Pràctiques de Programació

La memòria dinàmica i els punters

La memòria dinàmica i els punters

- En aquest mòdul es mostra dos conceptes importants de programació, especialment quan es programa en llenguatge C. L'objectiu d'aquest material no és donar conceptes teòrics, sinó que vegeu una visió més aplicada dels conceptes dels apunts.
  - Seguiu les explicacions per entendre els exemples que es mostren.
  - Analitzeu i assegureu-vos que enteneu els exemples de codi
    - Podeu escriure els programes, compilar-los i executar-los.
    - Intenteu fer canvis al codi per veure si enteneu què s'hi fa.
  - Intenteu resoldre els exercicis que es proposen.
    - Compareu la vostra solució amb la proposada.
    - Si no coincideixen i no sabeu si la vostra es bona o no, acudiu al vostre consultor.

- S'utilitzen representacions gràfiques que no són estàndard. Vosaltres podeu definir les vostres pròpies, només us mostrem una possibilitat.
  - Sempre que es treballa amb aquests conceptes és important fer un esquema gràfic de les dades.

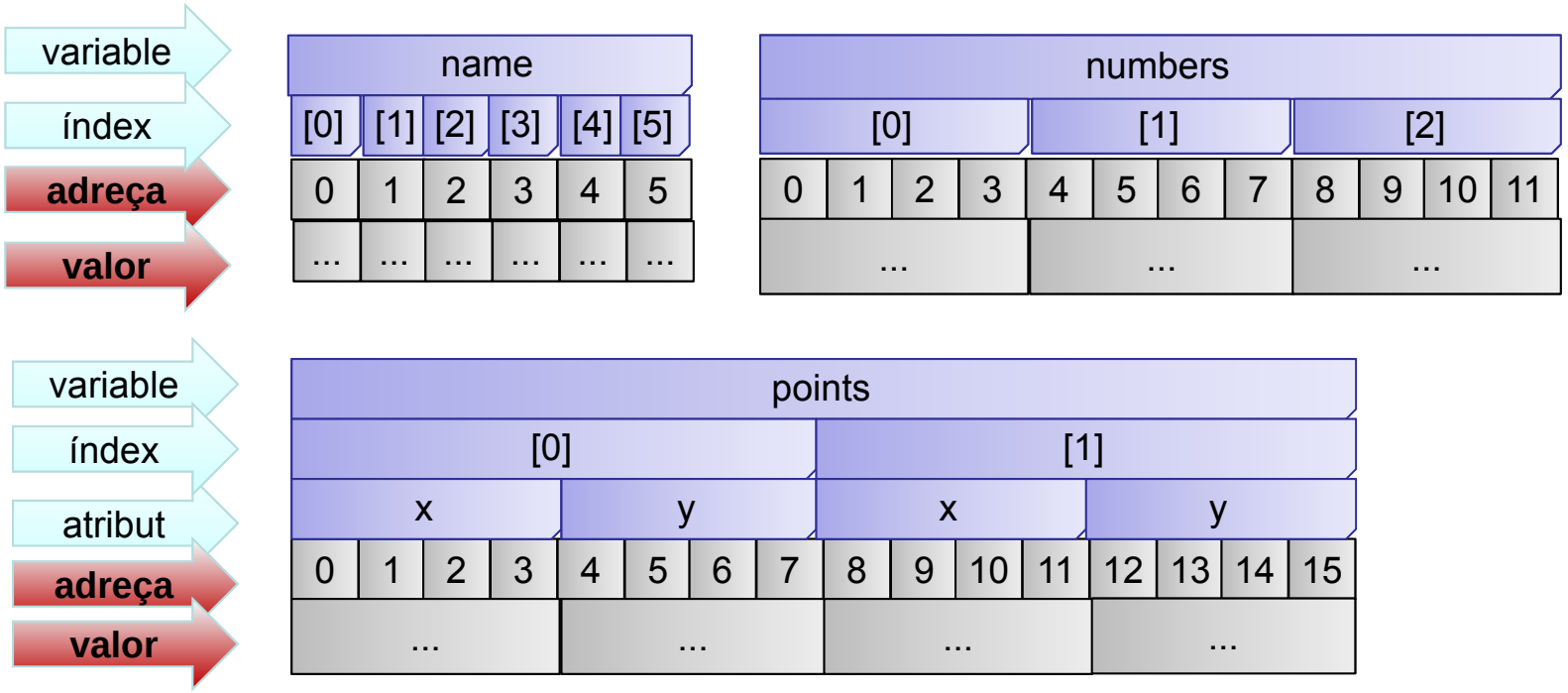
La memòria

- Fins ara hem vist:
  1. Com crear taules per emmagatzemar seqüències d'elements, tant elements de tipus estàndard (caràcters, enters, reals, ...) o qualsevol tipus de dades definides per nosaltres (coordenada, punt, persona, ...):

```
/* Seqüència de 6 caràcters */
char name[6];

/* Seqüència de 3 enters */
int numbers[3];

/* Seqüència de 2 punts */
struct {
        int x,y;
} tPoint;

tPoint points[2];
```

La memòria

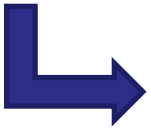
- Fins ara hem vist:
  2. Com queden organitzats els valors d'aquestes seqüències a memòria. És important que hagueu entès que la memòria està organitzada per unitats de mida byte, i que cada unitat té la seva pròpia adreça. Ens centrem principalment en les adreces i amb quins "noms" els hi podem fer referència (nom de la variable, índex en un array o atribut en una estructura). Els valors de moment no els considerem.



La memòria

- Fins ara hem vist:
  3. Com obtenir la mida en bytes d'un tipus, utilitzant la comanda **sizeof**. Fixeu-vos que en aquest cas utilitzem el nom de la variable com a paràmetre al **sizeof**. És l'equivalent a posar el tipus corresponent de la variable. Comproveu que coincideixen.

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

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

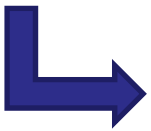


```
a) La mida de name es 6 bytes
b) La mida de numbers  es 12bytes
c) La mida de points es 16 bytes
```

Treballant amb les adreces

- En tots els exemples que hem vist fins ara, hem assumit que les adreces de memòria sempre comencen des de zero. En realitat mai es produirà aquesta situació, és més, veurem que l'adreça 0 (zero), coneguda també com a NULL, s'utilitza com a valor per defecte al inicialitzar un punter.
- Per tant, les adreces que hem utilitzat fins ara, són "relatives" a l'adreça inicial.
- Quan declarem una variable, el sistema operatiu busca un espai buit a la memòria, i l'assigna a la nostra variable. Per tant, cada cop que executem el nostre programa l'adreça de les variables serà diferent. Per obtenir una adreça de memòria disposem de l'operador **&**. En el codi següent es mostra un exemple. Fixeu-vos que els valors d'adreces es mostren generalment en Hexadecimal, i que els casos a i b tenen la mateixa adreça de memòria.

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



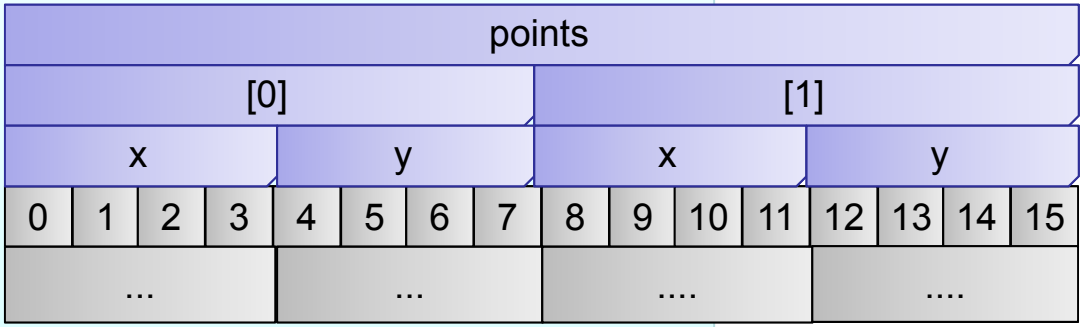
```
a) La adreça de <points> es 0027F7A8
b) La adreça de <points[0]> es 0027F7A8
c) La adreça de <points[0].y> es 0027F7AC
d) La adreça de <points[1].y> es 0027F7B4
```

(*) Els arrays/vectors tenen un significat diferent, per això en el cas de *points*, no s'utilitza &. Fixeu-vos que els casos a) i b) tenen la mateixa adreça. Veurem aquest cas quan parlem de punters.

Treballant amb les adreces

- Les adreces no deixen de ser valors numèrics, per tant, podem operar amb elles. Fixeu-vos que si posem les adreces relatives a l'adreça inicial (&points), obtenim els valors de l'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) Adreces relatives: points[0] => %d, points[1] => %d\n",aPoints0-aPoints,aPoints1-aPoints);
printf("b) Adreces relatives: points[0].x => %d, points[0].y => %d\n",aPoints0x-aPoints,aPoints0y-aPoints);
printf("c) Adreces relatives: points[1].x => %d, points[1].y => %d\n",aPoints1x-aPoints,aPoints1y-aPoints);
```



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

Els punters

- Fins ara hem vist com s'organitzen a memòria diferents tipus de dades. També teniu exemples de codi de com accedir a les adreces de memòria. Tota aquesta informació és important a l'hora d'entendre el funcionament d'un nou tipus de variables:

Els punters

- Els punters són simplement variables que en comptes de contenir un valor, contenen una adreça de memòria. En aquest sentit, es diu que un punter "apunta" a una certa posició de memòria.
- Els punters es declaren posant un asterisc "*****" davant del nom de la variable. Per exemple, en el següent codi, es declaren "punters a diferents tipus de dades". Fixeu-vos que la forma d'inicialitzar un punter és assignant-li el valor NULL (que correspon a l'adreça 0):

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

- Igual que tenim l'operador que ens dona l'adreça d'una variable, també podem obtenir el contingut d'una certa adreça (continguda en un punter), amb l'operador de "*des-referenciació*", que és el propi asterisc "*****". Fixeu-vos en el següent codi, intenteu entendre que es fa.

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

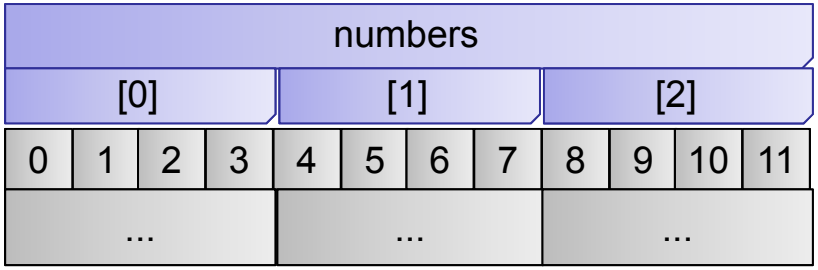
```
char  a='c';
char  *pA=&a;
*pA='b';
```

Nomenclatura: En una expressió del tipus **a=2**, s'acostuma a utilitzar el nom ***rvalue*** (*right value*, pronunciat *"are value"*) o objecte per la part dreta de l'igual (el valor 2), i ***lvalue*** (*left value*, pronunciat *"el value"*) per l'adreça (&a) on es guarda el valor.

Els punters

Autoavaluació:

1. A partir de la següent representació de memòria, indica quina seria la sortida de cadascuna de les següents sentències.



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

2. A partir dels exemples d'utilització de la comanda **sizeof**, intenta deduir quina seria la sortida del següent codi:

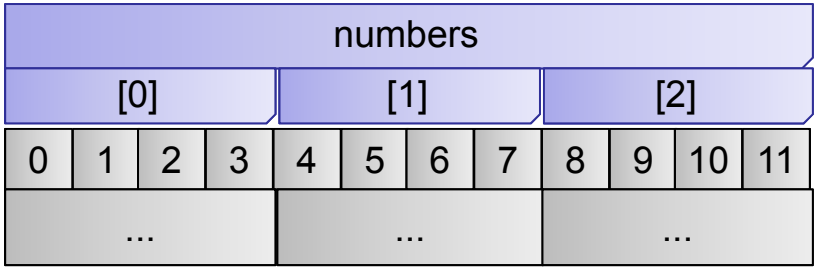
```
struct tPoint {
    int x,y;
};
tPoint points[5];
int vectorI[3] = {1, 2, 3};
char vectorC[5] = "hola";
tPoint *pPoint=points;
char  *pChar=vectorC;
int *pInt=vectorI;

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			
b			
c			

Els punters

Autoavaluació (Resposta):

1. A partir de la següent representació de memòria, indica quina seria la sortida de cadascuna de les següents sentències.



Expressió	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) Obtenim la adreça de la posició 2 del vector numbers, per tant, la expressió ens retorna l'adreça de numbers[2] = 8 (mirar taula).

2) Primer obtenim l'adreça de numbers, que és 0. La convertim a un enter, i per tant a partir d'aquest moment, tota operació serà com si es tractés d'un enter normal i corrent, i per tant, 0 + 1 = 1.

3) Seguint les explicacions dels casos anteriors, però en comptes de sumar un valor fix al valor del cas 1, sumem el nombre de bytes que ocupa un enter. Per tant, obtenim 0 + 4 = 4, que és diferent a 8, per tant el resultat és false.

*4) Com en el cas anterior, però ara sumem 0 + 2 * 4 = 8, que per tant ens dona un resultat true.*

Els punters

Autoavaluació (Resposta):

2. A partir dels exemples d'utilització de la comanda **sizeof**, intenta deduir quina seria la sortida del següent codi:

```
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 primer valor de cada cas és la mida d'un array d'un determinat tipus. Per exemple en el cas a, volem la mida del tipus "tPoint[5]", que s'obté amb:*

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

  La resta de casos són més simples:

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

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

Els punters

Autoavaluació (Resposta):

2. A partir dels exemples d'utilització de la comanda **sizeof**, intenta deduir quina seria la sortida del següent codi:

```
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 segon valor és la mida del punter. Recordeu que un punter emmagatzema una adreça de memòria, per tant, la seva mida serà la que ocupi una adreça de memòria, independentment del tipus al que faci referència. El valor dependrà del sistema en què treballeu:*
  - *En un sistema de 32bits, les adreces es guarden en 32bits => **4** bytes.*
  - *En un sistema de 64bits, les adreces es guarden en 64bits => **8** bytes.*

Curiositat: Si us fixeu en la mida de les adreces en cada sistema, veureu que en un sistema de 32bits, el nombre màxim d'adreces de memòria diferents serà 2^{32}, que correspon a 4.294.967.296 (≈4Gb). Aquest és el motiu pel qual aquests sistemes estan limitats a un màxim de 4Gb de RAM.

Els punters

Autoavaluació (Resposta):

2. A partir dels exemples d'utilització de la comanda **sizeof**, intenta deduir quina seria la sortida del següent codi:

```
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 i últim valor de cada cas és la mida del "contingut" del punter. En aquest punt és on afecta el tipus de punter que utilitzem, ja que cada punter "conté" un valor del seu tipus. Els valors corresponen a:*

  *a) sizeof(*pPoint) = sizeof(*(tPoint*)) = sizeof(tPoint) = 2 * sizeof(int) = 2 * 4 =* ***8***

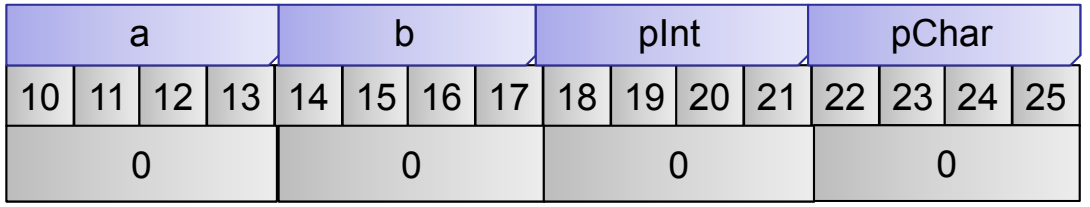
  *b) sizeof(*pChar) = sizeof(*(char*))= sizeof(char) =* ***1***

  *c) sizeof(*pInt) = sizeof(*(int*))= sizeof(int) =* ***4***

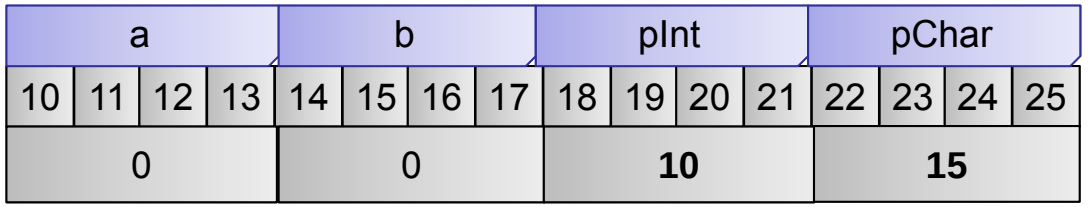
Els punters

- A continuació es mostra un codi simple de l'ús de punters, amb la evolució de la seva representació de la memòria.

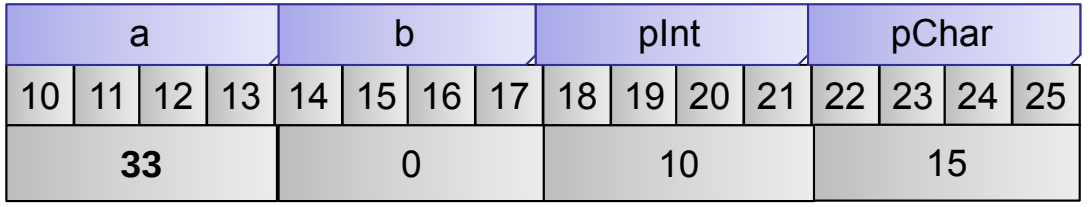
```
int a=0;
int b=0;
char  *pChar=NULL;
int *pInt=NULL;
```



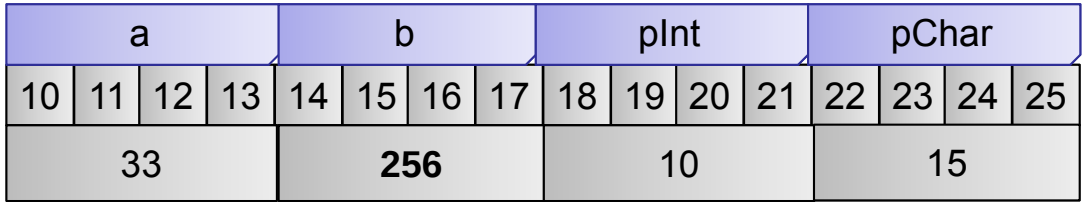

```
pInt=&a;
pChar=((char*)&b)+1;
```




```
*pInt=33;
```

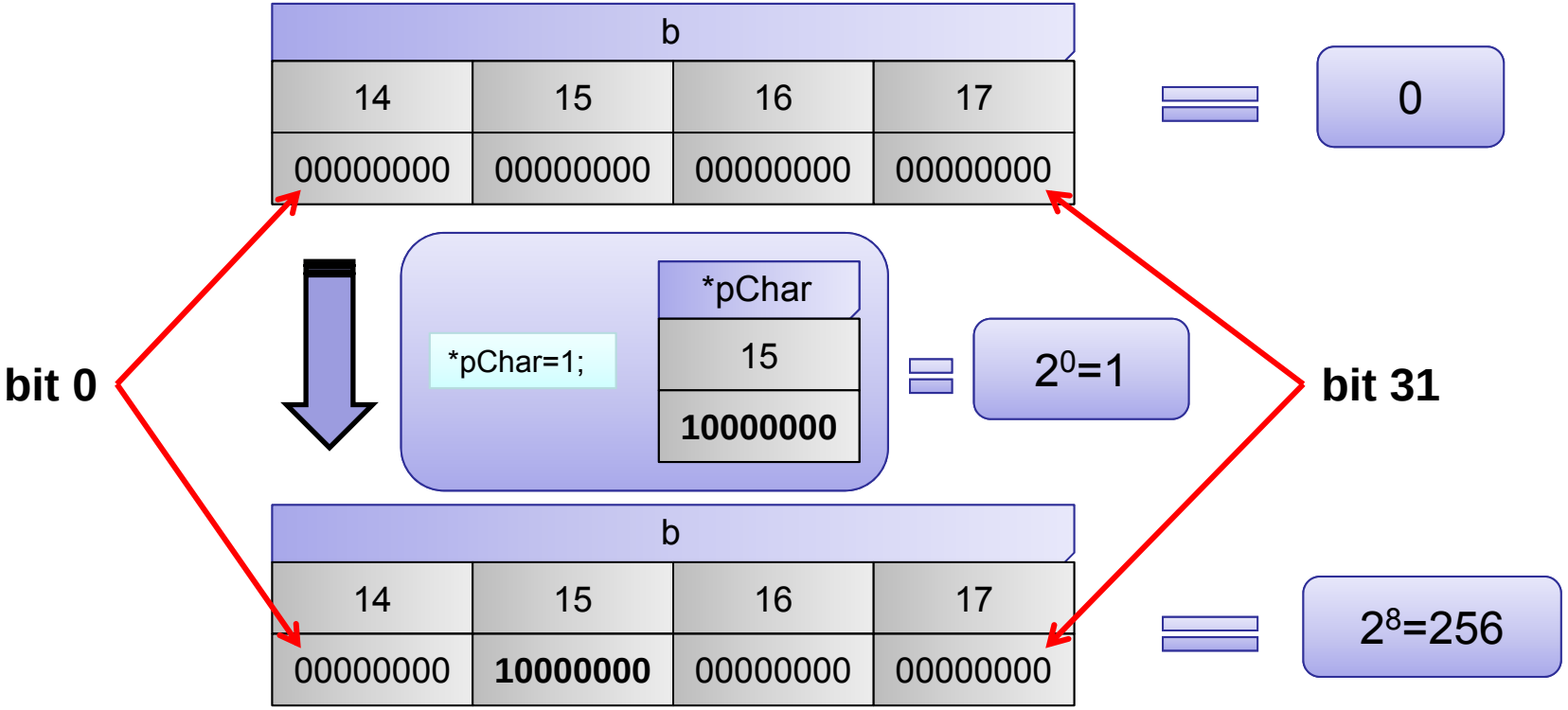



```
*pChar=1;
```



Els punters

- Fixeu-vos en l'últim cas. El punter pChar està "apuntant" a la posició de memòria 15, que correspon al segon byte de l'enter b. Si mirem a nivell de bit, la variable b correspon a la següent representació estàndard (complement a 2 amb el bit més significatiu a la dreta):



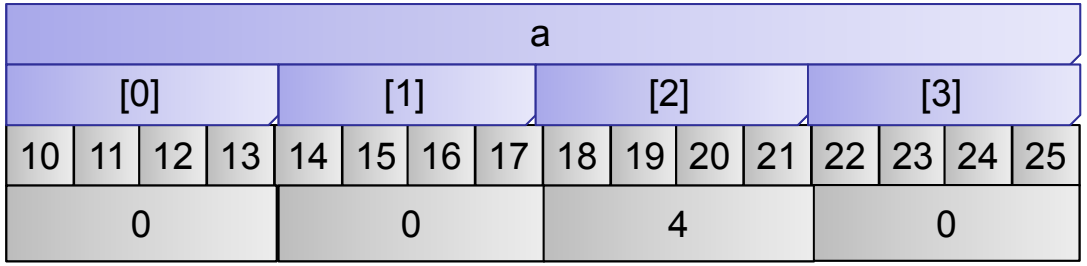
Nota: Tot i que no demanem que conegueu els formats dels valors, és molt útil saber que existeixen. El més important és tenir present que quan treballem amb punters, podem modificar valors directament a la memòria, el que ens confereix un gran control sobre les dades, però també afegeix un nivell de complexitat superior. Tingueu clar on apunten els vostres punters.

Els punters

- Una de les grans utilitats dels punters és poder recórrer seqüències.
  - **Indexació:** Permet accedir al contingut de la **N-èssima** unitat a partir d'una adreça inicial. Cada unitat correspon a la mida del tipus de dades que apunta el punter. En l'exemple següent, pInt conté el valor de l'adreça de a (10), però en utilitzar la indexació pInt[2], el que estem accedint és el **contingut** de l'adreça [10 + sizeof(int) * (2)] . Es mostren les dues alternatives:

```
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;
```



  - **Increment/decrement:** Avancem o retrocedim unitats, corresponent a la mida de dades que apunta el punter. Quan es suma o es resta directament un valor enter a un punter, l'estem desplaçant per unitats del seu tipus. L'exemple anterior es podria haver escrit com:

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

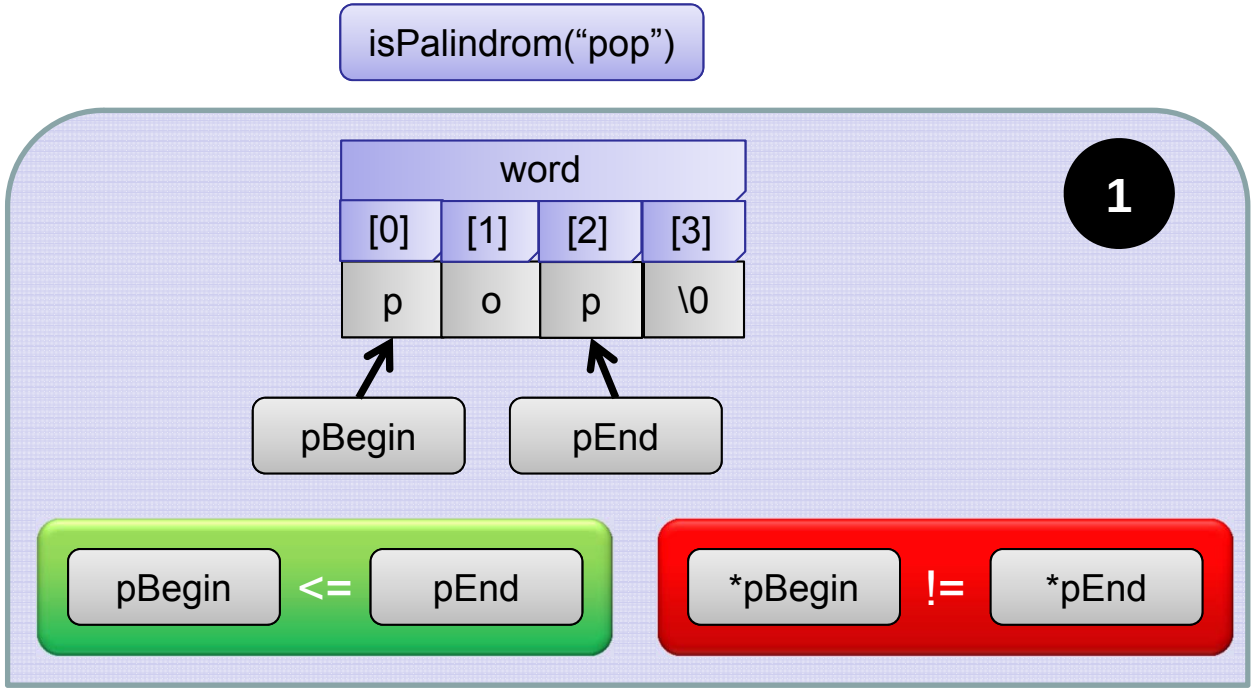
```
int a[4]={0,0,0,0};
int *pInt=a;
*(pInt+2)=4;
```

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

Els punters

- Una de les grans utilitats dels punters és poder recórrer seqüències.
  - **Exemple:** A continuació es mostra un exemple de punters per saber si una seqüència de caràcters és un palíndrom o no (cap i cua). També un exemple gràfic de com funcionaria amb una paraula curta. Fixeu-vos en una possible representació gràfica dels punters, com a nodes que "apunten" a un cert element, i on generalment no s'utilitzen adreces, ja que com ja hem vist, aquestes van canviant i es poden obtenir a partir dels noms de les variables.

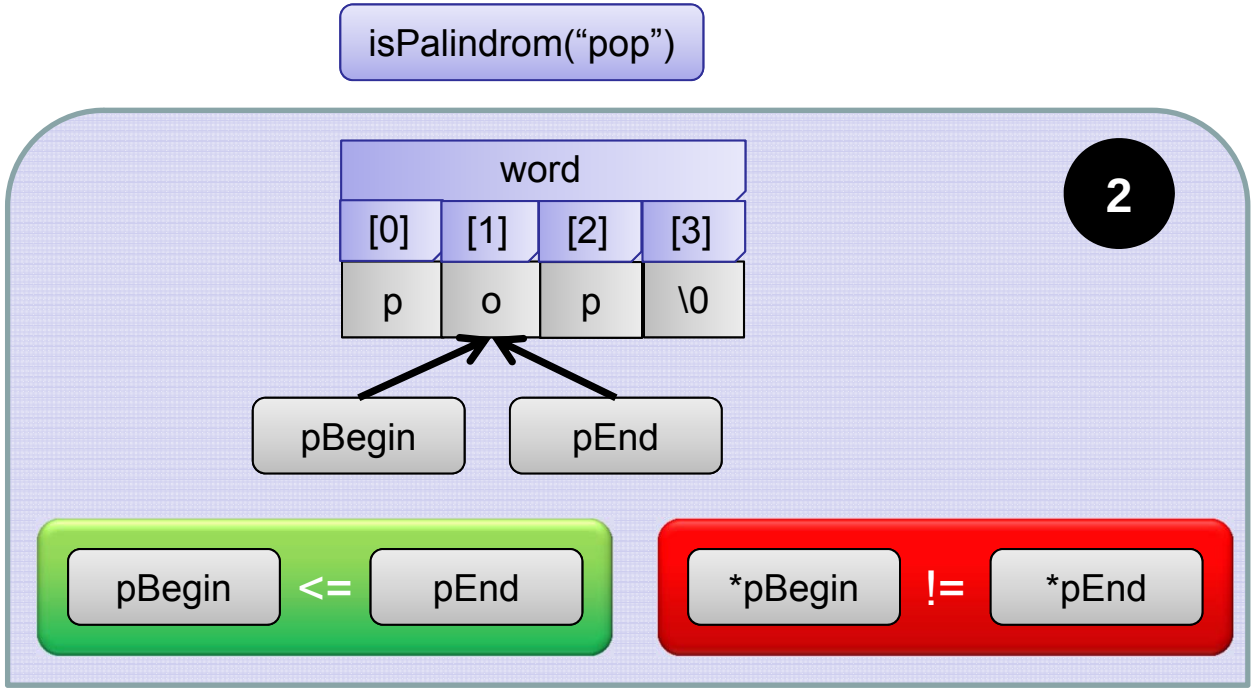
```
bool isPalindrom(char word[]) {
        char *pBegin=NULL;
        char *pEnd=NULL;

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



Els punters

- Una de les grans utilitats dels punters és poder recórrer seqüències.
  - **Exemple:** A continuació es mostra un exemple de punters per saber si una seqüència de caràcters és un palíndrom o no (cap i cua). També un exemple gràfic de com funcionaria amb una paraula curta. Fixeu-vos en una possible representació gràfica dels punters, com a nodes que "apunten" a un cert element, i on generalment no s'utilitzen adreces, ja que com ja hem vist, aquestes van canviant i es poden obtenir a partir dels noms de les variables.

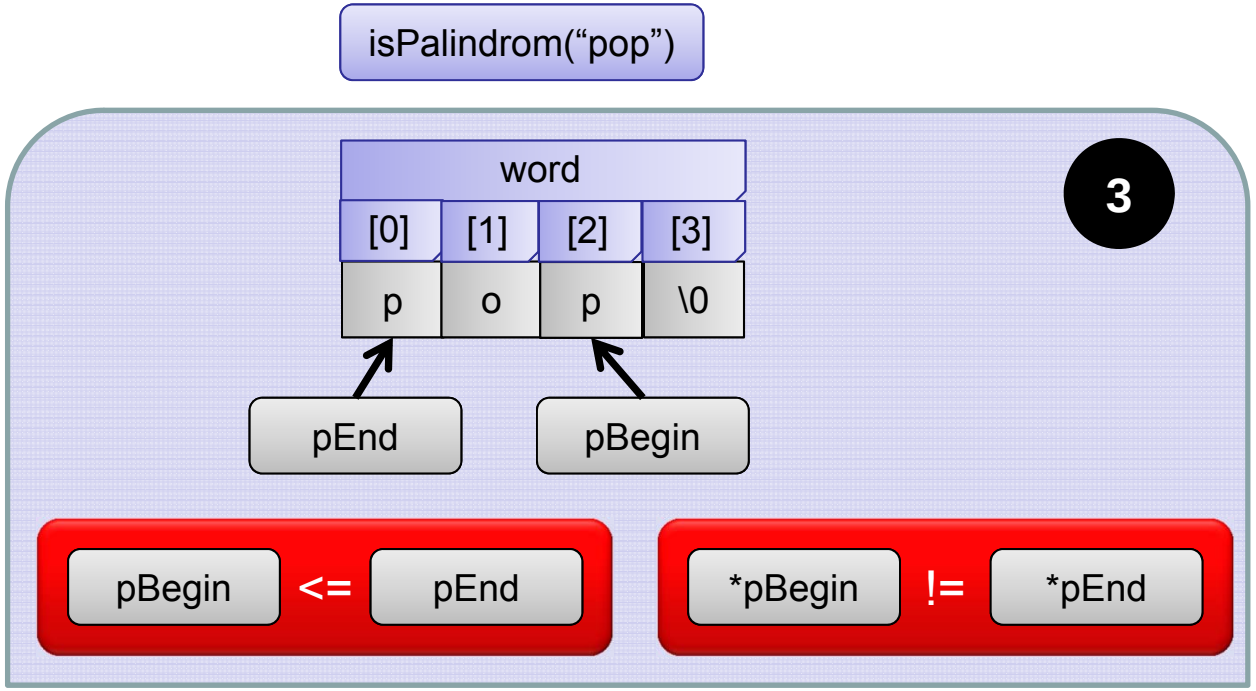
```
bool isPalindrom(char word[]) {
        char *pBegin=NULL;
        char *pEnd=NULL;

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



Els punters

- Una de les grans utilitats dels punters és poder recórrer seqüències.
  - **Exemple:** A continuació es mostra un exemple de punters per saber si una seqüència de caràcters és un palíndrom o no (cap i cua). També un exemple gràfic de com funcionaria amb una paraula curta. Fixeu-vos en una possible representació gràfica dels punters, com a nodes que "apunten" a un cert element, i on generalment no s'utilitzen adreces, ja que com ja hem vist, aquestes van canviant i es poden obtenir a partir dels noms de les 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;
}
```



Els punters

- Per acabar aquest apartat, recordar que els punters són simplement una variable que en comptes de contenir un valor, conté una adreça de memòria. Alguns dels usos més comuns dels punters estan relacionats amb:
  - ***Pas de paràmetres per referència:*** Són aquells paràmetres que poden patir modificacions dins d'un mètode. Per tant, quan cridem una acció/funció amb un paràmetre de sortida o de entrada/sortida, utilitzem punters.
  - ***Recorreguts i cerques:*** Quan es fan recorreguts i cerques en estructures complexes, generalment s'utilitzen punters.
  - ***Retorn d'estructures en funcions:*** La quantitat de memòria que pot retornar una funció és limitada. Per solucionar el problema, una pràctica habitual, és retornar un punter a la estructura, en comptes de la estructura en sí.
  - ***Memòria dinàmica:*** Zones de memòria que es reserven durant l'execució del programa, generalment perquè no coneixem la mida necessària en el moment de programar. Aquest tema es tracta a continuació.

- Existeix un tipus especial de punters (**void***), que es considera un punter genèric. No es pot utilitzar aquests punters per fer aritmètica, ja que no tenen un tipus associat, però es poden convertir amb un cast a qualsevol tipus de punter. S'utilitzen sobretot per definir estructures de dades genèriques o en funcions que treballen amb memòria. Veurem algun exemple més endavant.

Memòria dinàmica

- La memòria dinàmica és aquella memòria que es reserva en temps d'execució. Generalment s'utilitza quan durant el desenvolupament de la nostra aplicació no sabem quanta memòria necessitarem.
- Quan utilitzem memòria dinàmica, cal tenir en compte que hi ha dos processos implicats:
  - **Reserva:** Demanem al sistema operatiu que ens concedeixi un espai de memòria per a guardar-hi les nostres dades. Cal verificar que realment existeix aquest espai.
  - **Alliberar:** Li comuniquem al sistema operatiu que el nostre programa ja ha acabat d'utilitzar un cert espai de memòria i que per tant li retornem. A partir d'aquest moment ja no el podrem utilitzar.
- És molt important tenir en compte que la memòria és un recurs **finit i limitat**.

Memòria dinàmica

Reservar memòria:

- La reserva de memòria es fa mitjançant la comanda **malloc.** Necessitareu fer la inclusió de la llibreria stdlib.h (**#include** <stdlib.h>)

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

- A aquesta comanda se li passa **el nombre de bytes** que es necessiten. El sistema busca una zona de memòria on hi hagi aquesta quantitat de bytes lliures **consecutius** i ens retorna **l'adreça de la primera posició**.
  - Utilitzeu la comanda *sizeof* per obtenir les mides dels tipus bàsics
- En el cas de que no hi hagi cap zona amb aquesta quantitat de memòria, ens retornarà un NULL.
  - Cal comprovar **sempre** que s'ha obtingut la memòria demanada.
- A continuació es mostra una taula amb equivalències entre la definició estàtica de memòria (dels exemples anteriors) amb la forma de fer-ho amb memòria dinàmica (es mostra la declaració i la reserva de memòria).

Estàtic	Dinàmic	
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**));

Memòria dinàmica

Alliberar memòria:

- Tota la memòria reservada amb un **malloc** s'ha d'alliberar tant aviat com sigui possible. La comanda per alliberar memòria és **free**.

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

- A aquesta comanda se li passa **l'adreça a la primera posició**. El sistema alliberarà aquesta memòria, podent-la tornar a assignar quan es demani més memòria dinàmica.
- Cal tenir en compte els següents punts:
  - Quan un programa acaba, tota la memòria que ha reservat s'allibera automàticament.
  - Quan un mètode acaba, **NO** s'allibera la memòria dinàmica reservada, però si que s'alliberen els punters que s'hi ha declarat, per tant podem perdre l'adreça a la memòria demanada. Les zones de memòria reservades i no alliberades es coneixen amb el nom de **memory leaks**, i donat que no tenim la seva adreça, no les podem alliberar fins a que finalitzi la aplicació.

Memòria dinàmica

Treballar amb memòria dinàmica:

- Quan reservem memòria dinàmica, no tenim cap coneixement del contingut de la zona de memòria obtinguda. Generalment s'inicialitza cada estructura segons els valors per defecte que siguin necessaris, però existeixen mètodes per assignar valors a nivell de bytes. Per exemple, la comanda **memset** assigna el valor **value** als **n** primers bytes a partir de la posició apuntada per **ptr**.

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

- Fixeu-vos en els exemples anteriors, que la memòria reservada s'assigna a un punter. Per tant, podem utilitzar tota la aritmètica de punters i les seves propietats per treballar amb aquesta memòria, igual que fèiem en el cas de la memòria estàtica.

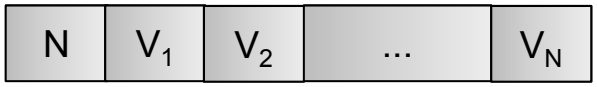
Memòria dinàmica

Autoavaluació:

1. Escriu un mètode, que llegeixi una seqüència d'enters de teclat, la guardi a memòria i ens retorni l'adreça on la ha guardat. Cal tenir en compte que el primer element de la seqüència d'entrada serà el nombre de valors que conté. El mètode també tindrà un paràmetre de sortida on retornarà el nombre d'elements de la seqüència. O sigui, la capçalera del mètode serà:

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

el format de la seqüència d'entrada serà:



i les dades a memòria haurien de quedar com (sense la mida):

Escriviu també el programa principal que cridi aquest mètode, i n'inverteixi l'ordre dels seus elements. Tingueu present que cal utilitzar correctament els mètodes per reservar i alliberar la memòria.

Memòria dinàmica

Autoavaluació (Resposta):

El primer que farem és escriure el mètode auxiliar. Fixeu-vos especialment a la part de reservar memòria (Allocate) i com s'accedeix als elements:

```
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;
}
```

Memòria dinàmica

Autoavaluació (Resposta):

Si us fixeu en la solució proposada, i la proveu, veureu que funciona correctament. Amb tot, ***no és una solució correcta****. Ens hem oblidat d'un pas clau quan treballem amb memòria dinàmica, comprovar que ens han donat la memòria que hem demanat. Per fer això, cal afegir la comprovació de que l'adreça retornada per la comanda malloc no és NULL.*

Cal decidir com gestionem aquest fet. En aquest cas, el que farem és retornar una longitud de -1, i un punter a NULL. D'aquesta manera, quan un mètode utilitzi aquesta funció, podrà saber si hi ha hagut un error de memòria.

La reserva de memòria quedaria com:

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

Memòria dinàmica

Autoavaluació (Resposta):

Anem a definir ara el programa principal, el qual utilitzi el mètode auxiliar i n'inverteixi l'ordre dels elements. Definim la part d'obtenció de les dades. Fixeu-vos en que declarem un enter per la Mida, i un punter per la seqüència. Cridem el mètode auxiliar per llegir el vector de valors, controlant el cas en que no s'ha pogut obtenir la memòria necessària per guardar la seqüència. Finalment, abans de sortir alliberem la memòria utilitzada:

```
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;
}
```

Memòria dinàmica

Autoavaluació (Resposta):

Finalment, invertirem l'ordre dels elements de la seqüència, utilitzant punters, de forma similar a com verificàvem si una paraula era un palíndrom o no. Fixeu-vos que necessitem 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 */
         ...
}
```

Memòria dinàmica

Autoavaluació (Resposta):

Per provar que tot funciona correctament, podeu definir un mètode per visualitzar la seqüència. Per exemple:

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

Memòria dinàmica

Altres conceptes:

- A part del **malloc**, existeix un altre mètode, anomenat **realloc**, que permet augmentar la mida d'una zona de memòria reservada prèviament (apuntada per ***ptr***), movent les dades a una altra zona de la memòria si és necessari . Retorna el punter a la nova zona de memòria.

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

- També disposem del mètode **calloc**, que funciona de forma similar al **malloc**, però inicialitza tota la zona de memòria reservada a zero. En aquest cas se li passa per separat el nombre d'elements i la mida de cada element, però el resultat continua essent una zona contigua de memòria de mida (num*size) bytes.

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

Memòria dinàmica

Errors típics:

- **Utilitzar un punter no inicialitzat**. Això pot passar en paràmetres d'entrada/sortida o sortida, o si no s'ha inicialitzat el valor d'un punter. Cal tenir en compte que si al punter no se li assigna el valor NULL, aquest pot contenir qualsevol valor, per tant, a efectes pràctics conté un valor no NULL i per tant es suposa que apunta a una zona correcta de memòria. Si l'utilitzem, podem canviar el contingut d'una zona de la memòria, perdent dades o provocant que el sistema falli. Per sort, el sistema operatiu ens protegeix de poder fer mal a la resta d'aplicacions, però la nostra aplicació no queda protegida.

- **Perdre la referència a una zona de memòria**, perdent per tant la possibilitat de poder-la alliberar. En aquest cas, al sortir del mètode s'elimina el punter **ptr**, però no la memòria reservada, i per tant ja no hi ha manera de poder-la alliberar, ja que hem perdut l'adreça en la qual es troba. El mateix passa si modifiquem l'adreça de **ptr** (per simplicitat s'ha obviat la comprovació de la memòria).

```
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;
}
```

Punters a estructures

- Un cas particular de l'ús de punters, es dona quan aquest apunta a una estructura. En aquests casos, podem utilitzar l'operador **"->"** per accedir directament als seus camps.

Per exemple, recordem la definició del tipus tPoint:

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

Si tenim un punter a una estructura d'aquest tipus:

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

Tenim dues possibilitats d'accedir als seus atributs (x i y), la primera és obtenir el contingut del punter, i per tant, una estructura de tipus tPoint i accedir-hi directament (fixa't que una possibilitat és accedir com si fos un vector d'un sol element):

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

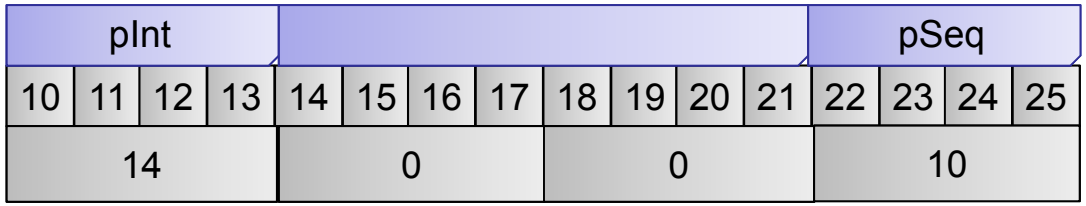
Una altra opció és accedir als atributs amb l'operador **"->"**:

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

Punters a punters

- Un cop hem vist els punters i com es gestiona la memòria dinàmica, ja estem preparats per veure un cas particular de punters: Els punters a punters.
- Fixeu-vos en el següent codi i la seva representació gràfica:

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

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



- Fixeu-vos amb que el contingut d'un punter a punter, no és un valor, sinó un punter, per tant, una adreça de memòria.

Punters a punters

- Un exemple clar de l'ús dels punters a punters són les seqüències de cadenes de caràcters. Imagineu que volem guardar els noms i les edats d'un conjunt de persones, agafant aquests valors per teclat. Si seguim l'exemple de l'exercici proposat anteriorment:
  - Les edats les podem guardar com una seqüència d'enters, per tant, si en tenim **N**, faríem:

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

  - En el cas dels noms, hem vist anteriorment, que per guardar una cadena de caràcters de longitud **L**, ho podíem fer:

```
char*pName=NULL;

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

  - Si combinem els dos exemples anteriors, podem veure que si volem guardar una seqüència d'**N** cadenes de caràcters de longitud $\mathbf{L_i}$ (on $\mathbf{L_i}$ és la longitud del i-èssim nom), faríem:

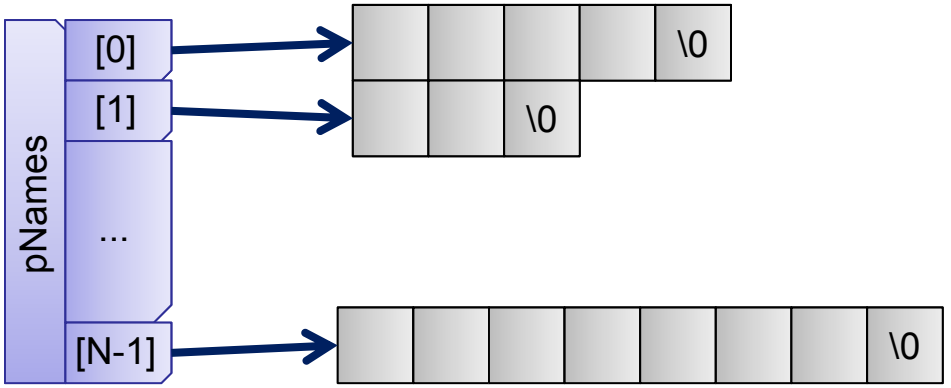
```
char**pNames=NULL;
pNames=(char**)malloc(N*sizeof(char*));
```

  - Per reservar la cadena de caràcters on guardarem l'i-èssim nom, ho farem amb:

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

Punters a punters

- Gràficament ho podem representar com una llista de punters, on cada punter apunta a una cadena de caràcters (per seguir el format estàndard, totes acabades amb el caràcter '\0'). Fixeu-vos que hem eliminat les adreces de memòria de l'esquema.



- Per alliberar aquesta estructura és imprescindible alliberar cada cadena de caràcters abans d'alliberar la llista de punters. Per tant, cal fer:

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

Punters a punters

- Un altre cas típic és la inicialització de memòria dinàmica dins d'un mètode. Per exemple, volem un mètode per obtenir una seqüència d'N valors aleatòris. Fixa't amb el següent exemple:
  1. Si el vector ja existeix (es diferent a NULL), allibera la memòria (no sabem la seva longitud).
  2. Reserva espai per a la nova llista de valors.
  3. Omple la llista de valors amb nombres aleatòris.

```
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 proves aquest codi, veuràs que no funciona. És més, és un exemple de ***memory leak***, ja que es perd la referència a la zona de memòria reservada. Anem a veure què està passant:

Punters a punters

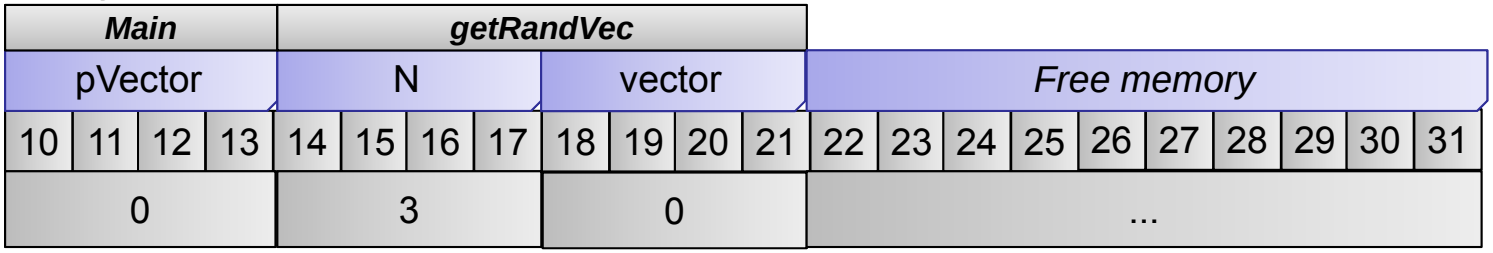
- Hem de suposar, que en el programa que cridarà al mètode getRandVec, tindrem un codi de l'estil:

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

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

- Per tant, a memòria tindrem alguna cosa de l'estil:

- En entrar al mètode getRandVec, cal afegir a la memòria els paràmetres, que són variables locals de la funció (per simplificar, assumirem que es guarden en el mateix espai de memòria, tot i que realment no és així). Fixeu-vos amb que *vector* agafa el valor de *pVector*, que es una adreça de memòria (NULL en aquest cas).



Punters a punters

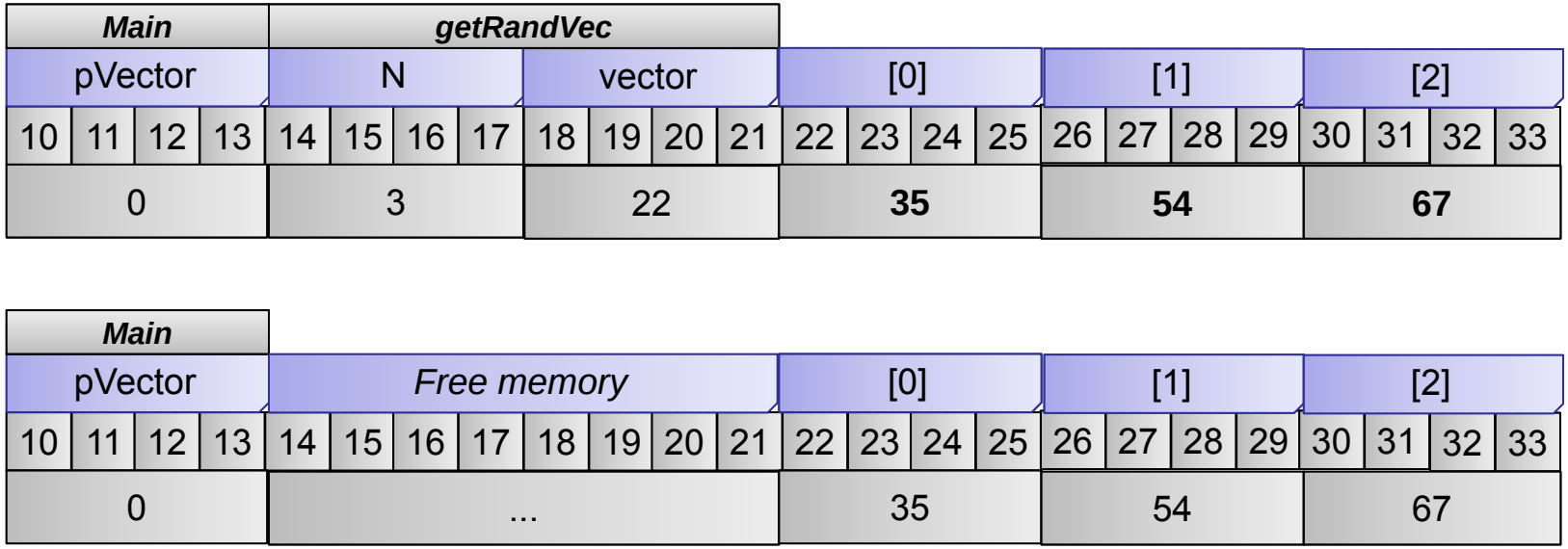
- Només començar es declara una nova variable entera i, que no tindrem en compte per simplicitat, i es reserva un espai de 3 enters, l'adreça resultant es guarda a ***vector***.

```
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				...				...				...			

Punters a punters

- Finalment s'assignen els valors aleatoris a les diferents posicions de memòria (per exemple 35, 54 i 67). En sortir, s'elimina totes les variables creades dins el mètode (les que queden per sota del requadre ***getRandVec***. Fixeu-vos que a les variables de Main no hi ha hagut cap canvi, per tant, pVector continua tenint un valor NULL. Tot i que la memòria continua estant ocupada, no sabem a quina adreça comença.



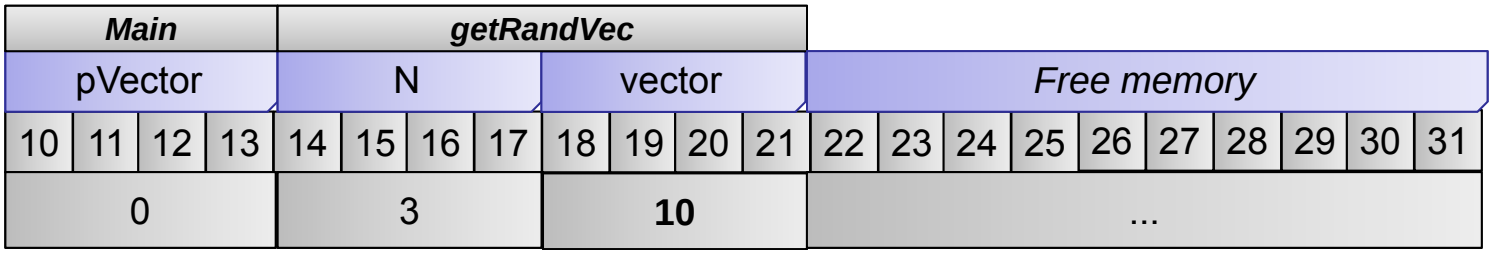
Punters a punters

- El problema està en que el cavi que es fa a l'adreça no repercuteix a la variable del mètode que fa la crida. Per solucionar el problema, cal utilitzar un **punter a punter**. El codi quedaria així:

```
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();
         }
}
```

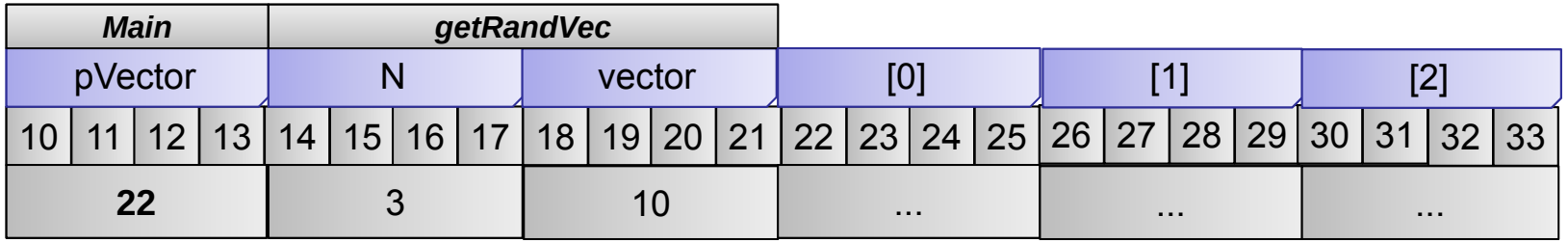
```
int main(void) {
        int *pVector=NULL;
        getRandVec(3,&pVector);
         free(pVector);
        return EXIT_SUCCESS ;
}
```

- Si repetim el procés, però ara considerant els canvis efectuats, tindrem que al començar a executar el mètode ***getRandVec*** la memòria estarà d'aquesta manera (novament es marquen els canvis en negreta):

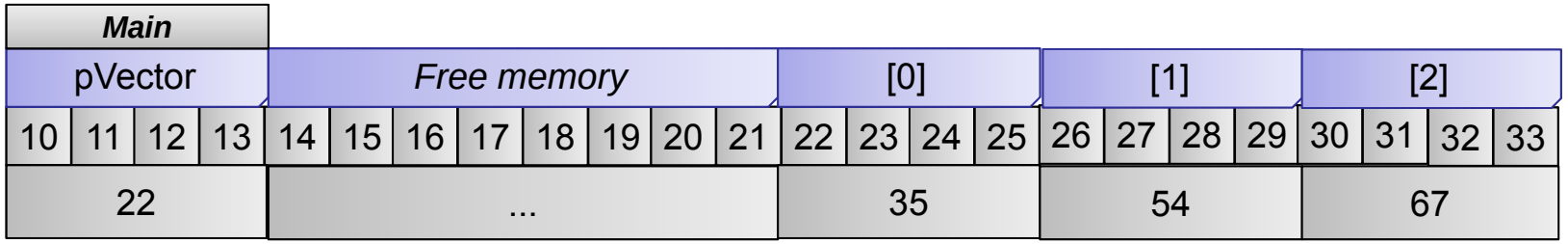


Punters a punters

- En aquest cas, l'adreça de la memòria reservada no es guarda a ***vector***, sinó al **contingut de vector**, i per tant, a l'adreça on ***vector*** apunta (a l'exemple l'adreça 10).



- Al sortir del mètode, s'eliminen les variables locals de ***getRandVec***, i ens queda:

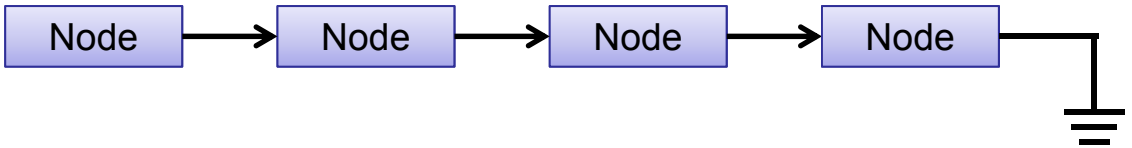


- Per tant podem accedir al vector de nombres aleatoris, ja que sabem que comença a la adreça de memòria 22 (valor de pVector).

Exercicis

Autoavaluació:

1. En els tipus abstractes de dades (TADs), s'acostuma a utilitzar memòria dinàmica i punters. Un dels TADs més simples és la **llista encadenada**. Una llista encadenada és una seqüència d'elements, on cada element "està encadenat" a següent element de la llista, permetent anar des del principi al final de la seqüència. Una possible representació gràfica és la següent, on tenim una sèrie de nodes, on cadascun apunta al seu següent i l'últim apunta a NULL (representat generalment amb el símbol de la figura).



Una forma simple de definir aquests nodes, per tal que puguin contenir qualsevol tipus d'element, és utilitzar punters genèrics. A més, cada node ha d'apuntar al seu següent element, per tant, un node pot estar definit amb una estructura tNode com la següent:

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

Exercicis

Autoavaluació:

Seguint les indicacions anteriors, es demana:

a) Implementa el mètode *addNode*, que donada una llista i un punter a un element, afegeixi al final de la llista el node corresponent. Cal considerar el cas inicial, en que la llista es NULL. La capçalera del mètode ha de ser:

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

b) Implementa el mètode *releaseList*, que donada una llista, elimini tots els seus nodes. Donat que no sabem de quin tipus són els elements, no eliminarem el contingut del node, només el node en si. La capçalera ha de ser:

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

c) Escriu un petit programa que demostri l'ús dels dos mètodes anteriors per guardar una llista de cadenes de caràcters. Crea també un mètode showStrList, que mostri el contingut de la llista. La capçalera d'aquest mètode ha de ser:

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

Exercicis

Autoavaluació (Resposta):

a) Per implementar el mètode addNode, cal considerar els dos casos possibles:

1. *La llista està buida (*list=NULL) i per tant cal afegir el primer node.*
2. *La llista conté algun element. En aquest cas, cal recórrer la llista fins a l'últim element i afegir-hi el nou element. L'últim element és aquell que té com a següent el valor NULL.*

En els dos casos, caldrà reservar memòria per a un nou node i guardar-hi el element obtingut. El codi es mostra a continuació:

```
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;
        }
}
```

Exercicis

Autoavaluació (Resposta):

b) El mètode releaseList, anirà recorrent tots els nodes de la llista i els anirà eliminant. Al final, la llista ha de quedar buida, per tant, el contingut del paràmetre list ha de ser NULL. L'únic que cal tenir en compte és que no podem eliminar un node abans d'haver apuntat al seu successor. En l'enunciat es diu que no hem d'alliberar els elements, pel que assumim que la memòria per aquests elements s'alliberarà en algun altre lloc. El codi es mostra a continuació:

```
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);
         }
}
```

Exercicis

Autoavaluació (Resposta):

c) *Primer definim el mètode per visualitzar la llista. Aquest mètode simplement anirà recorrent la llista i imprimirà el contingut del node:*

```
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;
        }
}
```

Finalment definim un exemple simple d'utilització dels mètodes anteriors. En aquest cas afegim colors a la llista. Donat que els colors els hem definit com a cadenes de caràcters constants, no necessitarem eliminar-ne la memòria.

```
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 ;
}
```