

## Contenido

1. Tipos de Datos Dinámicos: Punteros .....	3
1.1. Introducción.....	3
1.2. Variables. Nombres. Direcciones de memoria .....	3
1.2.1 El operador de dirección & .....	3
1.3. Concepto de puntero.....	4
1.4. Declaración de punteros.....	5
1.5. Asignación de valores a un puntero .....	5
1.6. Indirección de un puntero. Operador * .....	6
1.7. Inicialización de punteros .....	6
1.7.1. Los punteros NULL y void.....	7
1.7.2. Asignación de una dirección de memoria válida a una variable puntero .....	7
1.7.3. Asignación dinámica de memoria: función malloc .....	8
1.7.4. Liberación de espacio de memoria: función free .....	8
1.7.5. Asignación de memoria .....	9
1.7.6. Liberar memoria no cambia el valor del puntero .....	11
1.7.7. Alias.....	12
1.8. Estructuras y punteros.....	12
1.8.1. Campos de tipo puntero .....	12
1.8.2. Punteros a estructuras.....	13
1.9. Funciones y punteros.....	13
1.9.1. Punteros como argumentos de funciones.....	13
1.9.2. Punteros a datos locales .....	13
1.9.3. Punteros a punteros .....	15
2. Listas enlazadas .....	16
2.1. Listas simplemente enlazadas .....	16
2.1.1. Estructura de una lista enlazada.....	17
2.1.2. Añadir un nodo al inicio o cabeza de la lista:.....	17
2.1.3. Recorrer una lista.....	19
2.1.4. Añadir un nodo al final de la lista: .....	20
2.1.5. Insertar un nodo en una posición P de la lista.....	21
2.1.6. Eliminar el primer nodo de la lista (TP) .....	23
2.1.7. Eliminar el último nodo de la lista .....	24

2.1.8.	Eliminar un elemento de una posición P de la lista .....	25
2.1.9.	Implementación de las operaciones de listas simplemente enlazadas con recursión .....	26

## Unidad 7

### Tipos de datos dinámicos: Punteros

Temario: Asignación dinámica de memoria. Uso de punteros. Inicialización y asignación de punteros. Procedimientos para asignación y liberación de memoria. Tipos de datos recursivos. Listas enlazadas con punteros

#### Bibliografía:

- [1] Programación en C Metodología Algoritmos y Estructura de datos. McGraw-Hill. JOYANES\_AGUILAR
- [2] Introducción a C. Andrés Marzal e Isabel Gracia.
- [3] [http://sopa.dis.ulpgc.es/so/cpp/intro\\_c/](http://sopa.dis.ulpgc.es/so/cpp/intro_c/)

## 1. Tipos de Datos Dinámicos: Punteros

### 1.1.Introducción

Los punteros en C tienen la fama, en el mundo de la programación, de su dificultad, tanto en el aprendizaje como en su uso. En este capítulo trataremos de mostrar que los punteros no son más difíciles de aprender que cualquier otra herramienta de programación. El puntero, no es más que una herramienta muy potente que se utiliza para hacer más eficientes y flexibles los programas. Los punteros son, sin lugar a dudas, una de las razones fundamentales para que C sea tan potente y tan utilizado.

Una *variable puntero* (o *puntero*, como se llama normalmente) es una variable que contiene direcciones de memoria. Todas las variables vistas hasta este momento contienen valores de datos, por el contrario los punteros contienen valores que son direcciones de memoria donde se almacenan datos.

### 1.2.Variables. Nombres. Direcciones de memoria

Una variable, repasando lo visto en la unidad 2, puede ser caracterizada por seis atributos: nombre, dirección de memoria, valor almacenado, tipo de dato asociado, tiempo de vida y ámbito. Los atributos relevantes para el tema que se está estudiando son nombre, tipo de dato y dirección de memoria. Cuando la declaración de una variable se realiza de forma estática (como se vino trabajando hasta el momento), el compilador antes de la ejecución del programa, enlaza estos tres atributos con la variable. Para esto debe reservar memoria RAM suficiente para almacenar el tipo de dato asociado a esa variable. Cada vez que ejecuta el programa se asociará a la variable una dirección de memoria; puede ser la misma dirección, u otra y esto dependerá de la memoria disponible y de otros varios factores. Dependiendo del tipo de variable que se declare, se reservará más o menos memoria. Como ya se vio anteriormente, cada tipo de variable ocupa más o menos bytes de memoria. Por ejemplo, si se declara un `char`, se reserva 1 byte (8 bits). Cuando finaliza el programa todo el espacio reservado queda libre.

 [http://maxus.fis.usal.es/FICHAS\\_C.WEB/00xx\\_PAGS/0004.html](http://maxus.fis.usal.es/FICHAS_C.WEB/00xx_PAGS/0004.html)

#### 1.2.1 El operador de dirección &


Para averiguar la dirección de memoria asociada a una variable se utiliza el operador & (operador de dirección).

### Ejemplo 1:

```
1.  #include <stdio.h>
2.  int main () {
3.      int numero = 10 ;
4.      printf ("Direccion de numero = %p\n" , &numero);
5.      printf ("valor de numero = %i\n" , numero);
6.      return 0 ;
7.  }
```

Para mostrar la dirección de la variable se usa el modificador de tipo %p en lugar de %i. Sirve para escribir direcciones de punteros y variables. El valor se muestra en formato hexadecimal.

No hay que confundir el valor de la variable con la dirección donde está almacenada. La variable `numero` está almacenada en un lugar determinado de la memoria y ese lugar no cambia mientras se ejecuta el programa. El valor de la variable puede cambiar a lo largo del programa. Ese valor está almacenado en la dirección de la variable. El nombre (identificador) de la variable es ponerle un nombre a la zona de la memoria asignada por el compilador. Cuando en el programa se escribe `numero`, en realidad se está diciendo, "el valor que está almacenado en la dirección de memoria a la que se llama `numero`".

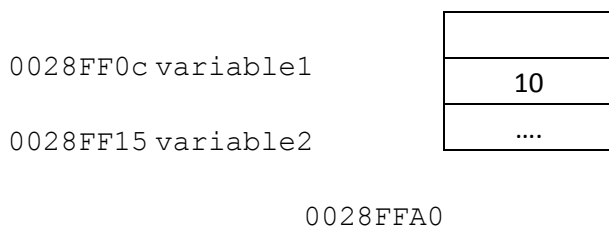
 Capítulo 10 Sección *Direcciones de Memoria*. Pág. 324. *Programación en C Metodología Algoritmos y Estructura de datos*. McGraw-Hill. JOYANES\_AGUILAR.

## 1.3. Concepto de puntero

El concepto de punteros tiene correspondencia en la vida diaria. Cuando se envía un mail, su información se entrega basada en un puntero que es la dirección de mail utilizada. Cuando se llama por teléfono a una persona, se utiliza un puntero (el número de teléfono que se marca). De esta manera, una dirección de correo y un número de teléfono tienen en común que ambos indican dónde encontrar algo. Son punteros a casillas de correo en un determinado servidor y teléfonos fijos, respectivamente. Un puntero en C también indica dónde encontrar algo, más específicamente dónde encontrar determinados datos dentro de la memoria. Un puntero contiene la dirección de memoria en donde se almacena un dato en un determinado momento de la ejecución de un programa.


Una variable de tipo puntero tiene asociados los mismos atributos que cualquier otra tipo de variable en un programa: nombre, dirección de memoria, valor, etc. Así como en la declaración `int variable1=10` se asocian todos los atributos de una variable, en la declaración de un puntero, supongamos llamado `variable2`, también se asocian. En el primer caso se asocia el nombre `variable1` a una dirección de memoria determinada por el compilador, podría ser `0028FF0c`, que permite almacenar un tipo de dato entero, en este caso `10`. En el caso de la variable de tipo puntero, se asocia un nombre, en este caso `variable2`, a una dirección de memoria que permite guardar como dato en vez de un número, cadena, registro, etc. (como en este caso el `10`), una dirección de memoria en donde efectivamente se almacena un dato.

Dirección	Memoria	Nombre
	10	
	0028FFA0	



Los punteros se rigen por estas reglas básicas:

- un puntero es una variable como cualquier otra;
- una variable puntero contiene una dirección que apunta a otra posición en memoria;
- en esa posición se almacenan los datos a los que apunta el puntero;

 Capítulo 10 Sección *Concepto de puntero (Apuntador)*. Pág. 325. *Programación en C Metodología Algoritmos y Estructura de datos*. McGraw-Hill. JOYANES\_AGUILAR.  
[http://platea.pntic.mec.es/vgonzale/cyr\\_0204/cyr\\_01/control/lengua\\_C/punteros.htm#que\\_son](http://platea.pntic.mec.es/vgonzale/cyr_0204/cyr_01/control/lengua_C/punteros.htm#que_son)

## 1.4. Declaración de punteros

Como se mencionó anteriormente, las variables de tipo puntero almacenan direcciones de memoria en donde se almacenan datos. Estos datos pueden ser de distinto tipo, numéricos, cadenas, vectores, registros, etc. Cada uno de estos tipos requiere de diferente espacio de memoria para poder ser almacenados, por lo tanto cuando se declara una variable de tipo puntero, la dirección de memoria que va a almacenar ese puntero debe ser capaz de albergar el tipo de dato requerido. Por esto al declarar un puntero se debe especificar el tipo de dato al que apuntará la variable puntero.

**Sintaxis:** <tipo\_dato> \* <nombre\_variable>

### Ejemplo 2:

```
int * num_ent;           /*num_ent puntero a entero*/  
float * num_real;        /*num_real puntero a punto flotante*/  
char * caracter;         /*caracter puntero a char*/
```

## 1.5. Asignación de valores a un puntero

Los punteros sólo pueden almacenar direcciones de memoria, por lo tanto sólo se podrán asignar direcciones a variables de tipo puntero. Se debe recordar que las direcciones almacenadas por variables de tipo puntero dependerán de los tipos de datos asociados al puntero, por lo tanto un puntero a entero sólo podrá contener una dirección de memoria que pueda almacenar un número entero, lo mismo ocurre con un puntero a char, a float o a cualquier otro tipo de dato. Para poder hacer estas asignaciones se utiliza el operador & de dirección.

### Ejemplo 3:

```
1.  #include <stdio.h>  
2.  int main () {  
3.  int num ;
```

```
4.     int * puntEntero;
5.     puntEntero = &num;
6.     printf ("direccion de num = %p\n" , &num);
7.     printf ("direccion de punt_num = %p\n" , puntEntero);
8.     }
```

## 1.6. Indirección de un puntero. Operador \*

La acción de obtener el valor al que apunta un puntero se denomina *indireccionar* el puntero (desreferenciar el puntero); para ello, se utiliza el operador de indirección \*.

### Ejemplo 4:

```
1.     #include <stdio.h>
2.     int main () {
3.         int num = 25 ;
4.         int * puntEntero ;
5.         puntEntero = &num ;
6.         printf ( "direccion de num = %p\n" , &num );
7.         printf ( "valor de num = %d\n" , num );
8.         printf ( "direccion de punt_num = %p\n" , puntEntero );
9.         printf ( "valor de punt_num = %d\n" , *puntEntero );
10.        return 0 ;
11.    }
```



Capítulo 10 Sección *Indirección de punteros*. Pág. 328. *Programación en C Metodología Algoritmos y Estructura de datos*. McGraw-Hill. JOYANES\_AGUILAR.

## 1.7. Inicialización de punteros

Se puede pensar a un puntero como una flecha. Esa flecha apunta a una dirección de memoria. Por ejemplo, si se declara el puntero:

```
char *puntero;
```

Se declara una "flecha" que apunta a una dirección de memoria que debe poder albergar un valor de tipo char. ¿A cual?. Aquí se presenta el primer problema práctico con los punteros. Tal cual como se declaró anteriormente, esa flecha apunta a cualquier dirección de memoria, al azar. Lo habitual es que sea la dirección de memoria 0 (cero), pero puede ser cualquiera.

Si inmediatamente después de declarar el puntero se intenta guardar algo en la dirección de memoria a la que apunta, como por ejemplo `*puntero = 'A'`; pueden ocurrir dos cosas:

- Que la dirección aleatoria a la que apunta puntero pertenezca al programa. En ese caso se guardará la 'A' en la dirección y aparentemente no habría pasado nada. Lo que realmente ocurre es que se está almacenando un byte, el número 65 (código ascii de la 'A') en algún sitio de la memoria asignada al programa (código, zona de variables, etc.). Al no poder controlar en qué sector de la memoria asignada al programa se almacenó el byte, puede presentarse un error en cualquier otro lado del programa, en un sitio aparentemente correcto.

- Que la dirección aleatoria a la que apunta puntero no pertenezca al programa. En el momento de la asignación el programa se "colgará" y dará un error de violación de memoria o similar.

Por esto el primer consejo práctico es:

Inicializar todos los punteros al declararlos utilizando la constante `NULL`

Ejemplo: `char *puntero = NULL;`


De esta manera, si no se asigna a la variable puntero una dirección de memoria adecuada, el programa dará error en el momento de utilizarla por primera vez, y no después, en otro lugar del programa. Resumiendo, un puntero nulo se utiliza para proporcionar a un programa un medio de conocer cuando una variable puntero no direcciona a un dato válido, siendo un mecanismo muy útil para depurar programas.

En la sección 1.11 de este apunte se profundiza el problema de la inicialización de punteros en la construcción de programas modularizados.

### 1.7.1. Los punteros `NULL` y `void`

Un puntero nulo, `NULL`, como se explicó en la sección anterior, no apunta a ninguna parte -dato válido- en particular, es decir, un puntero nulo no direcciona ningún dato válido en memoria.

Por otro lado, en C se puede declarar un puntero de modo que apunte a cualquier tipo de dato, es decir, **NO** se asigna a un tipo de dato específico. El método es declarar el puntero como un puntero `void *`, denominado puntero genérico.

 Capítulo 10 Sección *Punteros null y void*. Pág. 330. *Programación en C Metodología Algoritmos y Estructura de datos*. McGraw-Hill. JOYANES\_AGUILAR.

### 1.7.2. Asignación de una dirección de memoria válida a una variable puntero

Existen dos maneras de realizar la asignación de una dirección de memoria adecuada a un puntero.

**Opción 1:** Apuntarlo a una dirección de memoria ya reservada para el programa.

Para realizar esto basta asignar a la variable puntero la dirección de cualquier variable estática del mismo tipo que tengamos declarada en nuestro programa o igualarlo a otro puntero que ya esté apuntando a una dirección adecuada.

Ejemplo 5:

```
1.  #include <stdio.h>
2.  int main () {
3.      char unCaracter ;
4.      char * puntero = NULL ;
5.      puntero = & unCaracter ;
6.      return 0 ;
7.  }
```

Ahora `puntero` apunta a la dirección de memoria en la que está `unCaracter`. Se puede utilizar con seguridad la memoria a la que apunta `puntero`, sabiendo que lo que se asigne ahí también será accesible desde la variable en `unCaracter`.

Ejemplo 6:

```
1.  #include <stdio.h>
2.  int main () {
3.      char unCaracter ;
4.      char * puntero = NULL ;
5.      puntero = &unCaracter ;
6.      *puntero = 'A';    /* Ahora unCaracter también tiene una 'A' */
7.      return 0 ;
8.  }
```

**Opción 2:** Reservar una zona de memoria específica para la variable puntero y hacer que apunte a ella.

Se utilizará para esto dos funciones de la biblioteca estándar `stdlib.h`:

- `malloc` (abreviatura de *memory allocate*, que se puede traducir como *reservar memoria*): solicita un bloque de memoria del tamaño que se indique (en bytes);
- `free` (que en español significa *liberar*): libera memoria obtenida con `malloc`, es decir, la marca como disponible para futuras llamadas a `malloc`.
- Una vez reservada la zona de memoria, se puede utilizar con seguridad. Se debería liberar la memoria cuando ya no se la necesite. Para ello se utiliza la función `free()` a la que se le pasa la dirección de memoria que se quiere liberar.

### 1.7.3. Asignación dinámica de memoria: función `malloc`

**Sintaxis:** `<vble_ptr> = (<tipo_dato_ptr>) malloc(<tam_en_bytes>);`

**Comportamiento:** Adjudica espacio para un objeto, cuyo tamaño es especificado por `<tam_en_bytes>` y cuyo valor es indeterminado.

**Valor de retorno:** En caso de haber memoria disponible retorna un puntero que apunta a un bloque de memoria asignado, de lo contrario retorna nulo o cero.

### 1.7.4. Liberación de espacio de memoria: función `free`

**Sintaxis:** `free(<vble_ptr>);`

**Comportamiento:** El espacio apuntado por `<vble_ptr>` es desasignado, esto es, se pone disponible para otra asignación. Si `<vble_ptr>` es un puntero nulo, no se realizará ninguna acción. De lo contrario, si el argumento no corresponde a un valor válido, el comportamiento no está definido.

**Valor de retorno:** No retorna ningún valor.

Ejemplo 7:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  int main () {
5.      int * p = NULL ;
6.      p = (int*) malloc(sizeof(int));
```



```
6.      *p = 5;
7.      free ( p );
8.      p = NULL ;
9.      }
```

Primero se declaró un puntero, que aún no apunta a ningún sitio. Luego, el puntero, no su contenido, sino el propio puntero, es igual a un puntero tipo `int` que contiene la dirección en memoria con espacio para un `int`. La función `sizeof()` obtiene el espacio que ocupa aquello que se desee, si como argumento de esta función se pone `int`, como son 2 bytes, se asignan dos bytes de memoria. Por último, ahora que el puntero tiene contenido, se asigna un valor. Una vez utilizado el puntero se puede prescindir de la variable puntero y liberar la memoria utilizada aplicando la función `free()` a la variable puntero.

### Ejemplo 8:

El siguiente ejemplo ilustra cómo controlar que la memoria ha sido asignada correctamente.

```
1.      #include <stdio.h>
2.      #include <stdlib.h>
3.
4.      int main () {
5.          int * puntero = NULL ;
6.          int retorno;
7.          puntero = (int*) malloc (sizeof( int ) );
8.          if ( puntero == NULL ) { /*si hay error de asignación, se termina el programa*/
9.              printf ( "Error de asignación de memoria\n" );
10.             retorno=0;
11.         };
12.         else{
13.             *puntero = 10 ;
14.             printf ( "Mi número es %d" ,*puntero );
15.             free ( puntero );
16.             puntero = NULL ;}
17.             retorno=1;
18.         }
19.         return retorno;
20.     }
```



[http://es.wikibooks.org/wiki/Programaci%C3%B3n\\_en\\_C/Manejo\\_din%C3%A1mico\\_de\\_memoria](http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Manejo_din%C3%A1mico_de_memoria)  
[http://maxus.fis.usal.es/FICHAS\\_C.WEB/07xx\\_PAGS/0701.html#anchor39626](http://maxus.fis.usal.es/FICHAS_C.WEB/07xx_PAGS/0701.html#anchor39626)  
[http://sopa.dis.ulpgc.es/so/cpp/intro\\_c/introc75.htm](http://sopa.dis.ulpgc.es/so/cpp/intro_c/introc75.htm)

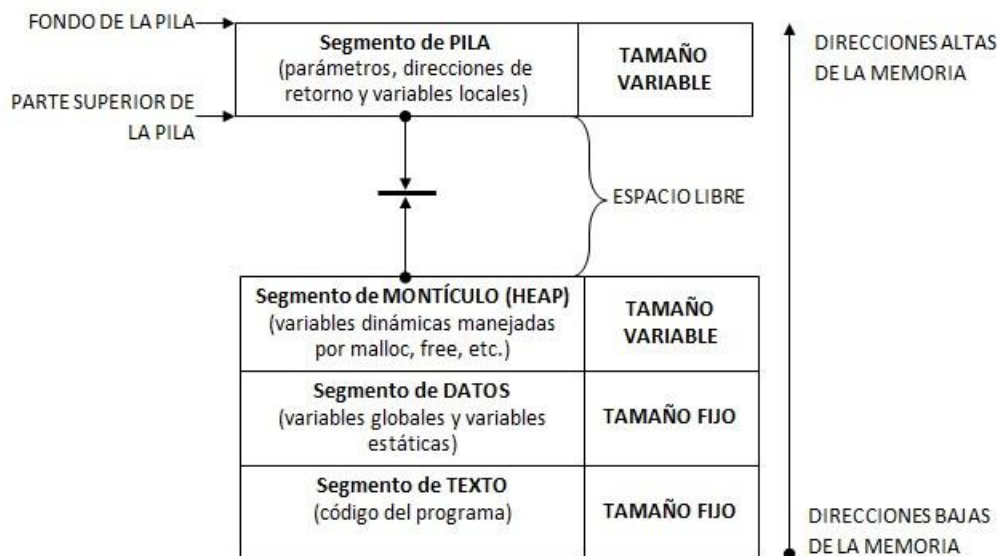
### 1.7.5. Asignación de memoria

En la sección anterior se explicó que existen dos formas de asignar una dirección de memoria válida a un puntero. En la opción 1 se utilizó la dirección de una variable preexistente en el programa para asignar su dirección a una variable tipo puntero utilizando el operador `&`. A la variable preexistente en el programa se le asignó una dirección de memoria en el momento de la compilación del código, esta dirección proviene de la

zona de memoria que está reservada para el programa que se está desarrollando. La memoria asignada recién es liberada cuando el programa finaliza su ejecución. El programador no tiene injerencia respecto del momento en el que se realiza la asignación, ni del lugar de donde proviene la memoria asignada y tampoco del momento en el que se la libera. Este tipo de asignación de memoria se llama *asignación estática de memoria* y es este tipo de asignación con la que venimos trabajando hasta ahora.

En la opción 2 se presenta las funciones `malloc` y `free`. Estas funciones permiten al programador decidir en qué momento de la ejecución del código la variable tendrá asignada una dirección de memoria válida utilizando la primera función. Con la segunda función, el programador puede liberar la memoria asignada en el momento que ya no la necesite más. Ahora bien, ¿de qué zona de la memoria provienen las direcciones asignadas con este método? Para poder responder esta pregunta se presenta, en primer lugar, la forma en la que se utiliza la memoria a la que tiene acceso una aplicación. Básicamente, la memoria RAM se descompone en tres zonas: pila, montículo y código.

- La **pila** (stack) es una zona de memoria que se asigna de forma exclusiva a la aplicación, su tamaño es estático, es decir que una vez asignada esta porción de memoria a la aplicación, ésta no crece ni decrece durante la ejecución de la aplicación. Esta memoria se parece bastante a las últimas hojas del cuaderno, las que se utilizan para hacer cálculos auxiliares (siempre se comienza utilizando la última hoja y se avanza hacia el inicio del cuaderno), como su nombre indica, mediante una pila o stack. Crece hacia abajo, hacia las direcciones bajas de memoria. Cuando la pila invade la zona usada del cuaderno, se produce una colisión entre pila y montículo.
- El código y los datos estáticos ocupan otra parte de la memoria asignada; el código es el programa en sí; la colección de instrucciones que hay que ejecutar y los datos estáticos son la información fija del programa (meses del año, tablas de factores de conversión, etc.). Esta información suele ubicarse por encima de la pila, para evitar invasiones, o bien al otro extremo de la memoria, por idéntica razón.
- Por último, el **montículo** (heap). Esta parte de la memoria es compartida y la administra el Sistema Operativo, que a petición de una aplicación, asigna de forma dinámica, por lo tanto la cantidad de memoria del montículo asignada a una aplicación varía durante la ejecución de la aplicación. Normalmente, la mayor parte de la memoria que utiliza una aplicación proviene del montículo, y es precisamente aquí donde se van a ir creando y destruyendo variables dinámicas durante la ejecución de la aplicación.



Las variables dinámicas se crean y destruyen a elección del programador. El objetivo que se persigue es racionalizar la utilización de un recurso escaso, la memoria RAM. El método que se emplea sigue los pasos indicados a continuación:

1. Reservar bloques de memoria.
2. Cargar información en ellos.
3. Procesar esa información.
4. Guardar los resultados obtenidos (en disco o en RAM).
5. Liberar la memoria reservada, que estará de nuevo a nuestra disposición para ser reservada en forma de otras variables.

De estos pasos se puede detectar otra recomendación importante al usar punteros con asignación dinámica de memoria:

Un programa debe efectuar una llamada a free por cada llamada a malloc

#### 1.7.6. Liberar memoria no cambia el valor del puntero

La llamada a free libera la memoria apuntada por un puntero, pero no modifica el valor de la variable que se le pasa. Cuando a un puntero por ejemplo, `*int unNumero`, se le asigna un valor, en este caso un valor numérico, por ejemplo `*unNumero=10`, y posteriormente se libera la memoria asignada al puntero `unNumero` ejecutando `free(unNumero)`, esa dirección se libera y pasa a estar disponible para eventuales llamadas a malloc, pero ¡`unNumero` sigue valiendo 10! ¿Por qué? Porque la función free posee su parámetro por valor, no por referencia, así que free no tiene forma de modificar el valor de `unNumero`. Por esto se recomienda:

Asignar el valor NULL después de una llamada a free, pues así se hace explícito que la variable puntero no apunta a nada.

Recordar, entonces, que es responsabilidad del programador asignar explícitamente el valor NULL a todo puntero que no apunte a memoria reservada.

 [http://maxus.fis.usal.es/FICHAS\\_C.WEB/11xx\\_PAGS/1101.html](http://maxus.fis.usal.es/FICHAS_C.WEB/11xx_PAGS/1101.html)

### 1.7.7. Alias

Las expresiones `unCaracter` y `*puntero` del Ejemplo 5 son equivalentes. Se dice entonces que `*puntero` es un alias de `unCaracter`. Nada impide crear múltiples alias de una variable, que podrían utilizarse desde distintos lugares de un programa:

#### Ejemplo 9:

```
float valor; float *alias_1,*alias_2,*alias_3;

alias_1 = alias_2 = alias_3 = &valor;
```

El uso de alias es potencialmente peligroso, pues permite modificar el valor de una variable empleando cualquiera de esos alias.

## 1.8. Estructuras y punteros

### 1.8.1. Campos de tipo puntero

Todo lo dicho hasta ahora para punteros, vale para aquellos declarados dentro de una estructura. Por ejemplo, se puede declarar una estructura como la siguiente:

#### Ejemplo 12:

```
1.  Typedef struct
2.  {
3.      char *nombre;
4.      int otroCampo;
5.  } Datos;
```

Si se declara una variable de tipo `Datos`, el puntero `nombre` está sin inicializar, por lo que se le debe asignar el valor `NULL`:

```
6.  Datos unNombre;
7.  unNombre.nombre = NULL;
```

y antes de usarlo, reservar espacio para este campo:

```
8.  unNombre.nombre = (char*)malloc(sizeof(char));
9.  o bien asignarlo a alguna variable adecuada.
10. unNombre.nombre = &algunaVariableAdecuada;
```

El problema principal con las estructuras surge cuando se las copia o asigna. Suponga el siguiente código:

```
1. struct Datos unNombre;
2. struct Datos otroNombre;
3. unNombre.nombre = NULL;
4. otroNombre.nombre = NULL;
5. ...
6. unNombre.nombre = (char*)malloc(sizeof(char));
```

7. ...

8. `otroNombre = unNombre;`

La última asignación copia todos los campos de la estructura `unNombre` en `otroNombre`, incluido el puntero interno. Ambos punteros van a apuntar a la misma dirección de memoria. Cambiar el contenido de uno de ellos implica cambiar el contenido del otro. El problema se presenta si se libera uno de ellos.

1. `free (unNombre.nombre);`

2. `unNombre.nombre = NULL;`

Con esta acción también se libera la memoria a la que apunta `otroNombre.nombre`, por lo que su contenido puede no ser válido. Utilizar o liberar posteriormente `otroNombre.nombre` puede generar problemas en la ejecución.

### 1.8.2. Punteros a estructuras

Un puntero también puede apuntar a una estructura. Se puede declarar un puntero a una estructura tal como se declara un puntero a cualquier otro objeto y se declara un puntero estructura tal como se declara cualquier otra variable estructura: especificando un puntero en lugar del nombre de la variable estructura.

Ejemplo 14:

```
1.     Typedef struct
2.     {
3.     char nombre [30] ;
4.     int edad;
5.     int altura;
6.     int peso;
7.     } persona;
8.     persona empleado = {"Amigo, Pepe", 47, 182, 85};
9.     persona *p; / * se crea un puntero de estructura * /
10.    p = &empleado;
```

Cuando se referencia un miembro de la estructura utilizando el nombre de la estructura, se especifica la estructura y el nombre del miembro separado por un punto (.). Por ejemplo para referenciar el nombre de una persona, se debe utilizar `empleado.nombre`.

Cuando la variable o el parámetro es un puntero a la estructura, para referenciar un campo de la estructura se utiliza `->` (flecha), por ejemplo `p->nombre`

## 1.9. Funciones y punteros

### 1.9.1. Punteros como argumentos de funciones

Cuando se pasa un parámetro a un módulo, este pasaje puede ser por valor, una copia del dato, o por referencia, la dirección de la memoria donde se encuentra el dato.

### 1.9.2. Punteros a datos locales

Al utilizar punteros se puede caer en la tentación de crear funciones que retornen punteros declarados localmente, asumiendo que programas como el siguiente son correctos.

**Ejemplo 15:**

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int* ingresarNum();
4.  int main() {
5.  int *a=NULL, *b=NULL, *c=NULL;

6.  a=ingresarNum();
7.  b=ingresarNum();
8.  c=ingresarNum();
9.  printf("Numero a: %d",*a);
10. printf("Numero b: %d",*b);
11. printf("Numero c: %d",*c);
12. return 0;
13. }
14. int* ingresarNum(){
15. int aux;
16. system("cls");
17. printf("Ingrese un numero: ");
18. scanf("%d",&aux);
19. return &aux;
20. }
21.
```

Este código es erróneo porque intenta inicializar los punteros invocando a una función que usan variables locales para retornar la dirección de memoria, en el ejemplo anterior, la variable aux tienen sentido dentro de la función, pero al terminar la función, desaparecen las variables locales. Cuando se intente usar el puntero devuelto por la función, esa dirección de memoria ya está libre y es posible que algún otro proceso la sobrescriba, haciendo que su valor sea aleatorio.

El programa correcto implementa asignación dinámica de memoria y sería:

**Ejemplo 16:**

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int ingresarNum(); /*retorna un entero, ya no retorna un puntero a entero*/
4.  int main() {
5.  int *a = NULL;
6.  int *b = NULL;
7.  int *c = NULL;
8.  /*asignación de una dirección válida de memoria*/
9.  a = (int*) malloc(sizeof(int));
10. b = (int*) malloc(sizeof(int));
```

```
11.  c = (int*) malloc(sizeof(int));
12.  /*ingreso de valores numéricos*/
13.  *a=ingresarNum();
14.  *b=ingresarNum();
15.  *c=ingresarNum();
16.  printf("Numero a: %d", *a);
17.  printf("Numero b: %d", *b);
18.  printf("Numero c: %d", *c);
19.  /*liberar memoria usada por los punteros a, b y c*/
20.  free(a);
21.  free(b);
22.  free(c);
23.  return 0;
24.  }
25.  int ingresarNum(){
26.  int aux;
27.  printf("Ingrese un numero: ");
28.  scanf("%d", &aux);
29.  return aux;
30.  }
```

### **1.9.3. Punteros a punteros**

Un puntero es una variable como las demás variables de C. Como toda variable, los punteros ocupan un lugar en memoria RAM y también tienen una dirección (la suya, no la que contienen). Esta dirección se puede almacenar en variables que son igualmente punteros, y cuya declaración es como puede verse en el ejemplo siguiente:

#### **Ejemplo 17:**

```
1. Tipo_base variable;
2. Tipo_base * ptb;
3. Tipo_base ** pptb; /*puntero a puntero*/
4. ptb== &variable;
5. pptb= &ptb;
```

El doble asterisco indica que `pptb` es un "doble puntero" o más exactamente el puntero de un puntero.

Aplicando el operador `*`, de indirección en las variables declaradas en el ejemplo 17, se obtienen las siguientes relaciones:

#### **Ejemplo 18:**

```
1. *pptb equivale a ptb
2. **pptb equivale a variable
```

Estas equivalencias son sumamente importantes: dado el puntero de un puntero, se puede modificar el puntero, del mismo modo que dado el puntero de una variable se puede modificar la variable.

En C, todos los parámetros por defecto pasan por valor. En particular, todos los parámetros que sean punteros pasarán por valor (aunque las variables que señalan pasen por referencia). Entonces, si se pasa un puntero como parámetro, el valor del parámetro pasa por valor y por tanto no es modificable desde el interior de la función. Esto puede ser grave para módulos que reciban un puntero de valor `NULL` y deseen reservar memoria dinámica internamente, devolviendo a través del puntero recibido la dirección del bloque reservado. Ahora bien, si se pasa un doble puntero como parámetro, entonces el doble puntero pasa por valor... pero el puntero que señala pasa por referencia! Esto abre la puerta para modificar punteros desde el interior de un módulo; basta pasarlos como doble puntero (esto es, basta pasar la dirección del puntero que se quiere modificar). Un puntero es una variable como otra cualquiera, y si se quiere que pase por referencia, debe pasarse a ese módulo la dirección del puntero (que es, un doble puntero).

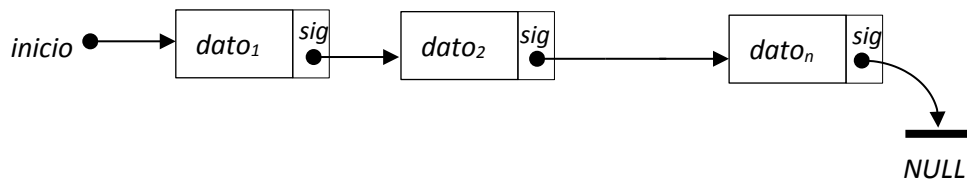
📖 Capítulo 10 Sección 10.4. Pág. 331. *Programación en C Metodología Algoritmos y Estructura de datos*. McGraw-Hill. JOYANES\_AGUILAR.  
[http://maxus.fis.usal.es/FICHAS\\_C.WEB/07xx\\_PAGS/0701.html#dobleindireccion](http://maxus.fis.usal.es/FICHAS_C.WEB/07xx_PAGS/0701.html#dobleindireccion)

## 2. Listas enlazadas

En esta sección se presenta la creación de listas con registros enlazados. Este tipo de listas ajustan su consumo de memoria al tamaño de la secuencia de datos que almacenan, contrariamente al uso de variables indizadas, que requieren una reserva previa de memoria y por lo tanto, una reserva superior a la requerida por los datos.

### 2.1. Listas simplemente enlazadas

Una lista simplemente enlazada es una secuencia de registros unidos por punteros de manera que cada registro contiene al menos dos campos, uno para el dato y otro para indicar cuál es el siguiente registro. El registro consta de uno o más campos con datos y un puntero: el que apunta al siguiente registro. El último registro de la secuencia apunta a `NULL`. Para conocer el inicio de la lista se debe almacenar la dirección de memoria del primer registro, llamado puntero maestro o inicio. El siguiente gráfico muestra la idea de lista enlazada:



Entonces, una lista simplemente enlazada es una secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un “enlace” o puntero.

Cada elemento (nodo) se compone de dos partes (campos) la:

- primera parte (campo) contiene información y es, un valor de un tipo genérico (dato); ▪
- segunda parte (campo) es un puntero (enlace) que apunta al siguiente elemento.

Para saber dónde está la primera entrada de una lista, se guarda en una variable de la memoria, la dirección de la primera entrada. Esta variable apunta al principio de la lista y suele llamarse puntero inicial o inicio. Si se quiere leer la lista, se comienza en la posición indicada por este puntero y se halla el primer nombre junto con un puntero a la siguiente entrada. Si se sigue este puntero se accede a la segunda entrada, y así sucesivamente hasta el final de la lista. Para detectar el final de la lista se utiliza un puntero nulo (`NULL`).



### 2.1.1. Estructura de una lista enlazada

El siguiente ejemplo es un programa para crear una lista de enteros. Los tipos de datos a definir son los siguientes:

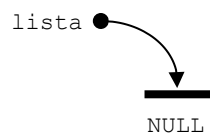
#### Ejemplo 20:

```
1. typedef struct Nodo{  
2.   int dato;  
3.   struct Nodo *sig;  
4. }tNodo;  
5. typedef tNodo *tPunteroNodo;
```

Un registro de tipo `tNodo` consta de dos elementos:

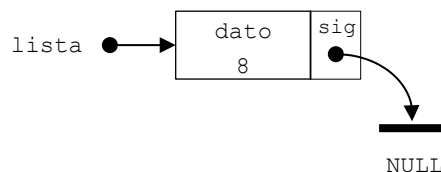
- un entero llamado `dato`, contiene el dato a registrar,
  - y un puntero a un elemento que es un puntero a un nodo con la misma estructura, es otro puntero a `struct Nodo`, (Observa que hay cierto nivel de recursión o autoreferencia en la definición: un `struct Nodo` contiene un puntero a un `struct Nodo`. Por esta razón, estas estructuras se denominan estructuras recursivas.)
6. Posteriormente se crea el puntero inicial, aquel en el que empieza la lista de enteros, llamada en el ejemplo, `lista`. Como pueden observar, no es más que un puntero a un elemento de tipo `tPunteroNodo`. Inicialmente, la lista está vacía, indicada explícitamente con la constante `NULL`:
7. `tPunteroNodo lista= NULL;`

Gráficamente, la situación se presenta así:



### 2.1.2. Añadir un nodo al inicio o cabeza de la lista:

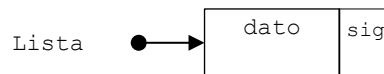
Se añade un nodo a la lista. El objetivo es pasar de la lista anterior a esta otra:



La creación del nuevo registro se realiza mediante asignación dinámica de memoria:

```
8. lista = (tNodo*) malloc(sizeof(tNodo));
```

El resultado es:



Ya se tiene el primer nodo de la lista, pero sus campos aún no tienen los valores finalmente.

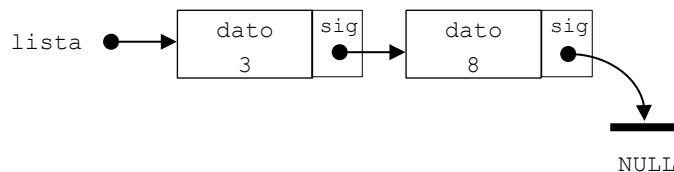
Por una parte, el campo dato debería contener el valor 8, y por otra, el campo sig debería apuntar a NULL:

#### Ejemplo 21:

```
1. tPunteroNodo lista = NULL;  
2. ...  
3. lista = (tNodo*) malloc(sizeof(tNodo));  
4. lista->dato = 8;  
5. lista->sig = NULL;
```

¿Cómo se agregan más nodos a la lista?

El resultado que se pretende obtener es el siguiente:

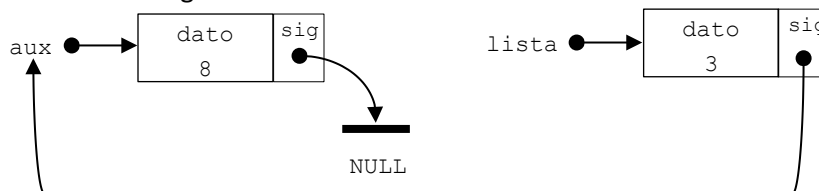


El siguiente código crea un nuevo nodo y lo enlaza en la lista, observar que se requiere de un puntero auxiliar para no perder la referencia a la lista,

#### Ejemplo 22:

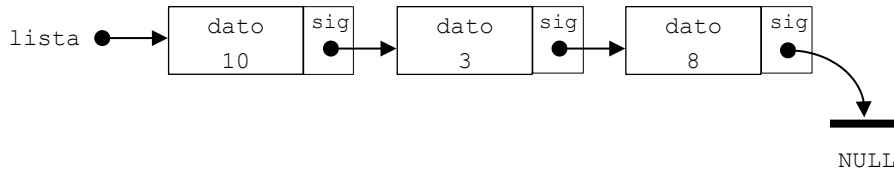
```
1. tPunteroNodo lista = NULL, aux ;  
2. ...  
3. aux = lista ;  
4. lista = (tNodo*) malloc(sizeof(tNodo));  
5. lista->dato = 3;  
6. lista->sig = aux ;
```

Gráficamente se realiza la siguiente tarea:



### 2.1.3. Recorrer una lista

Para recorrer una lista se inicia en la dirección de memoria inicial y se debe ir reemplazando esta dirección por la dirección del siguiente nodo mientras que el campo `sig` sea distinto de nulo. Por ejemplo, se tiene la siguiente lista:



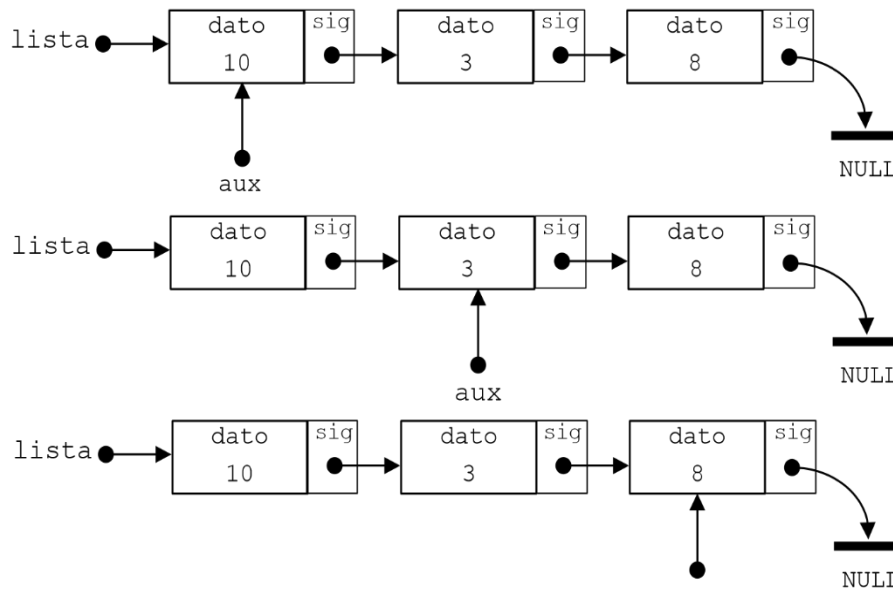
Para realizar el recorrido se inicia analizando el contenido apuntado por la variable `lista`, en este caso el campo `dato` almacena el número 10 y el campo `sig` almacena la dirección de memoria del siguiente nodo. Seguidamente se debe “mover” el puntero `lista` hacia la dirección de memoria apuntada por `sig`. Una vez que se traslada el puntero a la siguiente dirección, se verifica el contenido del campo `sig` de esta nueva dirección. Si éste es nulo, se detiene el proceso, se llegó al final de la lista, sino se mueve el puntero al siguiente nodo para verificar si apunta a una dirección nula o no. Se Repite estos pasos hasta llegar al final de la lista.

Durante este proceso el puntero cambia de una dirección de memoria a otra, por lo que si se utiliza la variable que apunta al inicio de la lista para hacer esto, se pierde el registro del lugar en donde comienza la lista, inhabilitando su acceso posterior. Por esto lo primero que se debe hacer, es una copia de la dirección de inicio de la lista en una variable auxiliar y trabajar sobre esta copia para recorrerla.

#### Ejemplo 23:

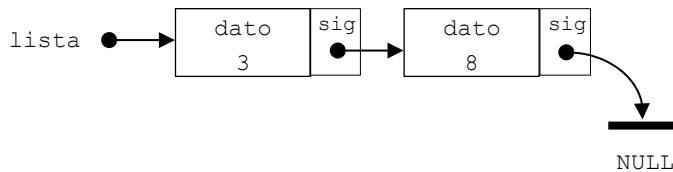
```
1. tPunteroNodo lista = NULL, aux, nuevo;
2. ...
3. aux = lista;
4. while (aux->sig != NULL)
5. aux=aux->sig;
```

Gráficamente:

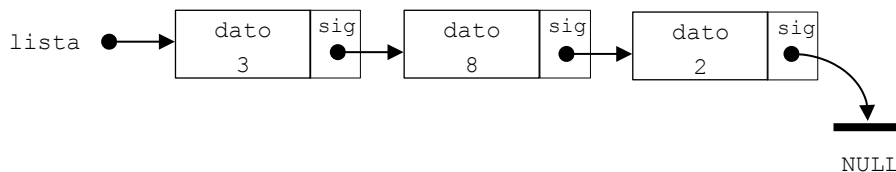


#### 2.1.4. Añadir un nodo al final de la lista:

Partiendo de una lista como la siguiente:



Se desea agregar un nodo al final:



Se debe recorrer la lista, utilizando para ello una variable auxiliar y luego enlazar el nodo a agregar, tal como lo muestra el siguiente ejemplo:

##### Ejemplo 24:

```
1. tPunteroNodo lista = NULL, aux, nuevo;  
2. ...  
3. /*recorrer la lista hasta el final*/  
4. aux = lista;  
5. while (aux->sig != NULL)  
6.     aux = aux->sig;
```

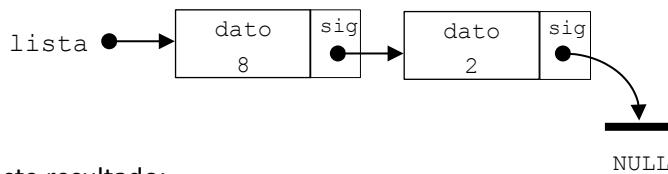
```
7.    /*crear el nuevo nodo (el que se desea insertar)*/
8.    nuevo = (tNodo*) malloc(sizeof(tNodo));
9.    nuevo->dato = 2;
10.   nuevo->sig = NULL;
11.   /*unir el nuevo nodo a la lista*/
12.   aux->sig = nuevo;
```

Este código no funciona correctamente cuando la lista está vacía ya que la expresión `aux->sig != NULL` no será correcta en ese caso por ser `lista = NULL`. Para solucionar esto basta agregar un control que verifique esta situación:

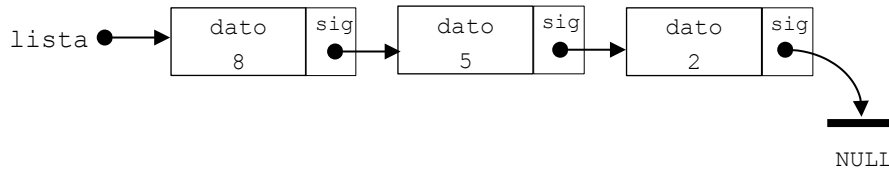
```
1.      tPunteroNodo aux, nuevo;
2.      ...
3.      if (lista == NULL) {
4.          lista = (tNodo*) malloc(sizeof(tNodo));
5.          lista->dato = valor;
6.          lista->sig = NULL;
7.      }
8.      else {
9.          /*recorrer la lista hasta el final*/
10.         aux = lista;
11.         while (aux->sig != NULL)
12.             aux = aux->sig;
13.         /*crear el nuevo nodo (el que se desea insertar)*/
14.         nuevo = (tNodo*) malloc( sizeof(tNodo) );
15.         nuevo->dato = 2;
16.         nuevo->sig = NULL;
17.         /*unir el nuevo nodo a la lista*/
18.         aux->sig = nuevo;
19.     }
```

### **2.1.5. Insertar un nodo en una posición P de la lista**

El objetivo es partir de la lista:



Y llegar a este resultado:



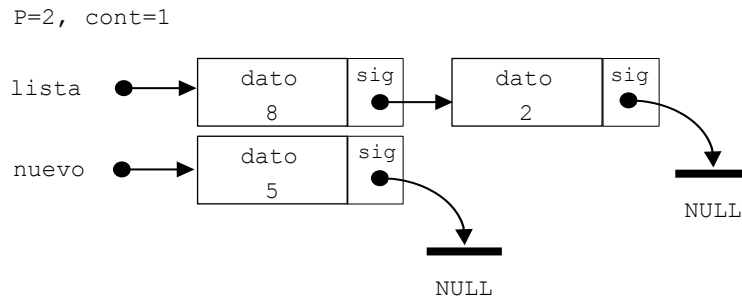
Tener presente que la posición de inserción no necesariamente es exactamente la del medio, puede ser cualquier posición P dentro de la lista. Por ahora se asume que P es distinto de 1.

Se debe crear el nuevo nodo cuyo campo dato almacenará el valor 5 y el campo sig el valor NULL. Luego se verifica si la lista está vacía. En el caso de estar vacía, lo único que debemos hacer es apuntar la variable lista a la variable nuevo. Si la lista no está vacía tenemos que recorrer la lista hasta llegar a la posición P de inserción. Una vez creado el nuevo nodo tenemos que “engancharlo” en la lista. Para esto el campo sig del nodo anterior al nodo de la posición P debe apuntar al nuevo nodo y el campo sig del nuevo nodo debe apuntar al nodo de la posición P. Notar que debemos conocer la dirección de memoria del nodo anterior al nodo de la posición P.

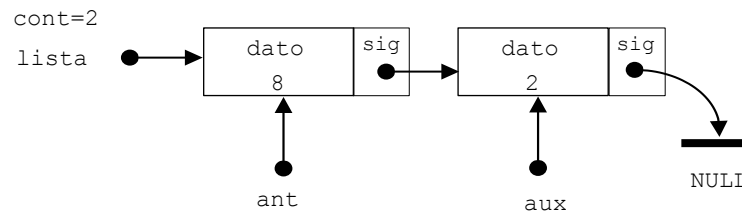
#### Ejemplo 25:

```
1. tPunteroNodo lista = NULL, aux, nuevo, ant = NULL;
2. int P, cont=1; /*lista posee la dir del primer nodo*/
3.     ...
4.     /*crear el nuevo nodo*/
5.     nuevo = (tNodo*) malloc(sizeof(tNodo));
6.     nuevo->dato = 2;
7.     nuevo->sig = NULL;
8.     if (lista != NULL){ /*si la lista no está vacía*/
9.         /*recorrer la lista hasta la posición P*/
10.        aux = lista;
11.        while ((cont != P)&&(aux->sig != NULL)){
12.            ant = aux;
13.            aux = aux->sig;
14.            cont += 1;
15.        }
16.        /*enganchar el nuevo nodo a la lista*/
17.        ant->sig = nuevo;
18.        nuevo->sig = aux;
19.    } else lista = nuevo; /*si la lista está vacía*/
```

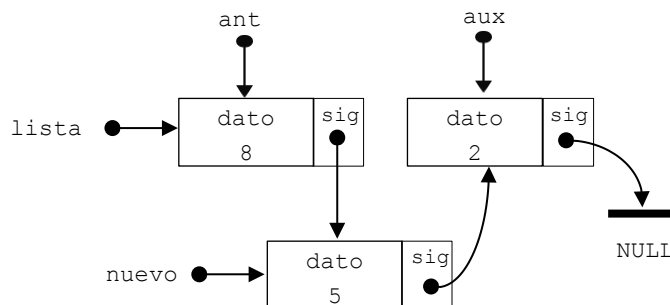
Gráficamente:



Después de recorrer la lista hasta la posición P

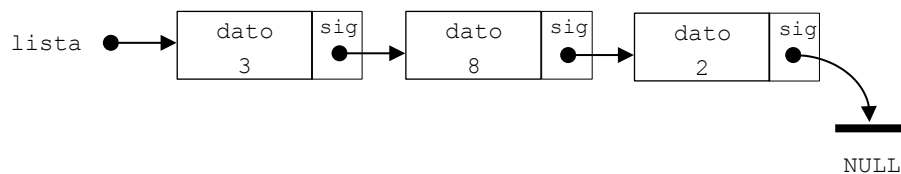


/\*enganchar el nuevo nodo\*/

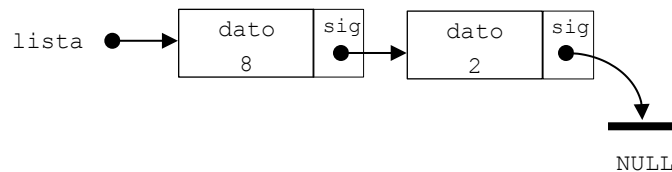


### 2.1.6. Eliminar el primer nodo de la lista (TP)

Veamos ahora cómo eliminar el primer elemento de una lista. El objetivo es partir de una lista como:



Y llegar a esta lista:



Se observa del gráfico que simplemente se debe apuntar la variable lista al segundo nodo de la lista, es decir el nodo apuntado por el campo sig del registro apuntado por la variable lista. Pero se debe hacer la modificación del inicio de la lista sin dejar inaccesible el nodo a eliminar ya que se tiene que liberar el espacio asignado al registro que se quiere eliminar. Para realizar este procedimiento de manera correcta se debe utilizar una variable auxiliar.

#### Ejemplo 26:

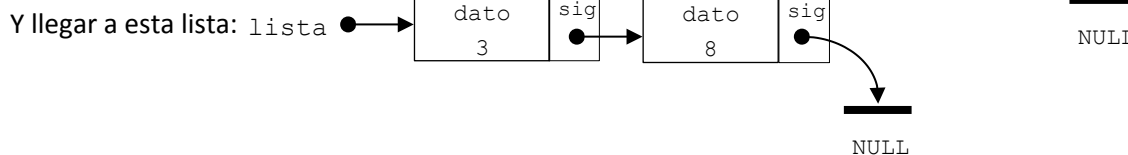
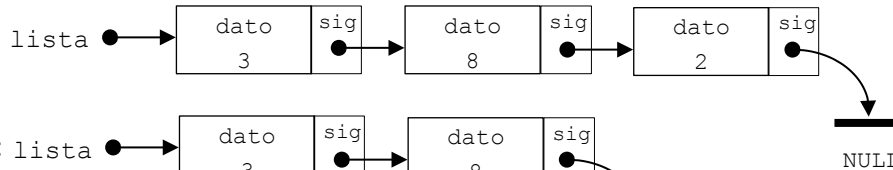
```
1. tPunteroNodo lista = NULL, aux;  
2. ...  
3. aux = lista->sig ;  
4. free(lista);  
5. lista = aux;
```

#### **2.1.7. Eliminar el último nodo de la lista**

Para borrar el último elemento de la lista nuevamente se debe dividir el trabajo en dos fases:

- localizar el último nodo de la lista para liberar la memoria que ocupa,
- hacer que el penúltimo nodo tenga como valor NULL en su campo sig.

El objetivo es partir de una lista como:



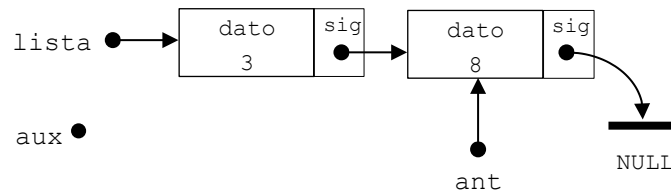
El código siguiente ilustra la resolución de esta situación

#### Ejemplo 27:

```
1. tPunteroNodo aux, ant;  
2. if (lista !=NULL){  
3. /*recorrer la lista hasta el final*/  
4. aux = lista;  
5. ant = NULL;  
6. while (aux->sig != NULL) {  
7. ant = aux;  
8. aux = aux->sig;  
9. }  
10. ant->sig = NULL;  
11. free(aux);  
12. }
```

Gráficamente, la lista resulta:





Cuando la lista tiene un único elemento el ciclo `while` no se ejecuta y por ende la variable `ant` queda con valor `NULL`. Esta situación provocará que el código de la línea 11 de un error. Para salvar esta situación se presenta el código completo:

```
1.     tPunteroNodo aux, ant;
2.     if (lista !=NULL){
3.         /*recorrer la lista hasta el final*/
4.         aux = lista;
5.         ant = NULL;
6.         while (aux->sig != NULL) {
7.             ant = aux;
8.             aux = aux->sig;
9.         }
10.        free(aux) ;
11.        if (ant==NULL) /*La lista tiene un solo elemento*/
12.            lista = NULL;
13.        else
14.            ant->sig = NULL;
15.    }
16.    ...
```

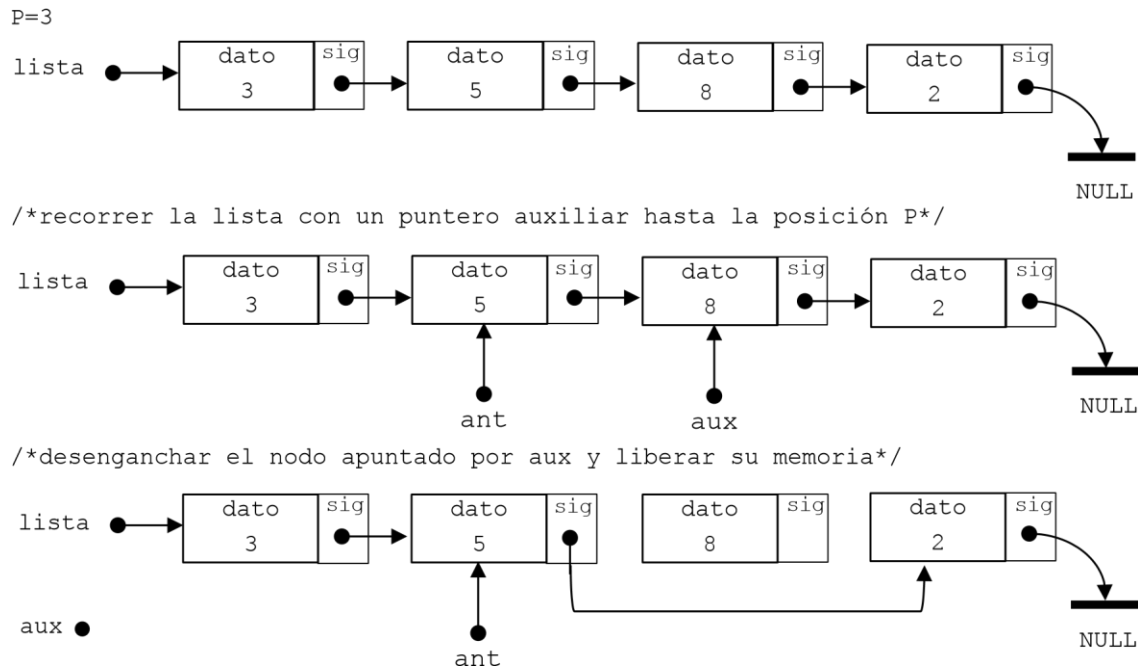
Cabe la siguiente recomendación:

Un caso especial que conviene estudiar explícitamente es el de la lista compuesta por un solo elemento.

Siempre que se accede a un campo de un puntero, por ejemplo, `ptr->sig` o `ptr->dato`, siempre hay que verificar si cabe alguna posibilidad de que `ptr` sea `NULL`; si es así, hay un problema que se debe solucionar.

### 2.1.8. Eliminar un elemento de una posición P de la lista

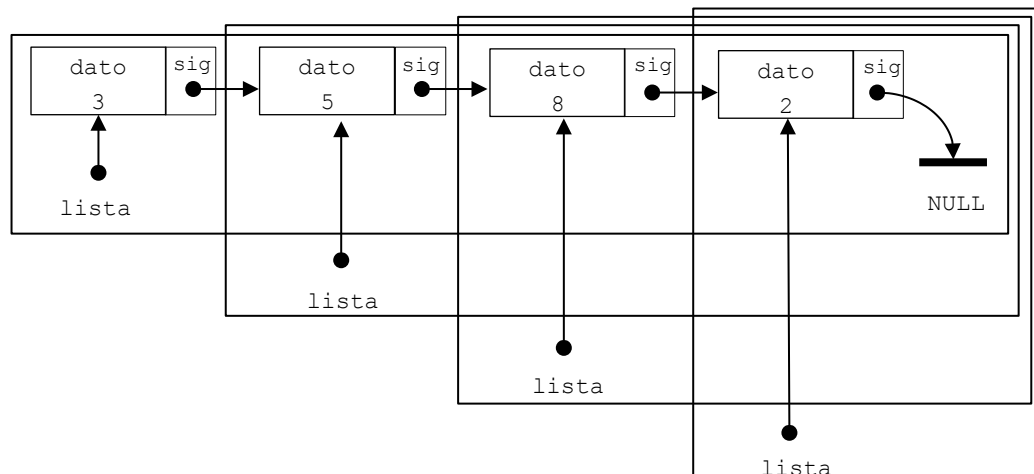
Gráficamente el problema se resuelve de la siguiente manera:



Queda como tarea para el lector implementar el código para este caso.

### 2.1.9. Implementación de las operaciones de listas simplemente enlazadas con recursión

La implementación de las operaciones sobre listas simplemente enlazadas puede simplificarse en gran medida utilizando algoritmos recursivos. Como ya se estudió, la recursión permite la definición de módulos más simples y fáciles de comprender y a esto se suma que las listas enlazadas se construyen en base a una estructura recursiva, por lo que el recorrido de las listas se desarrolla de forma natural con algoritmos recursivos, ya que cada nodo de la lista se puede considerar como la cabeza de una sublista menor. Gráficamente:



Una de las operaciones que se simplifica en gran medida utilizando un algoritmo recursivo es la que permite eliminar la primera ocurrencia de un valor dentro de la lista. Se muestra a continuación el código iterativo y el recursivo a modo de ejemplo:

#### Ejemplo 28:

```

1.      /*ALGORITMO ITERATIVO*/

```

```
void borra_primera_ocurrencia(tPunteroNodo *lista, int valor){
    tPunteroNodo ant, aux;
    if (!es_lista_vacia(*lista)){
        if ((*lista)->dato==valor){
            aux=*lista;
            *lista=(*lista)->sig;
            free(aux);
9.         }
        else
        {
            aux=*lista;
            while (aux!=NULL && aux->dato!=valor){
                ant=aux;
                aux = aux->sig;
            }
            if (aux!=NULL){
                ant->sig=aux->sig;
                free(aux);
            }
        }
    }
23. }

1.     /*ALGORITMO RECURSIVO*/
2.     void borra_primera_ocurrencia_recursivo(tPunteroNodo *lista, int valor){
3.         tPunteroNodo aux;
4.         if (!es_lista_vacia(*lista)){
5.             if ((*lista)->dato==valor){
6.                 aux=*lista;
7.                 *lista=(*lista)->sig;
8.                 free(aux); }
9.             else
10.                borra_primera_ocurrencia_recursivo(&(*lista)->sig, valor);
        } }
}
```