

Introducción a software propio de Ingeniería – Lenguaje C - REPASO

Programación - GITT

Carmen Nieves Ojeda Guerra

Elementos de un programa en C

La salida en pantalla: printf

- Para presentar en pantalla se usa la función *printf* que está en la biblioteca **<stdio.h>**
- Si se quiere mostrar un literal *string*:

```
printf("este es un literal string");
```

- Si se quiere mostrar valores que no son literales, se usan los **caracteres de sustitución**, como:

Carácter de sustitución	Tipo de dato representado
%d	entero
%c	carácter
%f	real
%s	cadena (string)
%p	puntero

Elementos de un programa en C

Tipos de datos estructurados

- C define **estructuras** de datos que agrupan campos de diferentes tipos. Su formato es:

```
struct nombre_de_la_estructura {  
    tipo_1 nombre_del_campo1;  
    tipo_2 nombre_del_campo2;  
    ...  
    tipo_N nombre_del_campoN;  
};
```

al conjunto de los datos se
le llama **registro**

declaración de una
estructura

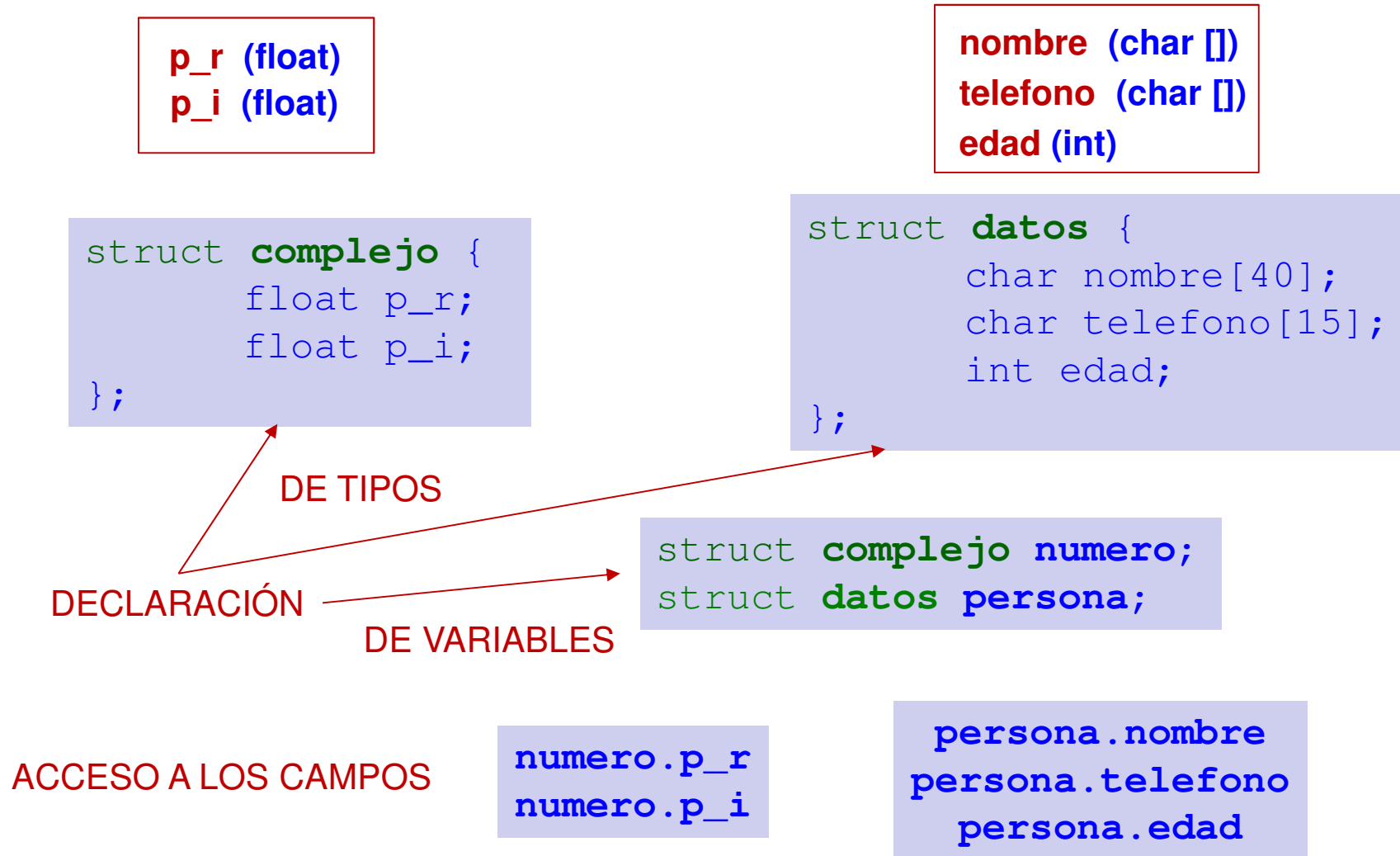
```
struct nombre_de_la_estructura var_estructura;
```

```
var_estructura.nombre_del_campo
```

acceso a los
campos

Elementos de un programa en C

Tipos de datos estructurados



Arrays y String en C

Declaración y acceso de arrays

DECLARACIÓN Y CREACIÓN:

```
tipo nombre_array[longitud];
```

```
tipo nombre_array[longitud_fila][longitud_columna];
```

ACCESO:

```
nombre_array[indice] = valor;
```

```
nombre_array[indice]
```

$0 \leq \text{indice} < \text{longitud}$

```
nombre_array[indice1][indice2] = valor;
```

```
nombre_array[indice1][indice2]
```

$0 \leq \text{indice1} < \text{longitud_fila}$

$0 \leq \text{indice2} < \text{longitud_columna}$

Arrays y String en C

Declaración y acceso de String

DECLARACIÓN Y CREACIÓN:

- C no tiene el tipo *String*
- Las cadenas de caracteres se almacenan en *array* de *char*

```
char nombre_cadena[longitud];
```

- Existe una biblioteca de funciones (**string.h**) de manejo de cadenas de caracteres

ACCESO:

Entre comillas simples (es un **char**)

```
nombre_cadena[indice] = valor;
```

```
nombre_cadena[indice]
```

$0 \leq \text{indice} < \text{longitud}$

Corresponde a un carácter individual

```
nombre_cadena
```

Corresponde la cadena completa

Arrays y String en C

Funciones de la biblioteca de C para el manejo de String

- La biblioteca *string.h* contiene funciones para manejar cadenas en C: **strcpy**, **strcat**, **strlen**, **strcmp**,...

```
char nombre_1[20], nombre_2[20], nombre_3[50];  
// se asigna valor a nombre_1
```

```
    nombre_2 = nombre_1; // es INCORRECTO  
    strcpy(nombre_2, nombre_1); // es CORRECTO
```

```
nombre_3 = nombre_1 + " " + nombre_2; // es INCORRECTO  
strcpy(nombre_3, nombre_1);  
strcat(nombre_3, " "); // es CORRECTO  
strcat(nombre_3, nombre_2);
```

Arrays y String en C

Conversión de String a número y de número a String

- La biblioteca *stdlib.h* contiene funciones para convertir *String* a números
- Algunas funciones son:
 - **atoi**: convierte una cadena a número entero (si la cadena no se puede convertir, se devuelve 0. Si puede convertir una parte, devuelve esa parte)
 - **atof**: convierte una cadena a número real (si la cadena no se puede convertir, se devuelve 0. Si puede convertir una parte, devuelve esa parte)
- Para convertir números a *String* se usa la función *sprintf* de la biblioteca *stdio.h*

```
sprintf(cadena, formato, numero)
```


Los punteros

Operaciones básicas con punteros

tipo *nombrePun: *nombrePun* es un puntero (apunta a) un valor de tipo **tipo**, es decir, guarda la dirección en memoria de un valor de tipo **tipo**

&nombreVar: La dirección en memoria de *nombreVar*

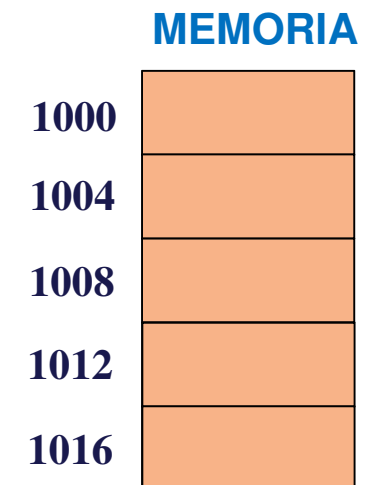
***nombrePun:** El contenido de la dirección en memoria almacenada en *nombrePun*

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```



Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    → int y = 5;  
      int z = 3;  
      int *nptr;  
      int *mptr;  
  
      nptr = &y;  
      z = *nptr;  
      *nptr = 7;  
      mptr = nptr;  
      mptr = &z;  
      *mptr = *nptr;  
      y = (*nptr) + 1;  
      z = ((*nptr)++) * 2;  
      z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	5	y
1004		
1008		
1012		
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    → int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	5	y
1004	3	z
1008		
1012		
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    → int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	5	y
1004	3	z
1008	?	nptr
1012		
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    → int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	5	y
1004	3	z
1008	?	nptr
1012	?	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    → nptr = &y;  
      z = *nptr;  
      *nptr = 7;  
      mptr = nptr;  
      mptr = &z;  
      *mptr = *nptr;  
      y = (*nptr) + 1;  
      z = ((*nptr)++) * 2;  
      z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	5	y
1004	3	z
1008	1000	nptr
1012	?	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    → z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	5	y
1004	5	z
1008	1000	nptr
1012	?	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    → *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	7	y
1004	5	z
1008	1000	nptr
1012	?	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    → mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	7	y
1004	5	z
1008	1000	nptr
1012	1000	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    → mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	7	y
1004	5	z
1008	1000	nptr
1012	1004	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    → *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	7	y
1004	7	z
1008	1000	nptr
1012	1004	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    → y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	8	y
1004	7	z
1008	1000	nptr
1012	1004	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    → z = ((*nptr) * 2);  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	8	y
1004	16	z
1008	1000	nptr
1012	1004	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:




```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    → ■: ((*nptr)++) ■;  
    z = (++(*nptr)) + 3;  
}
```

MEMORIA		
1000	9	y
1004	16	z
1008	1000	nptr
1012	1004	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
      ++(*nptr)  ;  
}
```

MEMORIA		
1000	10	y
1004	16	z
1008	1000	nptr
1012	1004	mptr
1016		

Los punteros

Ejercicio

Presentar cómo se modifica la memoria al ejecutar el siguiente código:

```
int main() {  
    int y = 5;  
    int z = 3;  
    int *nptr;  
    int *mptr;  
  
    nptr = &y;  
    z = *nptr;  
    *nptr = 7;  
    mptr = nptr;  
    mptr = &z;  
    *mptr = *nptr;  
    y = (*nptr) + 1;  
    z = ((*nptr)++) * 2;  
    z = (█(*nptr)) + 3;  
}
```

MEMORIA		
1000	10	y
1004	13	z
1008	1000	nptr
1012	1004	mptr
1016		

Los punteros

Punteros a estructuras

```
struct nombre_de_la_estructura {  
    tipo_1 nombre_del_campo1;  
    tipo_2 nombre_del_campo2;  
    ...  
    tipo_N nombre_del_campoN;  
};  
...  
struct nombre_de_la_estructura *ptr;  
  
...  
...ptr->nombre_del_campo1...  
...ptr->nombre_del_campo2...  
...
```

Los punteros

Punteros a estructuras

```
struct nombre_de_la_estructura {  
    tipo_1 nombre_del_campo1;  
    tipo_2 nombre_del_campo2;  
    ...  
    tipo_N nombre_del_campoN;  
};
```

first (struct nodo *)
last (struct nodo *)
size (int)

prev	item	next
(struct nodo *)	(tipo)	(struct nodo *)

```
struct listaDoble {  
    struct nodo *first;  
    struct nodo *last;  
    int size;  
};
```

```
struct nodo {  
    tipo item;  
    struct nodo *prev;  
    struct nodo *next;  
};
```

Los punteros

Punteros a estructuras

```
struct listaDoble {  
    struct nodo *first;  
    struct nodo *last;  
    int size;  
};
```

```
struct nodo {  
    tipo item;  
    struct nodo *prev;  
    struct nodo *next;  
};
```

DE TIPOS

DECLARACIÓN
DE VARIABLES

```
struct listaDoble lista;  
struct nodo *actual;
```

Estructura

Puntero a una estructura

lista es
estructura

actual es
puntero a
estructura

ACCESO A LOS CAMPOS

```
lista.first  
lista.last  
lista.size
```

```
actual->prev  
actual->item  
actual->next
```

Los punteros

Punteros a estructuras

```
6 #define TAM_NOMBRE 40
7 #define TAM_SERIES 10
8
9 struct Plataforma {
10     char nombre[TAM_NOMBRE];
11     char series[TAM_SERIES][TAM_NOMBRE];
12     int nTemporadas[TAM_SERIES];
13     int nSeries;
14     int nClientes;
15     float presupuesto;
16 };
17
18 struct nodo {
19     struct Plataforma item;
20     struct nodo *prev, *next;
21 };
```

```
struct Plataforma plataforma;
...
plataforma.nombre // array 1D de char
plataforma.series // array 2D de char
plataforma.series[i] // array 1D de char
plataforma.series[i][j] // char
plataforma.nTemporadas // array 1D de int
plataforma.nClientes // int
plataforma.presupuesto // float
...
struct nodo *actual; // puntero a struct
// nodo
actual->item // struct Plataforma
actual->prev // puntero a struct nodo
actual->next // puntero a struct nodo
...
actual->item.nombre // array 1D de char
actual->item.series // array 2D de char
actual->item.series[i] // array 1D de char
actual->item.series[i][j] // char
actual->item.nTemporadas // array 1D de int
actual->item.nSeries // int
actual->item.presupuesto // float
```

Asignación

Primitivos y estructuras

Todos los datos de tipo primitivo se pueden almacenar en una variable usando el operador de asignación

Asignación de estructuras

```
struct nombre_de_la_estructura var1, var2;  
...  
var2 = var1;
```

```
struct Plataforma plataforma1, plataforma2;  
... // se asigna valor a plataforma1  
plataforma2 = plataforma1;
```

Asignación

Primitivos y estructuras

Asignación de cadenas de caracteres (*String*)

```
strcpy(cadena_destino, cadena_origen);
```

```
char nombre1[40];  
char nombre2[40] = "Bugs Bunny";  
strcpy(nombre1, nombre2); // nombre1 almacena Bugs Bunny
```

Asignación de *arrays* (no cadenas)

```
memcpy(array_destino, array_origen, num_bytes_copiados);
```

```
int nTemporadas[TAM_SERIES];  
...  
memcpy( nTemporadas,  
        actual->item.nTemporadas,  
        TAM_SERIES*sizeof(int));
```

Las funciones

Definición y declaración

- Los programas en C se dividen en **funciones**
- Son similares a los métodos de Java, pero las funciones no forman parte de una clase
- Una función en C se distingue sólo por su nombre (**no existe sobrecarga**)
- Tienen el formato:

siempre es un tipo primitivo (los **punteros** son tipos primitivos) o una estructura. También puede ser **void**

```
tipo_del_resultado nombre_funcion(tipo_param1 param1,  
                                     tipo_param2 param2, ... ) {  
  
    /* Código de la función */  
  
}
```


Las funciones

Definición y declaración

- Para trabajar con funciones en C, hay que seguir los tres pasos siguientes:

1. Definir el prototipo:

```
27 void insertarPlataforma(struct Plataforma, int);
```

2. Declarar la función:

```
147 void insertarPlataforma(struct Plataforma item, int indice) {  
148     ...  
149 }
```

3. Llamar a la función:

```
37 int main() {  
38     struct Plataforma plataforma1 = {"Plataforma1", {"Serie1_1", "Serie1_2", "Serie1_3", "Serie1_4"},  
39     {3, 5, 1, 7}, 4, 10000, 350.67};  
40     insertarPlataforma(plataforma1, 0);  
41     ...  
42 }
```

Las funciones

Paso de parámetros por valor y por referencia

- Para poder modificar los valores de los parámetros, dentro de la función, se deben pasar **por referencia**
- Cuando se pasa un dato por referencia, se pasa el puntero al dato

```
#include <stdio.h>
void modificar(int *);

int main() {
    int i = 1;
    printf("\ni = %d antes de llamar a la función modificar", i);
    modificar(&i);
    printf("\ni = %d después de llamar a la función modificar", i);
}

void modificar(int *i) {
    printf("\ni = %d dentro de modificar", *i);
    *i = 9;
    printf("\ni = %d dentro de modificar", *i);
}
```

Se pasa la dirección del parámetro →

→ i es un **puntero** al valor pasado

Las funciones

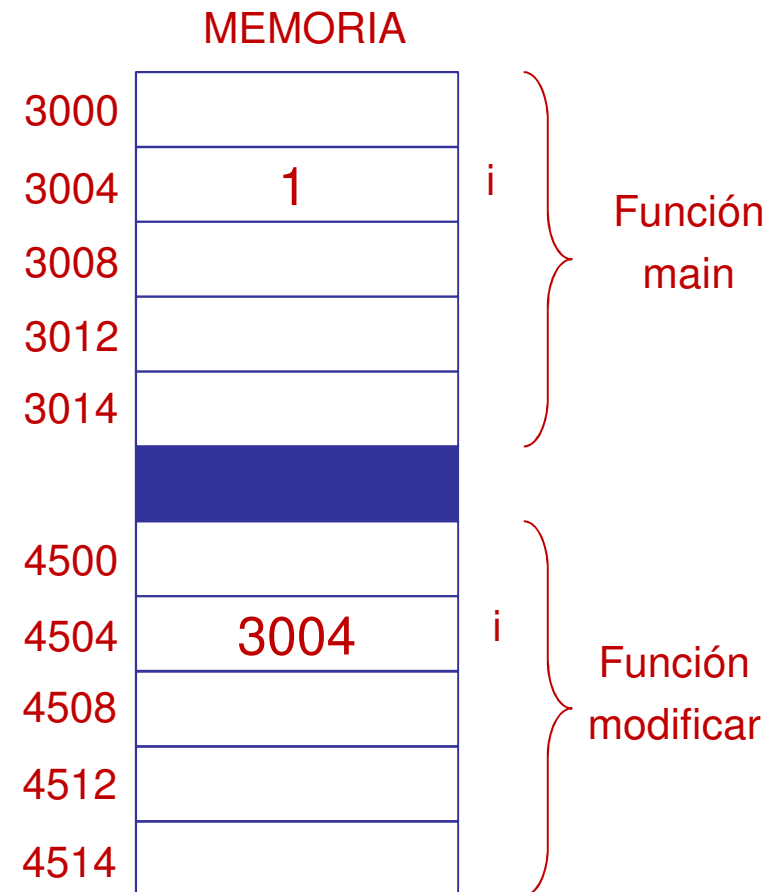
Paso de parámetros por valor y por referencia

```
#include <stdio.h>
void modificar(int *);

int main() {
    int i = 1;
    printf("\ni = %d antes de llamar a la función modificar", i);
    modificar(&i);
    printf("\ni = %d después de llamar a la función modificar", i);
}

void modificar(int *i) {
    printf("\ni = %d dentro de modificar", *i);
    *i = 9;
    printf("\ni = %d dentro de modificar", *i);
}
```

```
i = 1 antes de llamar a la función modificar
i = 1 dentro de modificar
i = 9 dentro de modificar
i = 9 después de llamar a la función modificar
```



Las funciones

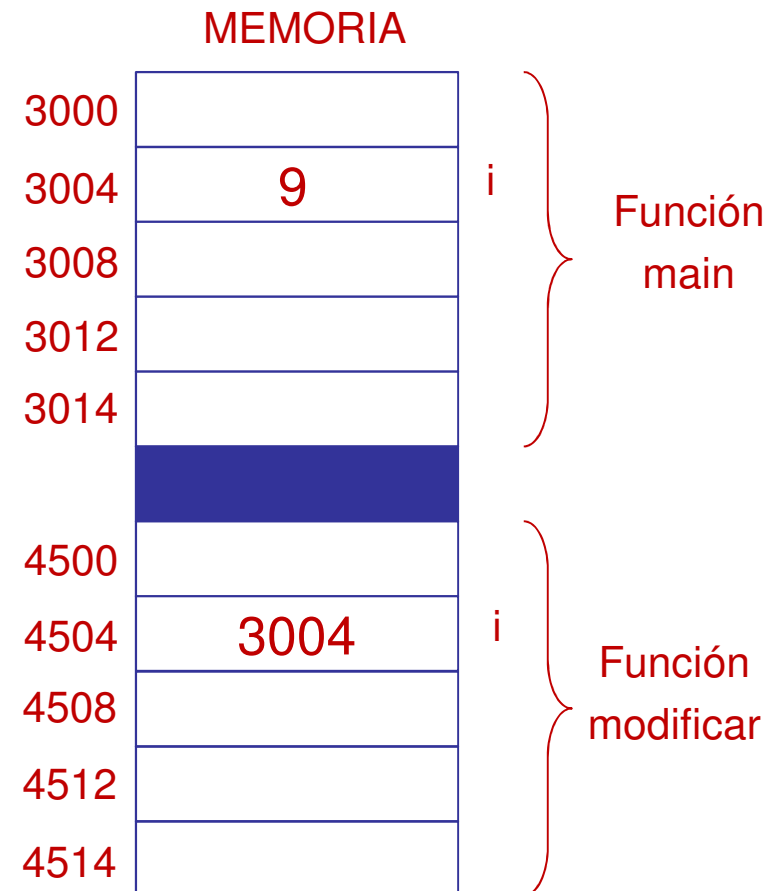
Paso de parámetros por valor y por referencia

```
#include <stdio.h>
void modificar(int *);

int main() {
    int i = 1;
    printf("\ni = %d antes de llamar a la función modificar", i);
    modificar(&i);
    printf("\ni = %d después de llamar a la función modificar", i);
}

void modificar(int *i) {
    printf("\ni = %d dentro de modificar", *i);
    *i = 9;
    printf("\ni = %d dentro de modificar", *i);
}
```

```
i = 1 antes de llamar a la función modificar
i = 1 dentro de modificar
i = 9 dentro de modificar
i = 9 después de llamar a la función modificar
```



Las funciones

Paso de parámetros por valor y por referencia

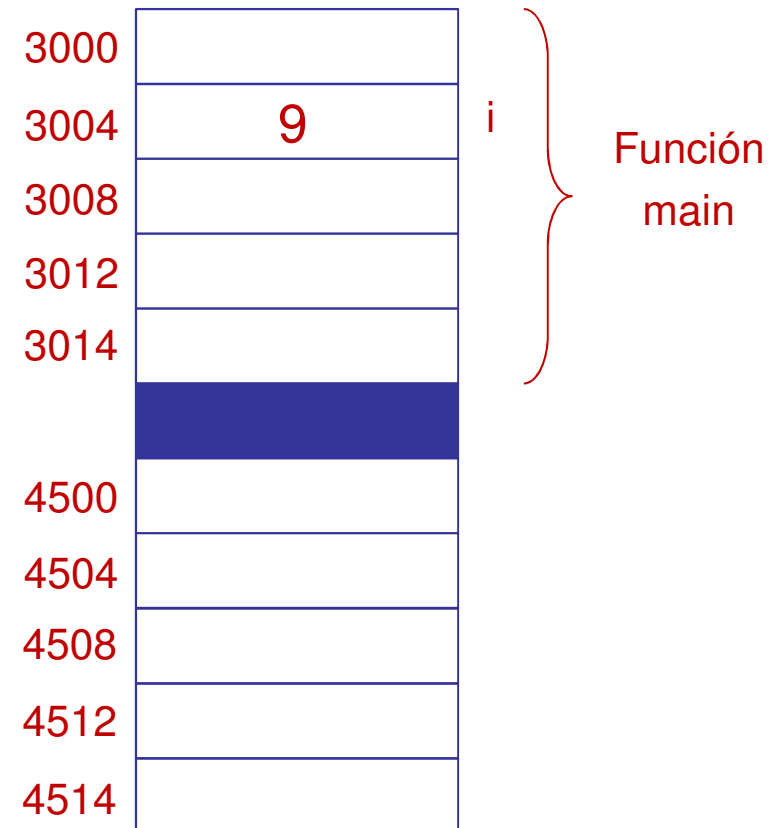
```
#include <stdio.h>
void modificar(int *);

int main() {
    int i = 1;
    printf("\ni = %d antes de llamar a la función modificar", i);
    modificar(&i);
    printf("\ni = %d después de llamar a la función modificar", i);
}

void modificar(int *i) {
    printf("\ni = %d dentro de modificar", *i);
    *i = 9;
    printf("\ni = %d dentro de modificar", *i);
}
```

```
i = 1 antes de llamar a la función modificar
i = 1 dentro de modificar
i = 9 dentro de modificar
i = 9 después de llamar a la función modificar
```

MEMORIA



Las funciones

La entrada por teclado

- Para leer cadenas:

```
fgets (cadena_de_caracteres, numero_de_caracteres, stdin);
```

- Para leer caracteres individuales:

```
char getchar ();
```

- Para leer números:

```
scanf (caracteres_de_sustitución, dirección_de_variable);
```

```
int numerol;  
char nombre[30], caracter;  
printf("Introduzca un número entero: ");  
scanf("%d", &numerol);  
getchar(); // para eliminar el enter  
printf("Introduzca el nombre: ");  
fgets(nombre, 30, stdin);  
nombre[strlen(nombre) - 1] = '\\0'; // para eliminar el enter y poner \\0  
printf("Introduzca el carácter: ");  
caracter = getchar();
```

Las funciones

Parámetros

**TODOS LOS ARRAYS SE PASAN POR REFERENCIA
(no hay que poner el operador & al pasarlos)**

```
#include <stdio.h>
#include <string.h>
#define LONGITUD_NOMBRE 30

void solicitarNombre(char []); // como parámetro

void solicitarNombre(char nombre[]) { // los arrays se pasan por referencia
    fgets(nombre, LONGITUD_NOMBRE, stdin);
    nombre[strlen(nombre) - 1] = '\0';
}

int main() {
    printf("Introduzca el nombre: ");
    char nombre[LONGITUD_NOMBRE];
    solicitarNombre(nombre);
    printf("\nEl nombre introducido es: %s\n", nombre);
}
```

Datos dinámicos vs Datos estáticos

Conceptos

Datos estáticos

- Tamaño constante durante la ejecución de un programa
- Se determinan en tiempo de compilación

Ejemplo: estructuras, *arrays*

Datos dinámicos

- Tamaño variable (o puede serlo) a lo largo de un programa
- Se crean y destruyen en tiempo de ejecución

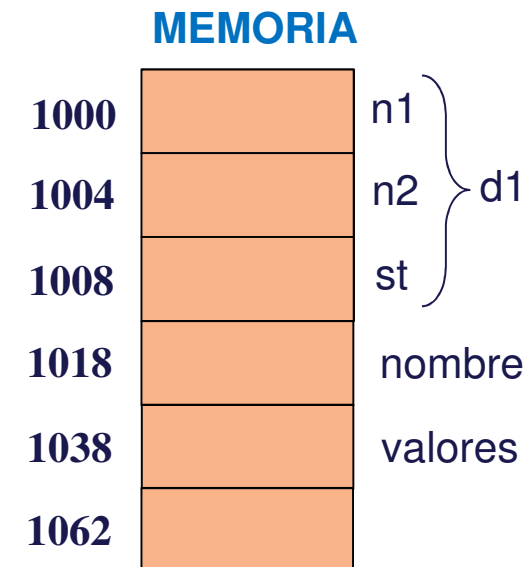
Ejemplo: listas enlazadas, pilas, colas

Datos dinámicos vs Datos estáticos

Conceptos

Datos estáticos

```
struct contact {  
    int n1;  
    float n2;  
    char st[10];  
} d1;  
  
char nombre[20];  
  
int valores[3][2];
```



Datos dinámicos vs Datos estáticos

Conceptos

Datos dinámicos

- Asignar memoria:

malloc devuelve la
dirección de memoria
asignada

```
tipo *nombrePun = (tipo *)malloc(num_elementos*sizeof(tipo));
```

Ejemplo:

```
char *nombre = (char *)malloc(20*sizeof(char));
```

Array de longitud
20 de *char*

- Liberar memoria:

```
free(nombrePun);
```

Ejemplo:

```
free(nombre);
```

Datos dinámicos

Funciones

SIEMPRE QUE UNA FUNCIÓN TENGA QUE RETORNAR UN *ARRAY* (NO A TRAVÉS DE PARÁMETROS), EL *ARRAY* DEBE HABER SIDO CREADO MEDIANTE ASIGNACIÓN DINÁMICA DE MEMORIA

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define LONGITUD_NOMBRE 30
char *solicitarNombre(); // retornado por la función
```

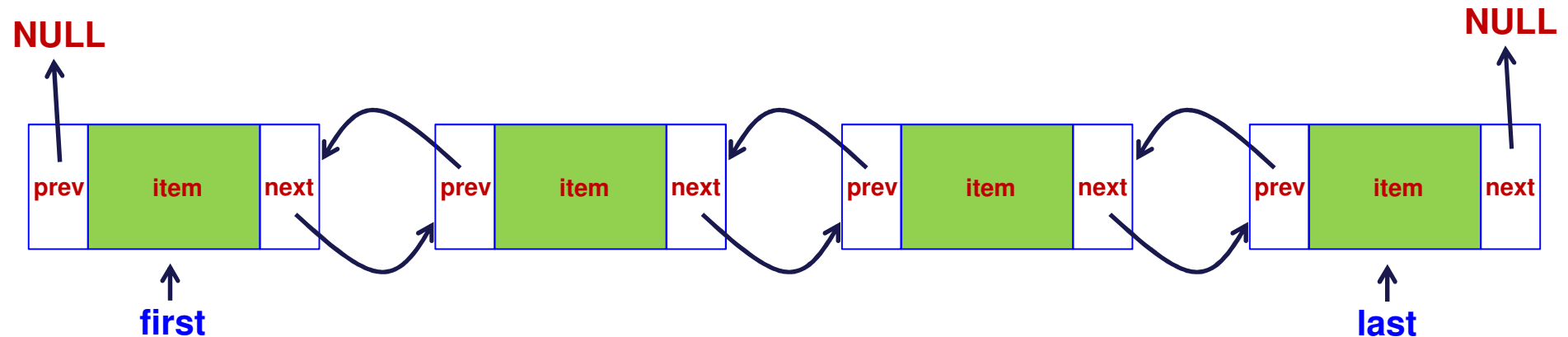
```
char *solicitarNombre() {
    char *nombre = (char *)malloc(LONGITUD_NOMBRE * sizeof(char));
    fgets(nombre, LONGITUD_NOMBRE, stdin);
    nombre[strlen(nombre) - 1] = '\0';
    return nombre;
}
```

```
tipo *nombrePun = (tipo *)malloc(num_elementos*sizeof(tipo));
```

```
int main() {
    printf("Introduzca el nombre: ");
    char *nombre = solicitarNombre();
    printf("\nEl nombre introducido es: %s\n", nombre);
}
```

Las listas enlazadas

Estructuras que definen la lista enlazada



```
struct nodo {  
    tipo item; // información de tipo no definido  
    struct nodo *prev, *next; // enlaces a otros nodos  
};
```

```
struct nodo *first = NULL; // la lista está vacía  
struct nodo *last = NULL; // la lista está vacía  
int size = 0; // la lista está vacía
```

Las listas enlazadas

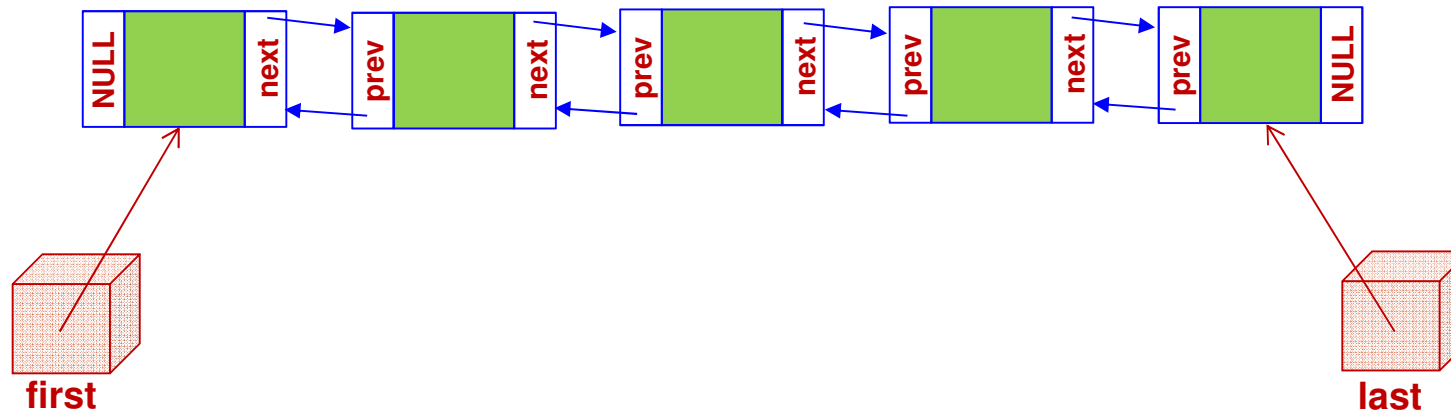
Relación entre arrays y listas enlazadas

Arrays	Listas enlazadas
<code>int i</code>	<code>struct nodo *actual</code>
<code>i = 0</code>	<code>actual = first</code>
<code>i = array.length-1</code> <code>(i = nElementos -1)</code>	<code>actual = last</code>
<code>i < nElementos</code>	<code>actual != NULL</code>
<code>array.length</code> o <code>nElementos</code>	<code>size</code>
<code>array[i]</code>	<code>actual -> item</code>
<code>i++</code>	<code>actual = actual -> next</code>
<code>i--</code>	<code>actual = actual -> prev</code>
<code>array[i] = valor</code>	<code>actual -> item = valor</code>

Es de tipo primitivo (si
es cadena hay que
usar strcpy)

Las listas enlazadas

Recorrido de los nodos de una lista enlazada



```
// instrucciones previas al bucle for
for (struct nodo *actual = first; actual != NULL; actual = actual->next) {
    // se hace alguna acción con el objeto actual
}
// instrucciones posteriores al bucle for
```

Las listas enlazadas

Búsqueda de un elemento en una lista enlazada

```
struct nodo *buscarElemento(Tipo_dado elemento) {  
    for (struct nodo *actual = first ; actual != NULL ; actual = actual->next) {  
        if (si el elemento es igual al item) {  
            return actual;  
        }  
    }  
    return NULL;  
}
```

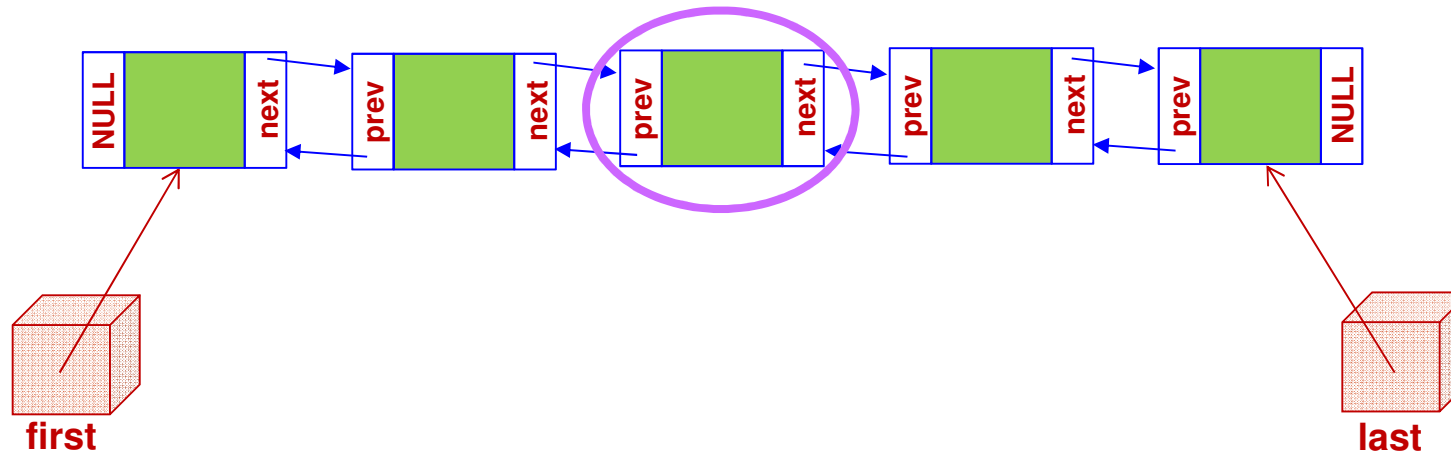
$\text{elemento} == \text{actual} \rightarrow \text{item}$

$\text{strcmp}(\text{elemento}, \text{actual} \rightarrow \text{item}) == 0$

Sí el tipo del elemento es una estructura, hay que comparar campo a campo de la estructura

Las listas enlazadas

Búsqueda del nodo de una posición dada en una lista enlazada

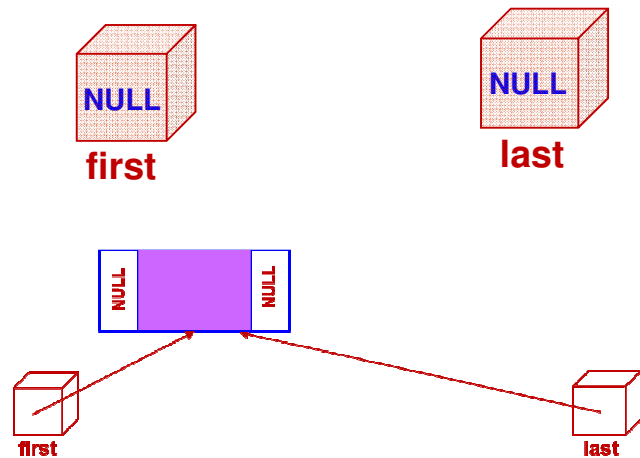


```
// instrucciones previas al bucle for
// indice >= 0 y indice < size (siendo size el número de nodos)
struct nodo *actual = first;
for (int j = 0 ; j < indice ; j++) {
    actual = actual->next;
}
// instrucciones posteriores al bucle for
```

indice
2

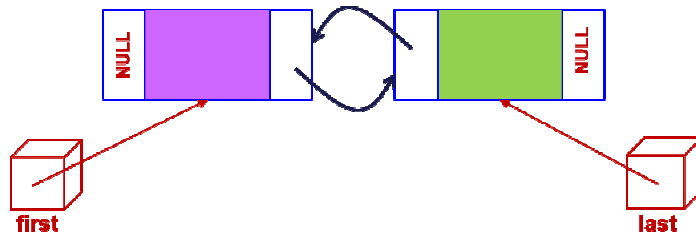
Las listas enlazadas

Inserción de un nodo en una lista enlazada



```
if (first == NULL) {  
    first = malloc(sizeof(struct nodo));  
    first->item = elemento;  
    first->prev = NULL;  
    first->next = NULL;  
    last = first;  
    size++;  
}
```

0 <= indice <= size

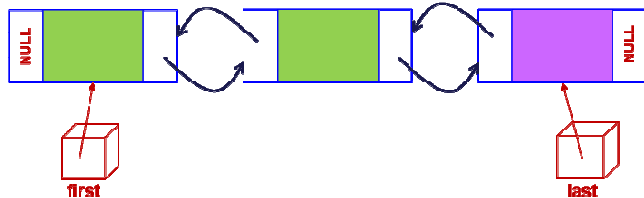


```
if (indice == 0) {  
    struct nodo *nuevoNodo = malloc(sizeof(struct nodo));  
    nuevoNodo->item = elemento;  
    nuevoNodo->prev = NULL;  
    nuevoNodo->next = first;  
    first->prev = nuevoNodo;  
    first = nuevoNodo;  
    size++;  
}
```

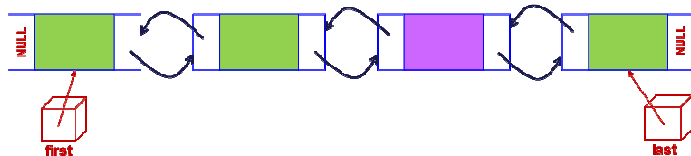
Las listas enlazadas

Inserción de un nodo en una lista enlazada

$0 \leq \text{indice} \leq \text{size}$



```
if (indice == size) {  
    struct nodo *nuevoNodo = malloc(sizeof(struct nodo));  
    nuevoNodo->item = elemento;  
    nuevoNodo->prev = last;  
    nuevoNodo->next = NULL;  
    last->next = nuevoNodo;  
    last = nuevoNodo;  
    size++;  
}
```

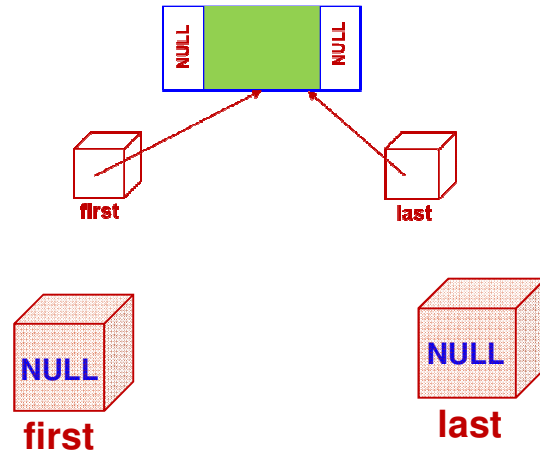


```
if ((indice > 0) && (indice < size)) {  
    struct nodo *actual = buscarNodo(indice);  
    struct nodo *anterior = actual->prev;  
    struct nodo *nuevoNodo = malloc(sizeof(struct nodo));  
    nuevoNodo->item = elemento;  
    nuevoNodo->prev = anterior;  
    nuevoNodo->next = actual;  
    actual->prev = nuevoNodo;  
    anterior->next = nuevoNodo;  
    size++;  
}
```

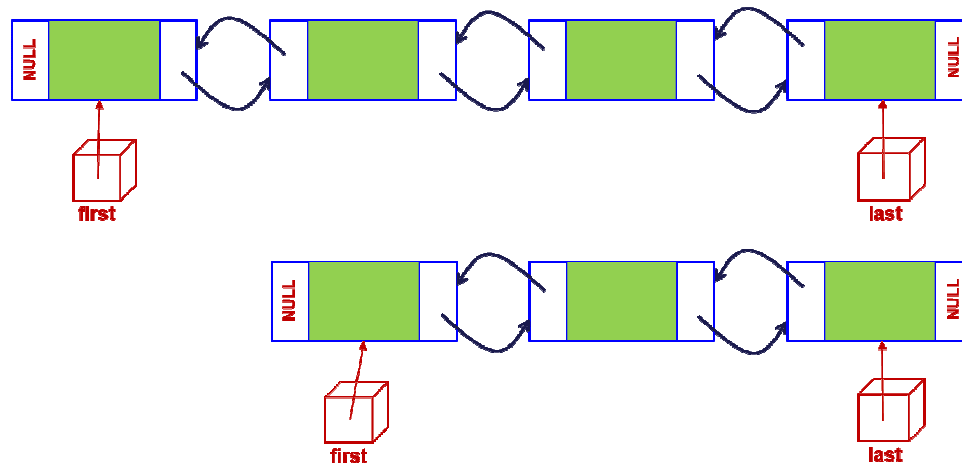
Las listas enlazadas

Eliminación de un nodo en una lista enlazada

$0 \leq \text{indice} < \text{size}$



```
if (first == last) {  
    free(first);  
    first = last = NULL;  
    size--;  
}
```

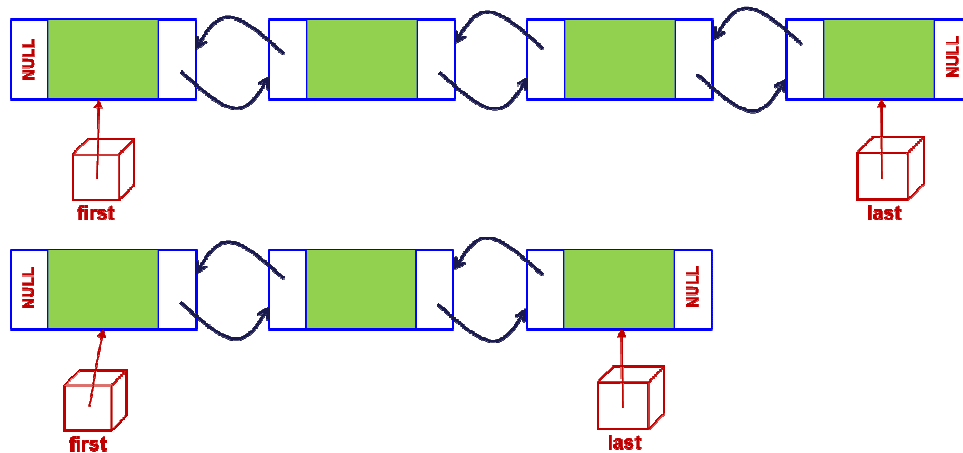


```
if (indice == 0) {  
    struct nodo *actual = first;  
    first = first->next;  
    first->prev = NULL;  
    free(actual);  
    size--;  
}
```

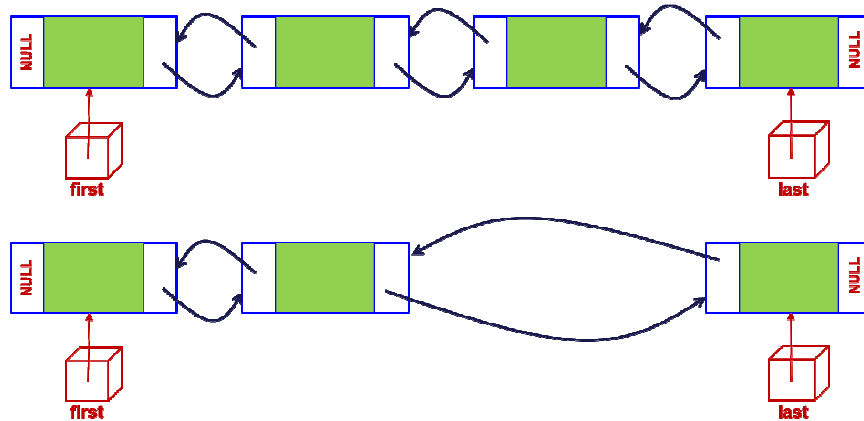
Las listas enlazadas

Eliminación de un nodo en una lista enlazada

$0 \leq \text{indice} < \text{size}$



```
if (indice == size - 1) {  
    struct nodo *actual = last;  
    last = last->prev;  
    last->next = NULL;  
    free(actual);  
    size--;  
}
```



```
if ((indice > 0) && (indice < size - 1)) {  
    struct nodo *actual = buscarNodo(indice);  
    struct nodo *anterior = actual->prev;  
    struct nodo *siguiente = actual->next;  
    anterior->next = siguiente;  
    siguiente->prev = anterior;  
    free(actual);  
    size--;  
}
```