# 4 Gestión dinámica de la memoria

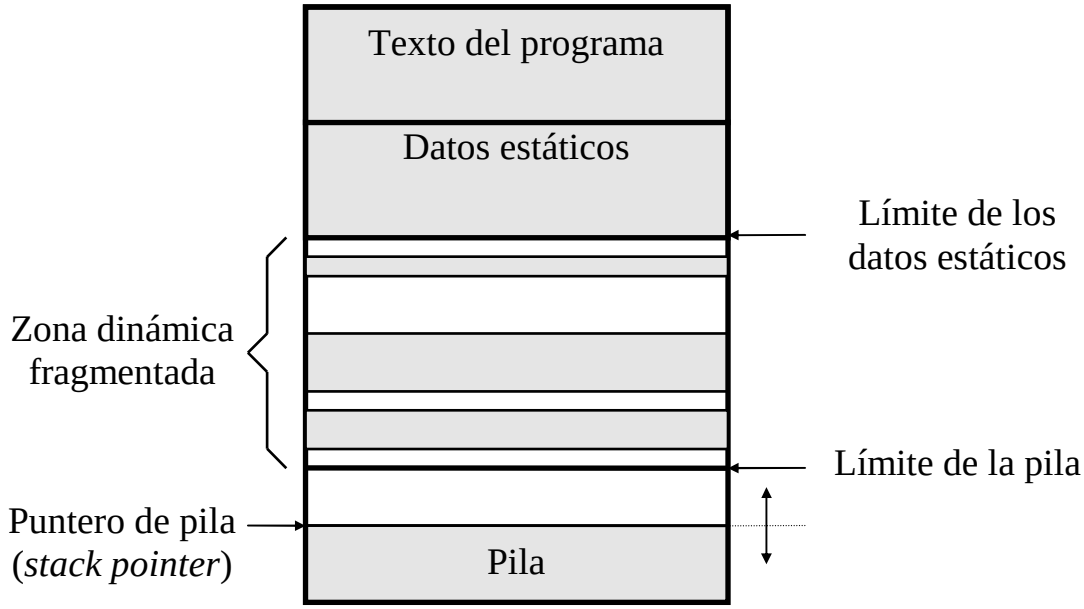
## Contenido

# 1. Introducción.

**Datos estáticos y dinámicos:**

- Datos *estáticos*: su tamaño y forma es constante durante la ejecución de un programa y por tanto se determinan en tiempo de compilación.
- Datos *dinámicos*: su tamaño y forma es variable (o puede serlo) a lo largo de un programa, por lo que se crean y destruyen en tiempo de ejecución. Esto permite dimensionar la estructura de datos de una forma precisa: se va *asignando* memoria en tiempo de ejecución según se va necesitando.

Cuando el sistema operativo carga un programa para ejecutarlo y lo convierte en proceso, le asigna cuatro partes lógicas en memoria principal: texto, datos (estáticos), pila y una zona libre. Esta zona libre (o *heap*) es la que va a contener los datos dinámicos, la cual, a su vez, en cada instante de la ejecución tendrá partes asignadas a los mismos y partes libres que *fragmentarán* esta zona, siendo posible que se agote si no se liberan las partes utilizadas ya inservibles. La pila también varía su tamaño dinámicamente, pero la gestiona el sistema operativo, no el programador.




Para trabajar con datos dinámicos necesitamos dos cosas:

1. **Subprogramas *predefinidos*** en el lenguaje que nos permitan gestionar la memoria de forma dinámica (asignación y liberación). En C estas funciones son **malloc()** y **free()**.
2. Algún tipo de dato con el que podamos acceder a esos datos dinámicos (ya que con los tipos vistos hasta ahora sólo podemos acceder a datos con un tamaño y forma ya determinados). Esto lo resolveremos con **el tipo de dato puntero**.

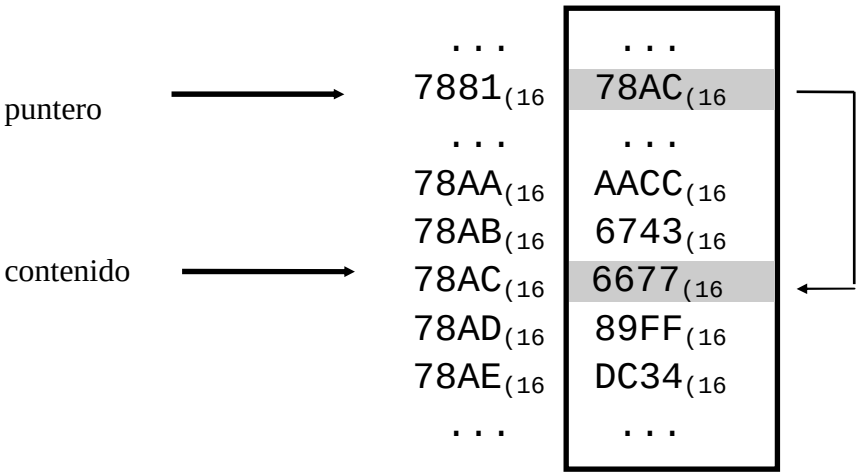
## 2. El tipo puntero y operaciones básicas con punteros.

Las variables de *tipo puntero* nos van a permitir, entre otras cosas, referenciar datos dinámicos. Tenemos que diferenciar claramente entre:

a) la variable referencia o apuntadora, de *tipo puntero*;

b) la variable *anónima* referenciada o apuntada, de cualquier tipo, tipo que estará asociado siempre al puntero.

Físicamente, un puntero no es más que una dirección de memoria. En el siguiente ejemplo se muestra el contenido de la memoria con un puntero que apunta a la dirección $78AC_{(16}$ , la cual contiene $6677_{(16}$:



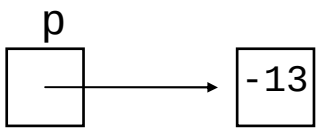

Definiremos un tipo puntero con el carácter asterisco (*) y especificando siempre el tipo de la variable referenciada: Ejemplo:

```
int *p; /* puntero a enteros */
```

O bien podemos definir un tipo y luego declarar la variable :

```
typedef int *PtrInt;    /* define el tipo de datos */
PtrInt p;               /* declara una variable */
```

Cuando `p` esté apuntando a un entero de valor `-13`, gráficamente lo representaremos así:




Para acceder a la variable apuntada hay que hacerlo a través de la variable puntero, ya que aquélla no tiene nombre (por eso es *anónima*). La forma de denotarla es `*p`. En el ejemplo `*p = -13` (y `p` = dirección de memoria de la celda con el valor `-13`, dirección que no necesitamos tratar directamente).

- Ejemplo de punteros a estructuras:

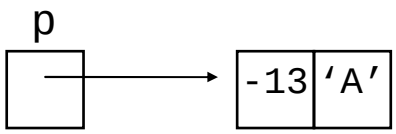
```
struct {
     int num;
     char car;
}TipoRegistro;
```

```
typedef TipoRegistro *TipoPuntero;
TipoPuntero p;
```

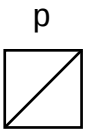
Así:

`p` es la dirección de una estructura con dos campos (tipo puntero)
`*p` es un registro con dos campos (tipo registro)
`(*p).num` es una variable simple (tipo entero)
`p->num` es una variable simple (tipo entero)
`(*p).car` es una variable simple (tipo carácter)
`p->car` es una variable simple (tipo carácter)
`&x` es la dirección de una variable x, siendo x, por ejemplo `int x;`

Gráficamente:




Si deseamos que una variable de tipo puntero (a algo), en un momento determinado de la ejecución del programa no apunte a nada, le asignaremos la palabra reservada `NULL` (`p = NULL`) y gráficamente lo representaremos:




**Operaciones básicas con Punteros**

Al definir una variable de tipo puntero, implícitamente podemos realizar las siguientes operaciones con ella:

1. *Acceso a la variable anónima*
2. *Asignación*
3. *Comparación*
4. *Paso como parámetros*

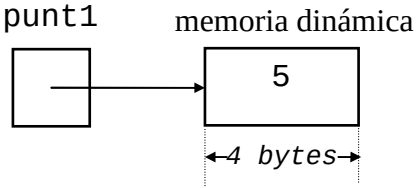
Utilizaremos el siguiente ejemplo para ver estas operaciones:

```
int *punt1, *punt2;
```

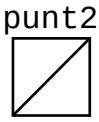
- El *acceso a la variable anónima* se consigue de la forma ya vista con el operador `*`, y la *asignación* se realiza de la misma forma que en los tipos simples. En el ejemplo, suponiendo que `punt1` ya tenga memoria reservada, podemos asignar valores a la variable anónima de la siguiente forma:

  ```
  *punt1 = 5;
  ```

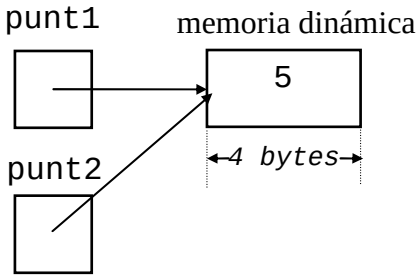
  Tras las asignación tendremos (suponiendo que un entero ocupa 4 bytes):



A una variable puntero también se le puede asignar la palabra reservada `NULL` (sea cual sea el tipo de la variable anónima): `punt2 = NULL;`, tras lo cual tendremos:




Se puede asignar el valor de una variable puntero a otra siempre que las variables a las que apuntan sean del mismo tipo: `punt2 = punt1;`




- La *comparación* de dos punteros, mediante los operadores `!=` y `==`, permite determinar si apuntan a la misma variable referenciada:

```
if (punt1 == punt2) /* expresión lógica */
```

o si no apuntan a nada:

```
if (punt1 == NULL)   /* expresión lógica */
```

- El *paso de punteros como parámetros* permite el paso de variables por refencia a funciones. Ejemplo:

```
#include <stdio.h>
void llamame(int *p);
void main() {
     int x;

     x=0;
     printf("El valor de x es %d\n", x);
     llamame(&x);
     printf("El nuevo valor de x es %d\n", x);
}
void llamame(int *p){
     *p=5;
}
```

## 3. Asignación y liberación dinámica de memoria.

Las operaciones básicas de gestión dinámica de la memoria son dos: **asignar** y **liberar** memoria. Para asignar memoria de forma dinámica existen tres funciones en lenguaje C y una para liberar memoria:

✓ **`malloc():`** Asigna un cierto espacio de memoria.

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

***Parámetros:***

*size:* Número de bytes que se asignan.

***Valor que devuelve:***

**malloc** devuelve un puntero a void que apunta al espacio de memoria asignado o **NULL** si no hay suficiente memoria disponible. Para devolver un puntero a otro tipo distinto de **void**, se debe usar un molde (*cast*) sobre el valor devuelto. Es conveniente comprobar siempre el valor devuelto por **malloc**, incluso si la cantidad de memoria que se ha pedido es pequeña.

- ✓ **realloc():** Cambia el tamaño del espacio de memoria previamente asignado.

```
Void *realloc(void *memblock,size_t size);
```

***Parámetros:***

*memblock:* Puntero a un bloque de memoria previamente asignado. Si es NULL, realloc se comporta como malloc. Si no es NULL, debería ser un puntero que ha sido devuelto por una llamada anterior a malloc o realloc.

*size:* Nuevo tamaño en bytes.

***Valor que devuelve:***

realloc devuelve un puntero a void que apunta al espacio de memoria reasignado (y posiblemente movido). Si no hay suficiente memoria para expandir el bloque al nuevo tamaño, el bloque original se deja sin cambiar y se devuelve NULL.

- ✓ **calloc():** Asigna espacio de memoria para un array de num elementos, cada uno de tamaño size bytes. Además cada elemento se inicializa a 0.

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

***Parámetros:***

*num:* Número de elementos.

*size:* Número en bytes de cada elemento.

***Valor que devuelve:***

calloc devuelve un puntero a void que apunta al espacio de memoria asignado.

- ✓ **free():** Libera el espacio de memoria asignado.

```
void free(void *memblock);
```

***Parámetros:***

*memblock:* Puntero a un bloque de memoria previamente asignado y que se quiere liberar.

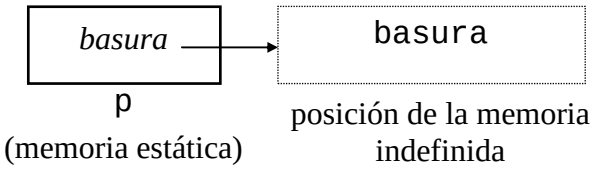
Todas las funciones anteriores están en malloc.h y en stdlib.h.

**EJEMPLO: Asignación de memoria dinámica a un entero.**

Cuando declaramos una variable de tipo puntero, por ejemplo

```
int *p;
```

estamos creando la variable `p`, y se le reservará memoria –estática– en tiempo de compilación; pero la variable referenciada o anónima no se crea. En este momento tenemos:

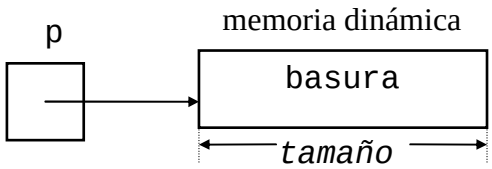



La variable anónima debemos crearla después mediante una llamada a la función **`malloc(),`** que se utiliza para asignar memoria dinámica en tiempo de ejecución. `malloc()` recibe el número de bytes que necesitan ser asignados si se encuentra memoria disponible. Si no hay memoria disponible para satisfacer la operación, `malloc()` devuelve un puntero a *NULL*.

Así:

**`p=(int *)malloc(sizeof(int));`**

donde `p` es una variable de tipo puntero y `sizeof(int)` es el tamaño en bytes del tipo de la variable anónima. En tiempo de ejecución, después de la llamada a este procedimiento, tendremos ya la memoria (dinámica) reservada pero sin inicializar:




Tras usar la memoria asignada a un puntero, hay que liberarla con la función **`free().`**

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

void main ()
{
   int a,*b;
  b=(int *)malloc(sizeof(int)); // Reserva Mm para un entero
  if(b==NULL) //SIEMPRE hay que comprobar que se le ha asignado al
              // puntero la memoria requerida. En otro caso se puede
              // estar escribiendo en memoria no asignada.
       printf("No hay memoria suficiente.");
  else{
    a=5;
    *b=6;
    printf("Dirección de a: %p. Valor de a: %d\n", &a,a);
    printf("Dirección b: %p. Valor al que apunta: %d\n", b, *b);
    free(b); // Es importante liberar la memoria
  }
}
```

## 4. Arrays dinámicos

Hasta ahora hemos trabajado con vectores y matrices estáticos, es decir, su tamaño queda establecido antes de la ejecución del programa, y permanece igual durante toda la ejecución.

Por ejemplo, supongamos las siguientes declaraciones:

```
#define MAX 20
int valores[MAX];
int num_elem;
```

El vector `valores` queda definido como un vector de 20 números enteros. Durante la ejecución del programa el vector puede tener menos de 20 valores. Por ello, usualmente disponemos de una variable entera (en este caso `num_elem`), que nos dice cuántos valores hay en cada momento en el vector. Si tenemos que recorrer todos los valores del vector, lo haríamos asi:

```
for (i=0; i<num_elem; i++)
    …
```

Este planteamiento tiene dos problemas:

a) Si durante todo el programa tenemos menos de 20 valores en el vector, estamos usando más espacio de memoria del estrictamente necesario (porque estamos usando espacio para 20 enteros, aunque necesitemos menos)
b) Si durante la ejecución del programa tenemos más de 20 valores, no podemos almacenarlos porque no hay espacio suficiente en el vector

El problema (a) no es especialmente grave a menos que el espacio ocupado por el vector o la matriz sea muy grande (hoy en día el espacio de memoria no es un problema grave en aplicaciones normales). En cambio, el problema (b) si que puede ser grave.

La forma de resolver esta situación es trabajar con vectores y matrices cuyo tamaño se define durante la ejecución, pudiendo crecer y decrecer para adaptarse a las necesidades de la aplicación. Para poder hacer esto, necesitamos operaciones de gestión dinámica de la memoria.

Veamos cómo se declaran y utilizan arrays dinámicos con varios ejemplos.

**EJEMPLO 1: Un array dinámico de enteros.**

El siguiente código construye un vector de enteros del tamaño que indica el usuario por teclado, rellena el vector con los datos que introduce el usuario, amplia el tamaño del vector y sigue rellenando, y finalmente, libera el espacio de memoria ocupado por el vector. Las operaciones clave han sido resaltadas en negrita.

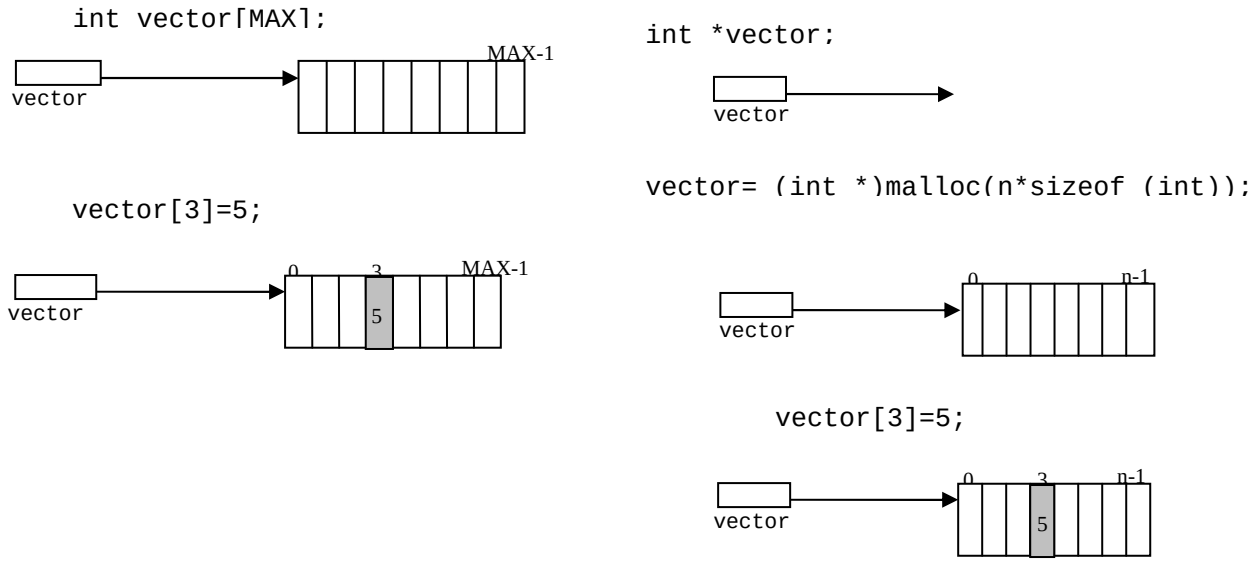
Como puede observarse, para trabajar con un vector dinámico de enteros, los primero que hay que hacer es declarar un puntero a un entero (en el ejemplo, el puntero se llama `vector`). Ese puntero, que inicialmente apunta a cualquier parte, deberá apuntar al sitio de memoria en que comienza el vector de enteros. Cuando hemos averiguado cuántos enteros debe contener el vector (en nuestro ejemplo `num_elem`), entonces usamos la función `malloc` para pedirle al computador que nos busque espacio de memoria suficiente para albergar un vector de ese tamaño. El parámetro de la función `malloc` justamente es el espacio de memoria necesario, y que en este caso es el espacio equivalente a `num_elem` datos cada uno de tamaño igual al que ocupa un número entero. La función `malloc` nos da la dirección de memoria donde comienza el espacio que ha reservado, y

asignamos esa dirección al puntero `vector`. Fijate que la palabra `malloc` está precedida por `(int *)` para indicar simplemente que la función `malloc` retornará un apuntador a un entero.

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

void main ()
{
  // declaramos un puntero a un entero
  int *vector;
  int num_elem=0;
  int i, mas;

  printf ("Escribe el numero de elementos del vector\n");
  scanf ("%d",&num_elem);

  // Ahora se el tamaño inicial del vector. Reservo espacio de memoria
  vector = (int *) malloc (num_elem*sizeof (int));
  if (vector == NULL)
       printf ("Operacion incorrecta");
  else
  {
       // cargo el vector con los datos leidos del teclado
       for (i=0; i<num_elem; i++)
       {
            printf ("Escribe siguiente elemento:\n ");
            scanf ("%d",&vector[i]);
       };

       // Añado más elementos
       printf ("¿Cuantos elementos quieres añadir?\n");
       scanf ("%d",&mas);

       // reajusto el tamaño del vector
       vector = (int *) realloc (vector, (num_elem+mas)*sizeof (int));
       if (vector == NULL)
            printf ("Operacion incorrecta");
       else
       {
            for (i=num_elem; i<num_elem+mas; i++)
            {
                 printf ("Escribe siguiente elemento:\n ");
                 scanf ("%d",&vector[i]);
            };
            num_elem = num_elem + mas;
       }
  }
  // ya no necesito el vector y libero el espacio de memoria
  free (vector);
}
```

Si la función `malloc` ha tenido problemas para buscar el espacio de memoria (por ejemplo, no ha encontrado espacio suficiente) entonces `malloc` retorna el valor `NULL`. Debemos, por tanto, asegurarnos de que la operación de reserva de memoria ha funcionado bien antes de empezar a usar el vector.

Si la operación de reserva de memoria ha funcionado bien, entonces podemos ya usar el vector exactamente igual que si fuese un vector estático, con la diferencia de que ahora el vector ocupa exactamente el espacio que ha elegido el usuario una vez iniciada la ejecución del programa.




La figura compara el funcionamiento de los vectores estáticos y dinámicos. A la izquierda se muestra cómo al declarar un vector estático el computador automáticamente reserva espacio en memoria según el tamaño establecido en la declaración, y crea la variable que apunta a ese espacio, para poder acceder después durante le programa. A la derecha se muestra como, en el caso de los vectores dinámicos, primero se crea la variable que apuntará al vector, después se reserva el espacio justo que se necesite, se hace que la variable apunte al espacio y se accede al vector con normalidad.

En la segunda parte del programa vemos qué hay que hacer en el caso de que necesitemos ampliar el tamaño del vector. Lo que haremos es usar la función `realloc`, a la que pasamos como parámetro el apuntador al inicio del vector cuyo tamaño queremos cambiar, y el nuevo tamaño que queremos que tenga ese vector. El computador buscará espacio para el nuevo vector y nos devolverá la dirección del sitio donde estará el vector. Como es lógico, las `num_elem` primeras posiciones del vector ampliado contendrán la misma información que había en el vector antes de la ampliación.

Finalmente, cuando ya he terminado de trabajar con el vector, podemos llamar a la función `free` para liberar el espacio de memoria ocupado por el vector, de forma que ese espacio pueda ser usado para otras operaciones.

**EJEMPLO 2: Un array de números reales que se reserva elemento a elemento.**

El siguiente programa lee y escribe los valores de un array de reales. El número de valores se conoce durante la ejecución. La memoria no se reserva toda a la vez sino elemento a elemento.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h> // para getch()

void main ()
{
   float *V=NULL;
  int N=0,i; char c;
  do{
    V=(float *)realloc((float *)V,(N+1)*sizeof(float));
    printf("Dame un valor>> "); scanf("%f",&V[N]);
    printf("Quieres introducir otro valor? (S/N >> ");
    c=getch();
    N++;
  }while(c=='S' || c=='s');
  for(i=0;i<N;i++) printf("\nValor %d >> %f\n",i,V[i]);
  free(V);
}
```

**EJEMPLO 3: Un array dinámico de estructuras.**

Veamos ahora un ejemplo más complejo, que trabaja con un vector de estructuras. El programa es muy parecido al ejemplo 1. Definimos primero algunos tipos de datos. El tipo `TPersona` es una estructura que contiene un nombre y una edad. Después definimos el tipo `Tlista`, que es un vector dinámico de personas. Un campo es el número de personas que tenemos en el vector, y el otro campo es el apuntador al inicio del vector.

En el programa reservamos espacio para un cierto número de personas, indicado por el usuario. Fíjate que ahora el espacio necesario corresponde al número de personas multiplicado por el tamaño que ocupa una persona (`sizeof (Tpersona)`). El resto de las operaciones del programa se entienden fácilmente.

**Ejercicios**:

1. **Construye un proyecto con el programa ejemplo y ejecútalo para comprobar que funciona correctamente. Ejecútalo después paso a paso y observa en la ventana del watch los cambios que se producen en la estructura** `mi_lista`**. Justo antes de la sentencia free observa en la ventana del watch todos los elementos del vector, para asegurarte de que los datos que has introducido se han almacenado correctamente en el vector.**

2. **Añade al programa el código necesario para ampliar el vector de personas (preguntar al usuario cuántas personas más quiere, ampliar el tamaño del vector y leer los datos de las nuevas personas.**

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

typedef char Tpalabra[20];
typedef struct {
  Tpalabra nombre;
  int edad;
} Tpersona;

typedef struct {
  int num_personas;
  Tpersona *personas;     // apuntador al vector dinámico de personas
} Tlista;

void main ()
{
  Tlista mi_lista;
  int i;


  printf ("Escribe el numero de personas del vector\n");
  scanf ("%d",&mi_lista.num_personas);

  // Ahora se el tamaño inicial del vector. Reservo espacio de memoria
  mi_lista.personas = (Tpersona *) malloc (mi_lista.num_personas*sizeof (Tpersona));
  if (mi_lista.personas == NULL)
       printf ("Operacion incorrecta");
  else
  {
       // cargo el vector con los datos leidos del teclado
       for (i=0; i<mi_lista.num_personas; i++)
       {
            printf ("Escribe el nombre:\n ");
            scanf ("%s",mi_lista.personas[i].nombre);
            printf ("Escribe la edad:\n ");
            scanf ("%d",&mi_lista.personas[i].edad);
       };

       // Añado más elementos


       // cuando termino de trabajar con el vector, libero el espacio de memoria
       free (mi_lista.personas);
  }
}
```

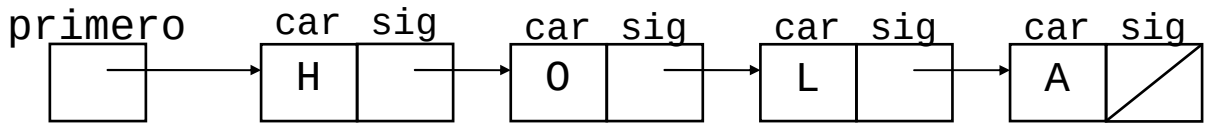
## 5. Listas enlazadas con punteros.

Lo que realmente hace de los punteros una herramienta potente es la circunstancia de que pueden apuntar a variables que a su vez contienen punteros. Cuando en cada variable anónima o *nodo* tenemos un solo puntero que apunta al siguiente nodo tenemos una *lista enlazada.*

**EJEMPLO: cadenas de caracteres de longitud variable.**

```
typedef struct nodo{
  char car;              /* Carácter */
  struct nodo *sig;   /* Al siguiente nodo */
}TipoRegistro;

TipoRegistro *primero;
```

Con estos tipos de datos podemos crear estructuras que no malgastan la memoria, y que sí malgastaría un array de longitud fija:




**EJEMPLO: lista enlazada de números enteros.**

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

typedef struct nodo
{
int dato;
struct nodo *enlace;
} TipoNodo;
 typedef TipoNodo *LISTA;

void mostrar_lista(LISTA ptr);
void insertar(LISTA& ptr, int elemento);

void main(){
  LISTA n1 = NULL;
  int elemento;

   // Inserta elementos hasta leer el cero
  do {
    printf("\nIntroduzca elemento: ");
    scanf(“%d”, &elemento);
    if(elemento != 0)
      insertar(n1, elemento);
  } while(elemento != 0);
  printf("\nLa nueva lista enlazada es: ");
 mostrar_lista(n1);
}
```

```
void mostrar_lista(LISTA ptr){
   while(ptr != NULL){
     printf("%d",ptr->dato);
     ptr = ptr->enlace;
   }
   printf("\n");
}

void insertar(LISTA& ptr, int elemento){
   LISTA p1, p2;

   p1 = ptr;
   if(p1 == NULL){
     p1 = malloc(sizeof(TipoNodo));
     if (p1 != NULL){
        p1->dato = elemento;
        p1->enlace = NULL;
        ptr = p1;
     }
   }
   else{
     while(p1->enlace != NULL)
        p1 = p1->enlace;
     p2 = malloc(sizeof(TipoNodo));
     if(p2 != NULL){
        p2->dato = elemento;
        p2->enlace = NULL;
        p1->enlace = p2;
     }
   }
 }
```

## Inserción y eliminación de nodos en una lista enlazada con punteros

A continuación se describen de forma detallada las siguientes operaciones sobre listas enlazadas:

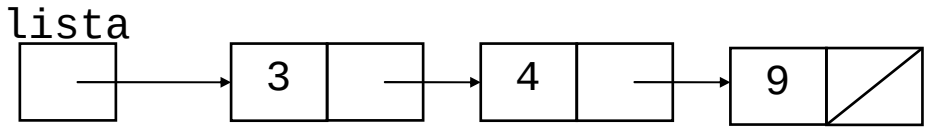
a) *Inserción de un nodo al principio*
b) *Insertar un nodo en una lista enlazada ordenada*
c) *Eliminar el primer nodo*
d) *Eliminar un nodo determinado*

Vamos a basarnos en las declaraciones de una lista enlazada de enteros:

```
typedef struct nodo{
    int dato;               /* Elemento entero */
    struct nodo *sig;   /* Al siguiente nodo */
}TipoNodo;

typedef TipoNodo *TipoLista;

TipoLista lista; /* Cabeza de la lista */
TipoLista nodo;  /* Nuevo nodo a insertar */
```

```
TipoLista ptr;   /* Puntero auxiliar */
```

Y partiremos del estado inicial de la lista:




- ***Inserción de un nodo al principio***

Los pasos a dar son los siguientes:

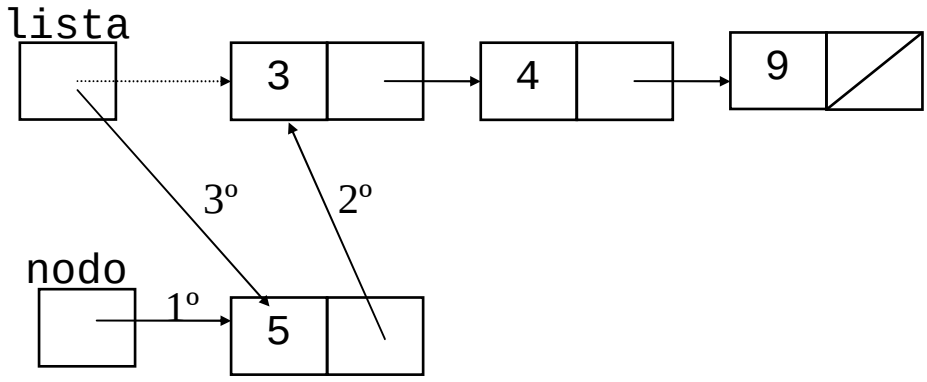
1) Reservar memoria para el nuevo nodo:
```
if ((nodo=(TipoLista) malloc(sizeof(TipoNodo)))==NULL)...
```

2) Asignar valor nuevo al nuevo nodo:
```
nodo->dato = 5 /* por ejemplo */
```

3) Enlazar la lista al siguiente del nuevo nodo:
```
nodo->sig = lista
```

4) Actualizar la cabeza de la lista:
```
lista = nodo
```

Gráficamente (la línea discontinua refleja el enlace anterior a la operación, y también se especifica el orden de asignación de valores a los punteros):




- ***Inserción de un nodo en una lista ordenada (ascendentemente)***

Los pasos a seguir son los siguientes:

1) Reservar memoria para el nuevo nodo:
```
if ((nodo=(TipoLista) malloc(sizeof(TipoNodo)))==NULL)...
```

2) Asignar valor nuevo al nuevo nodo:

```
nodo->dato = 5 /* por ejemplo */
```

3.a) Si la lista está vacía o el dato nuevo es menor que el de la cabeza, el nodo se inserta al principio y se actualiza la cabeza `lista` para que apunte al nuevo nodo.

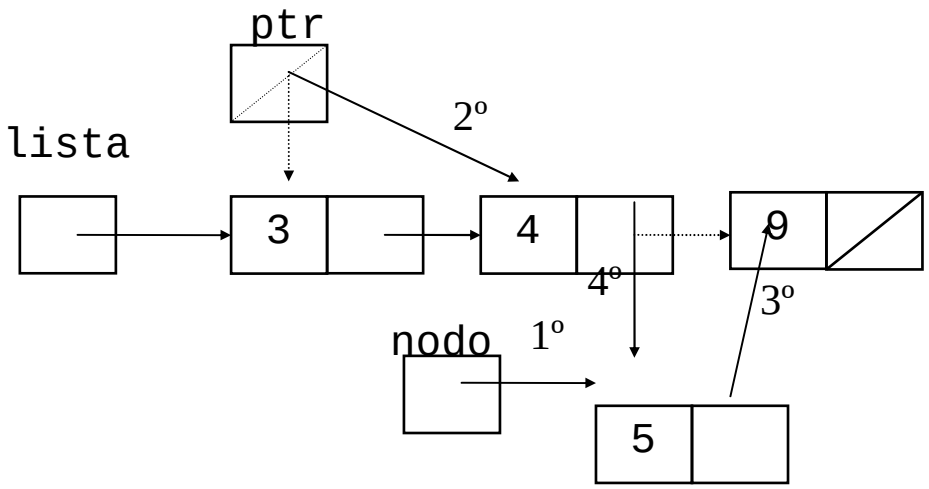
3.b) Si la lista no está vacía y el dato nuevo es mayor que el que hay en la cabeza, buscamos la posición de inserción usando un bucle del tipo:

```
while ((ptr->sig != NULL) &&
          (nodo->dato > (ptr->sig)->dato))
   ptr = ptr->sig;
/* ptr queda apuntando al predecesor */
```

y enlazamos el nodo:

```
nodo->sig = ptr->sig    /* con el sucesor */
ptr->sig = nodo     /* con el predecesor */
```

Gráficamente (observar el orden de asignación de punteros –el segundo paso son las sucesivas asignaciones al puntero `ptr`):




- ***Eliminar el primer nodo***

Los pasos son (suponemos lista no vacía):

1) Guardar el puntero a la cabeza actual:

```
ptr = lista
```

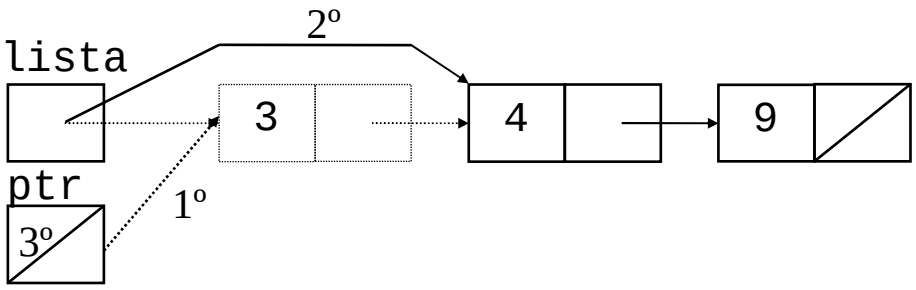
2) Avanzar una posición la cabeza actual:

```
lista = lista->sig
```

3) Liberar la memoria del primer nodo:

```
free(ptr)
```

Gráficamente (el tercer paso pone `ptr` a `NULL`):




- ***Eliminar un nodo determinado***

  Vamos a eliminar el nodo que contenga un valor determinado. Los pasos son los siguientes (suponemos lista no vacía):

  Caso a) El dato coincide con el de la cabeza:

  Se elimina como en la operación anterior.

  Caso b) El dato no coincide con el de la cabeza:

  1) Se busca el predecesor y se almacena en una variable de tipo puntero, por ejemplo, en `ant`. En otra variable, `ptr`, se almacena el nodo actual:

```
ant = lista;
ptr = lista->sig;
while ((ptr != NULL) &&
         (ptr->dato != valor)){
   ant = ptr;
   ptr = ptr->sig;
}
```

  2) Se comprueba si se ha encontrado, en cuyo caso se enlaza el predecesor con el sucesor del actual:

```
if (ptr != NULL){ /* encontrado */
   ant->sig = ptr->sig;
   free(ptr);
}
```

  Gráficamente quedaría (eliminando el nodo de valor 4):