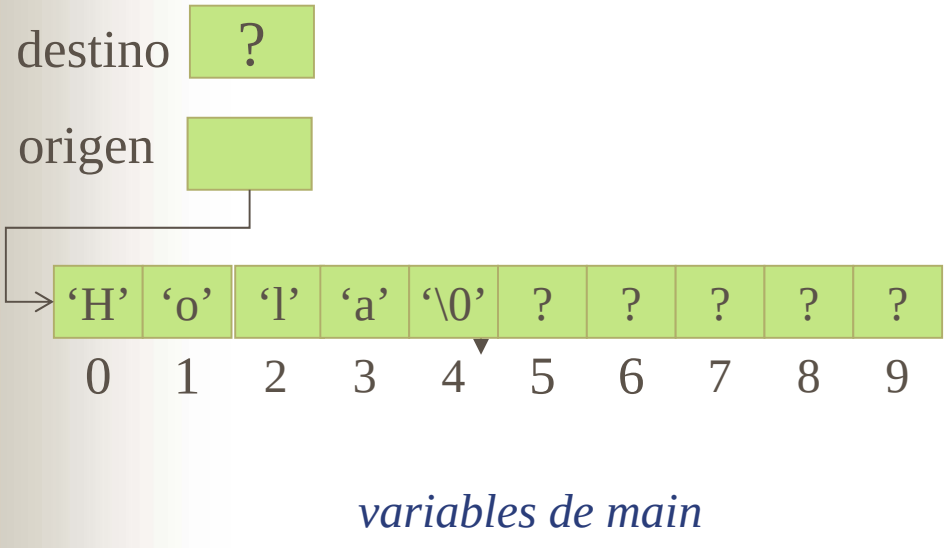
+1 para el '\0'

```
char* duplica(char* origen)
{
2  char* ptr;
3  if((ptr=(char*)malloc((strlen(origen)+1)*sizeof(char)))==NULL)
   {  printf("Error: no pudo asignarse memoria\n");
      ptr=NULL;
   }else
4    strcpy(ptr, origen);
   return(ptr);
}
int main()
{ char origen[10];
   char* destino;
   printf("Introducir cadena origen:\n");
1  gets(origen);
   destino = duplica(origen);
   printf("Origen: <%s>  Longitud: %d\n", origen, strlen(origen));
5  printf("Destino: <%s>  Longitud: %d\n", destino, strlen(destino));
   free(destino);
   return(0);
}
```

Otros ejemplos

1 *Antes de llamar a duplica*

stack ***heap***



variables de main

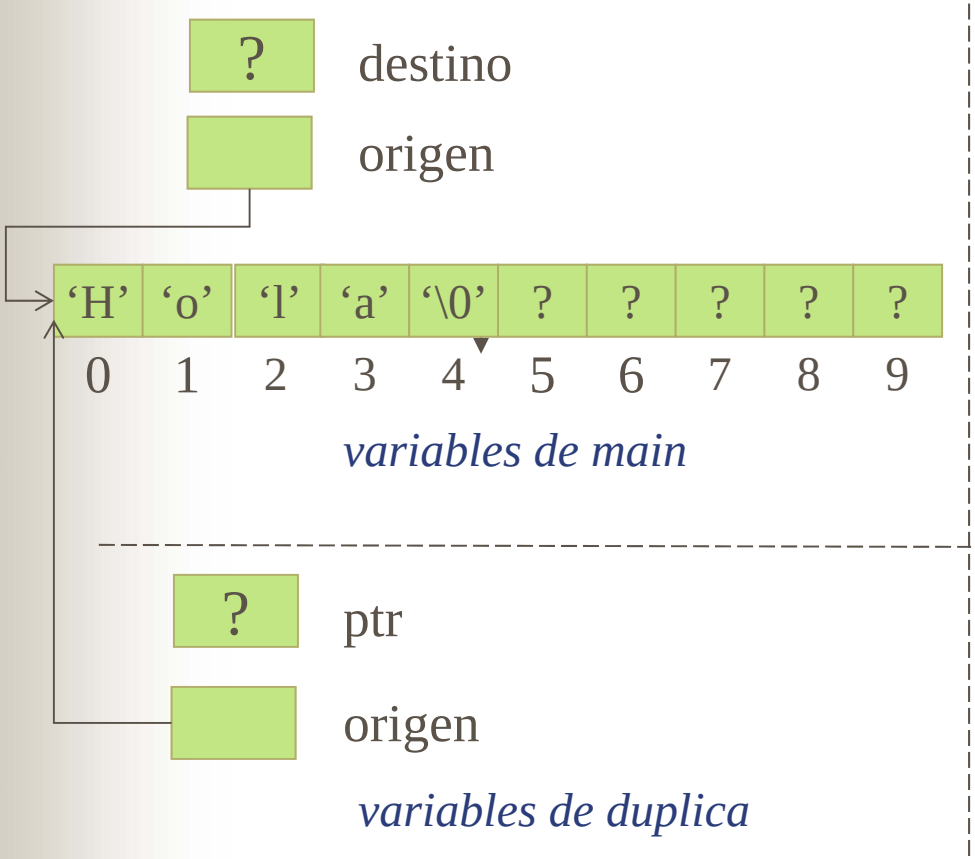
Otros ejemplos

2

Llamada a duplica

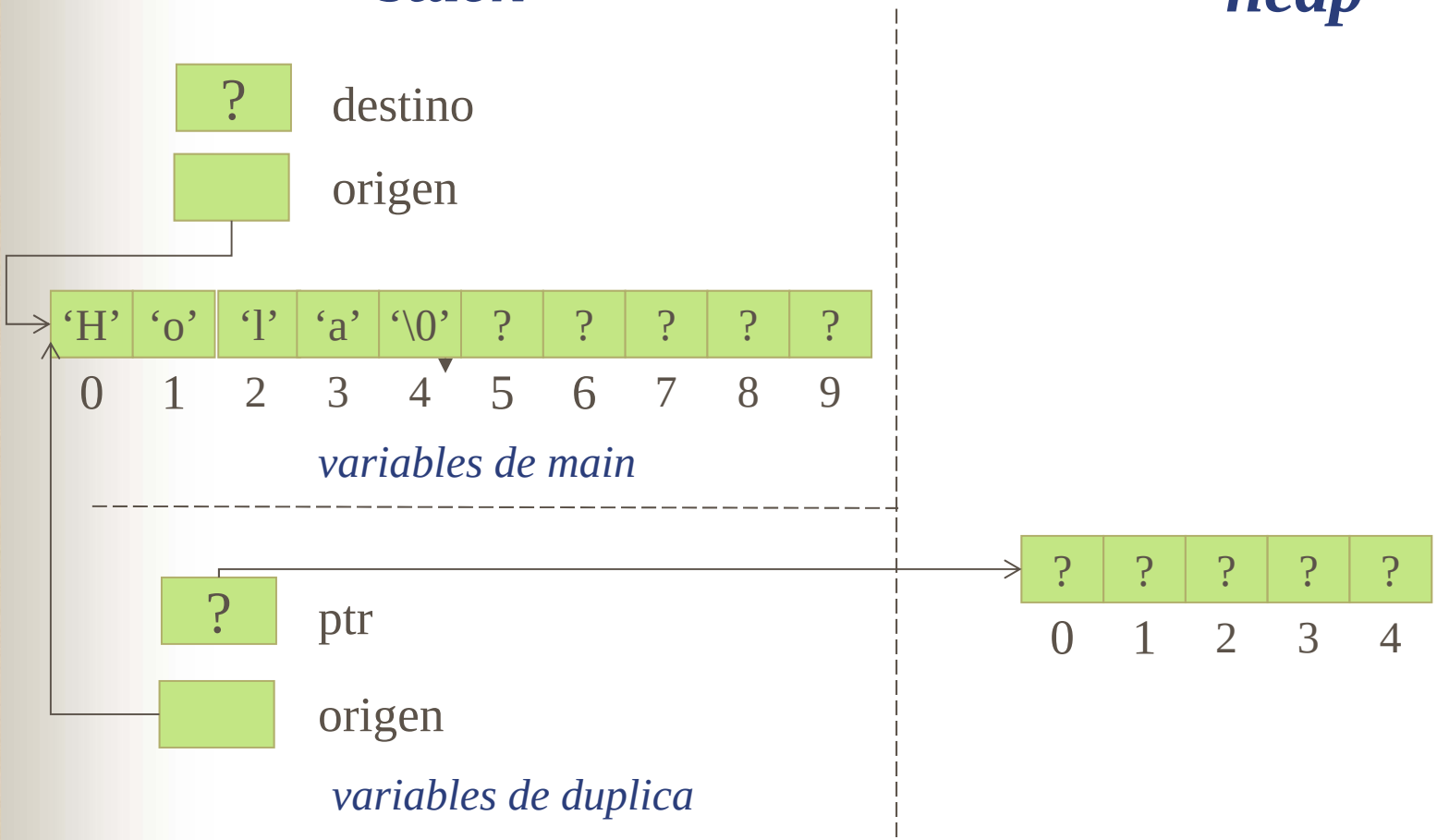
stack

heap



Otros ejemplos

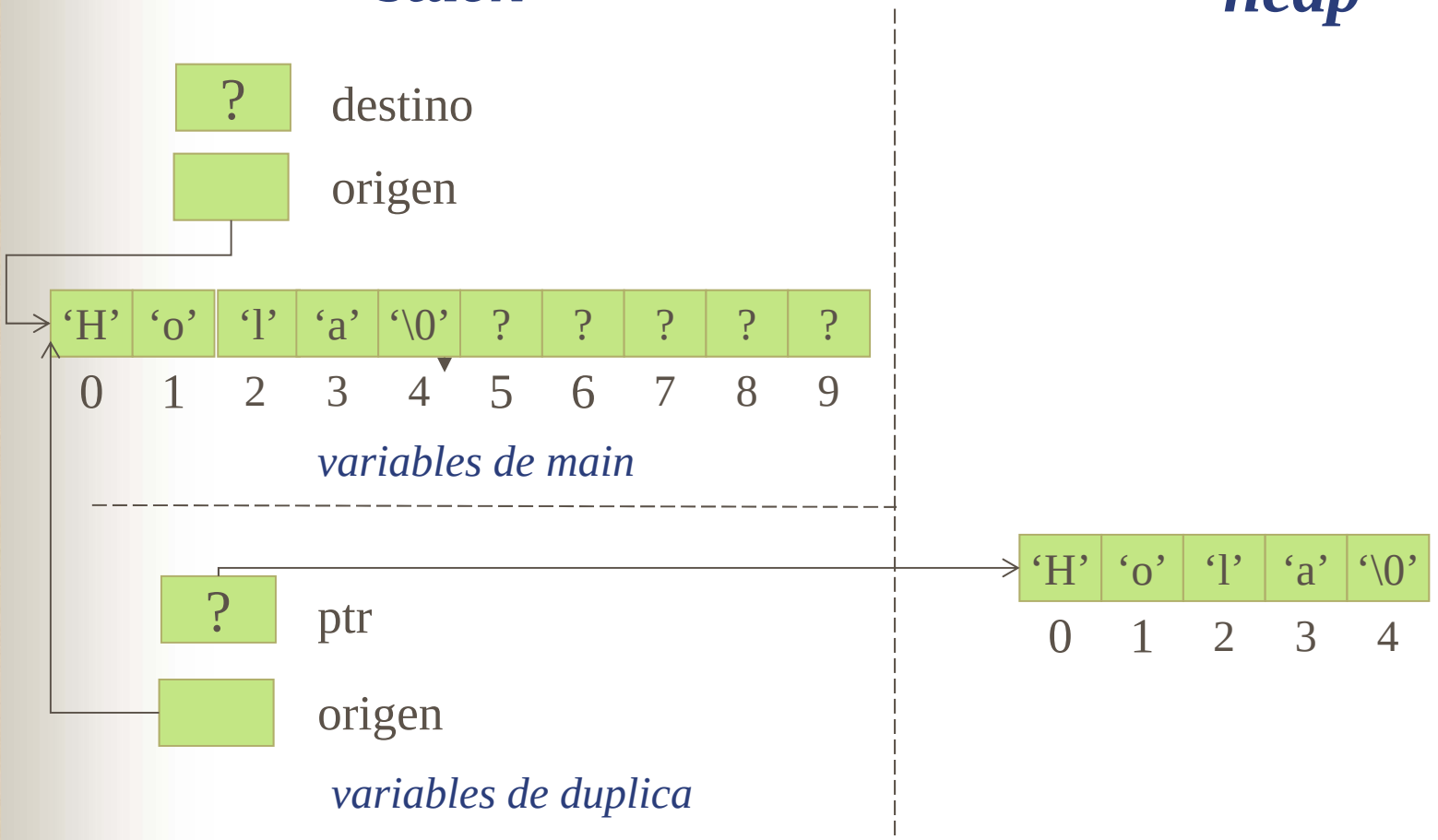
3 *Después de malloc*



Otros ejemplos

4

Después de strcpy



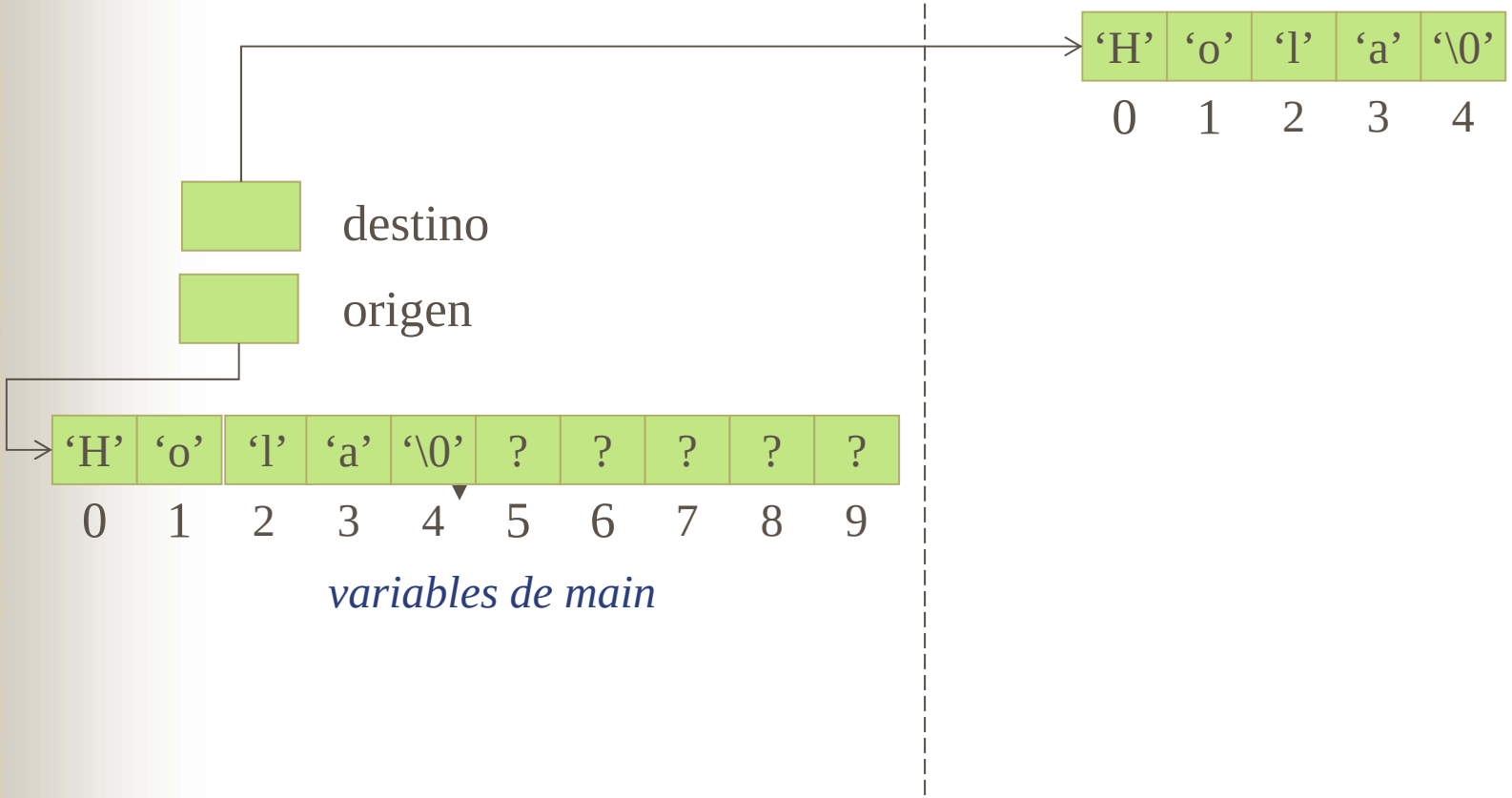
Otros ejemplos

5

Después de return

stack

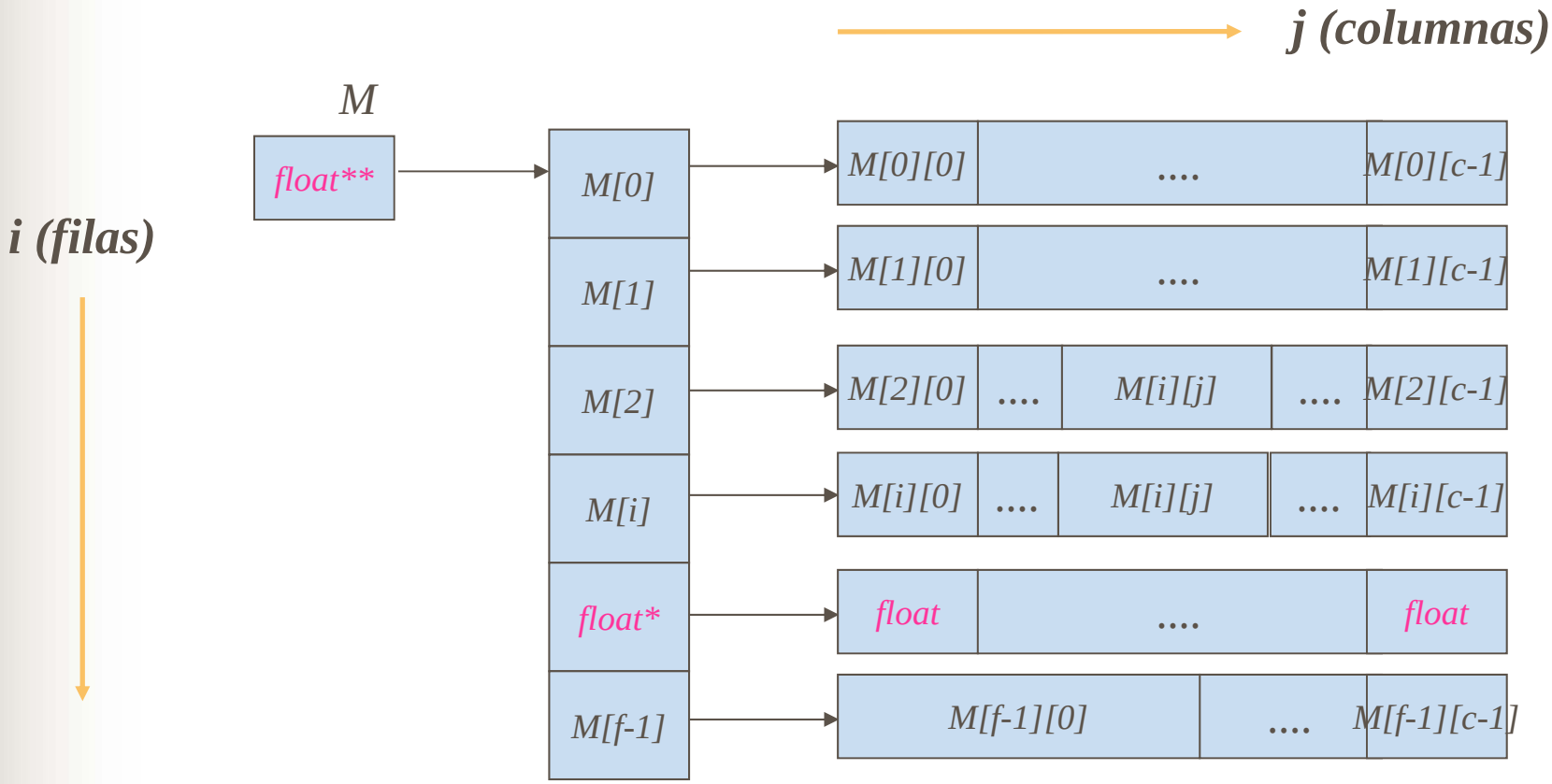
heap



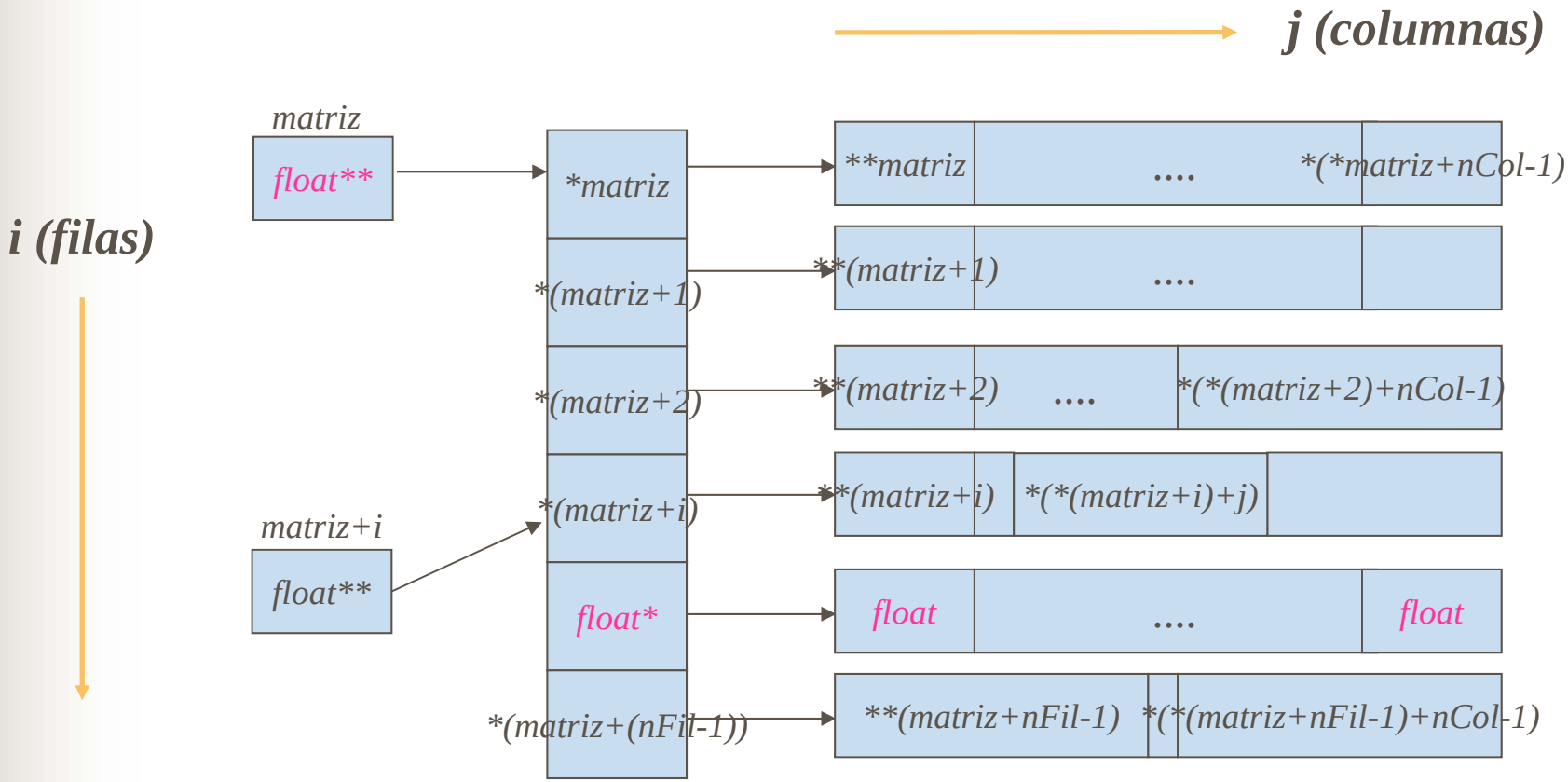
Matrices dinámicas

- Tenemos dos opciones:
  - Reservar y liberar la matriz por filas. Por simplicidad y parecido con lo que hemos visto hasta ahora, se recomienda esta opción
  - Reservar y liberar la matriz en un solo bloque

Reserva por filas



Reserva por filas



Reserva por filas

No olvidar añadir comprobación del valor devuelto por malloc!!

```
float** reservaMatrizDinamicaPorFilas(int nFil, int nCol)
{ float** Matriz;
  int i, j, error =0;
  Matriz=(float**)malloc(nFil*sizeof(float*));
  for(i=0; i<nFil;i++)
  {
    Matriz[i]=(float*)malloc(nCol*sizeof(float));
  }
  return(Matriz);
}
```

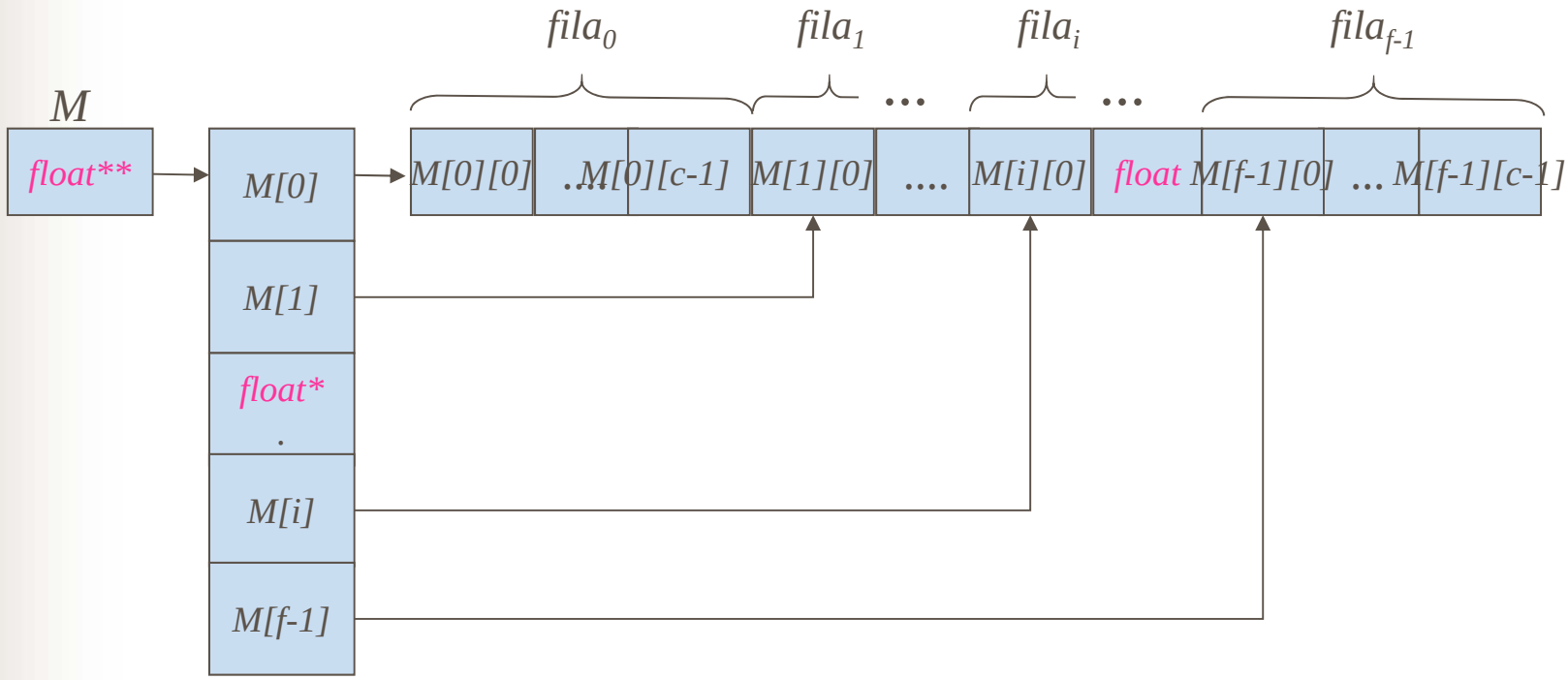
Reserva por filas

```
void liberaMatrizDinamicaPorFilas(float** Matriz, int nFil)
{
  int i;

  for(i=0; i<nFil; i++)
  {
    free(Matriz[i]);
  }
  free(Matriz);
}
```

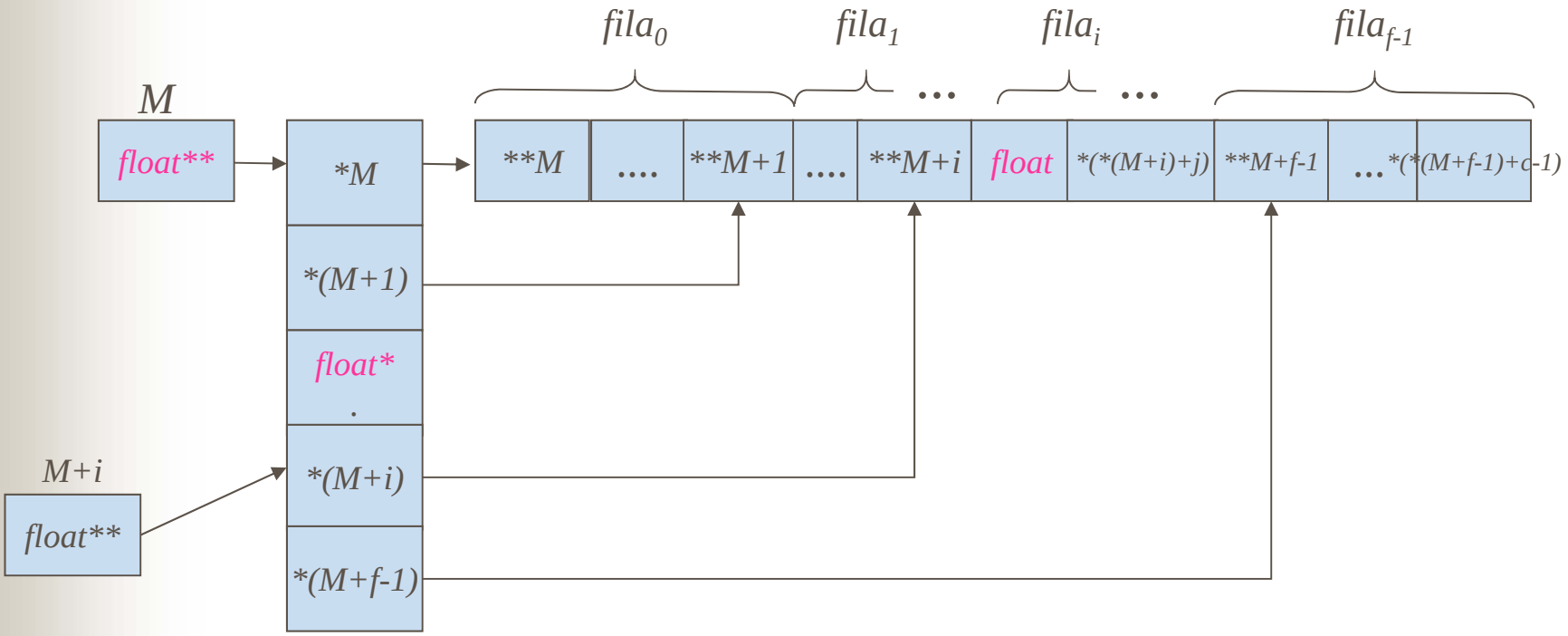
```
void liberaMatrizDinamicaPorFilas(float***
    Matriz, int nFil)
{
  int i;

  for(i=0; i<nFil; i++)
  {
    free((*Matriz)[i]);
  }
  free(*Matriz);
  (*Matriz) = NULL;
}
```

Si queremos poner Matriz a NULL, hay que pasarla por referencia

Reserva en un solo bloque



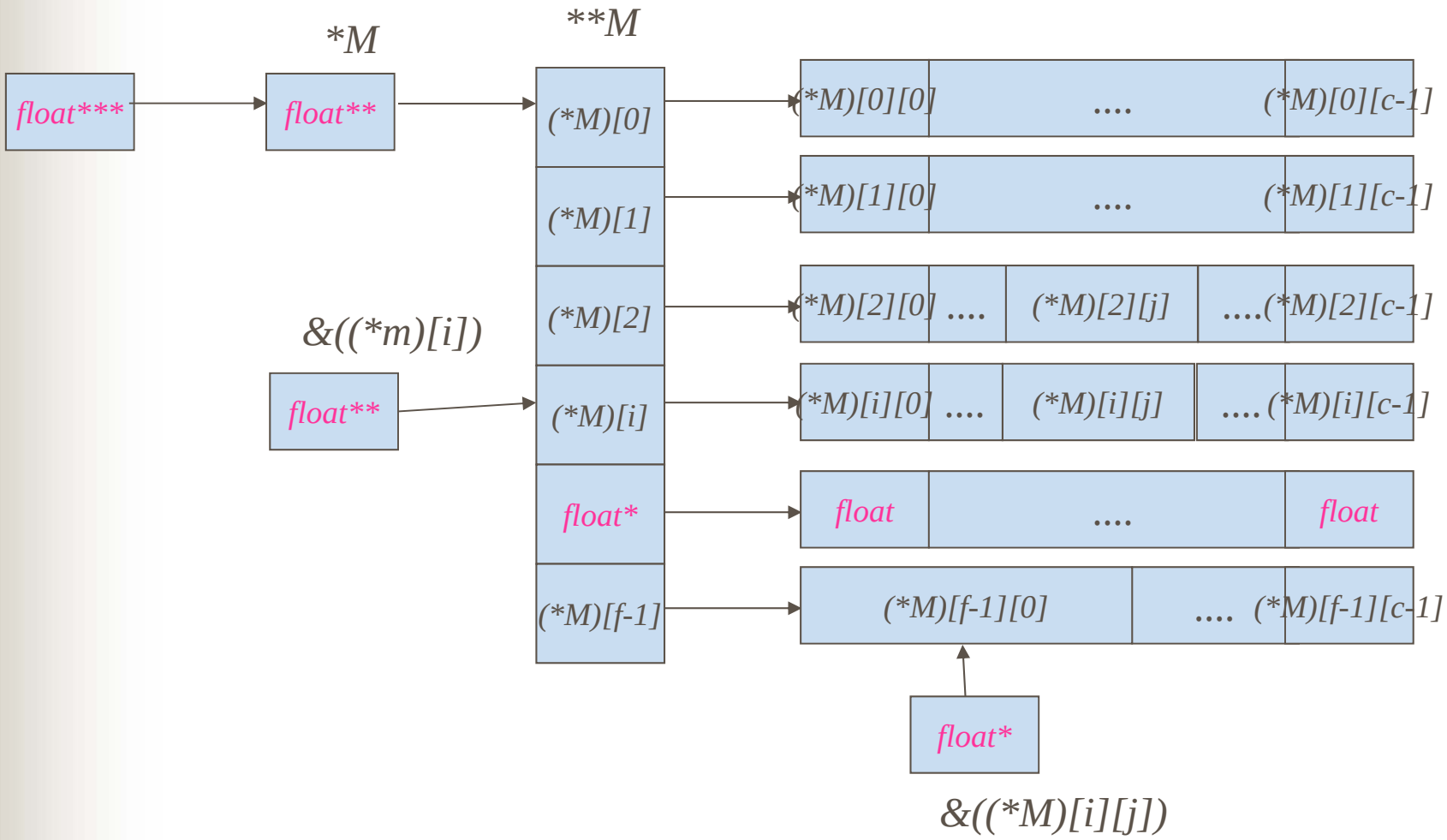
Reserva en un solo bloque



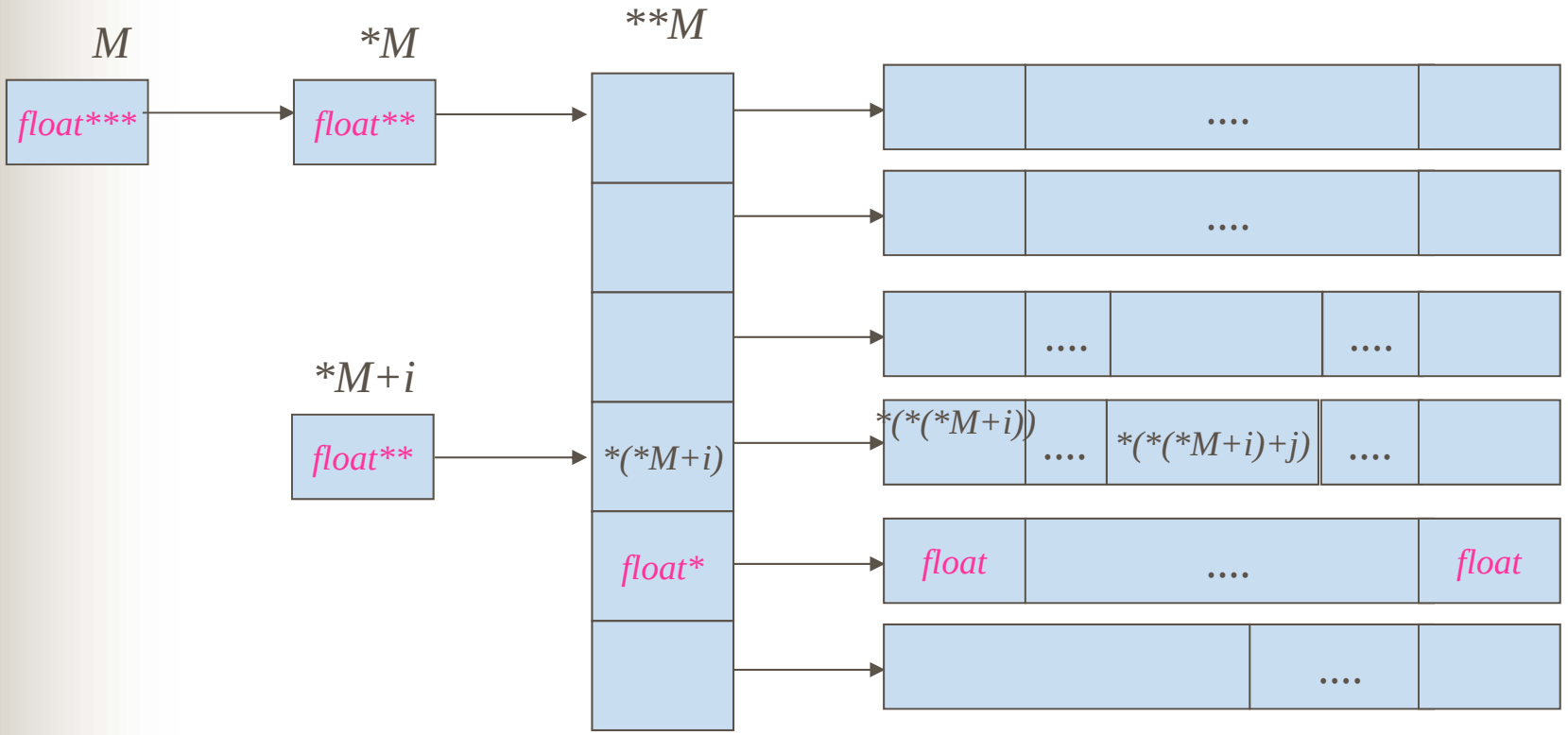
Reserva en un solo bloque

```
float** reservaMatrizDinamicaUnSoloBloque(int nFil, int nCol)
{  float** Matriz;
   int i;

   Matriz=(float**)malloc(nFil*sizeof(float*));
   Matriz[0]=(float*)malloc(nFil*nCol*sizeof(float));
   for(i=1; i<nFil; i++)
      Matriz[i]=Matriz[i-1]+nCol;
   return(Matriz);
}
void liberaMatrizDinamicaUnSoloBloque(float** Matriz)
{
  free(Matriz[0]);
  free(Matriz);
}
```

Reserva por referencia



Reserva por referencia



Reserva de matrices por referencia

```
void reservaMatrizDinamicaPorFilasRef(float*** Matriz,int
    nFil, int nCol)
{
  int i, j, error =0;
  *Matriz=(float**)malloc(nFil*sizeof(float*));
  for(i=0; i<nFil;i++)
  {
    (*Matriz)[i]=(float*)malloc(nCol*sizeof(float));
  }
}
```

Paso de matrices por referencia

- En ocasiones, es necesario pasar el puntero a la primera fila de la matriz por referencia:
  - Cuando queramos cambiar la dirección de comienzo de la matriz
  - Al crearla (si en lugar de devolverla con return la pasamos como parámetro)
  - Para que apunte a NULL

No hacer difícil lo fácil. Esto no es útil cuando sólo van a cambiar los valores de los elementos de la matriz

Punteros y matrices bidimensionales

- Igual que en el caso de los *arrays*, podemos utilizar tanto notación de punteros como notación de corchetes

```
int mat[DimF][DimC], **ptr;
ptr=mat
*ptr es un puntero a la primera fila
*(ptr+1) es un puntero a la segunda fila
**ptr es el valor mat[0][0]
**(ptr+1)es el valor mat[1][0]
*(*(ptr+1)+2) es el valor mat[1][2]
```

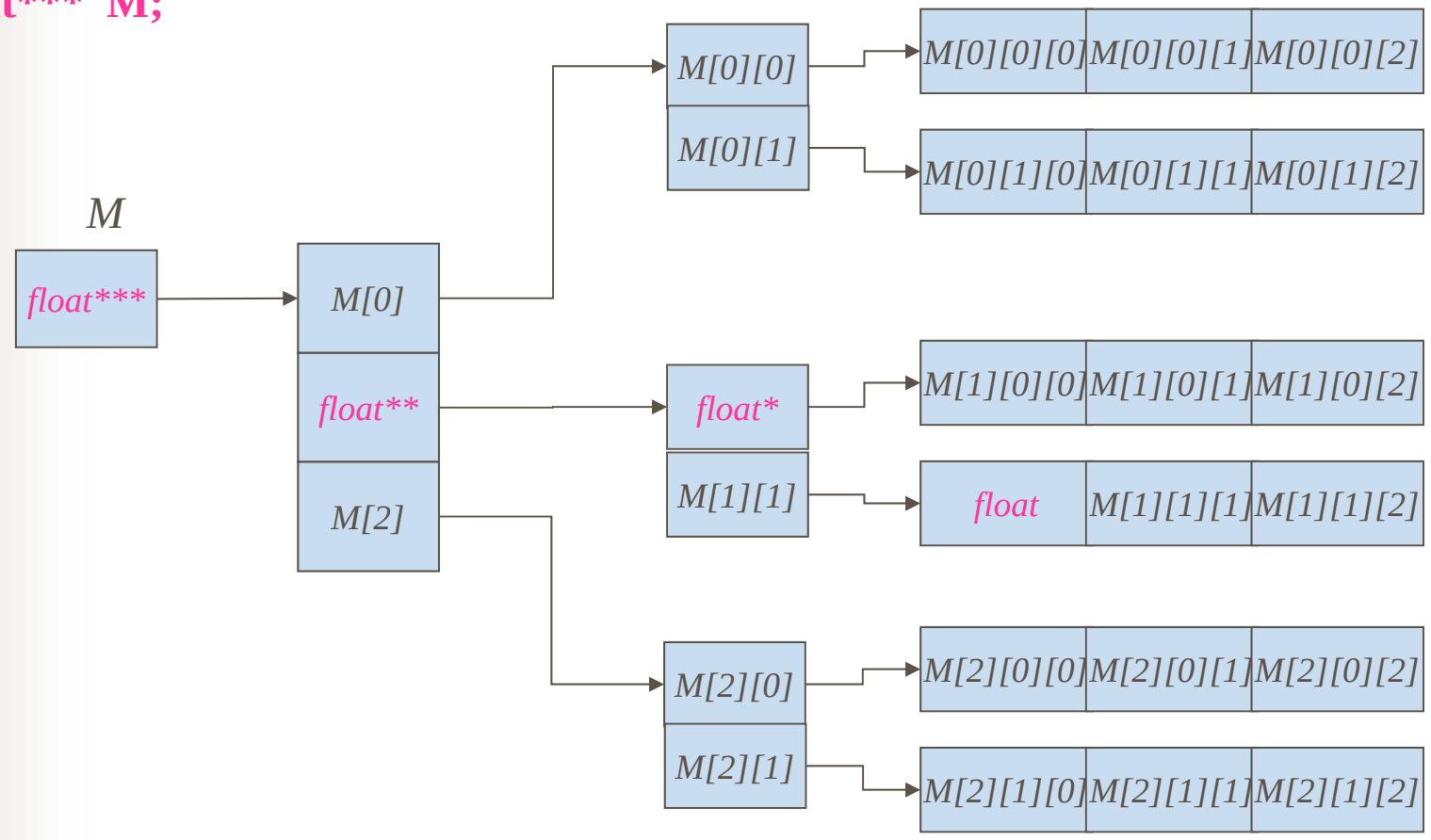
En general

ptr[i][j] ≈ * (*(ptr+i)+j) *bidimensional*

ptr[i][j][k] ≈ *(* (*(ptr+i)+j)+k) *tridimensional*

Matrices n-dimensionales

float* M;**



Matrices n-dimensionales

```
float*** reservaMatrizTridimensional(int nFil, int nCol, int nAlt)
{
  float*** Matriz;
  int i, j;

  Matriz=(float***)malloc(nFil*sizeof(float**));
  for(i=0; i<nFil;i++)
  {
       Matriz[i]=(float**)malloc(nCol*sizeof(float*));
       for(j=0; j<nCol; j++)
       {
         Matriz[i][j]=(float*)malloc(nAlt*sizeof(float));
       }
  }

  return(Matriz);
}
```

Matrices n-dimensionales

```
void liberaMatrizTridimensional(float*** Matriz,int nFil, int nCol)
{
  int i, j;

  for(i=0; i<nFil; i++)
  {
    for(j=0; j<nCol; j++)
    {
      free(Matriz[i][j]);
    }
    free(Matriz[i]);
  }
  free(Matriz);
}
```

Vectores y matrices de estructuras

```
struct dato* reservaVectorStr(int nEle)
{
  struct dato* ptr;

  if((ptr=(struct dato*)malloc(nEle*sizeof(struct
   dato)))==NULL)
  {
    printf("\nError en la reserva de memoria");
    exit(-1);
  }
  return(ptr);
}
```

```
struct dato
{
  int n;
};
```

Vectores y matrices de estructuras

```
void reservaVectorStrReferencia(struct dato** ptr, int nEle)
{
  if((*ptr=(struct dato*)malloc(nEle*sizeof(struct
   dato)))==NULL)
  {
    printf("\nError en la reserva de memoria");
    exit(-1);
  }
}
void liberaVectorStr(struct dato* ptr)
{
  free(ptr);
}
```

Vectores y matrices de estructuras

```
struct dato** reservaMatrizStr(int nFil, int nCol)
{ struct dato** ptr;
  int i;
  if((ptr=(struct dato**)malloc(nFil*sizeof(struct dato*)))==NULL)
  { printf("\nError en la reserva de memoria (1)");
    exit(-1);
  }
  for(i=0; i<nFil; i++)
  { if((ptr[i]=(struct dato*)malloc(nCol*sizeof(struct dato)))==NULL)
    {
      printf("\nError en la reserva de memoria (2)");
      exit(-1);
    }
  }
  return(ptr);
}
```

Vectores y matrices de estructuras

```
void reservaMatrizStrReferencia(struct dato*** ptr,int nFil,int nCol)
{ int i;
  if((*ptr = (struct dato**)malloc(nFil*sizeof(struct dato*)))==NULL)
  {
    printf("\nError en la reserva de memoria (1)");
    exit(-1);
  }
  for(i=0; i<nFil; i++)
  {
    if(((*ptr)[i] = (struct dato*)malloc(nCol*sizeof(struct
    dato)))==NULL)
    {
      printf("\nError en la reserva de memoria (2)");
      exit(-1);
    }
  }
}
```

Vectores y matrices de estructuras

```
void liberaMatrizStr(struct dato** ptr, int nFil)
{
  int i;

  for(i=0; i<nFil; i++)
  {
    free(ptr[i]);
  }
  free(ptr);
}
```

cadenasEstruct.c

Estructuras y cadenas

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct palabra{
   char * cadDinamica;
   int letras;
   char cadEstatica[3];
};

int main () {
   struct palabra pal1, pal2, pal3;
   char cadena[256];

   printf("Introduce una cadena: ");
   scanf("%s",&cadena);
   pal1.cadDinamica=(char*)malloc(sizeof(char)*(strlen(cadena)+1));
   strcpy(pal1.cadDinamica,cadena);
   pal1.letras=strlen(pal1.cadDinamica);
   strcpy(pal1.cadEstatica,"tu");

   pal2=pal1;
```

Estructuras y cadenas

```
    printf("Direccion memoria pal1.cadDinamica: %p\n",&(pal1.cadDinamica));
    printf("Direccion memoria pal2.cadDinamica: %p\n",&(pal2.cadDinamica));
    printf("Direcion de inicio de la cadena <%s> almacenada en pal1.cadDinamica:
      %p\n",pal1.cadDinamica,pal1.cadDinamica);
    printf("Direcion de inicio de la cadena <%s> almacenada en pal2.cadDinamica:
      %p\n",pal2.cadDinamica,pal2.cadDinamica);
    printf("Direcion de inicio de la cadena <%s> almacenada en pal1.cadEstatica:
      %p\n",pal1.cadEstatica,pal1.cadEstatica);
    printf("Direcion de inicio de la cadena <%s> almacenada en pal2.cadEstatica:
      %p\n",pal2.cadEstatica,pal2.cadEstatica);
}
```

```
Introduce una cadena: Direccion memoria pal1.cadDinamica: 0022FF60
Direccion memoria pal2.cadDinamica: 0022FF50
Direcion de inicio de la cadena <‡> almacenada en pal1.cadDinamica: 003E3FB0
Direcion de inicio de la cadena <‡> almacenada en pal2.cadDinamica: 003E3FB0
Direcion de inicio de la cadena <tu> almacenada en pal1.cadEstatica: 0022FF68
Direcion de inicio de la cadena <tu> almacenada en pal2.cadEstatica: 0022FF58
```

(c) Manz & Tankian http://tira.emezeta.com La tira cómica de Bit y Byte