# Curso de programación en C moderno
(II Edición)

Neira Ayuso, Pablo  Falgueras García, Carlos

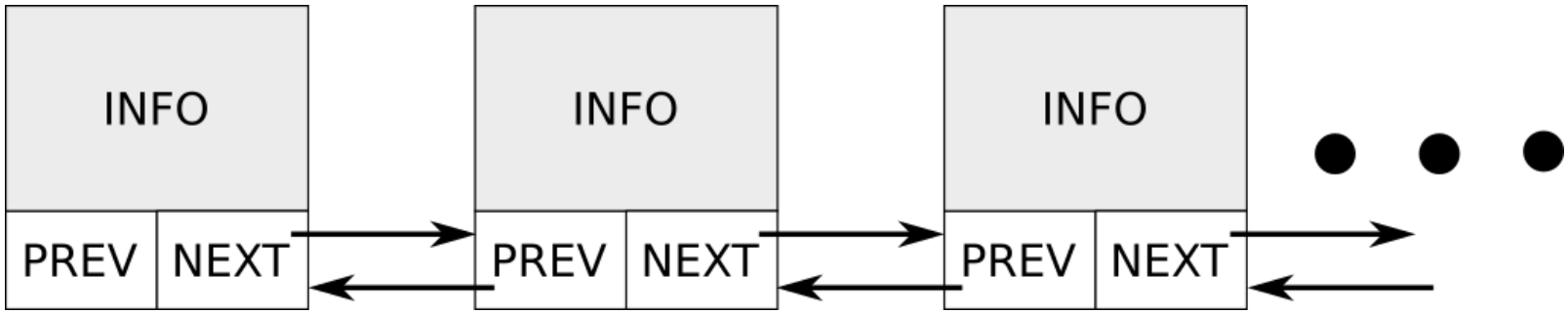
*Tema 9*

# Listas encadenadas

# Índice

# ¿Qué es una lista encadenada?






- Cada elemento guarda:
  1. Información
  2. Una referencia al siguiente elemento [y al anterior]
- Los elementos están dispersos en la memoria. Se reservan individualmente.

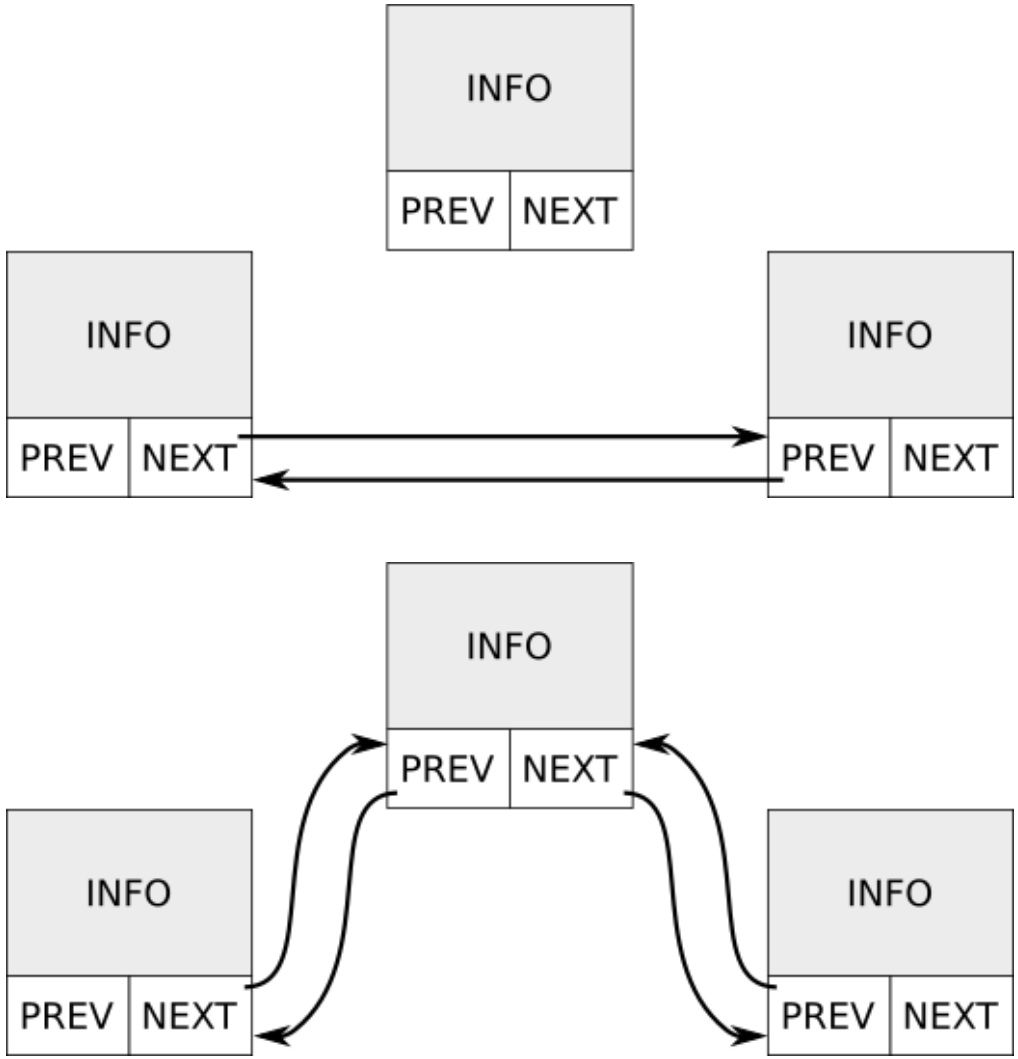
# ¿Cuándo son útiles?



- **No sabemos cuántos** elementos vamos a tener que guardar
- Vamos a recorrer los elementos de manera **secuencial**
- Necesitamos hacer **inserciones** y/o **eliminaciones** de elementos o sublistas

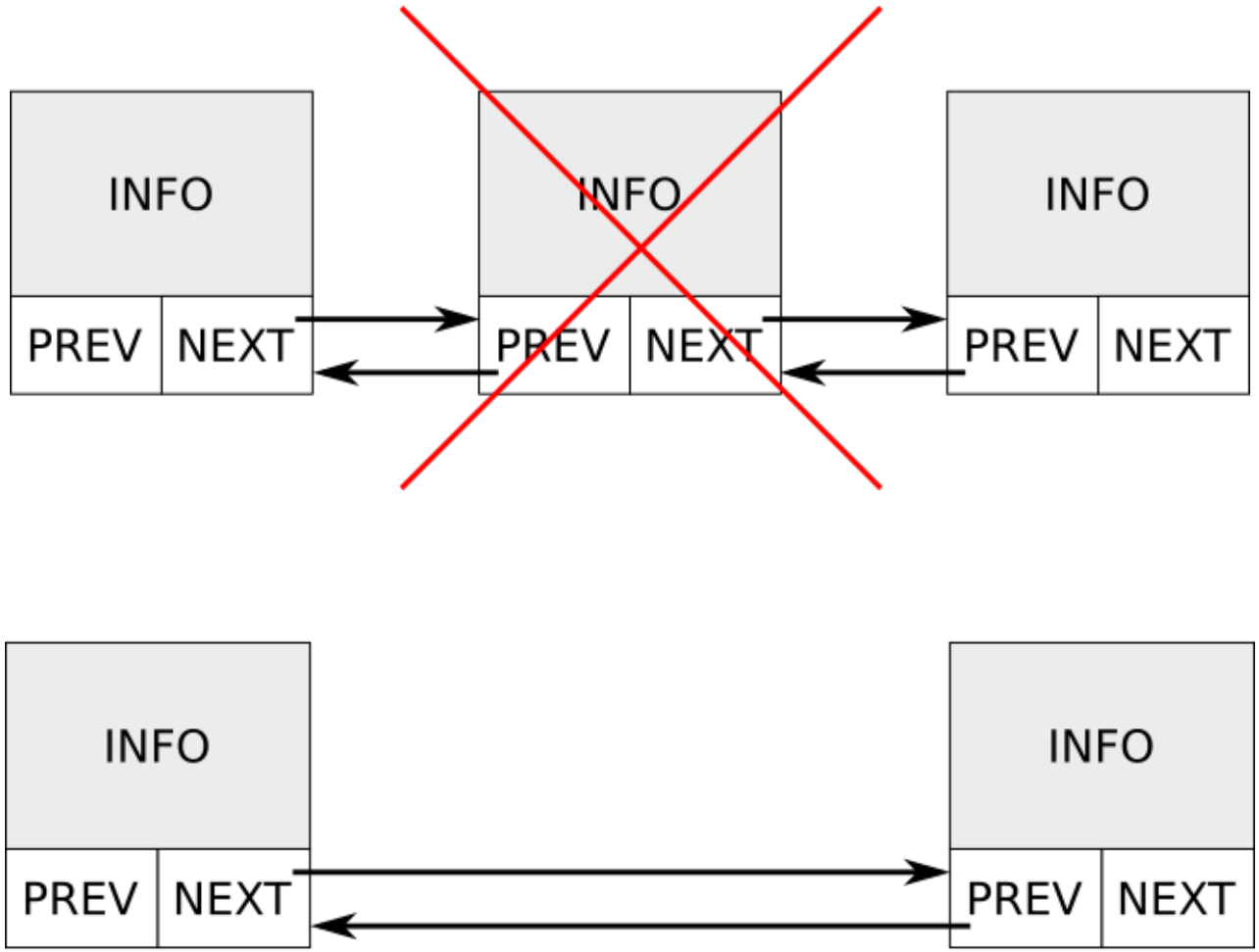
# Insertar





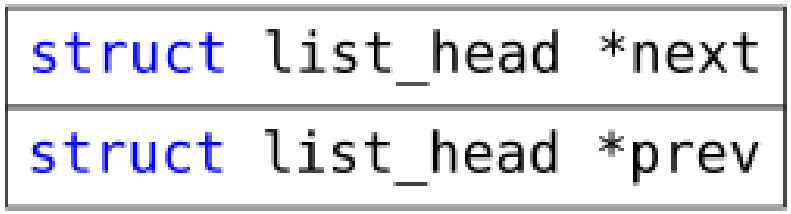
# Eliminar

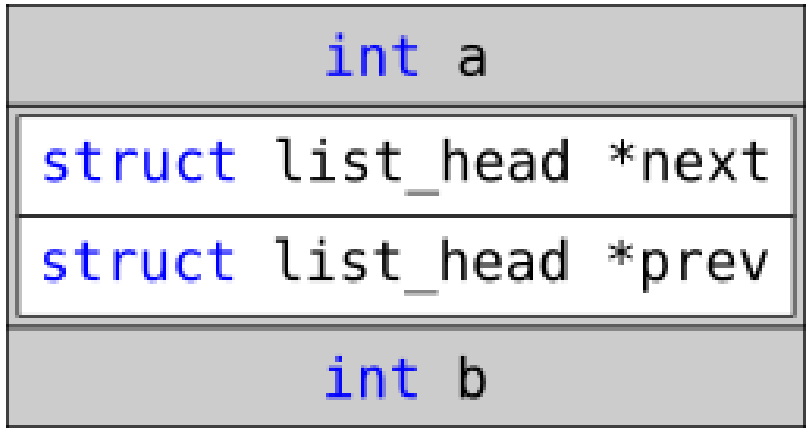




# Lista del Kernel





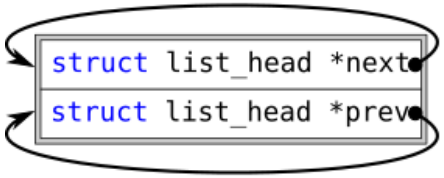




```
struct list_head {
  struct list_head *next;
  struct list_head *prev;
};
```

```
struct element {
  int a;
  struct list_head list;
  int b;
};
```

# Ejemplo

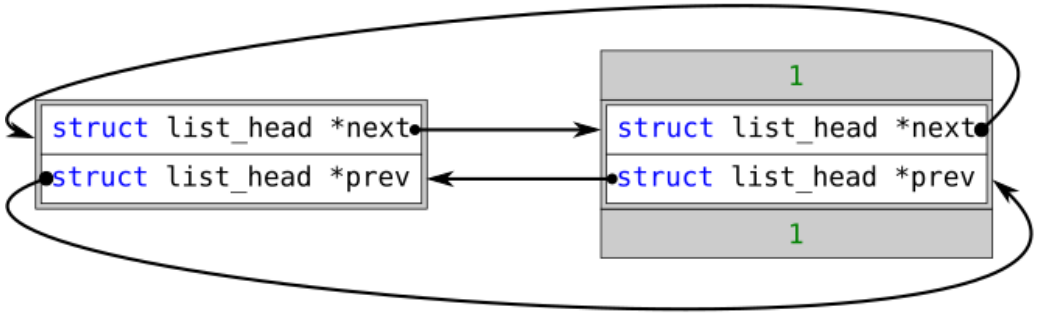





```c
int main()
{
  struct list_head list;
  struct element *element;

  INIT_LIST_HEAD(&list);

  element = element_alloc(1, 1);
  list_add(&(element->list), &list);

  element = element_alloc(2, 2);
  list_add(&(element->list), &list);

  list_for_each_entry(element, &list, list)
    printf("{%d, %d}\n", element->a, element->b);

  return 0;
}
```

# Ejemplo

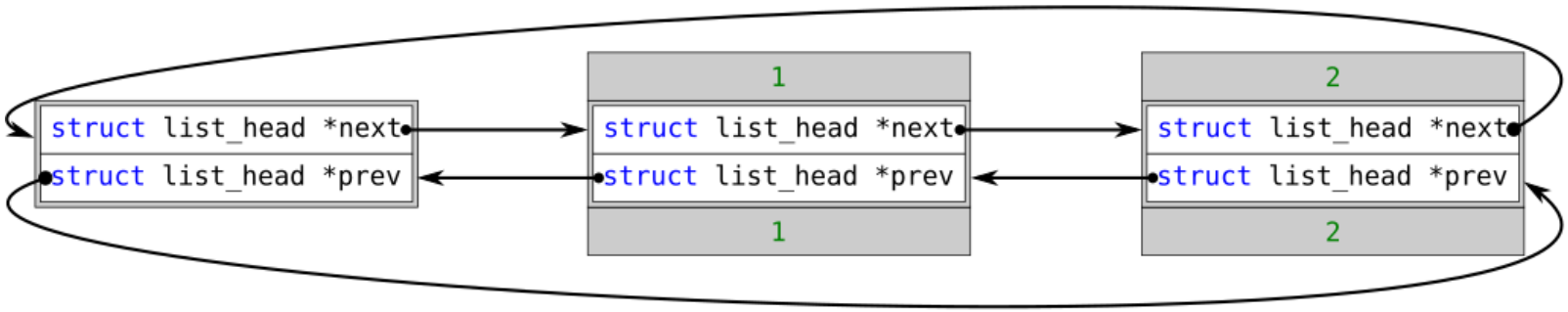





```c
int main()
{
   struct list_head list;
   struct element *element;

   INIT_LIST_HEAD(&list);

   element = element_alloc(1, 1);
   list_add(&(element->list), &list);

   element = element_alloc(2, 2);
   list_add(&(element->list), &list);

   list_for_each_entry(element, &list, list)
      printf("{%d, %d}\n", element->a, element->b);

   return 0;
}
```

# Ejemplo






```
1  int main()
2  {
3    struct list_head list;
4    struct element *element;
5
6    INIT_LIST_HEAD(&list);
7
8    element = element_alloc(1, 1);
9    list_add(&(element->list), &list);
10
11   element = element_alloc(2, 2);
12   list_add(&(element->list), &list);
13
14   list_for_each_entry(element, &list, list)
15     printf("{%d, %d}\n", element->a, element->b);
16
17   return 0;
18 }
```

# Ejemplo

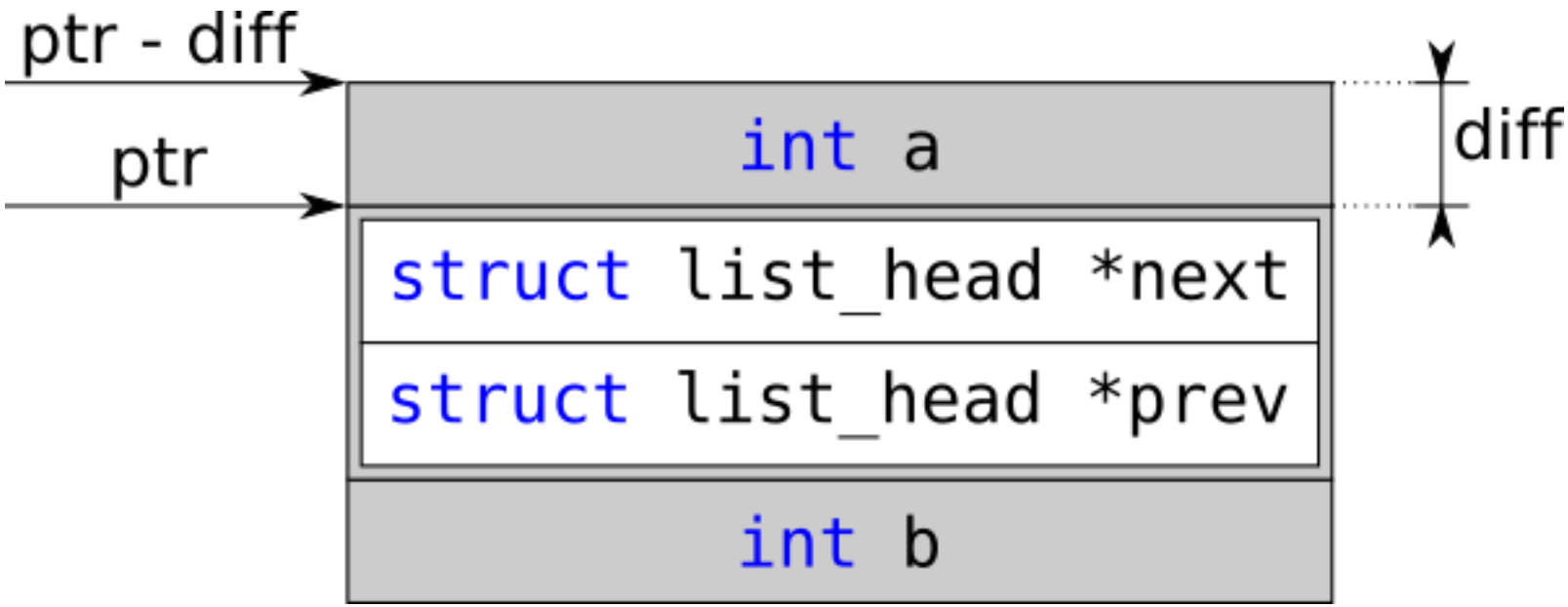


```
> list_example
{2, 2}
{1, 1}
```

```c
int main()
{
  struct list_head list;
  struct element *element;

  INIT_LIST_HEAD(&list);

  element = element_alloc(1, 1);
  list_add(&(element->list), &list);

  element = element_alloc(2, 2);
  list_add(&(element->list), &list);

  list_for_each_entry(element, &list, list)
    printf("{%d, %d}\n", element->a, element->b);

  return 0;
}
```

# ¿Cómo se obtiene el elemento a partir del `list_head`?






```
1 /**
2  * list_entry - get the struct for this entry
3  * @ptr:  the &struct list_head pointer.
4  * @type: the type of the struct this is embedded in.
5  * @member: the name of the list_struct within the struct.
6  */
7 #define list_entry(ptr, type, member) \
8   ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

# Funciones principales



- **`list_add`**: Añade **después** de la cabeza (**pila**)
- **`list_add_tail`**: Añade **antes** de la cabeza (**cola**)
- **`list_move`**: Mueve un elemento de una lista a **después** de la cabeza de otra (**pila**)
- **`list_move_tail`**: Mueve un elemento de una lista a **antes** de la cabeza de otra (**cola**)
- **`list_del`**: Borra el nodo que recibe
- **`list_splice`**: Une dos listas
- **`list_empty`**: Comprueba si una lista está vacía

# Macros principales



- **`INIT_LIST_HEAD`**: Inicializa la cabeza de una lista
- **`list_for_each`**: Recorre cada **nodo** de una lista
- **`list_for_each_safe`**: Recorre cada **nodo** de una lista y se puede borrar un elemento mientras se rocorre la lista.
- **`list_for_each_entry`**: Recorre cada **elemento** de una lista
- **`list_for_each_entry_safe`**: Recorre cada **elemento** de una lista y se puede borrar un elemento mientras se rocorre la lista.