

SISTEMAS INFORMÁTICOS

GITT, GIT, GIST Y GIEC

PRÁCTICA 5: Gestión de memoria dinámica

UAH, Departamento de Automática, ATC-SOL

OBJETIVOS

- Conocer el concepto de puntero y adquirir soltura en su utilización
- Realizar programas que creen, utilicen y liberen arrays dinámicos
- Gestionar el uso de memoria dinámica

TEMPORIZACIÓN

- Inicio de la práctica: Semana del 27 de noviembre
- Tiempo de desarrollo de la práctica: 3 semanas. Finalizarla antes del examen parcial 4
- Evaluación: ver fechas en el aula virtual

1. Conceptos de arrays dinámicos

Un array dinámico es un array de tamaño variable, lo cual significa que es posible añadir y eliminar elementos después de crear el array. Además, el número inicial de elementos es introducido desde teclado por el usuario en tiempo de ejecución¹.

Para trabajar con un array dinámico necesita declarar un puntero que apunte al tipo de dato que se almacenará en el array. A continuación debe reservar memoria para el puntero mediante una de las funciones `malloc()`, `calloc()` o `realloc()`. Una vez reservada la memoria, el puntero apunta al primer elemento del array y puede hacer uso de él mediante el almacenamiento y la lectura de datos. Analice los siguientes ejemplos que gestionan memoria dinámica y **realice las acciones que se solicitan**:

1.1 Ejemplo 1

El siguiente programa crea un array dinámico de números enteros, lo inicializa (con datos introducidos desde teclado) y visualiza por pantalla dicha información. Antes de finalizar el programa libera la memoria que fue reservada.

```
#include "cabecera.h"

int main()
{
    int nelementos, i, *p;

    // Lectura del número de elementos que formarán el array
    printf("Introduzca el numero de elementos\n");
    scanf("%d", &nelementos);

    // Reserva de memoria para nelementos
    p = (int *) malloc(nelementos*sizeof(int));
    if (p==NULL)
    {
        printf("Error al asignar memoria");
        return -1;
    }

    // Inicialización del array
    printf("\n\nIntroduzca los datos del array\n\n");
    for (i=0; i<nelementos; i++)
    {
        printf("\nDato %d: ", i+1);
        scanf("%d", &p[i]);
    }

    // Impresión de los datos por pantalla
    printf("Los datos almacenados en el array son:\n\n");
    for (i=0; i<nelementos; i++)
```

¹ Las características mencionadas no son permitidas por los arrays estáticos estudiados hasta ahora.

```
    printf("\t%d", p[i]);  
    printf("\n\n");  
  
    // Liberación de memoria  
    free(p);  
    MemoryManager_DumpMemoryLeaks();  
    system("pause");  
    return 0;  
}
```

Suponga que el usuario introduce el valor 5 como número de elementos del array (nelementos=5). Nada más ejecutar con éxito la reserva de memoria (función `malloc()`), el programa dispone de un array de 5 elementos que está apuntado por el puntero `p`. tal y como aparece en la figura 1.

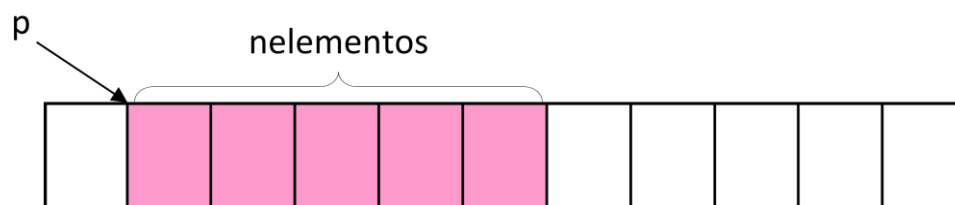


Figura 1. Representación gráfica de un array dinámico formado por 5 elementos

Suponga también que en el momento de inicializar el array, el usuario ha introducido desde teclado los siguientes valores: 76, 15, 45, 12 y 7. Estos datos son almacenados en cada una de las posiciones del array como muestra la figura 2.

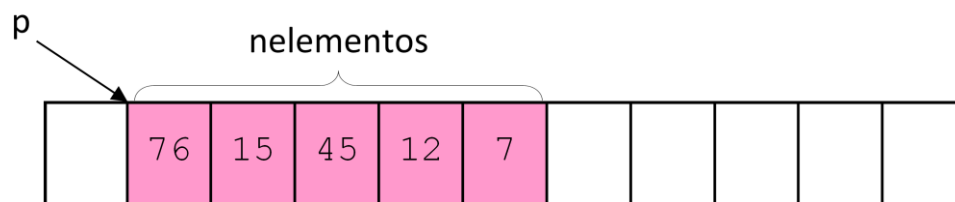
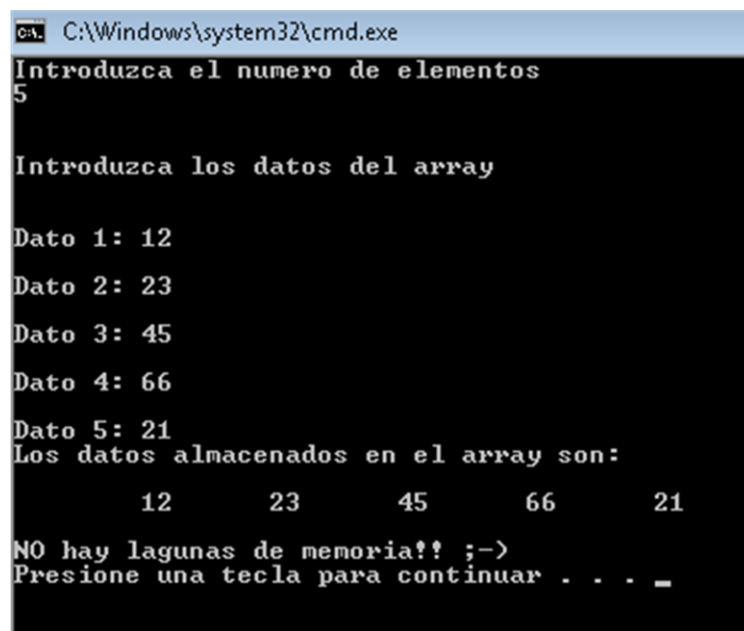


Figura 2. Representación gráfica del array una vez que han sido introducidos los valores iniciales

Después de liberar la memoria mediante la ejecución de la sentencia `free(p)`, el puntero sigue apuntando al mismo bloque de memoria, pero ya no puede ser utilizado pues, ya no se tiene acceso a esa zona. Si se intentara acceder se obtendría un error de ejecución. Realice las siguientes tareas:

1. Cree un proyecto, escriba el programa anterior² y ejecútelo, compare los resultados de ejecución con los datos mostrados en la figura 3.

² IMPORTANTE: Algunos errores relacionados con la gestión de memoria dinámica no son detectados por el compilador. Para solucionar este contratiempo **debe** utilizar la aplicación **MemoryManager** que se encuentra



```
C:\Windows\system32\cmd.exe
Introduzca el numero de elementos
5

Introduzca los datos del array

Dato 1: 12
Dato 2: 23
Dato 3: 45
Dato 4: 66
Dato 5: 21
Los datos almacenados en el array son:
      12      23      45      66      21

NO hay lagunas de memoria!! ;-)>
Presione una tecla para continuar . . . _
```

Figura 3. Imagen de la ejecución del ejemplo 1

2. Sustituya la función `malloc()` por la función `calloc()` según la siguiente sentencia y compruebe que el programa se ejecuta correctamente.

```
p = (int *) calloc(nelementos, sizeof(int));
```

3. La función `realloc()` añade nuevos elementos al final de un array dinámico. Esta función recibe un puntero al bloque de memoria al que se van a añadir los elementos y el tamaño total del array en bytes ((número de elementos que tenía + número de elementos que se van a añadir)*sizeof(int)). Retorna un puntero al comienzo del array.

Si el puntero que recibe es `NULL`, significa que el array no existe y en tal caso, la función `realloc()` lo creará. Sustituya la función `calloc()` por la función `realloc()`. Inicialice el puntero `p` a `NULL` y compruebe el buen funcionamiento del programa.

4. Intente acceder de nuevo a la zona de memoria después de liberada. Por ejemplo visualice de nuevo el array por pantalla después de la sentencia `free(p)`. ¿Que ocurre al ejecutar el programa? ¿Obtiene un error de ejecución? ¿Por qué? La razón, como se comentó anteriormente, es que esa zona de memoria ya no pertenece al programa y no está permitido acceder a ella.

publicada en el aula virtual junto con el enunciado de esta práctica. Puede consultar las instrucciones para un uso correcto en el **anexo** de esta práctica o en el aula virtual.

1.2 Ejemplo 2

El siguiente ejemplo añade un elemento al final del array dinámico del ejercicio anterior. Para ello reserva memoria para un elemento más y lee desde teclado el dato que va a albergar la posición de memoria añadida. El ejercicio se realiza a través de una función que recibe el array dinámico (puntero `p`) por referencia y su número de elementos (SIN contar el elemento que se va a añadir). Retorna **-1** si se produjo error al reservar memoria, o el **nuevo número de elementos** (contando el nuevo elemento añadido) si no se produjo error. Analice el código de la función `AnyadeElemento()`:

```
int AnyadeElemento(int **p, int nelementos)
{
    int *paux = *p;
    paux = (int *) realloc(*p, (nelementos + 1) * sizeof(int));
    if (paux != NULL)
    {
        printf("Introduce el nuevo valor\n\n");
        scanf("%d", &paux[nelementos]);
        *p = paux;
        return ++nelementos;
    }
    else
        return -1;
}
```

Según lo expuesto, el código que llama a la función tendrá el siguiente aspecto:

```
printf("Se va a a%cadir un nuevo elemento\n",164);
n = AnyadeElemento(&p, nelementos);
if (n!= -1)
    nelementos = n;
else
{
    printf("Error al reservar memoria\n");
    free(p);
    return -1;
}
```

Añada la función al proyecto del ejemplo anterior y sitúe el bloque de código correspondiente a la llamada a la función `AnyadeElemento()` en `main()`. **Compile y compruebe que todo se ejecuta sin errores y que se visualiza el mensaje “No hay lagunas de memoria”** generado por la aplicación `MemoryManager`.

Note que el puntero `p` es pasado por referencia a la función. En principio cabría pensar que no es necesario pasarlo por referencia, puesto que añadir un nuevo elemento en la última posición no implica el cambio de la dirección inicial del array, y, por tanto, `p` no variará. Si `p` no va a variar, entonces ¿por qué pasarlo por referencia? La razón que avala esta forma de actuar es que el puntero si puede variar porque puede valer inicialmente `NULL` o la función `realloc()` puede reubicar el nuevo bloque de memoria correspondiente al array

en otro lugar de la memoria, porque en el lugar actual no hay más espacio disponible para el nuevo tamaño requerido; en este caso, mueve los elementos del array a la nueva dirección, libera la zona de memoria anterior y retorna la nueva dirección.

Note que la función `main()` necesita dos de los datos con los que la función `AnyadeElemento()` ha estado operando: el número de elementos del array y el puntero al primer elemento del array. El primero es retornado por la función `AnyadeElemento()` y el segundo es pasado por referencia para que si fuera uno de los dos posibles casos estudiados anteriormente, la función `main()` pueda tener acceso al nuevo puntero.

Tenga en cuenta esta información para **codificar una nueva versión de la función `AnyadeElemento()`**. La nueva función retornará el puntero al primer elemento del array y recibirá en sus parámetros los siguientes datos:

1. Un puntero al comienzo del array: en esta nueva versión no es necesario pasar el puntero por referencia, puesto que la función retornará el puntero al comienzo del array.
2. Un puntero al número de elementos: debe ser pasado por referencia, puesto que no es retornado por la función.

Sustituya la versión antigua de la función `AnyadeElemento()` por la versión actual, pero no destruya la versión anterior. Guarde ambas versiones, compárelas y reflexione acerca del motivo por el cual las dos son aptas para la función `main()`.

El párrafo siguiente recapitula acerca de los dos tipos de función realizadas, con el fin de fijar ideas o de aclararlas.

Como se ha expuesto, los datos que necesita la función `main()` para operar con un array dinámico son: **un puntero al primer elemento del array y el número de elementos**. Como también se vió, la primera versión de la función proporcionó estos datos a `main()` retornando el número de elementos y recibiendo por referencia el puntero al comienzo del array. Pues bien, la segunda versión de la función también proporciona a `main()` los mismos datos, pero en esta ocasión retorna el puntero al comienzo del array y recibe por referencia el número de elementos del array. Se da por supuesto que el código de la función se debe construir para que añada un nuevo elemento al array pero siempre de acuerdo a los argumentos recibidos. Ambos prototipos de funciones son igualmente válidos. Además, no son los únicos prototipos válidos, imagine, por ejemplo, que la función recibe por referencia el puntero al array y por valor el número de elementos y retorna -1 si no se ha podido reservar memoria o 0 si se ha añadido el elemento correctamente. En este caso, al pasar el puntero por referencia, `main()` puede percibir cualquier cambio que se haga sobre él. Por otro lado, `main()` también tiene acceso al número de elementos del array, puesto que conoce si se ha añadido un elemento correctamente (valor retornado por la función es 0) y en ese caso la propia función `main()` puede incrementar en uno el número de elementos del array.

Con este ejercicio se pretende que se de cuenta de que existen diversas opciones para realizar un código correctamente, siempre que considere de forma conjunta la llamada a la función y el código de esta.

1.3 Ejemplo 3

Este ejemplo elimina un elemento situado en una posición intermedia del array dinámico.

La supresión de un elemento del array se realiza utilizando de nuevo la función `realloc()`, aunque en este caso, debe considerar que el número de elementos del array será **el número de elementos que tenía menos uno**. La función `realloc()` actúa sobre el último elemento del array, es decir, borrará el elemento situado en la última posición. Esto no satisface nuestro deseo de borrar un elemento de una posición genérica k . Para solucionar el problema, antes de borrar el último elemento, se deben desplazar una posición a la izquierda todos los elementos que están situados a la derecha de la posición del elemento a eliminar (posición k). Una vez desplazados, se obtiene un array donde el elemento situado en la posición k ha sido destruido (cuando se copió encima el elemento situado a su derecha) y donde el último elemento está repetido. Este es el momento de llamar a la función `realloc()` para eliminar el último elemento.

A continuación se explica este planteamiento de forma más detallada con un ejemplo concreto. Observe las figuras 4 y 5 que representan gráficamente el proceso y lea de nuevo, cuantas veces sea necesario, el párrafo anterior hasta que quede totalmente entendido. Suponga que el array contiene los elementos mostrados en la figura 4 y se desea eliminar el elemento de la posición 6 ($k=6$, dato=2).

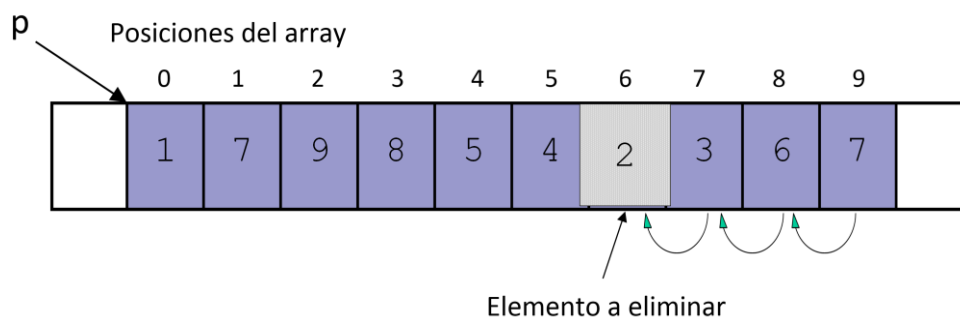


Figura 4. Array de 10 números de tipo entero. El elemento a eliminar se encuentra en la posición 6

La primera operación a realizar es el desplazamiento una posición a la izquierda de los elementos situados a la derecha del 2. Esta acción se realizará con un bucle. En la primera iteración del bucle se desplazará el número 3, que se grabará en la posición donde está el dato 2. Se perderá el dato 2, pero no importa, pues este es el dato que se quiere eliminar (observe la figura 5). Ahora el dato 3 estará repetido (en las posiciones 6 y 7). En la segunda iteración del bucle se desplazará el número 6 que se copiará en la posición 7 destruyendo el dato 3. Tampoco importa porque el dato 3 ya estaba almacenado en la posición 6. El elemento repetido ahora será el 6 pero al desplazar el ultimo elemento hacia la izquierda, es este último el que queda repetido (dato 7). Una vez finalizado el bucle, los datos del array han sido reorganizados para que sea posible eliminar el último elemento

mediante `realloc()`. Sin embargo el elemento realmente eliminado ha sido el dato 2 y los datos mantienen el orden que tenían inicialmente.

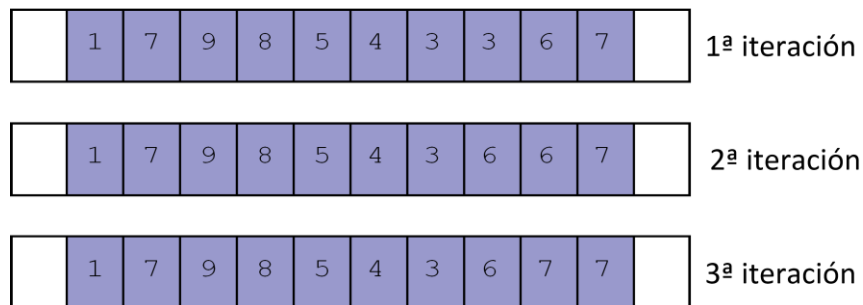


Figura 5. Situación del array después del desplazamiento que se realiza en cada iteración del bucle

La función `EliminaElemento()` mostrada a continuación codifica el algoritmo descrito anteriormente. Recibe el array (puntero a `p`), el número de elementos y la posición a eliminar del array, y retorna el número de elementos.

```
int EliminaElemento(int *p, int nelementos, int k)
{
    int i;

    for (i=k; i<=nelementos-2; i++)
        p[i]= p[i+1];
    p = (int *) realloc(p, (nelementos - 1) * sizeof(int));

    nelementos = nelementos-1;
    return nelementos;
}
```

Note que eliminar un elemento del array no requiere memoria adicional, todo lo contrario, se trata de liberar memoria, por lo tanto `realloc()` nunca necesitará mover el bloque de datos a otra zona de memoria y el puntero apuntando al inicio del array siempre será el mismo. Todo ello implica que cuando se va a eliminar un elemento no sea necesario un puntero auxiliar para trabajar con `realloc()`. ¿Significa esto que no es necesario pasar el puntero al principio del array por referencia a la función `EliminaElemento()`, puesto que este no se va a ver modificado? La respuesta es si, aunque con alguna reserva. Si el elemento a eliminar es el único elemento del array, `realloc()` retornará `NULL` y liberará la memoria. En este caso el puntero si cambia su valor, si bien esta circunstancia puede ser detectada desde fuera de la función a través del número de elementos del array.

Preste especial atención a la condición del bucle que realiza el desplazamiento de los datos: debe elegirse con cuidado para no sobrepasar los límites del array, puesto que la sentencia que ejecuta el bucle accede hasta la posición `i+1`. Debido a que el último valor que toma `i` es `nelementos-2`, si a este valor le sumamos 1: `nelementos-2+1 = nelementos-1`, que efectivamente se corresponde con la última posición del array, no excediendo así sus límites.


```
for (i=k; i<=nelementos-2; i++)  
    p[i]= p[i+1];
```

Añada la función al proyecto y la correspondiente llamada dentro de la función `main()` y ejecute el programa.

2. Ejercicio 1: gestión de los alumnos de una clase

Debe programar una aplicación que gestione un array dinámico de estructuras para almacenar información relativa a los alumnos de una clase. Cada estructura del array contiene los datos de un alumno que consisten en su nombre y su nota. Considere el siguiente tipo de estructura para la realización del programa:

```
#define MAX_CARAC 30  
  
typedef struct  
{  
    char nombre[MAX_CARAC];  
    float nota;  
}tficha;
```

La figura 6 representa el array dinámico una vez creado e inicializado.

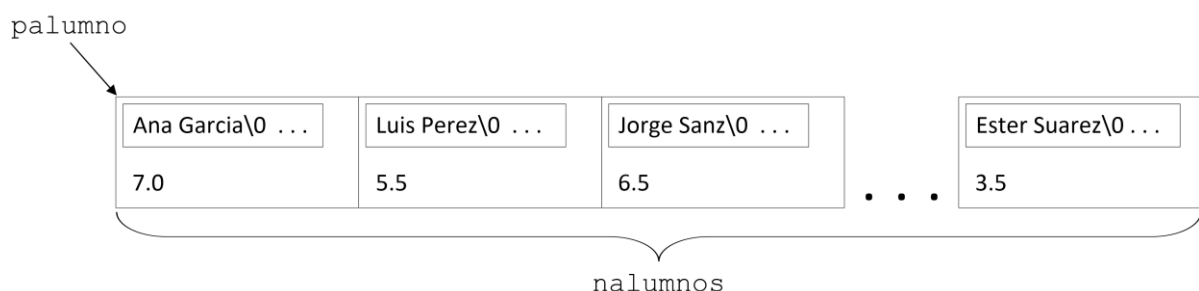


Figura 6. Array dinámico de estructuras con los datos de los alumnos de una clase

Las acciones que debe realizar la función `main()` son las siguientes:

- Lectura desde teclado del número de elementos del array (número de alumnos que hay en clase). El código debe asegurar que el número de alumnos leído es mayor que 0.
- Creación del array dinámico de estructuras mediante la función `malloc()`.
- Llamada a una función que inicialice el array (lectura de los datos desde teclado).
- Llamada a una función que visualice los datos del array.
- Liberación de la memoria.

2.1 Codificación de la aplicación

Cree un proyecto y añada el siguiente código en la función `main()` completando las sentencias que faltan.

```
int main()
{
    tficha *palumno;
    int nalumnos;

    // lectura del número de alumnos

    /* AÑADA AQUÍ LAS SENTENCIAS NECESARIAS PARA LEER EL NÚMERO DE
       ALUMNOS (NÚMERO MAYOR QUE 0) */

    // creación del array dinamico

    /* AÑADA AQUÍ LAS SENTENCIAS NECESARIAS PARA CREAR EL ARRAY */

    // Inicialización del array dinámico

    LeeAlumnos(palumno, nalumnos);

    // Visualización del array dinámico

    MuestraAlumnos(palumno, nalumnos);

    // Liberacion de memoria

    free(palumno);

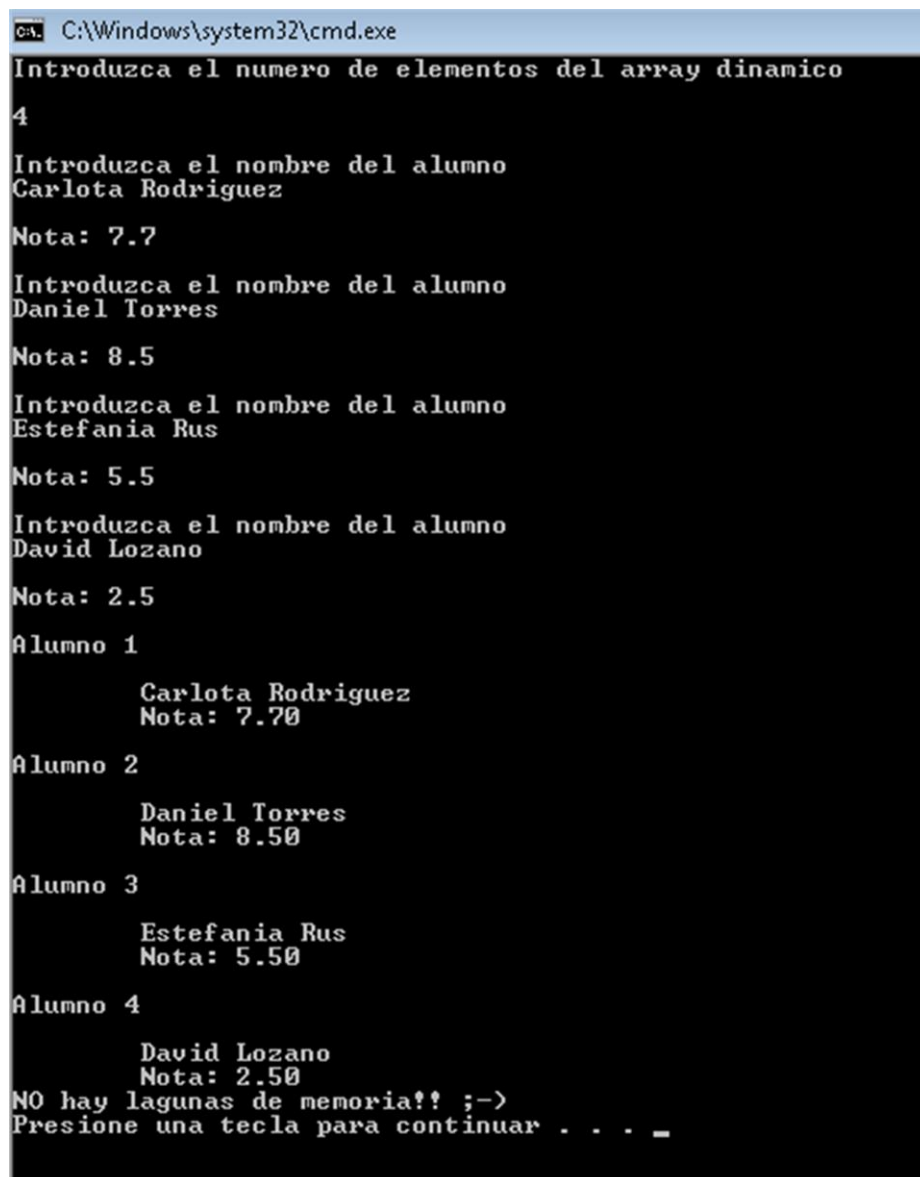
    MemoryManager_DumpMemoryLeaks();
    system("pause");
    return 0;
}
```

Observe las llamadas a las funciones, deduzca sus prototipos y escribalas en un nuevo fichero de código (`funciones.c`). Tenga en cuenta las siguientes indicaciones para una programación eficiente:

1. La función `LeeAlumnos()` lee desde teclado los datos de **todos** los alumnos. Para ello debe recorrer el array mediante un bucle y realizar una llamada a la función `LeeAlumno()` que se encarga de leer desde teclado los datos de **un** alumno. `LeeAlumno()` debe retornar una estructura con los datos leídos. Piense acerca de la siguiente cuestión: ¿Necesita esta función recibir algún argumento? Responda a la pregunta y **escriba ambas funciones** (`LeeAlumnos()` y `LeeAlumno()`).

2. La función `MuestraAlumnos()` muestra por pantalla los datos de **todos** los alumnos. Para ello debe recorrer el array mediante un bucle y realizar una llamada a la función `MuestraAlumno()` que se encarga de mostrar por pantalla los datos de **un** alumno. **Escriba ambas funciones** (`MuestraAlumnos()` y `MuestraAlumno()`).
3. Para leer una cadena de caracteres desde teclado debe utilizar la función `LeeCadena()` con la que ya trabajó en la práctica 4. Añádala al programa.

La figura 7 presenta la ejecución del programa con el fin de ayudarle en la comprensión del mismo.



```
C:\Windows\system32\cmd.exe
Introduzca el numero de elementos del array dinamico
4
Introduzca el nombre del alumno
Carlota Rodriguez
Nota: 7.7
Introduzca el nombre del alumno
Daniel Torres
Nota: 8.5
Introduzca el nombre del alumno
Estefania Rus
Nota: 5.5
Introduzca el nombre del alumno
David Lozano
Nota: 2.5
Alumno 1
    Carlota Rodriguez
    Nota: 7.70
Alumno 2
    Daniel Torres
    Nota: 8.50
Alumno 3
    Estefania Rus
    Nota: 5.50
Alumno 4
    David Lozano
    Nota: 2.50
NO hay lagunas de memoria!! ;->
Presione una tecla para continuar . . . _
```

Figura 7. Ventana de consola donde se ha ejecutado el programa

2.2 Pasando la estructura por referencia al leer un alumno desde teclado

Modifique el programa anterior de forma que la función `LeeAlumno()` no retorne ningún valor. Por lo tanto, la función que la ha llamado (`LeeAlumnos()`), no recibirá ningún dato. ¿Cómo puede entonces acceder al nombre y la nota del alumno que han sido leídos por la función `LeeAlumno()`? La respuesta es: pasando por referencia la estructura a la función, es decir, pasando la dirección de memoria donde se almacena la estructura. La función `LeeAlumno()` debe recoger esta dirección de memoria en un puntero y trabajar con el puntero para leer los datos. De esta forma la lectura de los datos será también visible desde la función `LeeAlumnos()`.

Realice la modificación, compile y ejecute de nuevo el programa. Tenga la precaución de guardar el código de las dos versiones de la función por si necesitara consultarlos más adelante.

Si ha realizado y le funcionan todas las actividades propuestas hasta aquí, ya ha aprendido a gestionarr arrays dinámicos de estructuras. No obstante, el campo de la estructura donde se almacena el nombre del estudiante es un array estático de 30 caracteres. Esto quiere decir que el nombre de cada estudiante ocupa 30 caracteres en memoria independientemente de su tamaño. Por ejemplo, el tamaño del nombre “Ana Perez” es 10 (incluyendo el carácter fin de cadena), por lo que se estarían desperdiciando 20 posiciones de memoria. Lo mismo ocurriría para todos aquellos nombres cuyo tamaño sea menor que 30. Este problema se puede solventar fácilmente creando arrays dinámicos de caracteres para almacenar los nombres de los alumnos, en lugar de arrays estáticos de tamaño fijo. El siguiente apartado explica la forma de llevarlo a cabo.

2.3 Array dinámico de estructuras y arrays dinámicos de cadenas de caracteres como campos de las estructuras

Para desarrollar este ejercicio debe cambiar el tipo de dato que alberga el nombre del estudiante, que pasará a ser un puntero a char. Considere el siguiente tipo de dato:

```
typedef struct
{
    char *pnombre;
    float nota;
}tficha;
```

En el apartado 2.1.1 programó un array dinámico de estructuras (observe la figura 6 donde se representó gráficamente). Utilizando el nuevo tipo de dato tendrá tantos arrays dinámicos como alumnos haya en el aula más uno. En efecto, por un lado tendrá el array dinámico de estructuras y por otro lado tendrá un array dinámico de caracteres por cada alumno, donde se almacenará su nombre. Los arrays dinámicos de caracteres estarán

apuntados por el campo `pnombre` (puntero a char) de cada estructura, es decir, `palumno[i].pnombre`³.

Tenga en cuenta que para que un puntero pueda almacenar información, es necesario que previamente se haya reservado memoria para el puntero (inicialización del puntero). La figura 8 representa el array dinámico de estructuras, una vez que se ha reservado memoria para el array de estructuras, pero todavía no ha sido reservada memoria para el campo `pnombre` de las estructuras, es decir, para los nombres de los alumnos:

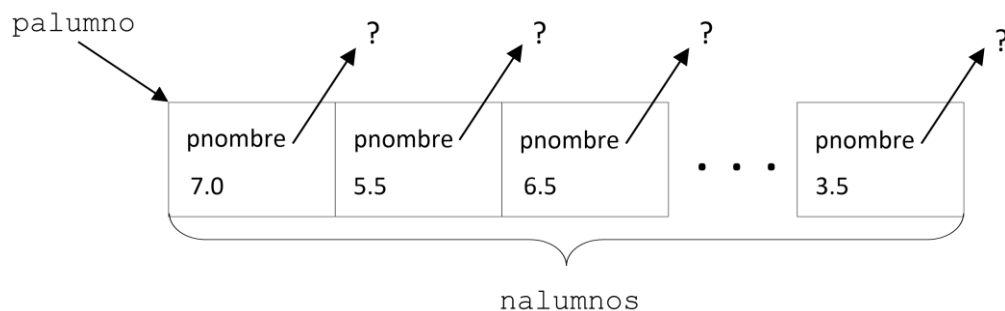


Figura 8. Array dinámico de estructuras cuyo campo `pnombre` es un puntero a char

Un análisis de la figura 8 puede llevarle a plantearse la siguiente cuestión ¿dónde apuntan los punteros `pnombre` de cada estructura? Estos punteros no apuntan todavía a una zona de memoria válida. Por tanto, los nombres de los alumnos no puede ser leídos directamente en el campo `pnombre`, puesto que no hay memoria reservada para estos punteros, a diferencia del apartado anterior donde al tratarse de un array estático la memoria no se reserva, sino que es asignado un número fijo de elementos en tiempo de compilación. Por tanto, previa a la lectura del nombre, debe realizarse la reserva de memoria para el campo `pnombre` de la estructura que se va a leer.

Si piensa cual debe ser la forma de actuar para realizar la reserva de memoria, puede surgir la siguiente inquietud: ¿Cómo puedo reservar memoria dinámica para una cadena de caracteres que todavía no ha sido introducida por teclado y por lo tanto no se conoce el número de caracteres que la componen? Para solventar esta dificultad primero debe leer el nombre desde teclado y almacenarlo en un array estático temporal. A continuación debe reservar memoria para tantos caracteres como tiene el nombre que acaba de leer y finalmente, copiar el nombre leído en la memoria reservada. Más concretamente debe proceder de la siguiente forma:

1. Lea desde teclado el nombre del alumno y almacenelo en el array estático de 30 caracteres que a tal efecto declare en la función `LeeAlumno()`.

³ Observe que el nombre del campo `pnombre` siempre debe ir asociado a una estructura (`palumno[i].pnombre`), no puede utilizarlo como si fuera una variable independiente (`pnombre`), pues no lo es y obtendría un error de compilación informándole de que la variable `pnombre` no ha sido declarada.

- Utilice `malloc()` para reservar el espacio de memoria del array dinámico donde va a almacenar el nombre del alumno que acaba de leer. Suponga un alumno genérico que se encuentra en una posición `k` del array de estructuras (ver figura 9). La memoria a reservar debe quedar apuntada por el campo `pnombre` de la estructura cuya posición es `k`, es decir: `palumno[k].pnombre`

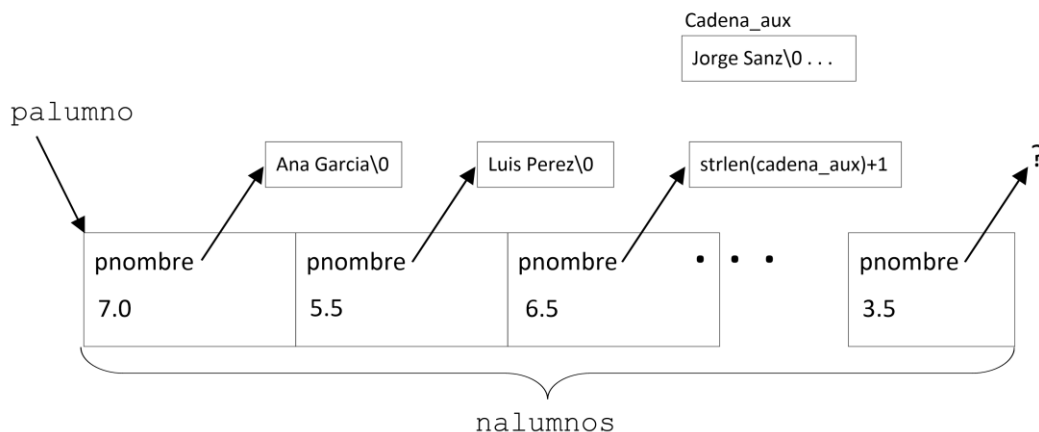


Figura 9. Reserva de memoria y lectura desde teclado del nombre de un alumno

Pero, ¿para cuantos caracteres reservo memoria? la respuesta es: para tantos caracteres como tiene el nombre que acaba de leer desde teclado y que almacené en el array estático. Y ¿cómo sé el número de caracteres que tiene ese nombre? El número de caracteres se puede calcular usando la función `strlen()`. Reserve memoria para `strlen() + 1` caracteres, de forma que pueda ser incluido también el carácter `'\0'`, que no fue tenido en cuenta por la función `strlen()`.

- Utilice la función `strcpy()` para copiar la cadena desde el array estático al espacio de memoria que acaba de reservar (figura 10).

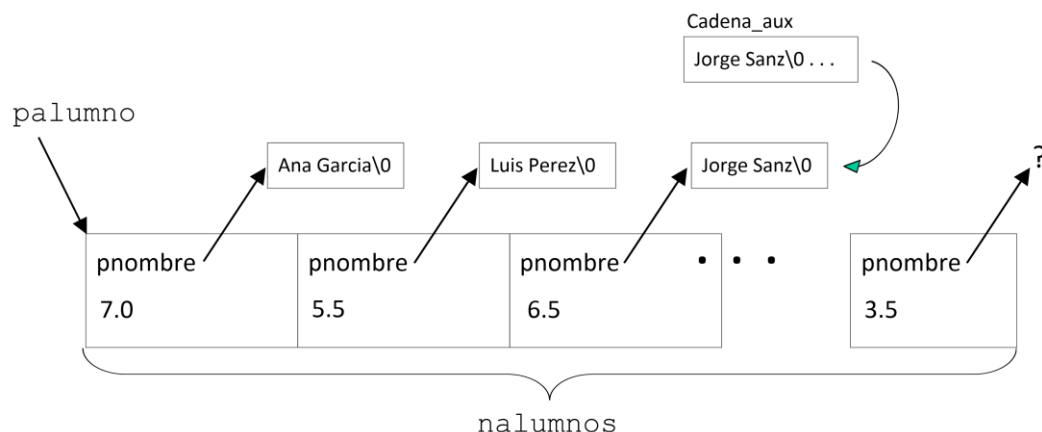


Figura 10. Copia del nombre del alumno desde la cadena estática al array dinámico creado para un alumno concreto

De esta forma, aunque se desperdicia memoria en el array estático que lee la cadena de caracteres inicialmente, pero solo se desperdicia memoria una sola vez, puesto que este array se utiliza para leer todas las cadenas de caracteres.

La modificación que requiere el programa para la realización de este apartado afecta a la lectura de datos (`LeeAlumnos()` y `LeeAlumno()`) y a la liberación de la memoria. Considere los siguientes prototipos de las funciones `LeeAlumno()` y `LeeAlumnos()` y `LiberaMemoria()`:

```
int LeeAlumno(tficha *palumno);
int LeeAlumnos(tficha *pficha, int n);
void LiberaMemoria(tficha *palumno, int nalumnos);
```

La función `LeeAlumno()` recibe por referencia el alumno a leer y retorna -1 si se produce error al reservar memoria o 0 si se ejecuta correctamente.

La función `LeeAlumnos()` recibe el puntero a la zona de memoria reservada para el array de estructuras y el número de alumnos que fue introducido desde teclado. Retorna el número de alumnos cuyo nombre ha sido leído sin errores. Si detecta un error al reservar la memoria para el nombre, debe finalizar el bucle. Recuerde que al dar por finalizado un bucle antes de ejecutar la totalidad de las iteraciones, la variable que indica el número de iteraciones ejecutadas (y por tanto el número de nombres leídos) es el índice del bucle.

En cuanto a la liberación de memoria, tenga en cuenta que debe ejecutar tantas veces la sentencia `free()` como veces haya ejecutado la sentencia `malloc()` y además, debe hacerlo en orden inverso. Es decir, si reservó memoria para el array de estructuras en primer lugar y para los nombres de los alumnos en segundo lugar, en el momento de liberar la memoria debe liberar primero los punteros que apuntan a los nombres (bucle for) y finalmente el puntero que apunta al array dinámico de estructuras. Esto debe ser así, puesto que para liberar los nombres de los alumnos necesita pasar por cada estructura para acceder al puntero `pnombre` (`palumno[i].pnombre`). Si liberase el puntero que apunta al array de estructuras en primer lugar ya no podrá acceder a esa zona de memoria para liberar los nombres de los alumnos, quedando estos sin liberar y por lo tanto sin poder ser utilizada por ningún otro programa. **Escriba las tres funciones:** `LeeAlumnos()`, `LeeAlumno()` y `LiberaMemoria()`.

Considere la siguiente funcion main():

```
int main()
{
    tficha *palumno;
    int nalumnos, nal_leidos;

    // lectura del número de alumnos
```

```
    /* AÑADA AQUÍ LAS SENTENCIAS NECESARIAS PARA LEER EL NÚMERO DE
```

```
ALUMNOS (NÚMERO MAYOR QUE 0) */

// creación el array dinamico

/* AÑADA AQUÍ LAS SENTENCIAS NECESARIAS PARA CREAR EL ARRAY */
// Inicialización del array dinámico

nal_leidos=LeeAlumnos(palumno, nalumnos);

if (nal_leidos < nalumnos)
{
    printf("Error el reservar la memoria de los nombres\n");
    if (nal_leidos == 0)
    {
        free(palumno);
        return -1;
    }
}

// Visualización del array dinámico

MuestraAlumnos(palumno, nal_leidos);

// Liberacion de memoria

LiberaMemoria(palumno, nal_leidos);

MemoryManager_DumpMemoryLeaks();
system("pause");
return 0;

}
```

3. Ejercicio 2: gestión de una cadena de tiendas de ropa

Realice una aplicación para gestionar una cadena de tiendas de ropa (como por ejemplo Zara, Mango, etc.). La aplicación llevará el control de la ropa de todas las tiendas que forman la cadena, permitiendo realizar cada una de las operaciones ofrecidas a través de un menú. Junto con el enunciado de la práctica se proporcionan todos los ficheros de código:

- main.c
- EntradaDatos.c
- VisualizarDatos.c
- BorradoDatos.c
- cabecera.h

Descarguelos desde el aula virtual a la carpeta donde ha creado el proyecto, mediante los siguientes pasos:

1. Situese con el ratón sobre el nombre del fichero a descargar.
2. Pulse el botón derecho y seleccione “Guardar enlace como”. Aparecerá una ventana que le permitirá elegir la carpeta donde desea descargar los ficheros.
3. Seleccione la carpeta donde ha creado su proyecto y pulse Guardar. Los ficheros se guardarán en la carpeta elegida y podrá añadirlos al proyecto mediante la opción del compilador “Proyecto/Añadir un elemento existente”.

Para la elaboración de este ejercicio se han definido los siguientes tipos de datos que se encuentran definidos en el fichero cabecera:

El tipo enumerado `tColor` define los colores que puede tener una prenda de ropa.

```
typedef enum
{
    blanco, amarillo, naranja, rosa, rojo, verde, azul, morado,
    marrón, negro
}tColor;
```

El tipo enumerado `tTipoPrenda` establece los distintos tipos de prendas que pueden venderse en las tiendas.

```
typedef enum
{
    camiseta, camisa, pantalon, falda, vestido, zapatos, jersey,
    chaqueta
}tTipoPrenda;
```

El tipo de estructura `tPrenda` define los campos que caracterizan a una prenda concreta. Tenga en cuenta que una prenda queda definida de forma única por su referencia y se caracteriza por tener un tipo, color y talla determinados. El número de unidades de la prenda es el número de piezas existentes en una tienda, con la misma referencia y que por tanto, tienen el mismo color, tipo, talla.

```
typedef struct
{
    char        ref[15];        // Referencia única de una prenda. Ej.: BX4432167990L
    char        talla[5];       // Talla. Ej.: XXL, 42, etc.
    tColor       color;         // Color de la prenda. Ej.: negro
    tTipoPrenda tipo;           // Tipo de prenda. Ej.: jersey
    char        h_m;            // Hombre o mujer. Ej.: m
    int          n_uds;          // Número de unidades existentes. Ej.:4
    float        precio;        // Precio de venta. Ej.: 29.99
}tPrenda;
```

El tipo de estructura `tContacto` establece el contacto de las tiendas y de la oficina de la cadena y viene determinado por la dirección postal, el teléfono y la dirección de correo electrónico.

```
typedef struct
{
    char    direccion[50];    // Dirección postal. Ej.:c/ Mayor, 4
    char    tfno[10];        // Teléfono. Ej.: 606454232
    char    e_mail[50];      // Teléfono. Ej.: info@zara.es
}tContacto;
```

El tipo de estructura `tTienda` define una tienda de forma única a través de su código y datos de contacto y permite acceder tanto al array dinámico de las prendas de ropa que la tienda tiene para su venta.

```
typedef struct
{
    int      codigo;          // Código único de identificación de la tienda. Ej.: 1
    tContacto contacto;       // Estructura con dirección, teléfono y correo-e
    int      n_prendas;       // Nº total de prendas existentes para su venta. Ej.: 130
    tPrenda  *p_prendas;     // Puntero al comienzo del array de prendas de la tienda
}tTienda;
```

El tipo de dato `tCadena` proporciona información de una cadena de tiendas.

```
typedef struct
{
    tContacto contacto;       // dirección, teléfono y correo-e de la oficina de gestión
    int      n_tiendas;      // Número total de tiendas
    tTienda  *p_tiendas;     // Puntero al inicio del array dinámico de las tiendas
}tCadena;
```

Observe que uno de los campos de la estructura `tTienda` es un puntero a la estructura `tPrenda`. Esto significa que el puntero `p_prendas` de cada tienda apuntará a un array dinámico que contendrá todas las prendas de la tienda. La figura 11 representa todos los arrays dinámicos de la aplicación y la conexión entre ellos.

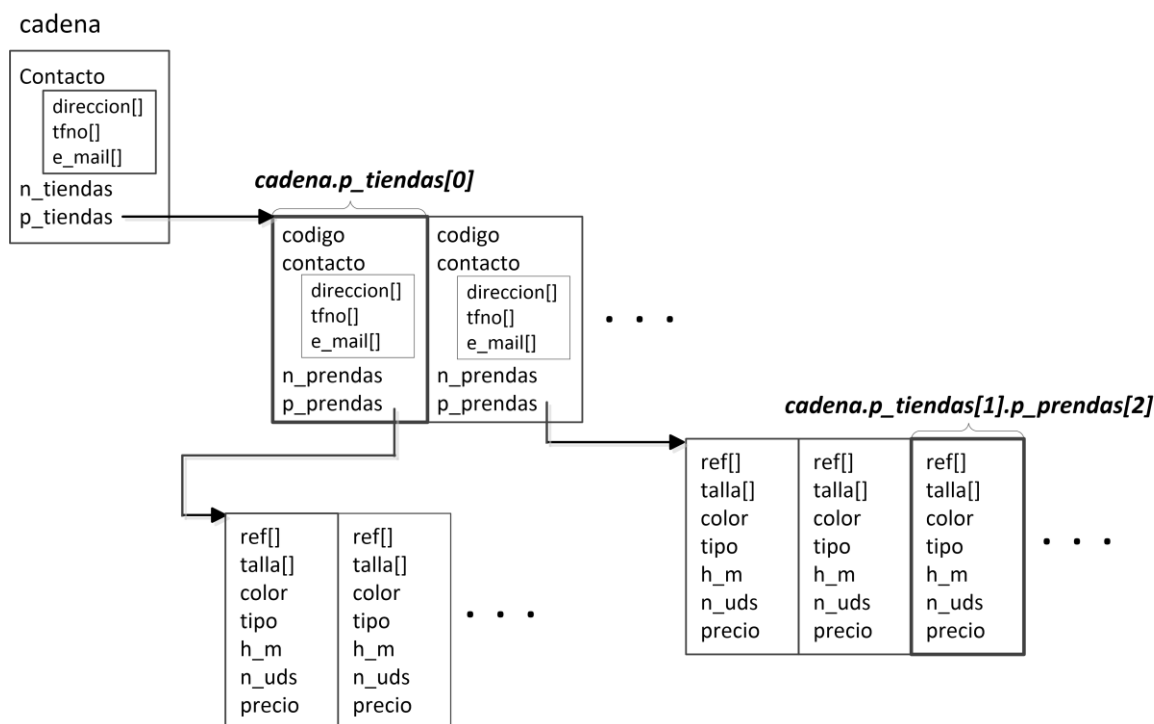


Figura 11. Representación gráfica de los datos de una cadena de tiendas

Analice el código de la función `main()` y tenga en cuenta **los prototipos de las funciones incluidos en el fichero cabecera** para **desarrollar todas las funciones** del programa a excepción de las funciones `Menu()`, `LeeCadena()` y `ErrorMemoria()` que se proporcionan en los ficheros descargados.

La variable `cadena` declarada en `main()` es una estructura que contiene el contacto de la oficina, el número de tiendas que componen la cadena y el puntero al comienzo del array dinámico de tiendas.

Escriba cada función en uno de los ficheros descargados, según le corresponda de acuerdo a su funcionalidad.

La función `LeeCadena()` presenta alguna variación con respecto a su homóloga utilizada en ejercicios anteriores: tiene un nuevo parámetro (puntero a char) donde recibe una parte del mensaje que va a visualizar por pantalla antes de leer la cadena desde teclado. Por ejemplo si se desea leer desde teclado un nombre, la llamada a la función será la siguiente:

```
LeeCadena(nombreaux, "un nombre", 50);
```

donde `nombreaux` es la cadena de caracteres que albergará el nombre leído, `"un nombre"` es una parte del mensaje que la función `LeeCadena()` imprimirá por pantalla y

50 el tamaño máximo de la cadena `nombreaux`. Cuando se ejecute `LeeCadena()` imprimirá el mensaje “Introduzca un nombre” antes de la lectura de datos.

Tenga en cuenta que los mensajes que se deben visualizar por pantalla en caso de **error al reservar memoria** son todos mostrados por la función `ErrorMemoria()`. Por tanto, las funciones que se encarguen de la reserva de memoria se limitarán a retornar el valor -1 en caso de error.

3.1 Opción 1: Crear array de tiendas

Realice la función `CrearTiendas()` que crea el array dinámico de tiendas. La función lee desde teclado el número de tiendas y el contacto de la oficina donde se realiza la gestión y reserva memoria para el array dinámico de tiendas. Retorna -1 si se produce error al reservar memoria ó 0 si no se produce ningún error.

Esta función **no** inicializa los datos del array dinámico y por tanto **no** se debe crear el array dinámico de prendas de las tiendas. Solo debe inicializarse el campo `p_prendas` de las tiendas con el valor `NULL`.

Considere la posibilidad de leer el campo `contacto` mediante una llamada a la función `LeerContacto()`, la cual debe codificar previamente. Consulte su prototipo en el fichero cabecera.

3.2 Opción 2: Inicializar tiendas

Realice la función `IniTiendas()` que inicializa la cadena de tiendas, esto es, lee desde teclado los campos de la estructura `tTienda` para cada una de las tiendas que forman la cadena.

La función `IniTiendas()` debe llamar a las dos funciones siguientes:

1. La función `IniTienda()`, que se encarga de leer desde teclado los campos de **UNA** estructura de tipo `tTienda` que le ha sido pasada por referencia. La función, además, reserva memoria para el array dinámico de prendas de la tienda. Retorna -1 si ocurre un error al reservar memoria ó 0 si todo se ejecuta correctamente.
2. La función `IniPrendasUnaTienda()`, que inicializa todas las prendas de una tienda. A su vez, esta función debe hacer uso de la función `IniPrenda()`, que inicializa **UNA** prenda.

Asimismo, para la lectura del campo `contacto`, la función `IniTienda()` debe llamar a la función `LeerContacto()`, codificada en el apartado anterior.

Debe programar las funciones `Initiendas()`, `IniTienda()`, `IniPrendasUnaTienda()` e `IniPrenda()`.

3.3 Opción 3: Visualizar los datos de las tiendas

Realice la función `VisuCadena()` que visualiza por pantalla los datos de las tiendas, incluídas todas las prendas. Desarrolle la función `VisuTienda()` que debe ser llamada por `VisuCadena()`. A su vez, `VisuTienda()` debe llamar a `VisuPrenda()`.

3.4 Opción 4: Abrir una tienda nueva

Realice la función `AnyadirTienda()` que añade una nueva tienda al array dinámico de tiendas y la inicializa. Retorna -1 si se produce error en la reserva dinámica de memoria ó 0 si no ocurre ningún error. La función `AnyadirTienda()` debe realizar las siguientes tareas:

- Añadir memoria al array dinámico de tiendas para la nueva tienda mediante la función `realloc`.
- Inicializar la tienda añadida utilizando la función `IniTienda()`, codificada en el apartado 3.2.
- Inicializar las prendas de la nueva tienda haciendo uso de la función `IniPrendasUnaTienda()`. A su vez, esta función debe llamar a la función `IniPrenda()`. Ambas funciones fueron codificada en el apartado 3.2.

Como en apartados anteriores, para la lectura del campo `contacto` debe realizar una llamada a la función `LeerContacto()`.

3.5 Opción 5: Eliminar un tipo de prenda de todas las tiendas

Se desea retirar de todas las tiendas un tipo de prenda determinado. Programe la función `EliminarPrendaDeTiendas()` que elimina un tipo de prenda de todas las tiendas. Esta función debe realizar las siguientes tareas:

1. Leer desde teclado del tipo de prenda a eliminar.
2. Recorrer todas las prendas de todas las tiendas y si el tipo de cada prenda coincide con el tipo de prenda leído desde teclado, debe realizar una llamada a la función `EliminaPrenda()`, pasándole el puntero al comienzo de las prendas de la tienda, la posición de la prenda a eliminar y el número de prendas de la tienda.
3. Decrementar el número de prendas de la tienda.
4. Si el número de prendas resultante después del decremento es 0, debe poner a NULL el puntero que apunta a las prendas.

Programe las funciones `EliminarPrendaDeTiendas()` y `EliminaPrenda()`.

Anexo

Instrucciones de uso del MemoryManager

Pasos a seguir para insertar Memory Manager en su programa:

1. Descargue de la página web de la asignatura el fichero `obj` correspondiente a la versión de Microsoft Visual Studio que tenga instalada en su ordenador (`MemoryManager-vcXXXX-d.obj`).
2. Descargue de la página web de la asignatura el fichero `MemoryManager.h`.
3. Copie ambos ficheros en el directorio donde ha creado su proyecto (donde tiene los ficheros de código) y añádalos al proyecto (opción agregar ficheros existentes).
4. Añada la sentencia: `#include "MemoryManager.h"` en el fichero cabecera de su programa.
5. Añada la sentencia: `MemoryManager_DumpMemoryLeaks()`; a la función `main()` de su programa. Sitúela después de liberar la memoria y antes de abandonar el programa. Debe repetir este paso en todos los puntos del código desde donde se de por finalizado el programa.

Ejemplo de uso con Lenguaje C (SSII):

```
#include <stdio.h>
#include <stdlib.h>
#include "MemoryManager.h"

int main()
{
    /* declaraciones de variables */

    /* cuerpo de la función main */

    MemoryManager_DumpMemoryLeaks();

    system("pause");
    return 0;
}
```