Fundamentos de Programación

La memoria dinámica y los punteros

La memoria dinámica y los punteros

- En este modulo se muestra dos conceptos importantes de programación, especialmente cuando se programa en lenguaje C. El objetivo de este material no es dar conceptos teóricos, sino que veáis una visión más aplicada de los conceptos de los apuntes.
  - Seguid las explicaciones para entender los ejemplos que se muestran.
  - Analizad y aseguraros de que habéis entendido los ejemplos de código
    - Podéis escribir los programas, compilarlos y ejecutarlos.
    - Intentad hacer cambios al código para ver si entendéis qué hace.
  - Intentad resolver los ejercicios que se proponen.
    - Comparad vuestra solución con la propuesta.
    - Si no coincide y no sabéis si la vuestra es correcta o no, acudid a vuestro consultor.

- Se utilizan representaciones gráficas que no son estándar. Vosotros podéis definir las vuestras propias, sólo os mostramos una posibilidad.
  - Siempre que se trabaja con estos conceptos es importante hacer un esquema gráfico de los datos.

http://hdl.handle.net/10609/xxx

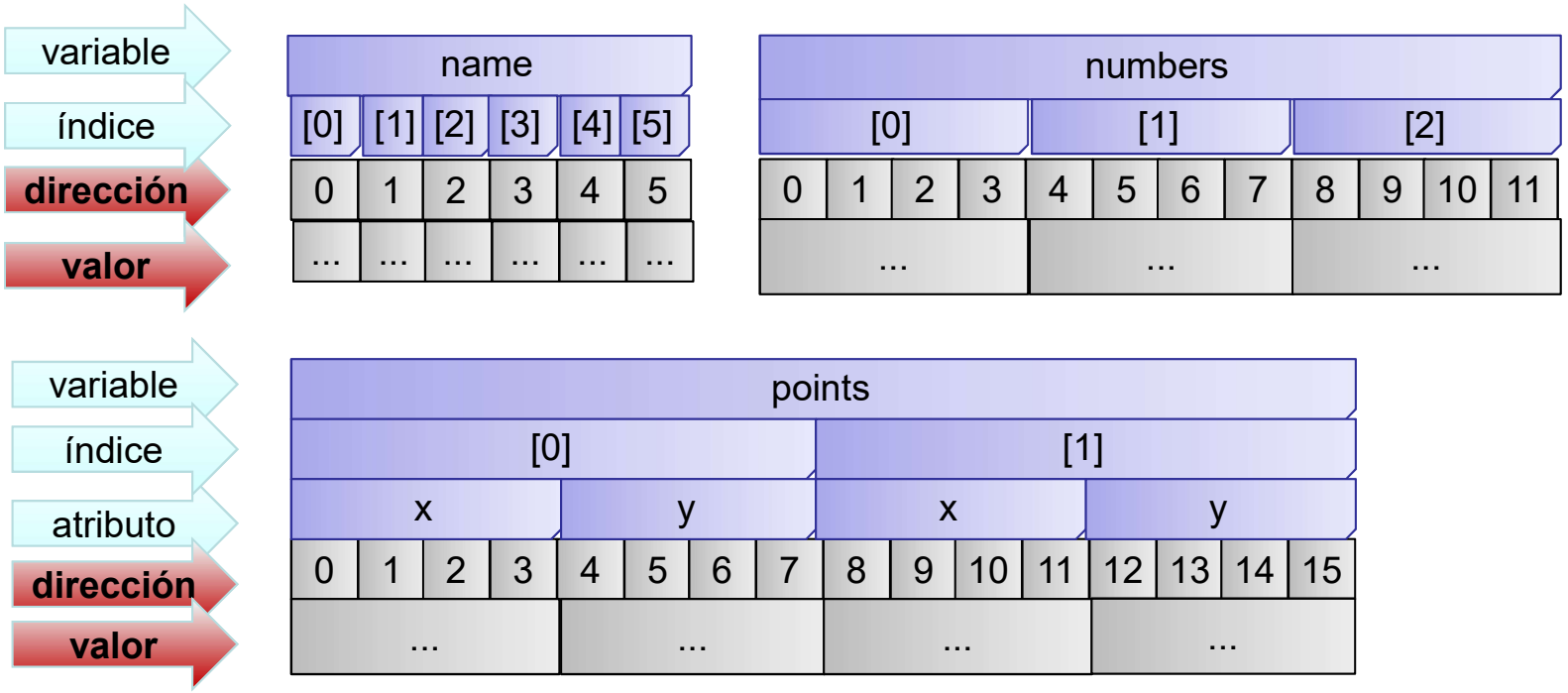
La memoria

- Hasta este momento hemos visto:
  1. Como crear tablas para almacenar secuencias de elementos, tanto elementos de tipos de datos estándar (caracteres, enteros, reales, …) o cualquier tipo de datos definidos por nosotros (coordenada, punto, persona, …):

```
/* Secuencia de  6 caracteres */
char name[6];

/* Secuencia de 3 enteros */
int numbers[3];

/* Secuencia de 2 puntos */
struct {
        int x,y;
} tPoint;

tPoint points[2];
```

La memoria

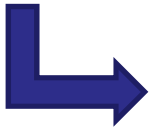
- Hasta este momento hemos visto:
  2. Como quedan organizados los valores de estas secuencias en memoria. Es importante haber entendido que la memoria se organiza por unidades de tamaño byte, y que cada unidad tiene su propia dirección. Nos centraremos principalmente en las direcciones y con qué "nombres" podemos referenciarlas (nombre de variable, índice en un array o atributo en una estructura). Los valores de momento no los consideramos.



La memoria

- Hasta este momento hemos visto:
  3. Como obtener el tamaño en bytes de un tipo de datos, utilizando el comando **sizeof**. Fijaros que en este caso utilizamos el nombre de la variable como parámetro del método **sizeof**. Es el equivalente a poner el tipo correspondiente de la variable. Comprueba que coinciden.

```
char name[6];
int numbers[3];
struct tPoint {
        int x,y;
};
tPoint points[2];

    printf("a) El tamaño de  <%s> es %d bytes\n",“name",sizeof(name));
    printf("b) El tamaño de <%s> es %d bytes\n",“numbers",sizeof(numbers));
    printf("c) El tamaño  de <%s> es %d bytes\n",“points",sizeof(points));
```

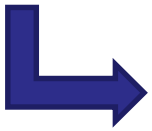


```
a) El tamaño de name es 6 bytes
b) El tamaño de numbers  es 12 bytes
c) El tamaño de points es 16 bytes
```

Trabajando con las direcciones

- En todos los ejemplos que hemos visto hasta ahora, se ha asumido que las direcciones de memoria siempre empiezan desde cero. En realidad nunca se producirá tal situación, es más, veremos que la dirección 0 (cero), conocida también como NULL, es el valor por defecto al inicializar un puntero.
- Por tanto, las direcciones que hemos utilizado hasta ahora, son “relativas” a la dirección inicial.
- Cuando declaramos una variable, el sistema operativo busca un espacio vacío en la memoria, y lo asigna a nuestra variable. Por lo tanto, cada vez que ejecutamos nuestro programa las direcciones de las variables serán distintas. Para obtener una dirección de memoria disponemos del operador **&**. En el código siguiente se muestra un ejemplo. Fijaros que los valores de las direcciones se muestran generalmente en Hexadecimal, y que los casos a y b tienen la misma dirección de memoria.

```
struct tPoint {
        int x,y;
};
tPoint points[2];
    printf("a) La dirección de <%s> es %p \n",“points“,points);
    printf(“b) La dirección de <%s> es %p \n",“points[0]",&(points[0]));
    printf(“c) La dirección de <%s> es %p \n",“points[0].y",&(points[0].y));
    printf(“d) La dirección de <%s> es %p \n",“points[1].y",&(points[1].y));
```



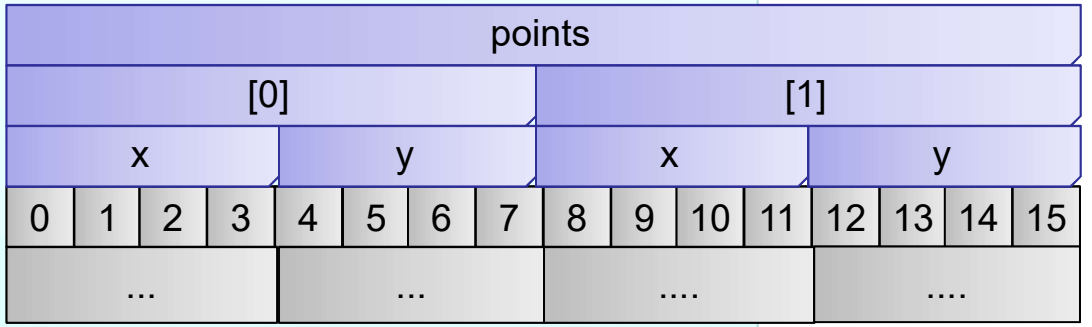
```
a) La dirección de <points> es 0027F7A8
b) La dirección de <points[0]> es 0027F7A8
c) La dirección de <points[0].y> es 0027F7AC
d) La dirección de <points[1].y> es 0027F7B4
```

(*) Los arrays/vectores tienen un significado distinto, por este motivo en el caso de *points*, no se utiliza &. Fijaros que los casos a) y b) tienen la misma dirección. Veremos este caso cuando hablemos de punteros.

Trabajando con las direcciones

- Las direcciones no dejan de ser valores numéricos, por lo tanto, podemos operar con ellos. Fijaros que si ponemos las direcciones relativas a la dirección inicial (&points), obtenemos los valores del esquema:

```
struct tPoint {
        int x,y;
};
tPoint points[2];
int aPoints;
int aPoints0, aPoints0x, aPoints0y;
int aPoints1, aPoints1x, aPoints1y;

aPoints=(int)&points;
aPoints0=(int)&(points[0]);
aPoints0x=(int)&(points[0].x);
aPoints0y=(int)&(points[0].y);
aPoints1=(int)&(points[1]);
aPoints1x=(int)&(points[1].x);
aPoints1y=(int)&(points[1].y);

printf("a) Direcciones relativas: points[0] => %d, points[1] => %d\n",aPoints0-aPoints,aPoints1-aPoints);
printf("b) Direcciones relativas: points[0].x => %d, points[0].y => %d\n",aPoints0x-aPoints,aPoints0y-aPoints);
printf("c) Direcciones relativas: points[1].x => %d, points[1].y => %d\n",aPoints1x-aPoints,aPoints1y-aPoints);
```



a) Direcciones relativas: points[0] => 0, points[1] => 8
b) Direcciones relativas: points[0].x => 0, points[0].y => 4
c) Direcciones relativas: points[1].x => 8, points[1].y => 12

Los punteros

- Hasta ahora hemos visto como se organizan en memoria los distintos tipos de datos. También tenéis ejemplos de código de cómo acceder a las direcciones de memoria. Toda esta información es importante para entender el funcionamiento de un nuevo tipo de variables:

Los punteros

- Los punteros son simplemente variables que en lugar de almacenar un valor, almacenan una dirección de memoria. En este sentido, se dice que un puntero "apunta" a una cierta posición de memoria.
- Los punteros se declaran utilizando un asterisco "*****" delante del nombre de la variable. Por ejemplo, en el siguiente código se declaran "punteros a distintos tipos de datos". Fijaros que la forma de inicializar un puntero es asignándole el valor NULL (que corresponde a la dirección 0):

```
char  *pChar=NULL;
int *pInt=NULL;
float *pFloat=NULL;
```

- Igual que tenemos el operador que nos da la dirección de una variable, también podemos obtener el contenido de una cierta dirección (contenida en un puntero), utilizando el operador de "*desreferenciación*", que es el mismo asterisco "*****". Fijaros en el siguiente código, e intentad entender qué hace:

Pista: Al final, la variable *a* tiene el valor 'b'.

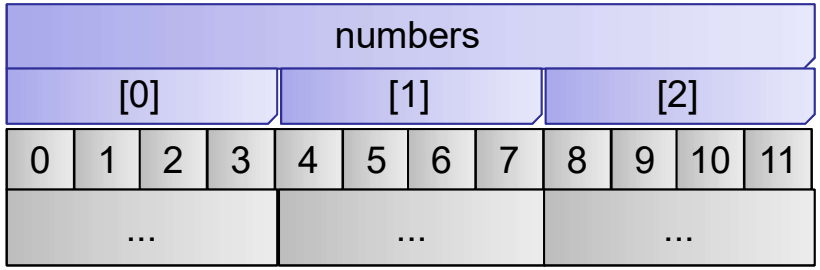
```
char  a='c';
char  *pA=&a;
*pA='b';
```

Nomenclatura: En una expresión del tipo **a=2**, se suele utilizar el nombre ***rvalue*** (*right value*, pronunciado *"are value"*) o objeto para la parte derecha del igual (el valor 2), y ***lvalue*** (*left value*, pronunciado *"el value"*) para la dirección (&a) donde se guarda el valor.

Los punteros

Autoevaluación:

1. A partir de la siguiente representación de memoria, indica cual seria la salida de las siguientes sentencias:



Expresión	Valor
&(numbers[2])	
((int)(numbers)) + 1	
(&(numbers[2])==((int)numbers)+sizeof(int))	
(&(numbers[2])==((int)numbers)+2* sizeof(int))	

2. A partir de los ejemplos de utilización del comando **sizeof**, intenta deducir cual seria la salida del siguiente código:

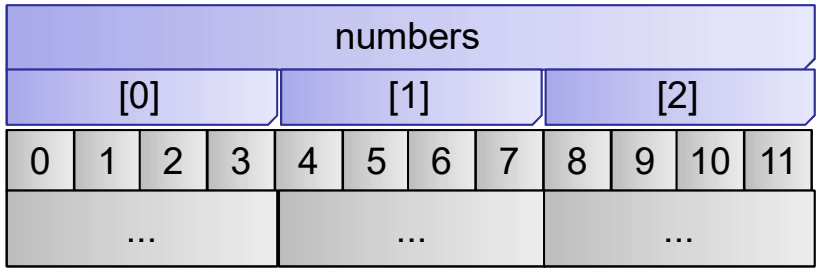
```
struct tPoint {
    int x,y;
};
tPoint points[5];
int vectorI[3] = {1, 2, 3};
char vectorC[5] = "hola";
tPoint *pPoint=points;
char  *pChar=vectorC;
int *pInt=vectorI;

printf("a) [%d,%d,%d]\n",sizeof(points),sizeof(pPoint), sizeof(*pPoint));
printf("b) [%d,%d,%d]\n",sizeof(vectorC),sizeof(pChar), sizeof(*pChar));
printf("c) [%d,%d,%d]\n",sizeof(vectorI),sizeof(pInt), sizeof(*pInt));
```

Caso	Valor		
a			
b			
c			

Los punteros

Autoevaluación(Respuesta):

1. A partir de la siguiente representación de memoria, indica cual seria la salida de las siguientes sentencias.



Expresión	Valor
&(numbers[2])	**8**
((int)(numbers)) + 1	**1**
((int)&(numbers[2])==((int)numbers)+sizeof(int))	**false**
((int)&(numbers[2])==((int)numbers)+2* sizeof(int))	**true**

1) Obtenemos la dirección de la posición 2 del vector numbers, por tanto, la expresión nos retorna la dirección de numbers[2] = 8 (mirar tabla).

2) Primero obtenemos la dirección de numbers, que es 0. La convertimos a un entero, y por lo tanto a partir de este momento, toda operación será como si se tratara de un entero normal y corriente, y por tanto, 0 + 1 = 1.

3) Siguiendo las explicaciones de los casos anteriores, pero en lugar de sumar un valor fijo al valor del caso 1, sumamos el numero de bytes que ocupa un entero. Per lo tanto, obtenemos 0 + 4 = 4, que es distinto a 8, por lo tanto el resultado es false.

*4) Como en el caso anterior, pero ahora sumamos 0 + 2 * 4 = 8, que por lo tanto nos da un resultado true.*

Los punteros

Autoevaluación(Respuesta):

2. A partir de los ejemplos de utilización del comando **sizeof**, intenta deducir cual seria la salida del siguiente código:

```
struct tPoint {
    int x,y;
};
tPoint points[5];
int vectorI[3] = {1, 2, 3};
char vectorC[5] = "hola";
tPoint *pPoint=points;
char  *pChar=vectorC;
int *pInt=vectorI;

printf("a) [%d,%d,%d]\n",sizeof(points),sizeof(pPoint), sizeof(*pPoint));
printf("b) [%d,%d,%d]\n",sizeof(vectorC),sizeof(pChar), sizeof(*pChar));
printf("c) [%d,%d,%d]\n",sizeof(vectorI),sizeof(pInt), sizeof(*pInt));
```

Caso	Valor		
a	40	4	8
b	5	4	1
c	12	4	4

- *El primer valor de cada caso es el tamaño de un array de un determinado tipo. Por ejemplo en el caso a, queremos el tamaño del tipo "tPoint[5]", que se obtiene con:*

  *5 * sizeof(tPoint) = 5 * (2 * sizeof(int)) = 5 * (2 * 4) =* ***40***

  El resto de casos son más simples:

  *sizeof(char[5])= 5 * sizeof(char) = 5*1 =* ***5***

  *sizeof(int[3]) = 3 * sizeof(int) = 3*4 =****12***

Los punteros

Autoevaluación(Respuesta):

2. A partir de los ejemplos de utilización del comando **sizeof**, intenta deducir cual seria la salida del siguiente código:

```
struct tPoint {
    int x,y;
};
tPoint points[5];
int vectorI[3] = {1, 2, 3};
char vectorC[5] = "hola";
tPoint *pPoint=points;
char  *pChar=vectorC;
int *pInt=vectorI;

printf("a) [%d,%d,%d]\n",sizeof(points),sizeof(pPoint), sizeof(*pPoint));
printf("b) [%d,%d,%d]\n",sizeof(vectorC),sizeof(pChar), sizeof(*pChar));
printf("c) [%d,%d,%d]\n",sizeof(vectorI),sizeof(pInt), sizeof(*pInt));
```

Cas	Valor		
a	40	4	8
b	5	4	1
c	12	4	4

- *El segundo valor es el tamaño del puntero. Recordad que un puntero almacena una dirección de memoria, por lo tanto, su tamaño será lo que ocupe una dirección de memoria, independientemente del tipo al que haga referencia. El valor dependerá del sistema en que trabajéis:*
  - *En un sistema de 32bits, las direcciones se guardan en 32bits => **4** bytes.*
  - *En un sistema de 64bits, las direcciones se guardan en 64bits => **8** bytes.*

Curiosidad: Si os fijáis en el tamaño de las direcciones en cada sistema, veréis que en un sistema de 32bits, el numero máximo de direcciones distintas será 2^{32}, que corresponde a 4.294.967.296 (≈4Gb). Este es el motivo por el cual estos sistemas están limitados a un máximo de 4Gb de RAM.

Los punteros

Autoevaluación(Respuesta):

2. A partir de los ejemplos de utilización del comando **sizeof**, intenta deducir cual seria la salida del siguiente código:

```
struct tPoint {
    int x,y;
};
tPoint points[5];
int vectorI[3] = {1, 2, 3};
char vectorC[5] = "hola";
tPoint *pPoint=points;
char  *pChar=vectorC;
int *pInt=vectorI;

printf("a) [%d,%d,%d]\n",sizeof(points),sizeof(pPoint), sizeof(*pPoint));
printf("b) [%d,%d,%d]\n",sizeof(vectorC),sizeof(pChar), sizeof(*pChar));
printf("c) [%d,%d,%d]\n",sizeof(vectorI),sizeof(pInt), sizeof(*pInt));
```

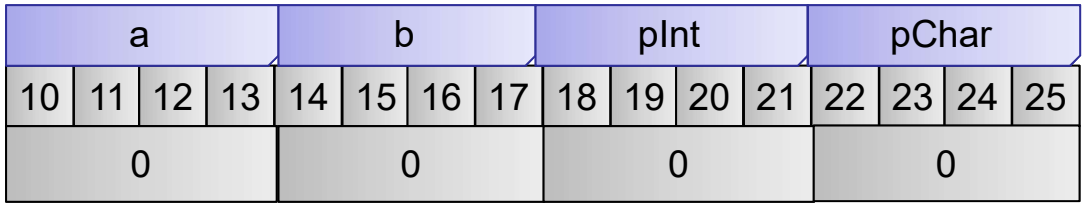
Cas	Valor		
a	40	4	8
b	5	4	1
c	12	4	4

- *El tercer y último valor de cada caso es el tamaño del "contenido" del puntero. En este punto es donde afecta el tipo de puntero que utilizamos, ya que cada puntero "contiene" un valor de su tipo. Los valores corresponden a:*
  - *a) sizeof(*pPoint) = sizeof(*(tPoint*)) = sizeof(tPoint) = 2 * sizeof(int) = 2 * 4 = **8***
  - *b) sizeof(*pChar) = sizeof(*(char*))= sizeof(char) = **1***
  - *c) sizeof(*pInt) = sizeof(*(int*))= sizeof(int) = **4***

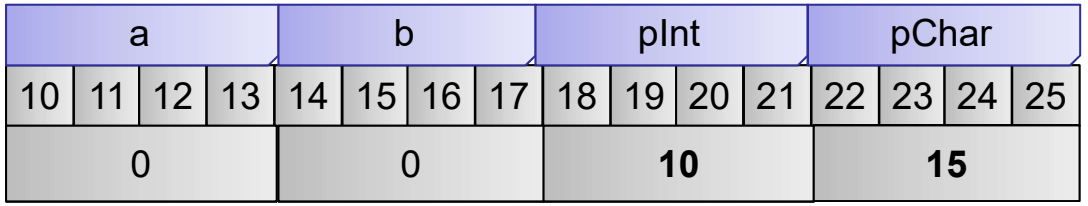
Los punteros

- A continuación se muestra un código simple del uso de punteros, con la evolución de su representación en memoria.

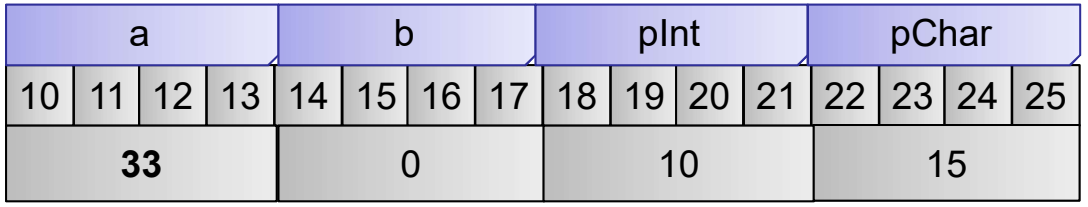
```
int a=0;
int b=0;
char  *pChar=NULL;
int *pInt=NULL;
```



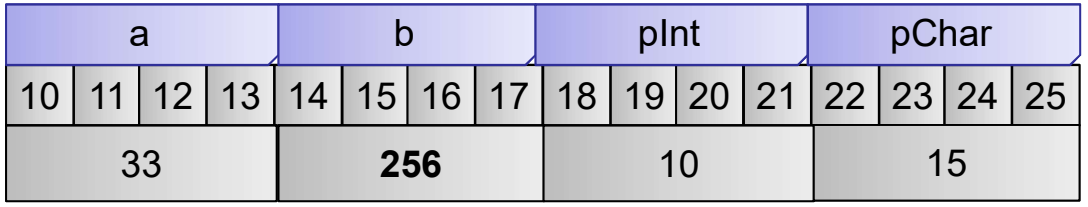

```
pInt=&a;
pChar=((char*)&b)+1;
```




```
*pInt=33;
```

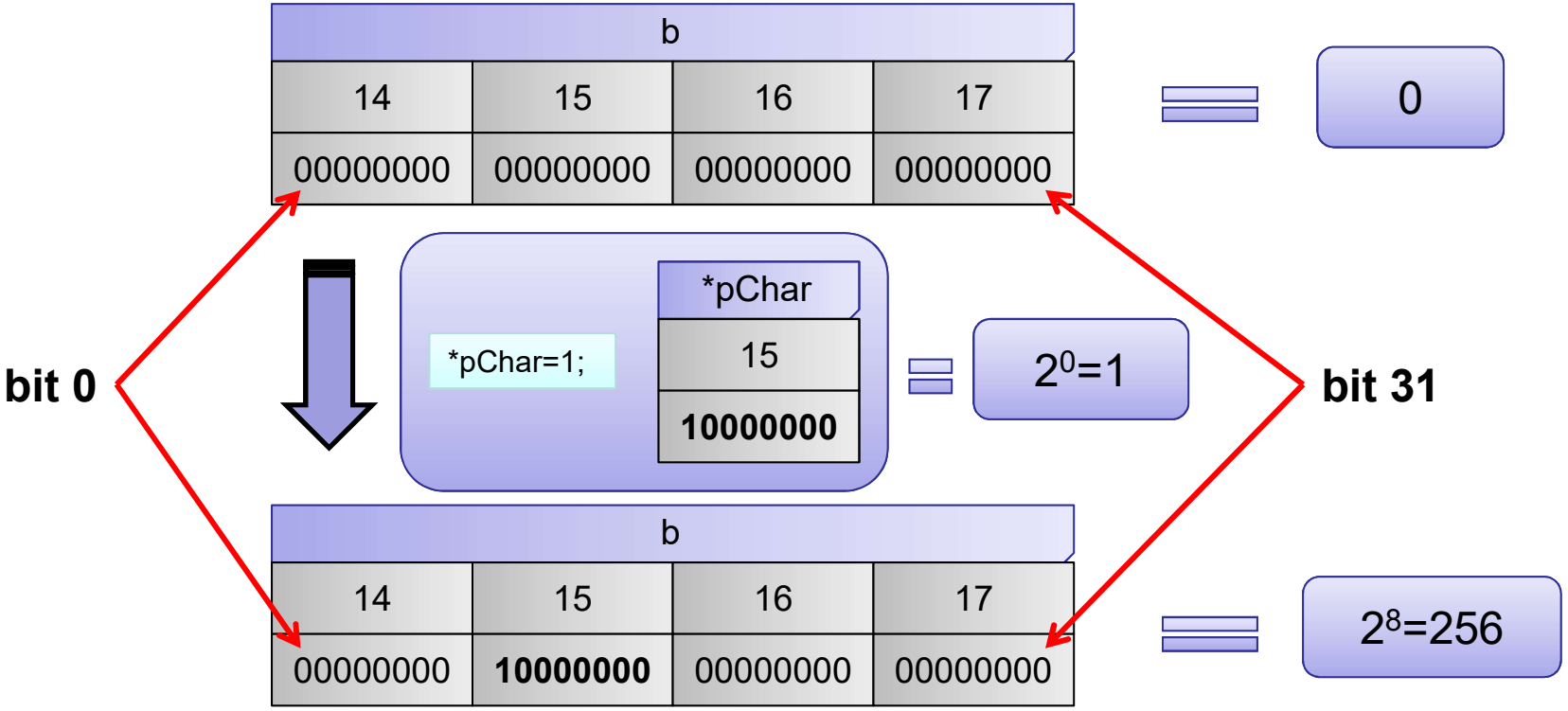



```
*pChar=1;
```



Los punteros

- Fijaros en el ultimo caso. El puntero pChar está “apuntando” a la posición de memoria 15, que corresponde al segundo byte del entero b. Si miramos nivel de bit, la variable b corresponde a la siguiente representación estándar (complemento a 2 con el bit más significativo a la derecha):



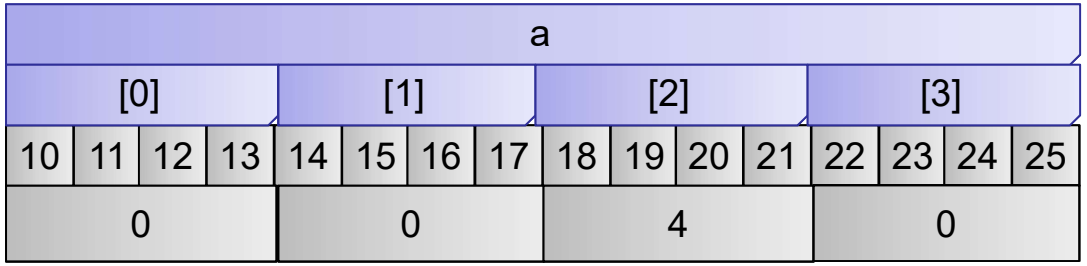
Nota: Aunque no pedimos que conozcáis los formatos de los valores, es muy útil saber que existen. Lo mas importante es tener presente que cuando se trabaja con punteros, podemos modificar valores directamente en la memoria, lo que nos confiere un gran control sobre los datos, pero también añade un nivel de complejidad superior. Tened claro dónde apuntan vuestros punteros.

Los punteros

- Una de las grandes utilidades de los punteros es poder recorrer secuencias.
  - **Indexación:** Permite acceder al contenido de la **N-ésima** unidad a partir de una dirección inicial. Cada unidad corresponde al tamaño del tipo de datos al que apunta el puntero. En el ejemplo siguiente, pInt contiene el valor de la dirección de a (10), pero en utilizar la indexación pInt[2], lo que estemos accediendo es el **contenido** de la dirección [10 + sizeof(int) * (2)] . Se muestran dos alternativas:

```
int a[4]={0,0,0,0};
int *pInt=(int*)((int)a+2*sizeof(int));

*pInt=4;
```

```
int a[4]={0,0,0,0};
int *pInt=a;

pInt[2]=4;
```



  - **Incremento/decremento:** Avanzamos o retrocedemos unidades, correspondientes al tamaño del tipo al que apunta el puntero. Cuando se suma o resta directamente un valor entero a un puntero, lo estamos desplazando por unidades de su tipo. El ejemplo anterior se podría haber escrito de las siguientes formas:

```
int a[4]={0,0,0,0};
int *pInt=a+2;
*pInt=4;
```

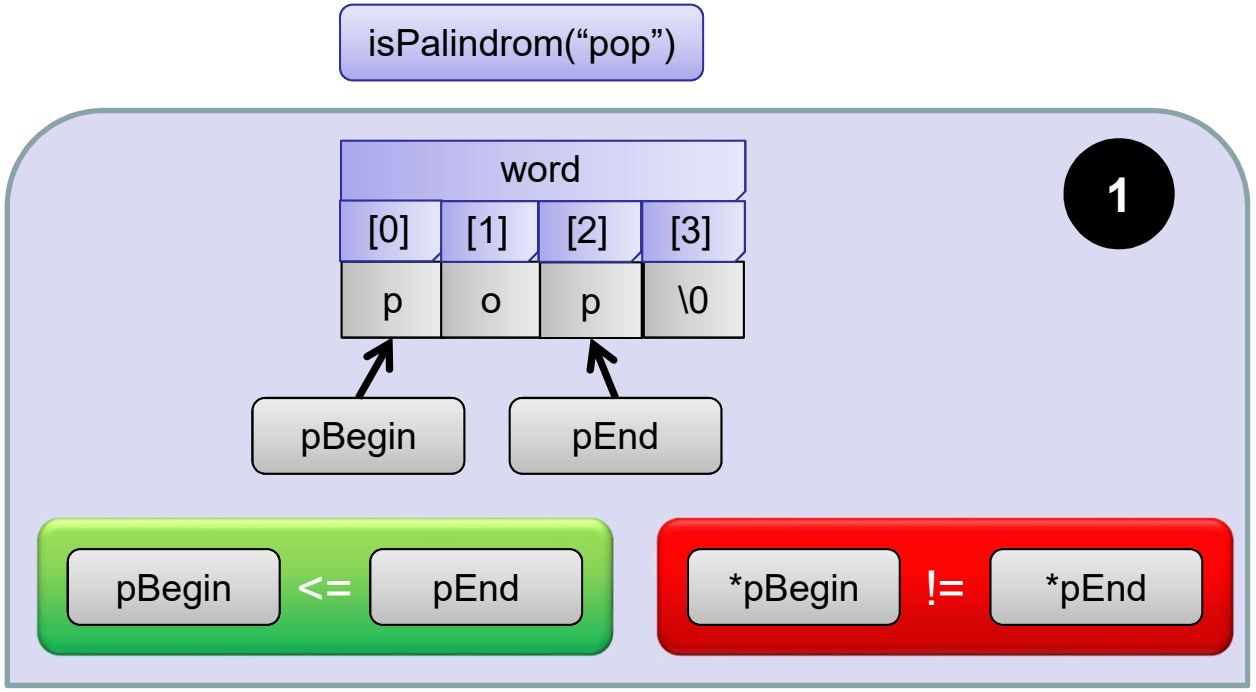
```
int a[4]={0,0,0,0};
int *pInt=a;
*(pInt+2)=4;
```

```
int a[4]={0,0,0,0};
int *pInt=a;
pInt++;
pInt++;
*pInt=4
```

Los punteros

- Una de las grandes utilidades de los punteros es poder recorrer secuencias.
  - **Ejemplo:** A continuación se muestra un ejemplo de punteros para saber si una secuencia de caracteres es palíndroma. También un ejemplo gráfico de como funcionaría con una palabra corta. Fijaros en una posible representación grafica de los punteros, como nodos que "apuntan" a un elemento determinado, y donde generalmente no se utilizan las direcciones, ya que como hemos visto, estas van cambiando y se pueden obtener a partir de los nombres de variables.

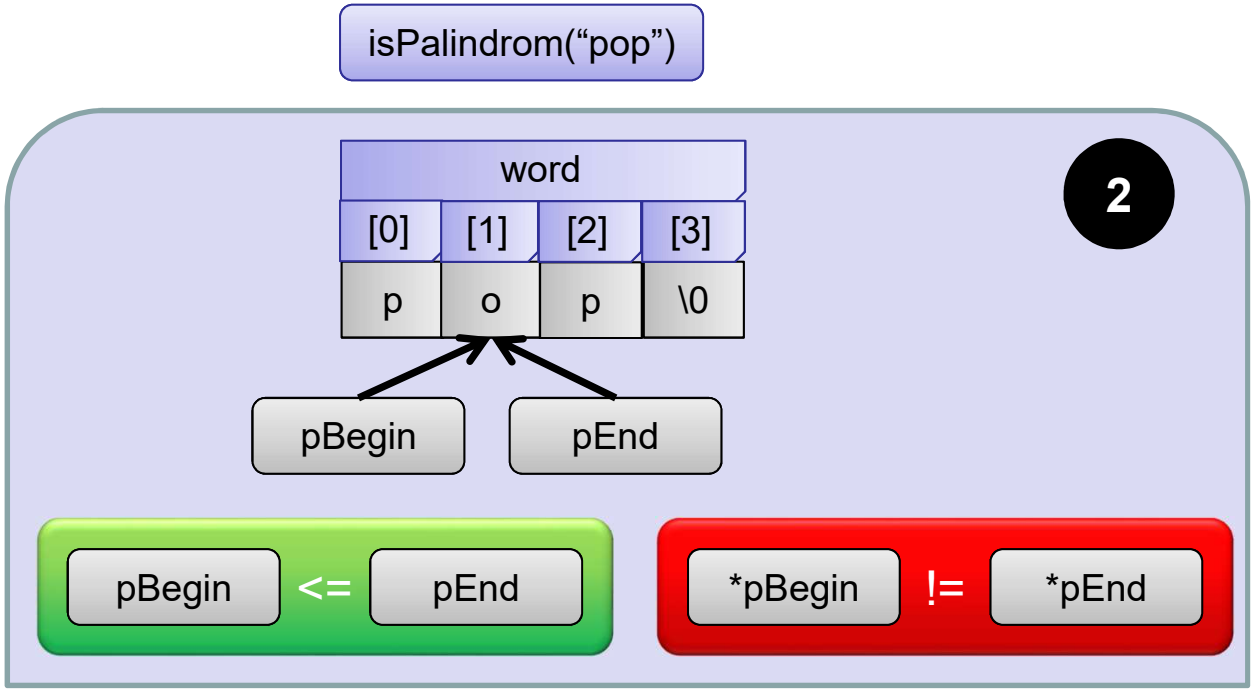
```
bool isPalindrom(char word[]) {
        char *pBegin=NULL;
        char *pEnd=NULL;

        pBegin=word;
        pEnd=&(word[strlen(word)-1]);
        while(pBegin<=pEnd) {
          if(*pBegin!=*pEnd) return false;
              pBegin++;
              pEnd--;
          }
          return true;
}
```



Los punteros

- Una de las grandes utilidades de los punteros es poder recorrer secuencias.
  - **Ejemplo:** A continuación se muestra un ejemplo de punteros para saber si una secuencia de caracteres es palíndroma. También un ejemplo gráfico de como funcionaría con una palabra corta. Fijaros en una posible representación grafica de los punteros, como nodos que "apuntan" a un elemento determinado, y donde generalmente no se utilizan las direcciones, ya que como hemos visto, estas van cambiando y se pueden obtener a partir de los nombres de variables.

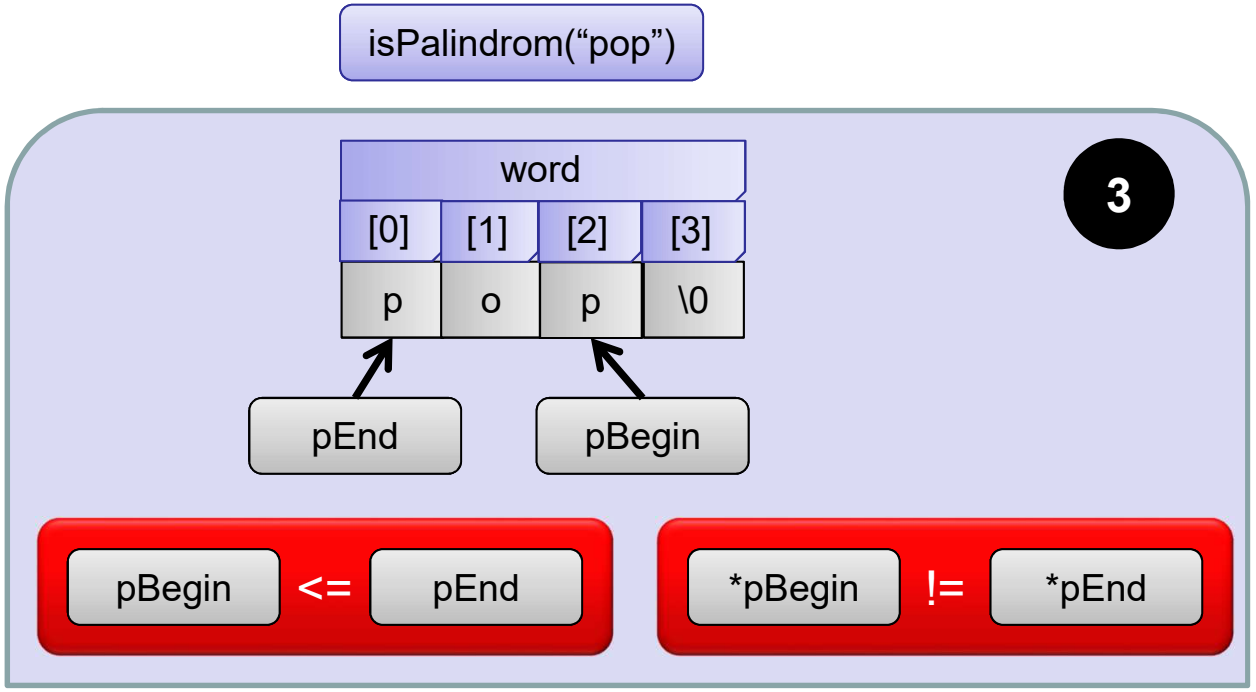
```
bool isPalindrom(char word[]) {
        char *pBegin=NULL;
        char *pEnd=NULL;

        pBegin=word;
        pEnd=&(word[strlen(word)-1]);
        while(pBegin<=pEnd) {
          if(*pBegin!=*pEnd) return false;
               pBegin++;
               pEnd--;
          }
          return true;
}
```



Los punteros

- Una de las grandes utilidades de los punteros es poder recorrer secuencias.
  - **Ejemplo:** A continuación se muestra un ejemplo de punteros para saber si una secuencia de caracteres es palíndroma. También un ejemplo gráfico de como funcionaría con una palabra corta. Fijaros en una posible representación grafica de los punteros, como nodos que "apuntan" a un elemento determinado, y donde generalmente no se utilizan las direcciones, ya que como hemos visto, estas van cambiando y se pueden obtener a partir de los nombres de variables.

```
bool isPalindrom(char word[]) {
        char *pBegin=NULL;
        char *pEnd=NULL;

        pBegin=word;
        pEnd=&(word[strlen(word)-1]);
        while(pBegin<=pEnd) {
          if(*pBegin!=*pEnd) return false;
              pBegin++;
              pEnd--;
          }
          return true;
}
```



Los punteros

- Para acabar este apartado, recordad que los punteros son simplemente una variable que en lugar de contener un valor, contiene una dirección de memoria. Algunos de los usos mas comunes de los punteros están relacionados con:
  - ***Paso de parámetros por referencia:*** Son aquellos parámetros que pueden sufrir modificaciones dentro de un método. Por lo tanto, cuando llamamos a una acción/función con un parámetro de salida o de entrada/salida, utilizamos punteros.
  - ***Recorridos y búsquedas:*** Cuando se hacen recorridos y búsquedas en estructuras complejas, generalmente se utilizan punteros.
  - ***Retorno de estructuras en funciones:*** La cantidad de memoria que puede retornar una función es limitada. Para solucionar el problema, una práctica habitual es retornar un puntero a la estructura, en lugar de la estructura en sí.
  - ***Memoria dinámica:*** Zonas de memoria que es reservan durante la ejecución del programa, generalmente porque no conocemos el tamaño necesario en el momento de programar. Este tema se trata a continuación.

- Existe un tipo especial de puntero (**void***), que es considera un puntero genérico. No se puede utilizar éste puntero para hacer aritmética, ya que no tienen un tipo asociado, pero se pueden convertir a cualquier con un cast a cualquier tipo de puntero. Se utilizan sobretodo para definir estructuras de datos genéricas o en funciones que trabajan con memoria. Veremos algún ejemplo más adelante.

Memoria dinámica

- La memoria dinámica es esa memoria que se reserva en tiempo de ejecución. Generalmente se utiliza cuando durante el desarrollo de nuestra aplicación no conocemos cuanta memoria necesitaremos.
- Cuando utilizamos memoria dinámica, se debe tener en cuenta que hay dos procesos implicados:
  - **Reserva:** Pedir al sistema operativo que nos conceda un espacio de memoria para guarda en él nuestros datos. Se debe verificar que realmente existe este espacio.
  - **Liberación:** Le comunicamos al sistema operativo que nuestro programa ya ha acabado de utilizar un determinado espacio de memoria y que por lo tanto se lo devolvemos. A partir de ese momento ya no lo podemos volver a utilizar.
- Es muy importante tener en cuenta que la memoria es un recurso **finito y limitado**.

Memoria dinámica

Reservar memoria:

- La reserva de memoria se hace mediante el comando **malloc.** Necesitaréis hacer la inclusión de la librería stdlib.h (**#include** <stdlib.h>)

```
void *malloc(size_t size);
```

- A este comando se le pasa **el número de bytes** que se necesitan. El sistema busca una zona de memoria donde esté disponible esta cantidad de bytes libres **consecutivos** y nos retorna **la dirección de la primera posición.**
  - Utilizad el comando *sizeof* para obtener los tamaños de los tipos básicos
- En el caso de que no haya ninguna zona con esta cantidad de memoria, nos retornara un NULL.
  - Se debe comprobar **siempre** que se ha obtenido la memoria pedida.
- A continuación se muestra una tabla con equivalencias entre la definición estática de memoria (de los ejemplos anteriores) con la forma de hacerlo con memoria dinámica (se muestra tanto la declaración como la reserva de memoria).

Estática	Dinámica	
char name[25];	**char*** name = NULL;	name = (**char***) malloc(25 * sizeof(**char**));
int vectorInt[15];	**int*** vectorInt = NULL;	vectorInt = (**int***) malloc(15 * sizeof(**int**));
tPoint points[2];	**tPoint*** points = NULL;	points = (**tPoint***) malloc(2 * sizeof(**tPoint**));

Memoria dinámica

Liberar memoria:

- Toda la memoria reservada con un **malloc** se debe liberar tan pronto como sea posible. El comando para liberar memoria es **free**.

```
void free(void*ptr);
```

- A este comando se le paso **la dirección a la primera posición**. El sistema liberará esta memoria, pudiendo volver a asignarla cuando se pida mas memoria dinámica.
- Se debe tener en cuenta los siguientes puntos:
  - Cuando un programa termina, tota la memoria que ha reservado se libera automáticamente.
  - Cuando un método termina, **NO** se libera la memoria dinámica reservada, pero si que se liberan los punteros que se hayan declarado, por lo tanto podemos perder la dirección a la memoria pedida. Las zonas de memoria reservadas y no liberadas se conocen con el nombre de **memory leaks**, y dado que no tenemos su dirección, no las podremos liberar hasta que acabe la aplicación.

Memoria dinámica

Trabajar con memoria dinámica:

- Cuando reservamos memoria dinámica, no tenemos ningún conocimiento del contenido de la zona de memoria obtenida. Generalmente se inicializa cada estructura según los valores por defecto que sean necesarios, pero existen métodos para asignar valores a nivel de bytes. Por ejemplo, el comando **memset** asigna el valor **value** a los **n** primeros bytes a partir de la posición apuntada por **ptr**.

```
void* memset(void* ptr, int value, size_t n);
```

- Fijaros en los ejemplos anteriores, que la memoria reservada se asigna a un puntero. Por lo tanto, podemos utilizar toda la aritmética de punteros y sus propiedades para trabajar con esta memoria, igual que lo hacíamos en el caso de la memoria estática.

Memoria dinámica

Autoevaluación:

1. Escribe un método, que lea una secuencia de enteros de teclado, la guarde en memoria y nos retorne la dirección donde la ha guardado. Se debe tener en cuenta que el primer elemento de la secuencia de entrada será el numero de valores que contiene la secuencia. El método también tendrá un parámetro de salida donde retornará el numero de elementos de la secuencia. O sea, la cabecera del método será:

```
int* readIntSequence(int *length);
```

el formato de la secuencia de entrada será:

y los datos en memoria deberían quedar como (sin el tamaño):

Escribid también el programa principal que llame a este método, e invierta el orden de sus elementos. Tened presente que se debe utilizar correctamente los métodos para reservar y liberar la memoria.

Memoria dinámica

Autoevaluación (Respuesta):

Lo primero que haremos es escribir el método auxiliar. Fijaros especialmente en la parte de reservar memoria y como se accede a los elementos:

```
int* readIntSequence(int *length) {
        int* vector=NULL;
        int N=0;
        int i=0;

        /* Read the number of elements */
        scanf("%d",&N);
        /* Allocate the memory */
        vector=(int*) malloc(N * sizeof(int));
        /* Read the elements */
        for(i=0;i<N;i++) {
                /* Read the element*/
                scanf("%d",&(vector[i]));
        }
        /* Set the output parameter */
        *length=N;
        /* Return the memory location */
        return vector;
}
```

Memoria dinámica

Autoevaluación (Respuesta):

Si os fijáis en la solución propuesta, y la probáis, veréis que funciona correctamente. Aún y así, ***no es una solución correcta****. Nos hemos olvidado de un paso clave cuando se trabaja con memoria dinámica, comprobar que se nos ha concedido la memoria que hemos pedido. Por este motivo, se debe añadir la comprobación de que la dirección devuelta por el comando malloc no es NULL.*

También se debe decidir cómo se gestiona este hecho. En este caso, lo que haremos es retornar una longitud de -1, y un puntero a NULL. De este modo, cuando un método utilice esta función, podrá saber si se ha producido algún error de memoria.

La reserva de memoria quedaría así:

```
int* readIntSequence(int *length) {
        …
        /* Allocate the memory */
        vector=(int*) malloc(N * sizeof(int));
        if(vector==NULL) {
                *length=-1;
                 return NULL;
        }
        …
}
```

Memoria dinámica

Autoevaluación (Respuesta):

Vamos a definir ahora el programa principal, el que utilizará el método auxiliar definido anteriormente, e invierta el orden de los elementos. Definimos la parte de obtención de los datos. Fijaros en que declaramos un entero para el tamaño, y un puntero para la secuencia. Llamamos al método auxiliar para leer el vector de valores, controlando el caso en que no se ha podido obtener la memoria necesaria para guardar la secuencia. Finalmente, antes de salir liberamos la memoria utilizada:

```
int main(void) {
        int N=0;
        int *pSeq=NULL;

        /* Read the sequence */
        pSeq=readIntSequence(&N);
        if(N<0) {
                /* Show error message*/
                printf("Not enough memory\n");
                return EXIT_FAILURE;
        }
        /* Invert Sequence */
        .....
        /* Free used memory */
        free(pSeq);
        /* End the program*/
        return EXIT_SUCCESS;
}
```

Memoria dinámica

Autoevaluación (Respuesta):

Finalmente, invertiremos el orden de los elementos de la secuencia, utilizando punteros, de forma similar a como verificábamos si una palabra era palíndroma o no. Fijaros en que necesitamos una variable auxiliar.

```
int main(void) {
        ...
        int *pBegin=NULL,*pEnd=NULL;
        int auxValue;

        /* Read the sequence */
        ...
        /* Invert Sequence */
        if(N>0) {
            /* Point to the first and last elements */
            pBegin=pSeq;
            pEnd=&(pSeq[N-1]);
            /* Invert all elements */
            while(pBegin<pEnd) {
                /* Swap the elements */
                auxValue=*pBegin;
                *pBegin=*pEnd;
                *pEnd=auxValue;
                /* Move pointers */
                pBegin++;
                pEnd--;
            }
        }
        /* Free used memory */
        ...
}
```

Memoria dinámica

Autoevaluación (Respuesta):

Para probar que todo funciona correctamente, podéis definir un método para visualizar la secuencia. Por ejemplo:

```
void showSeq(int *vector,int N) {
        int i=0;
        printf("[");
        for(i=0;i<N;i++) {
              printf(" %d",vector[i]);
        }
        printf(" ]\n");
}
```

Memoria dinámica

Otros conceptos:

- A parte del **malloc**, existe otro método, llamado **realloc**, que permite aumentar el tamaño de una zona de memoria reservada previamente (apuntada por ***ptr***), moviendo los datos a otra zona de la memoria se es necesario. Devuelve el puntero a la nueva zona de memoria.

```
void* realloc(void* ptr, size_t newSize);
```

- También disponemos del método **calloc**, que funciona de forma similar al **malloc**, pero inicializa tota la zona de memoria reservada a cero. En este caso se le pasa por separado el nombre de elementos y el tamaño de cada elemento, pero el resultado sigue siendo una zona contigua de memoria de tamaño (num*size) bytes.

```
void* calloc( size_t num, size_t size );
```

Memoria dinámica

Errores típicos:

- **Utilizar un puntero no inicializado**. Esto puede pasar en parámetros de entrada/salida o de salida, o si no se ha inicializado el valor de un puntero. Se debe tener en cuenta que si al puntero no se le asigna el valor NULL, este puede contener cualquier valor, por lo tanto, a efectos prácticos contiene un valor no NULL y por lo tanto se supone que apunta a una zona correcta de memoria. Si lo utilizamos, podemos cambiar el contenido de una zona de memoria desconocida, perdiendo datos o provocando que el sistema falle. Por suerte, el sistema operativo nos protege de poder dañar al resto de aplicaciones, pero nuestra aplicación no queda protegida.
- **Perder la referencia a una zona de memoria**, perdiendo la posibilidad de poder liberarla. En este caso, al salir del método se elimina el puntero **ptr**, pero no la memoria reservada, y por lo tanto no hay forma de poder liberarla, ya que hemos perdido la dirección en que se encuentra. Lo mismo pasa si modificamos la dirección de **ptr** (por simplicidad se ha obviado la comprobación de la memoria).

```
void f1(int N) {
        int *ptr=NULL;
        int i=0;
        ptr=malloc(N * sizeof(int));
        for(i=0;i<N;i++) {
                scanf("%d",&(ptr[i]));
        }
        / * Use values in the sequence */
        return;
}
```

```
void f2(int N) {
        int *ptr=NULL;
        int i=0;
        ptr=malloc(N * sizeof(int));
        for(i=0;i<N;i++) {
                scanf("%d",ptr);
                ptr++;
        }
        / * Use values in the sequence */
        return;
}
```

Punteros a estructuras

- Un caso particular del uso de punteros, se da cuando este apunta a una estructura. En estos casos, podemos utilizar el operador **"->"** para acceder directamente a sus campos.

Por ejemplo, recordad la definición del tipo tPoint:

```
struct  tPoint {
      int x;
      int y;
}
```

Si tenemos un puntero a una estructura de este tipo:

```
tPoint* point=(tPoint*)malloc(sizeof(tPoint));
```

Tenemos dos posibilidades de acceder a sus atributos (x e y), la primera es obtener el contenido del puntero, y por consiguiente, una estructura de tipo tPoint y acceder a ella directamente (fíjate que una posibilidad es acceder como si fuera un vector de un solo elemento):

```
(*point).x=10;
point[0].y=15;
```

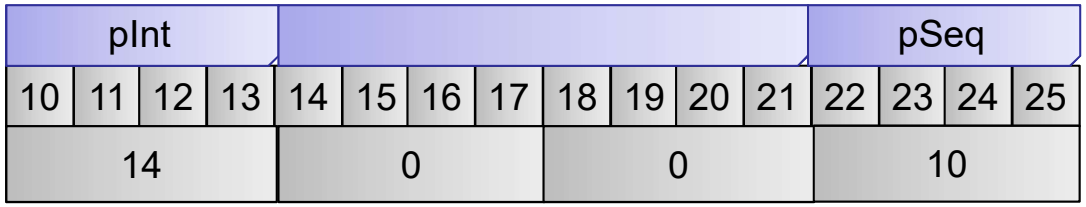
Otra opción es acceder a los atributos con el operador **"->"**:

```
point->x=10;
point->y=15;
```

Punteros a punteros

- Una vez vistos los punteros y cómo se gestiona la memoria dinámica, ya estamos preparados para ver un caso particular de punteros: Los punteros a punteros.
- Fijaros en el siguiente código y en su representación gráfica:

```
int *pInt=NULL;
int **pSeq=NULL;

*pSeq=pInt;
pInt=calloc(2,sizeof(int));
```



- Fijaros con que el contenido de un puntero a puntero, no es un valor, sino un puntero, por lo tanto una dirección de memoria.

Punteros a punteros

- Un ejemplo claro del uso de punteros a punteros son las secuencias de caracteres. Imaginad que queremos guardar los nombres y las edades de un conjunto de personas, obteniendo estos valores por teclado. Si seguimos el ejemplo proporcionado anteriormente:
  - Las edades las podemos guardar como una secuencia de enteros, per tanto, si tenemos **N**, haríamos:

```
int *pAges=NULL;
pAges=(int*)malloc(N*sizeof(int));
```

  - En el caso de los nombres, hemos visto anteriormente, que para guardar una cadena de longitud **L**, lo podemos hacer:

```
char*pName=NULL;

pName=(char*)malloc(L*sizeof(char));
```

  - Si combinamos los dos ejemplos anteriores, podemos ver que si queremos guardar una secuencia de **N** cadenas de caracteres de longitud $\mathbf{L_i}$ (donde $\mathbf{L_i}$ es la longitud del i-ésimo nombre), haríamos:

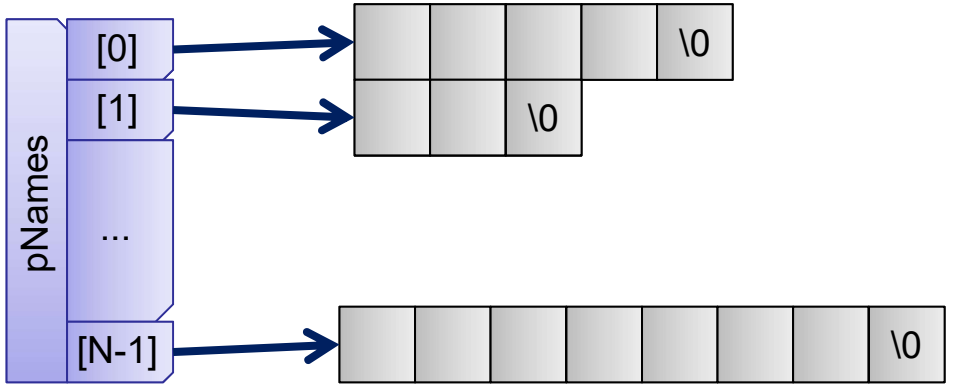
```
char**pNames=NULL;
pNames=(char**)malloc(N*sizeof(char*));
```

  - Para reservar la cadena de caracteres donde guardamos el i-ésimo nombre, I haríamos así:

```
pNames[i]=(char*) malloc(Li * sizeof(char));
```

Punteros a punteros

- Gráficamente lo podemos representar como una lista de punteros, donde cada puntero apunta a una cadena de caracteres (para seguir el formato estándar, todas acabadas con el carácter '\0'). Fijaros que hemos eliminado las direcciones de memoria del esquema.



- Para liberar esta estructura es imprescindible liberar cada cadena de caracteres antes de liberar la lista de punteros. Por lo tanto se debe hacer:

```
/* Free elements */
for(i=0;i<N;i++) {
        free(pNames[i]);
}
/* Free list of pointers*/
free(pNames);
```

Punteros a punteros

- Otro caso típico es la inicialización de memoria dinámica dentro de un método. Por ejemplo, queremos un método para obtener una secuencia de N valores aleatorios. Fíjate en el siguiente ejemplo:
  1. Si el vector ya existe (es distinto a NULL), liberamos la memoria (no sabemos su longitud).
  2. Reservamos memoria para la nueva lista de valores.
  3. Llenamos la lista de valores con números aleatorios.

```
void getRandVec(int N, int* vector) {
        int i=0;
        if(vector!=NULL) {
                free(vector);
        }
        vector=(int*)malloc(N*sizeof(int));
         for(i=0;i<N;i++) {
                 vector[i]=rand();
         }
}
```

- Si probáis este código, veréis que no funciona. Además, es un ejemplo de ***memory leak***, ya que se pierde la referencia a la zona de memoria reservada. Vamos a ver qué esta pasando:

Punteros a punteros

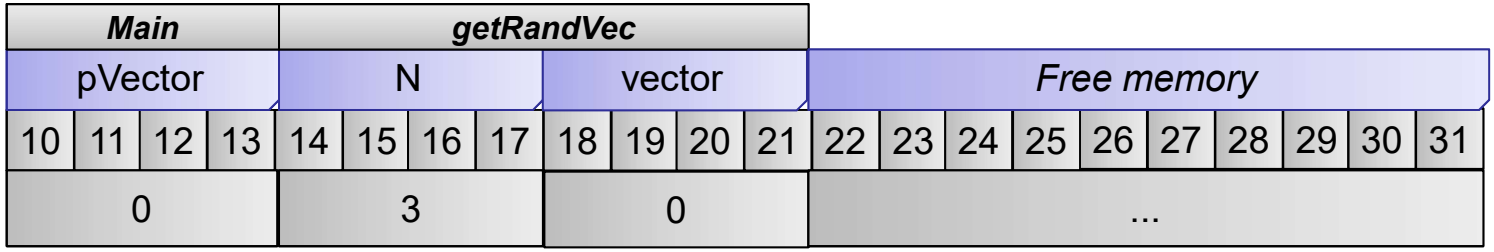
- Debemos suponer, que en el programa que llamará al método getRandVec, tendremos un código del estilo:

```
int main(void) {
        int *pVector=NULL;
        getRandVec(3,pVector);
         free(pVector);
        return EXIT_SUCCESS ;
}
```

```
void getRandVec(int N, int* vector) {
        ....
}
```

- Por lo tanto, en la memoria tendremos algo de este estilo:

- Al entrar en el método getRandVec, se debe añadir a la memoria los parámetros, que son variables locales de la función (para simplificar, asumiremos que se guardan en el mismo espacio de memoria, aunque en realidad no es así). Fijaros que *vector* coge el valor de *pVector*, que es una dirección de memoria (NULL en este caso).



Punteros a punteros

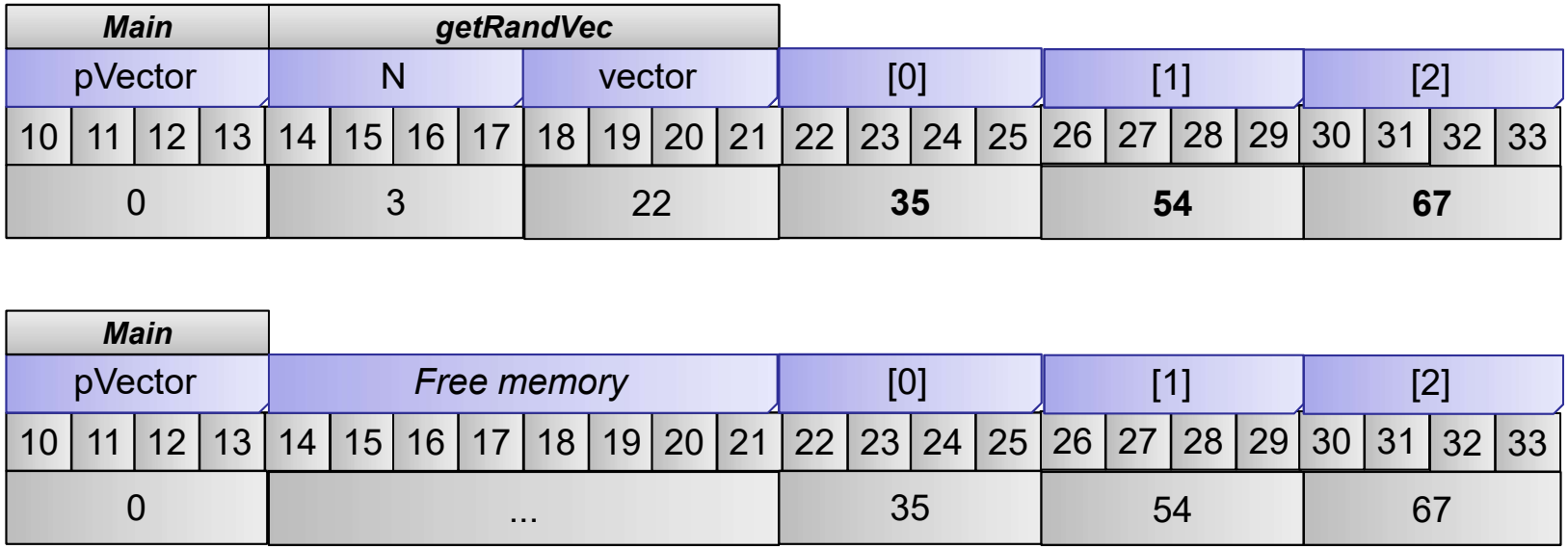
- Sólo empezar, se declara una nueva variable entera i, que no tendremos en cuenta por simplicidad, y se reserva un espacio de 3 enteros, la dirección resultante se guarda en ***vector***.

```
void getRandVec(int N, int* vector) {
        int i=0;
        if(vector!=NULL) {
                free(vector);
        }
        vector=(int*)malloc(N*sizeof(int));
         for(i=0;i<N;i++) {
                vector[i]=rand();
         }
}
```

Main				***getRandVec***																			
pVector				N				vector				[0]				[1]				[2]			
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
0				3				22				...				...				...			

Punteros a punteros

- Finalmente se asignan los valores aleatorios a las diferentes posiciones de memoria (por ejemplo los valores 35, 54 y 67). Al salir, se eliminan todas las variables creadas dentro del método (las que están debajo del recuadro ***getRandVec***. Fijaros que en las variables de Main no ha habido ningún cambio, por lo tanto, *pVector* continúa teniendo un valor NULL. Aunque la memoria sigue estando ocupada, no sabemos en qué dirección empieza.



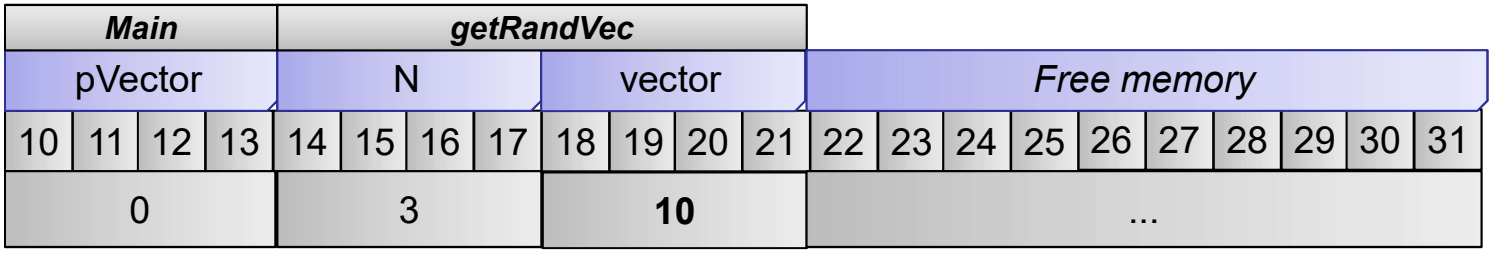
Punteros a punteros

- El problema está en que el cambio que se hace en la dirección no repercute a la variable del método que hace la llamada. Para solucionar el problema, se debe utilizar un **puntero a puntero**. El código quedaría así:

```
void getRandVec(int N, int** vector) {
        int i=0;
        if(*vector!=NULL) {
                free(*vector);
        }
        *vector=(int*)malloc(N*sizeof(int));
         for(i=0;i<N;i++) {
                (*vector)[i]=rand();
         }
}
```

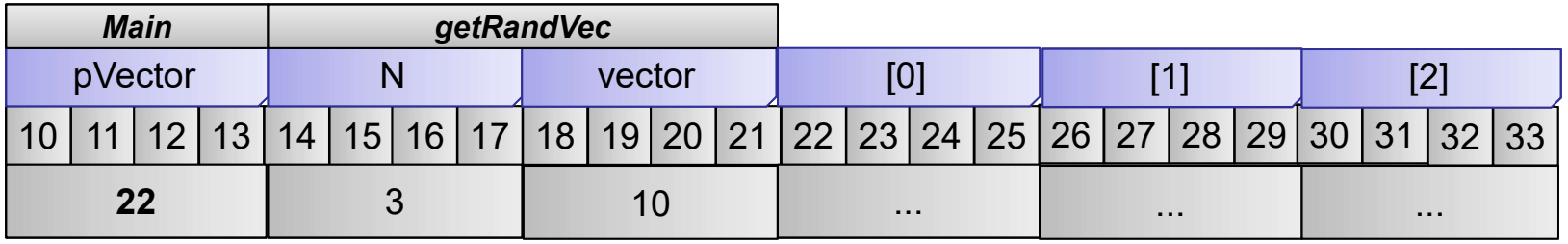
```
int main(void) {
        int *pVector=NULL;
        getRandVec(3,&pVector);
         free(pVector);
        return EXIT_SUCCESS ;
}
```

- Si repetimos el proceso, pero ahora considerando los cambios efectuados, tenemos que al empezar a ejecutar el método ***getRandVec*** la memoria estará de esta manera (nuevamente se marcan los cambios en negrita):

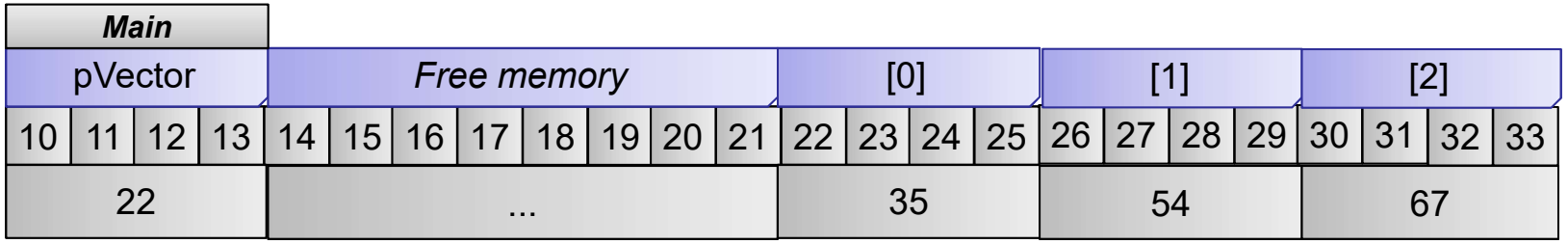


Punteros a punteros

- En este caso, la dirección de la memoria reservada no se guarda en ***vector***, sino en el **contenido de vector**, y por lo tanto, a la dirección donde ***vector*** apunta (en el ejemplo la dirección 10).



- Al salir del método se eliminan las variables locales de ***getRandVec***, y nos queda:

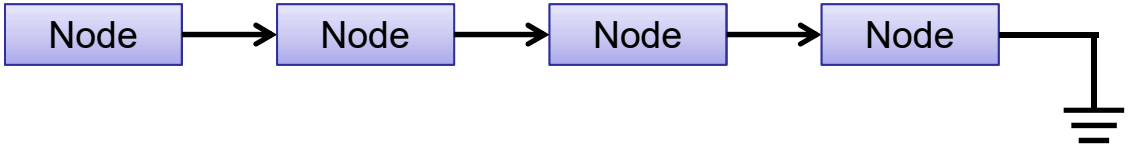


- Por tanto podemos acceder al vector de números aleatorios, ya que sabemos que empiezan en la dirección de memoria 22 (valor de pVector).

Ejercicios

Autoevaluación:

1. En los tipos abstractos de datos (TADs), se suele utilizar memoria dinámica y punteros. Uno de los TADs mas simples es la **lista encadenada**. Una lista encadenada es una secuencia de elementos, donde cada elemento "está encadenado" al siguiente elemento de la lista, permitiendo ir desde el principio al final de la secuencia. Una posible representación gráfica es la siguiente, donde tenemos una serie de nodos, donde cada nodo apunta al siguiente, y el último nodo apunta a NULL (representado generalmente con el símbolo de la figura).



Una forma simple de definir estos nodos, para que puedan contener cualquier tipo de elemento, es utilizar punteros genéricos. Además, cada nodo debe apuntar a su sucesor, por lo que un nodo se puede definir con una estructura tNode como la siguiente:

```
struct  tNode {
        void* element;
        tNode* next;
};
```

Ejercicios

Autoevaluación :

Siguiendo las indicaciones anteriores, se pide:

a) Implementa el método *addNode*, que dada una lista y un puntero a un elemento, añada al final de la lista el nodo correspondiente. Se debe considerar el caso inicial, en el que la lista es NULL. La cabecera del método debe ser:

```
void addNode(tNode** list, void* element);
```

b) Implementa el método *releaseList*, que dada una lista, elimine todos sus nodos. Dado que no sabemos de que tipo son los elementos, no eliminaremos el contenido del nodo, sólo el nodo en si. La cabecera debe ser:

```
void releaseList(tNode** list);
```

c) Escribe un pequeño programa que demuestre el uso de los métodos anteriores para guardar una lista de cadenas de caracteres. Crea también un método showStrList, que muestre el contenido de la lista. La cabecera de este método debe ser:

```
void showStrList(tNode* list);
```

Ejercicios

Autoevaluación (Respuesta):

a) Para implementar el método addNode, se debe considerar los dos casos posibles:

1. *La lista está vacía (*list=NULL) y por lo tanto se debe añadir el primer nodo.*
2. *La lista contiene algún elemento. En este caso, se debe recorrer la lista hasta el último elemento y añadir el nuevo elemento. El último elemento es ese que tiene como siguiente un NULL.*

En los dos casos, se deberá reservar memoria para un nuevo nodo y guardar en él el elemento obtenido. El código se muestra a continuación:

```
void  addNode(tNode** list, void* element)  {
        tNode* pLast=NULL;
        tNode* newNode=(tNode*)malloc(sizeof(tNode));

        /* Create the new node */
        newNode->element=element;
        newNode->next=NULL;

        /* Consider empty list */
        if(*list==NULL) {
                /* The new element becomes the list */
                *list=newNode;
        } else {
                /* Search for last node */
                pLast=*list;
                while(pLast->next!=NULL) {
                        pLast=pLast->next;
                }
                /* Add the new element */
                pLast->next=newNode;
        }
}
```

Ejercicios

Autoevaluación (Respuesta):

b) El método releaseList, irá recorriendo todos los nodos de la lista y los irá eliminando. Al final, la lista debe quedar vacía, por lo tanto, el contenido del parámetro list debe ser NULL. Lo único que se debe tener en cuenta es que no podemos eliminar un nodo antes de haber apuntado a su sucesor. En el enunciado se dice que no debemos liberar los elementos, por lo que asumimos que la memoria de estos elementos se libera en algún otro lugar. El código se muestra a continuación:

```
void  releaseList(tNode** list)  {
        tNode* pList=NULL;

        /* While the list is not empty */
        while(*list!=NULL) {
                /* Store the first node */
                pList=*list;

                 /* Remove the first node from the list */
                 *list=pList->next;

                  /* Destroy the first node */
                  free(pList);
         }
}
```

Ejercicios

Autoevaluación (Respuesta):

c) *Primero definimos el método para visualizar la lista. Este método simplemente irá recorriendo la lista e imprimirá el contenido del nodo:*

```
void  showStrList(tNode* list) {
        tNode* pList=list;

        /* While the list is not empty */
        while(pList!=NULL) {
                /* Show element */
                printf( "%s\n",pList->element);

                /* Move to next node */
                pList=pList->next;
        }
}
```

Finalmente definimos un ejemplo simple de utilización de los métodos anteriores. En este caso añadimos colores a la lista. Dado que los colores de definen como cadenas de caracteres constantes, no necesitaremos liberar su memoria.

```
int main(void) {
        tNode* colorList=NULL;

        /* Add three nodes to the list */
        addNode(&colorList,"blue");
        addNode(&colorList,"red");
        addNode(&colorList,"green");

        /* Show and releas list */
        showStrList(colorList);
        releaseList(&colorList);

        return EXIT_SUCCESS ;
}
```