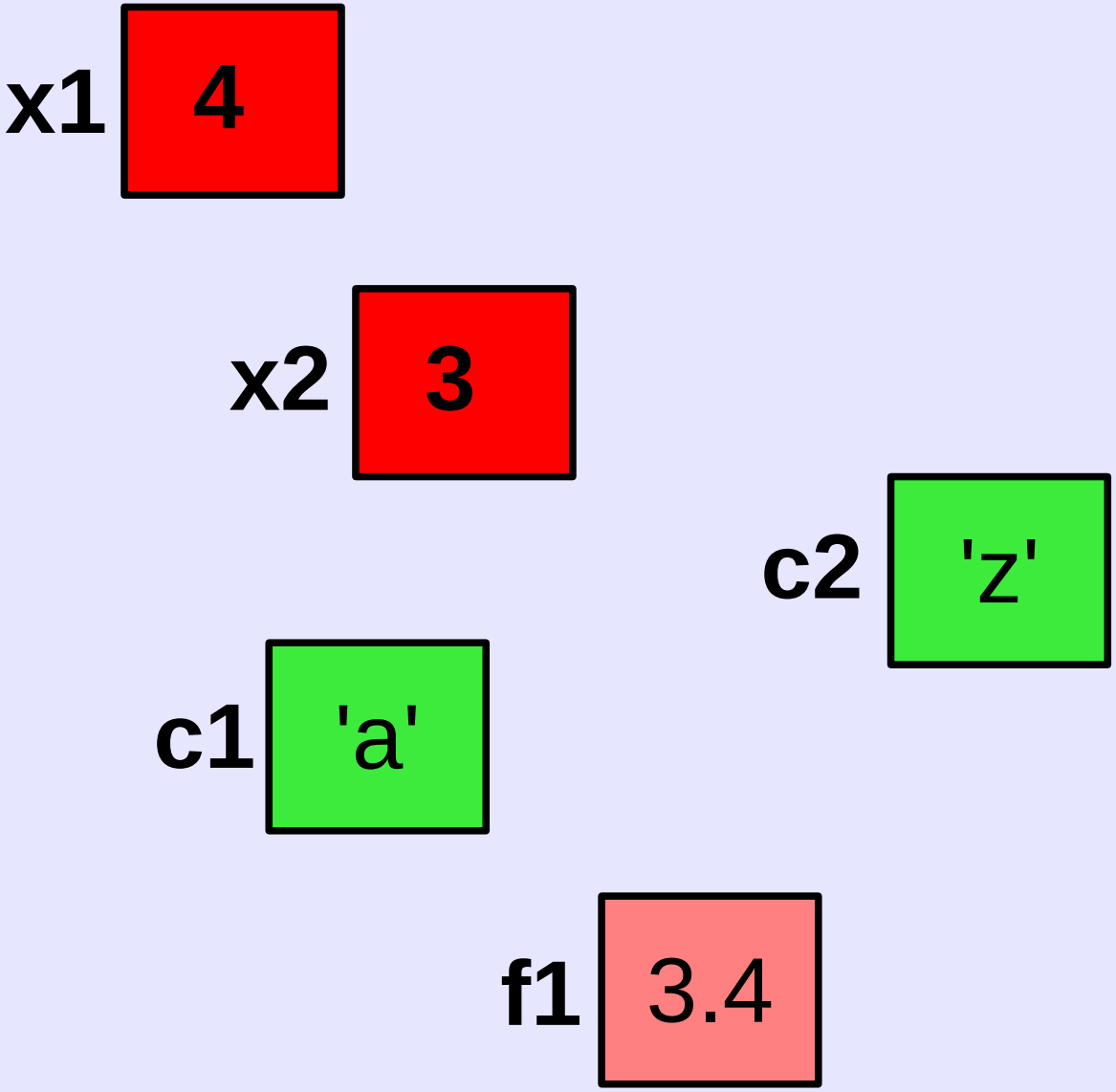
Introducción a punteros en C

Algoritmos y Estructuras de Datos II

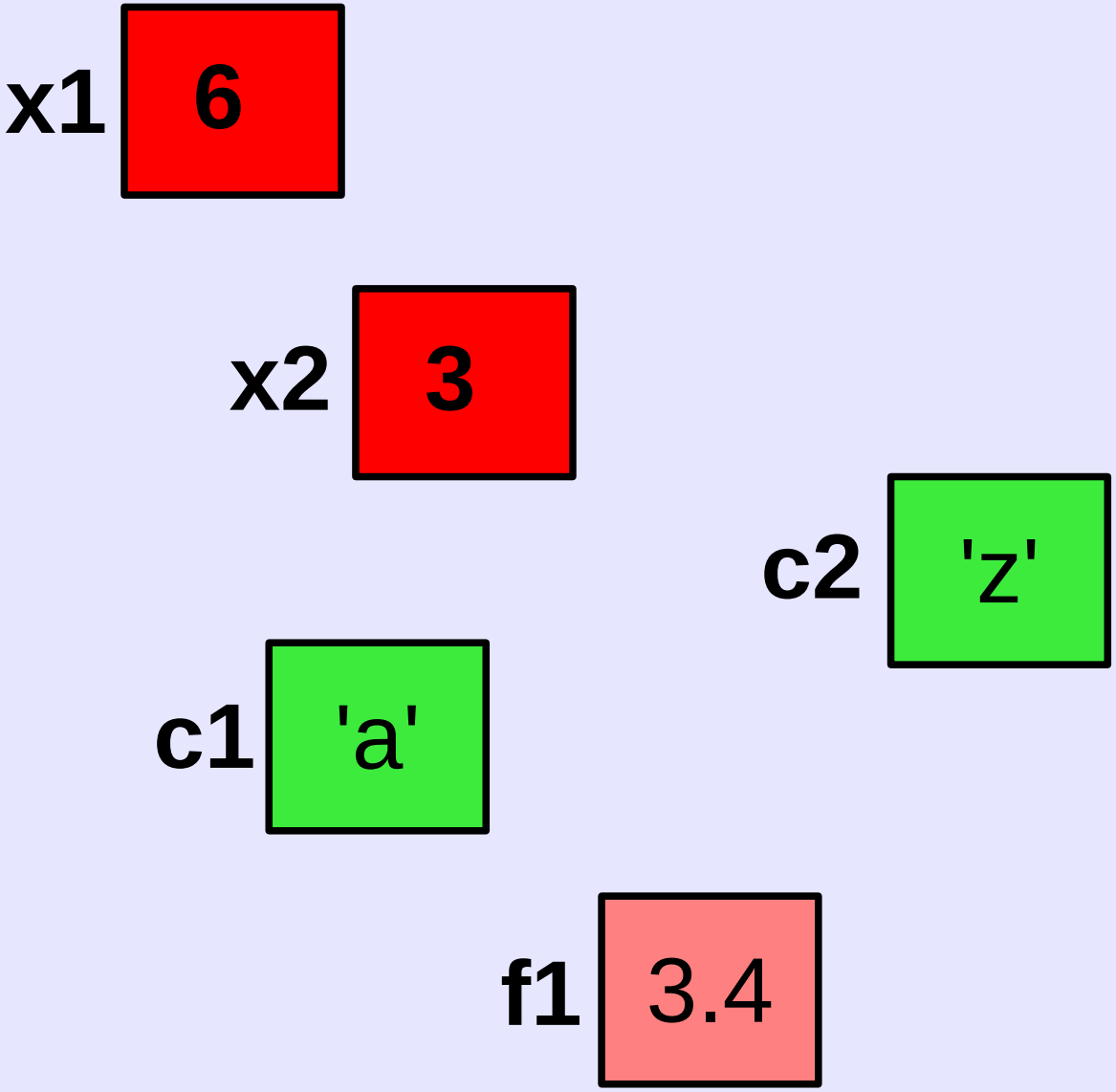
Leonardo M. Rodríguez

PARTE I: Uso básico de punteros

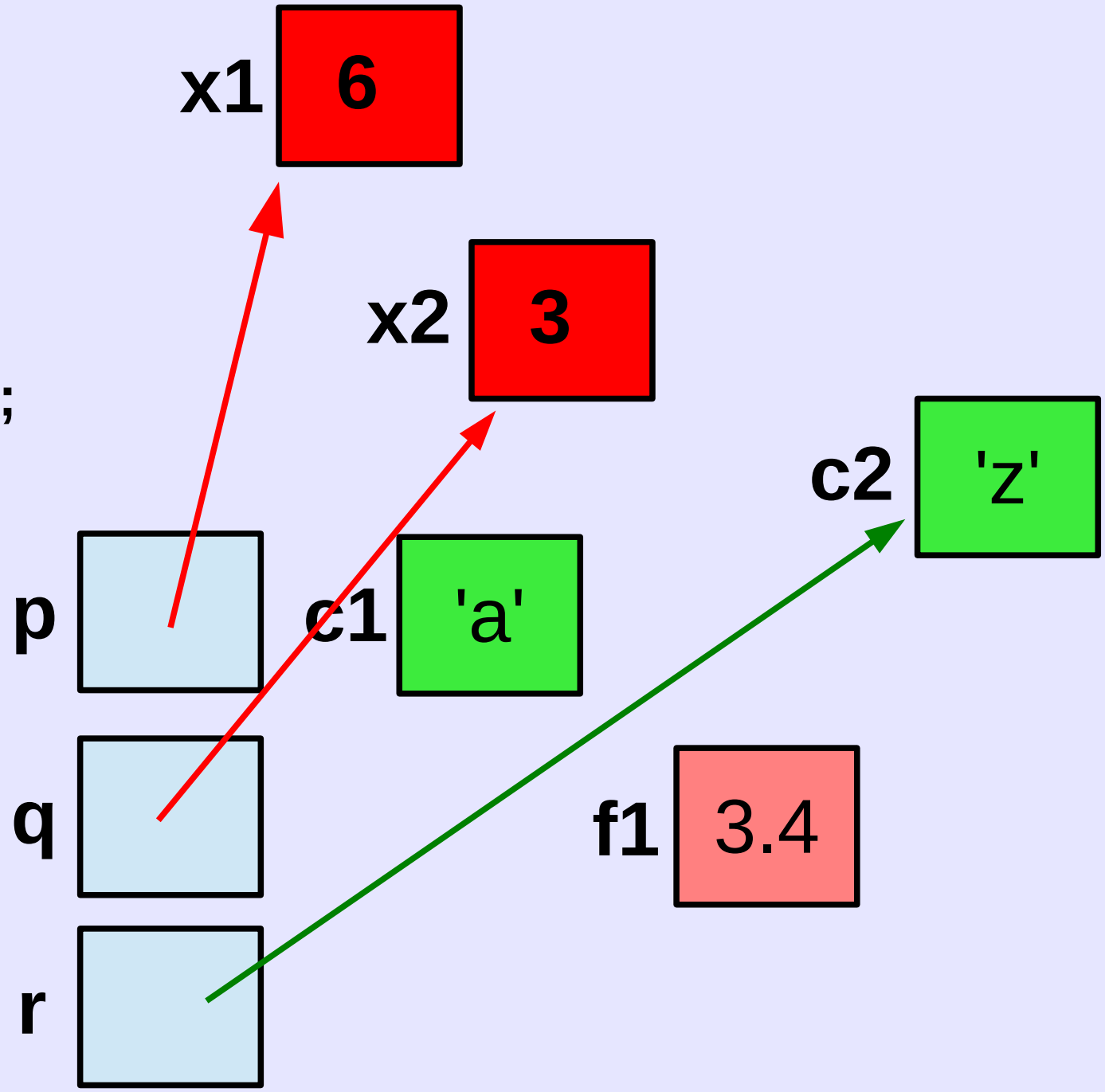
```
int x1 = 4;
int x2 = 3;

char c1 = 'a';
char c2 = 'z';

float  f1 = 3.4;

x1 = x2 * 2;
```



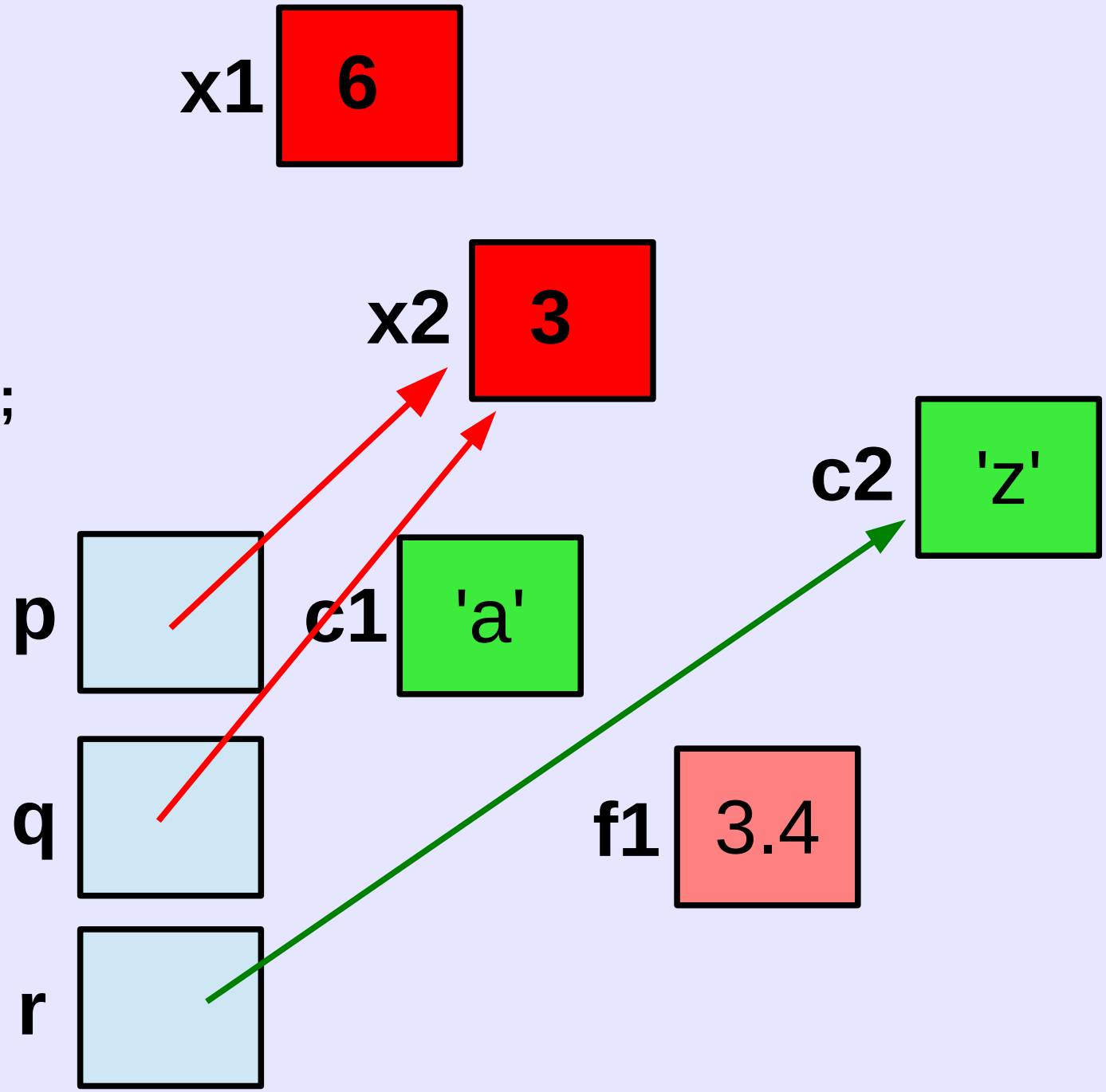
```
int x1 = 4;
int x2 = 3;

char c1 = 'a';
char c2 = 'z';

float  f1 = 3.4;

x1 = x2 * 2;
```



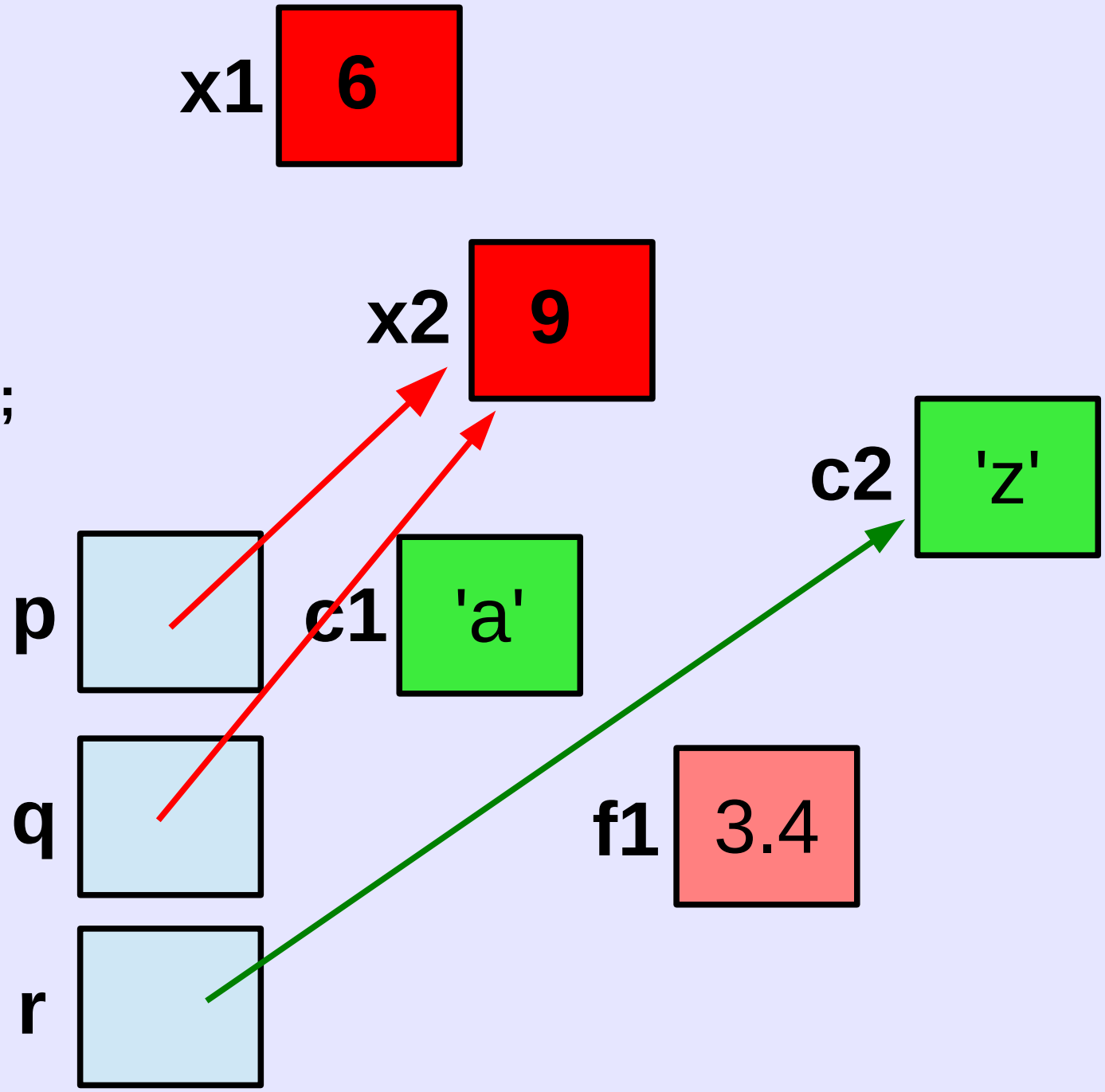
```
int *p = & x1;
int *q = & x2;
char *r = & c2;

p = q;
```



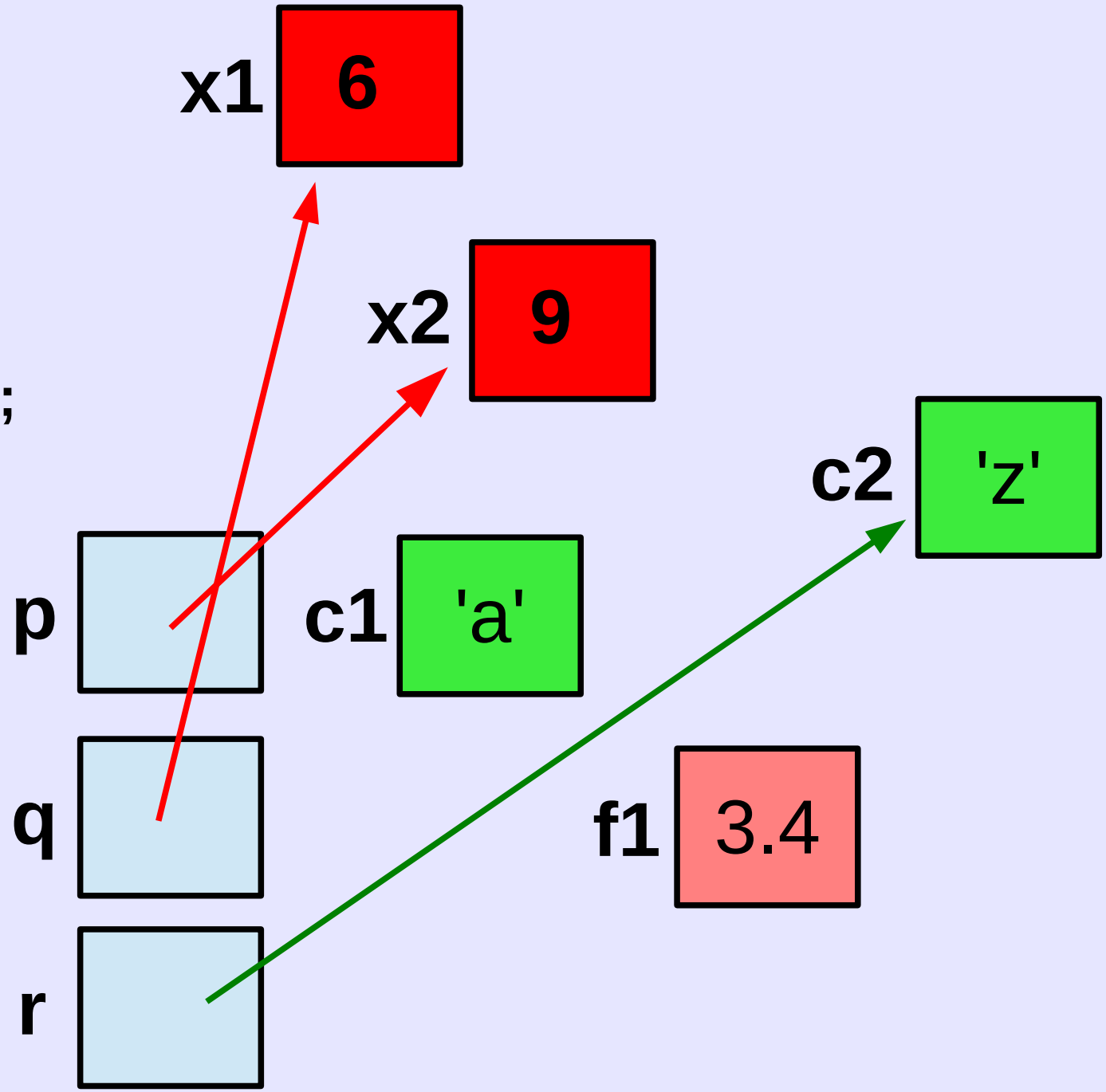
```
int *p = & x1;
int *q = & x2;
char *r = & c2;

p = q;
*p = 9;
```



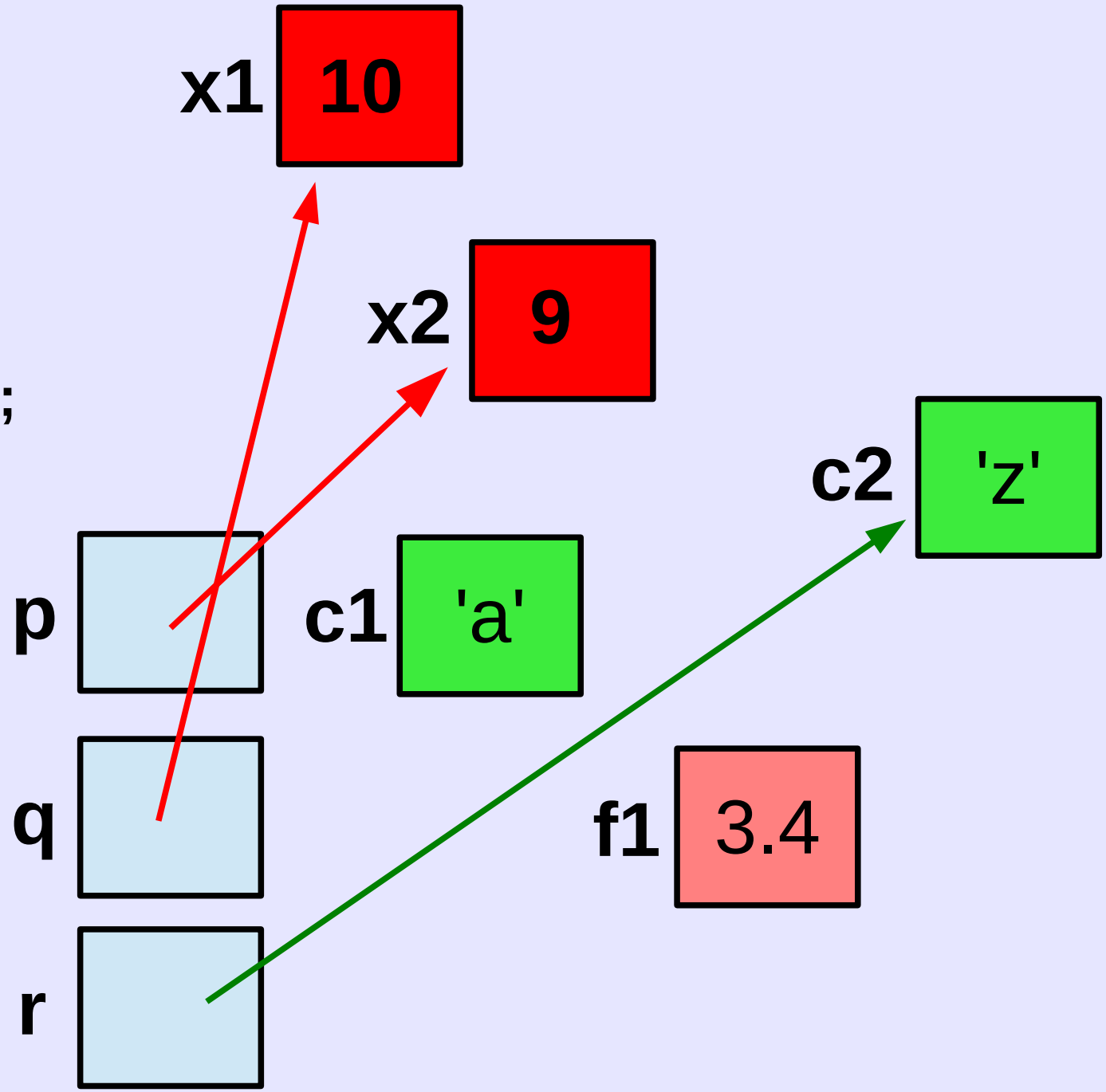
```
int *p = & x1;
int *q = & x2;
char *r = & c2;

p = q;
*p = 9;
q = & x1;
```



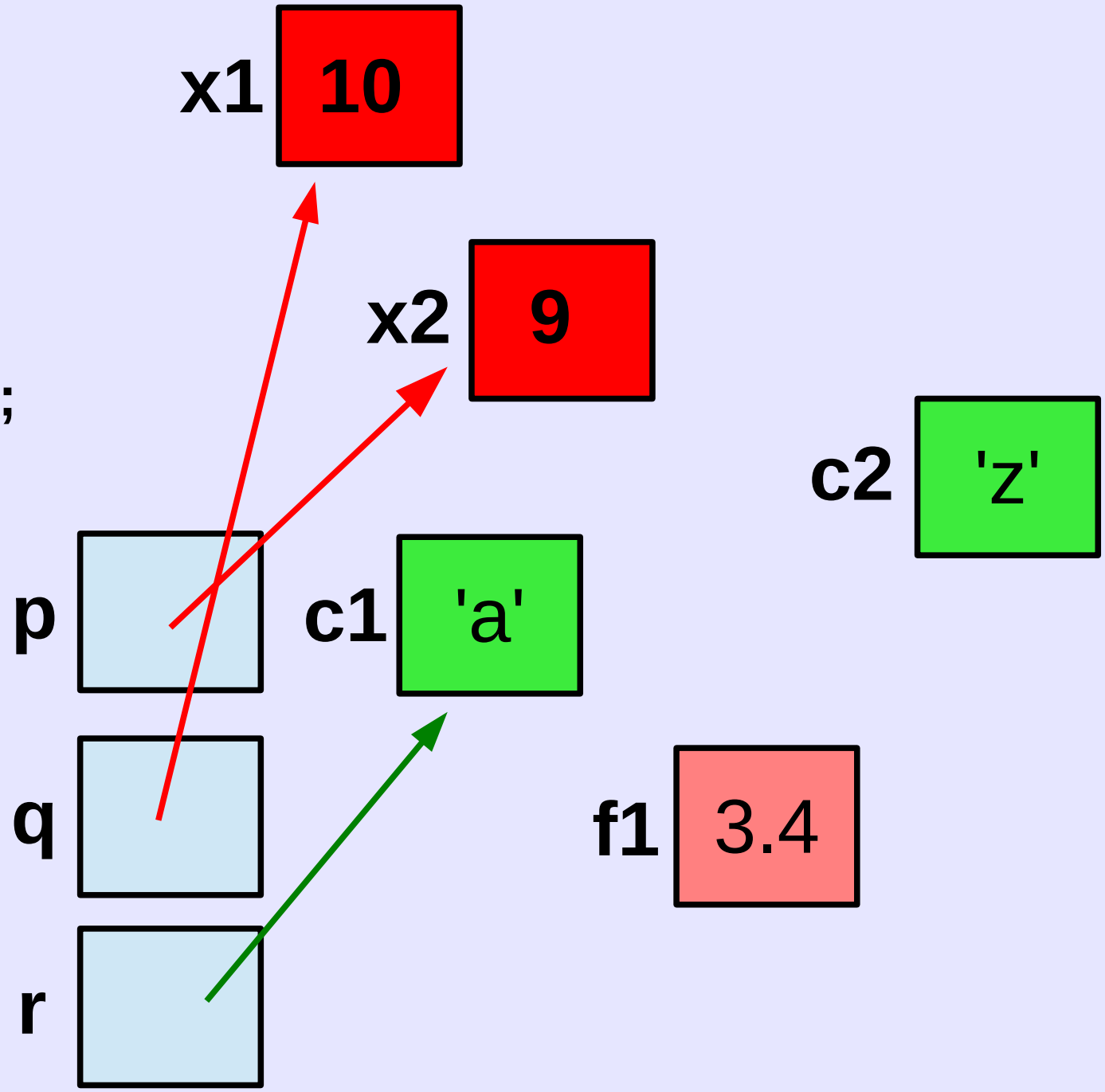
```
int *p = & x1;
int *q = & x2;
char *r = & c2;

p = q;
*p = 9;
q = & x1;
*q = *p + 1;
```



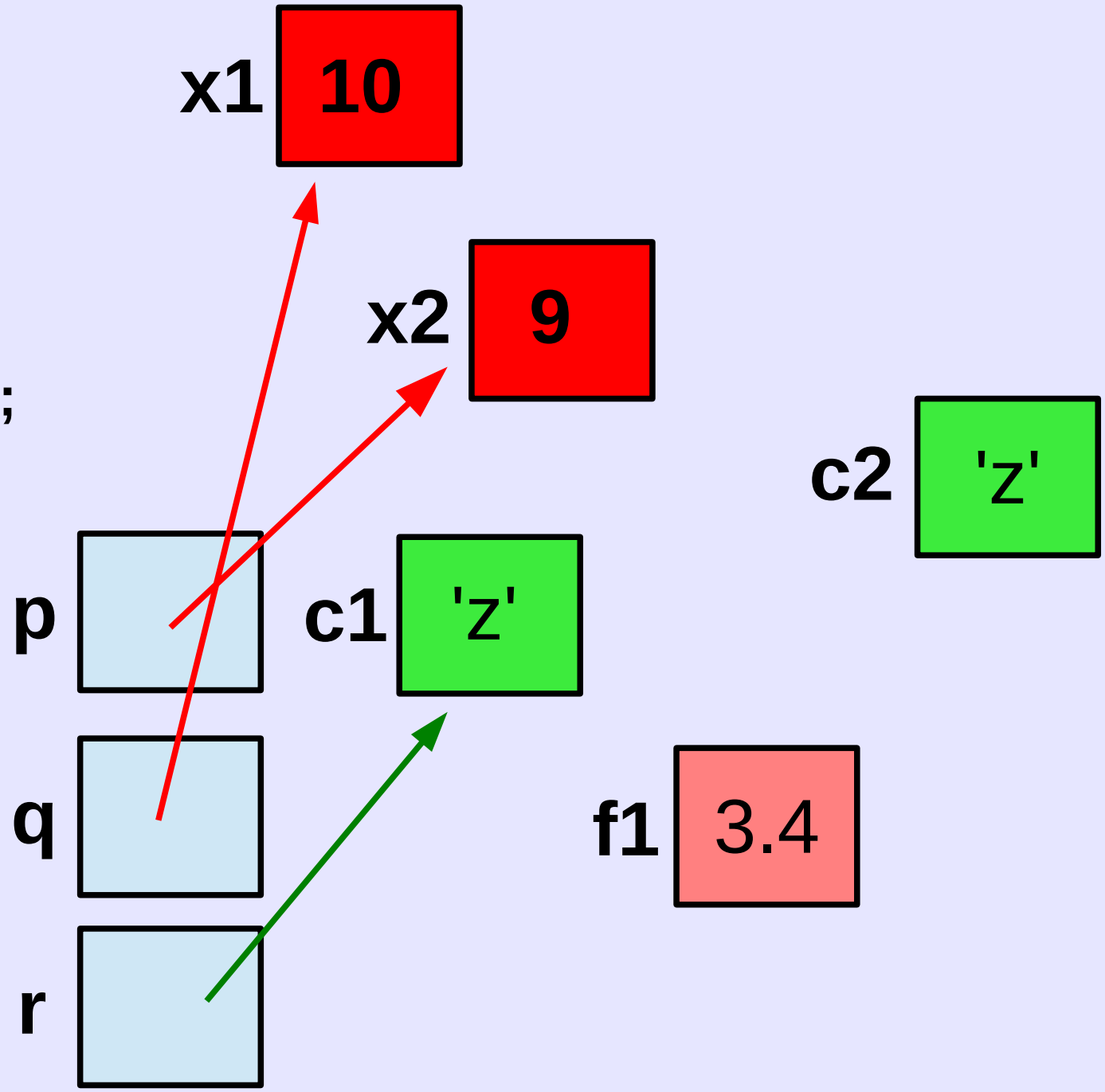
```
int *p = & x1;
int *q = & x2;
char *r = & c2;

p = q;
*p = 9;
q = & x1;
*q = *p + 1;
r = & c1;
```



```
int *p = & x1;
int *q = & x2;
char *r = & c2;

p = q;
*p = 9;
q = & x1;
*q = *p + 1;
r = & c1;
*r = c2;
```



```
int *p = & x1;
int *q = & x2;
char *r = & c2;

p = q;
*p = 9;
q = & x1;
*q = *p + 1;
r = & c1;
*r = c2;
```



Checkpoint

int *p; Declaración de un puntero.

En este caso la variable p es un puntero a una variable de tipo entero. El tipo de los punteros (color de la flecha) es muy importante.

(&) Operador de referencia (“crear una flecha”)

(*) Operador de dereferenciación (“seguir la flecha”)

PARTE II: Pasaje de referencias

```
int main (void) {
    int x = 4;
    mitad(x);
    printf("%d\n", x);
    return 0;
}


void mitad(int y) {
        y = y / 2;
}
```

¿Qué número imprime en pantalla este programa?

x 4

```
int main (void) {
    int x = 4;
    mitad(x);
    printf("%d\n", x);
    return 0;
}


void mitad(int y) {
        y = y / 2;
}
```

```
int main (void) {
    int x = 4;
    mitad(x);
    printf("%d\n", x);
    return 0;
}

void mitad(int y) {
        y = y / 2;
}
```

```
int main (void) {
    int x = 4;
    mitad(x);
    printf("%d\n", x);
    return 0;
}

void mitad(int y) {
        y = y / 2;
}
```

```
int main (void) {
    int x = 4;
    mitad(x);
    printf("%d\n", x);
    return 0;
}
```

IMPRIME 4!

x **4**

```
void mitad(int y) {
        y = y / 2;
}
```

y **2**

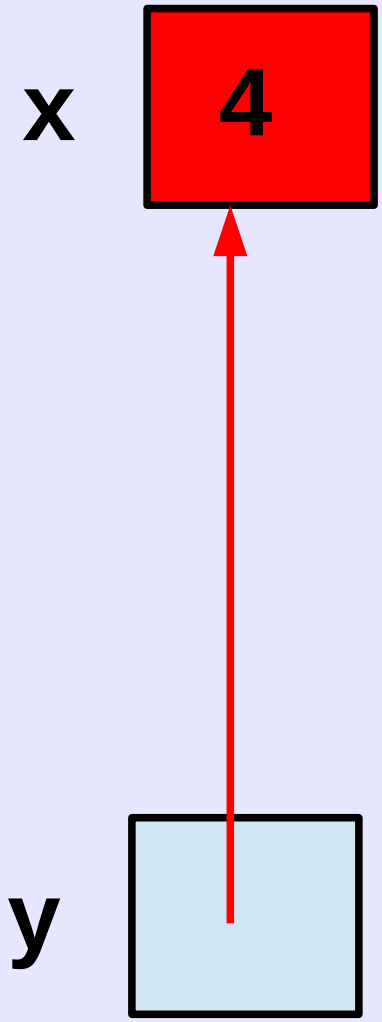
```
int main (void) {
    int x = 4;
    mitad(&x);
    printf("%d\n", x);
    return 0;
}


void mitad(int *y) {
        *y = *y / 2;
}
```

¿Qué número imprime en pantalla este programa?

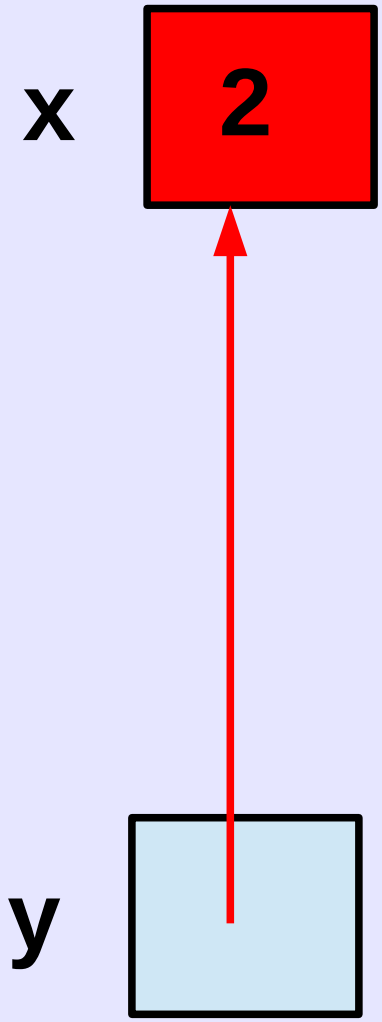
x 4

```
int main (void) {
    int x = 4;
    mitad(&x);
    printf("%d\n", x);
    return 0;
}


void mitad(int *y) {
        *y = *y / 2;
}
```

```
int main (void) {
    int x = 4;
    mitad(&x);
    printf("%d\n", x);
    return 0;
}


void mitad(int *y) {
        *y = *y / 2;
}
```

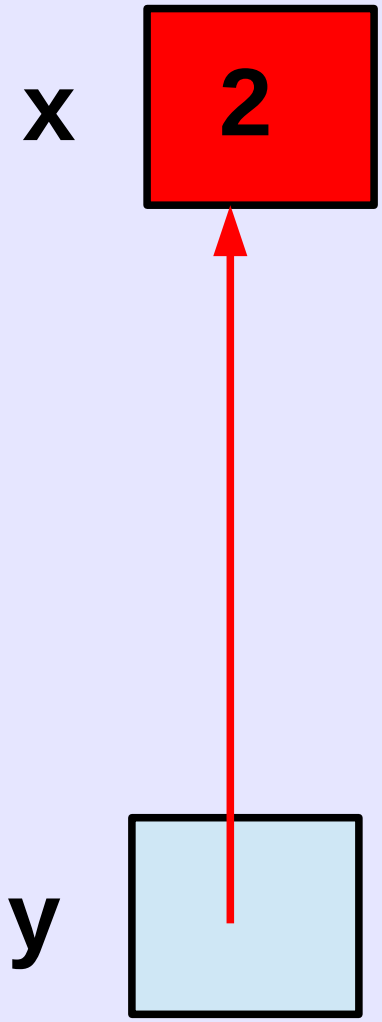


```
int main (void) {
    int x = 4;
    mitad(&x);
    printf("%d\n", x);
    return 0;
}


void mitad(int *y) {
        *y = *y / 2;
}
```



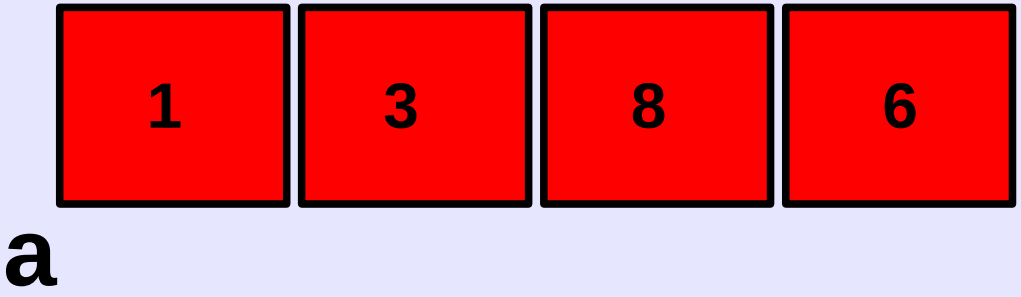
```
int main (void) {
    int x = 4;
    mitad(&x);
    printf("%d\n", x);
    return 0;
}

void mitad(int *y) {
        *y = *y / 2;
}
```

IMPRIME 2!

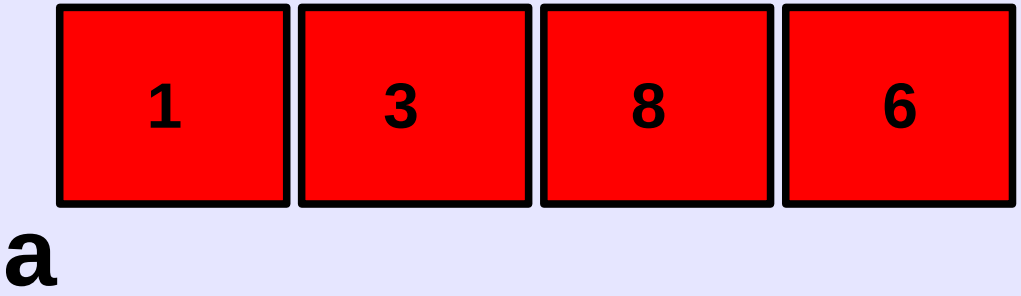


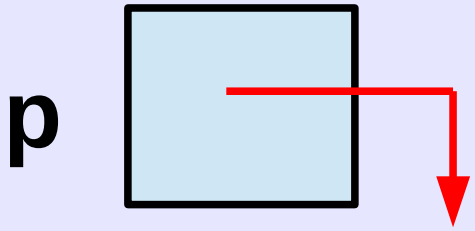
PARTE III: Punteros y arreglos

```
int a[4] = {1,3,8,6};

int *p = NULL;
```

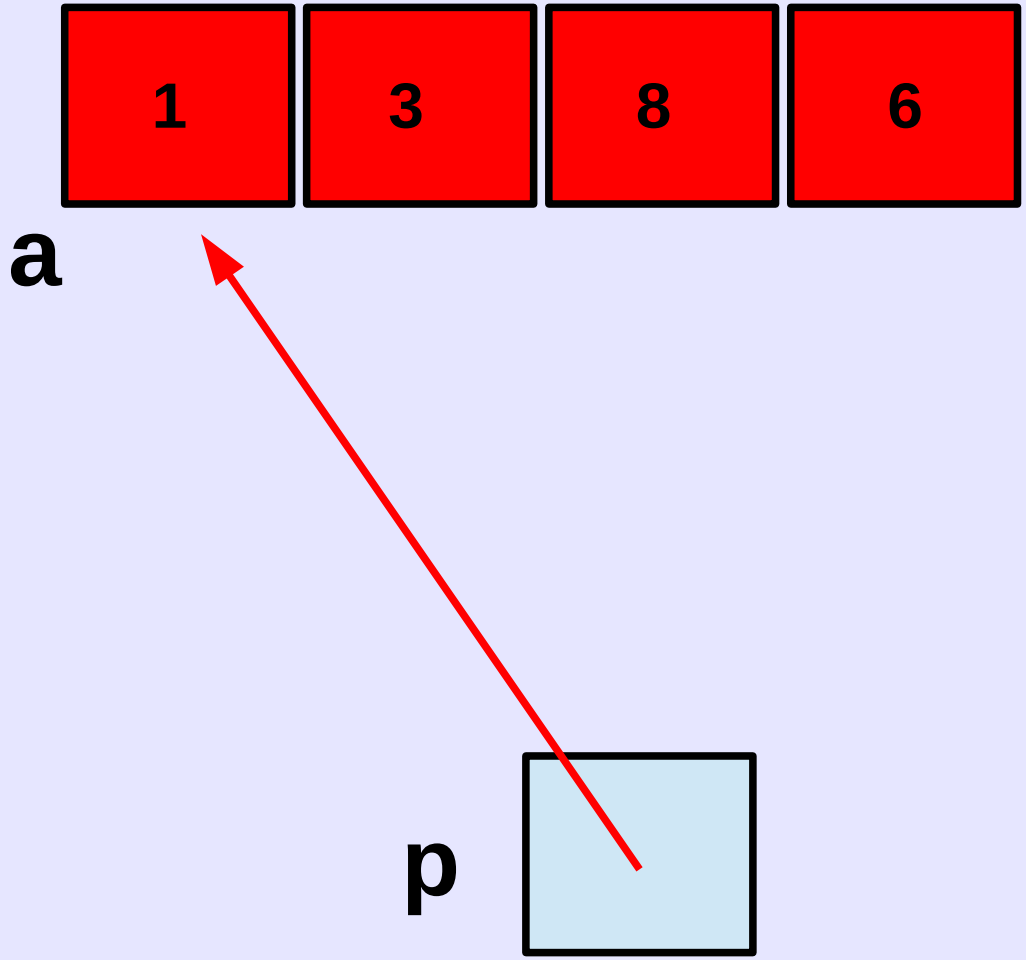


```
int a[4] = {1,3,8,6};

int *p = NULL;

p = a;
```

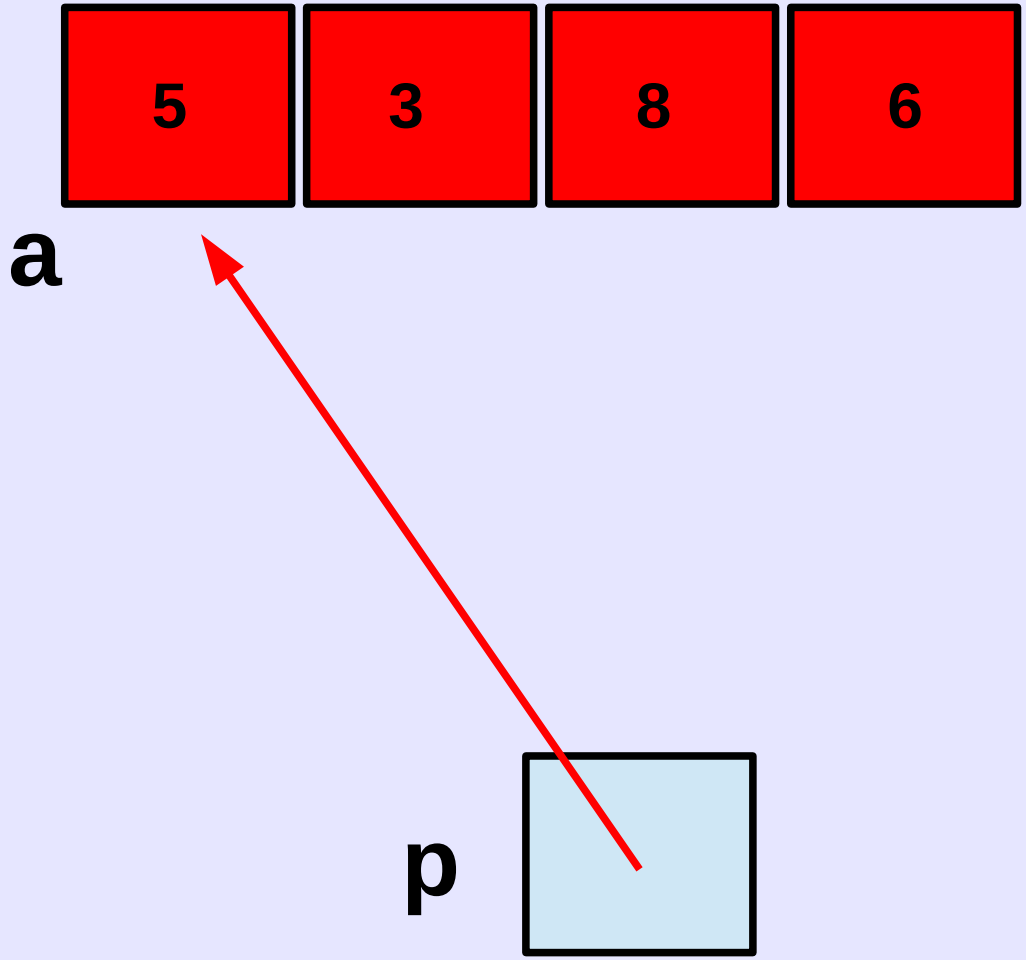




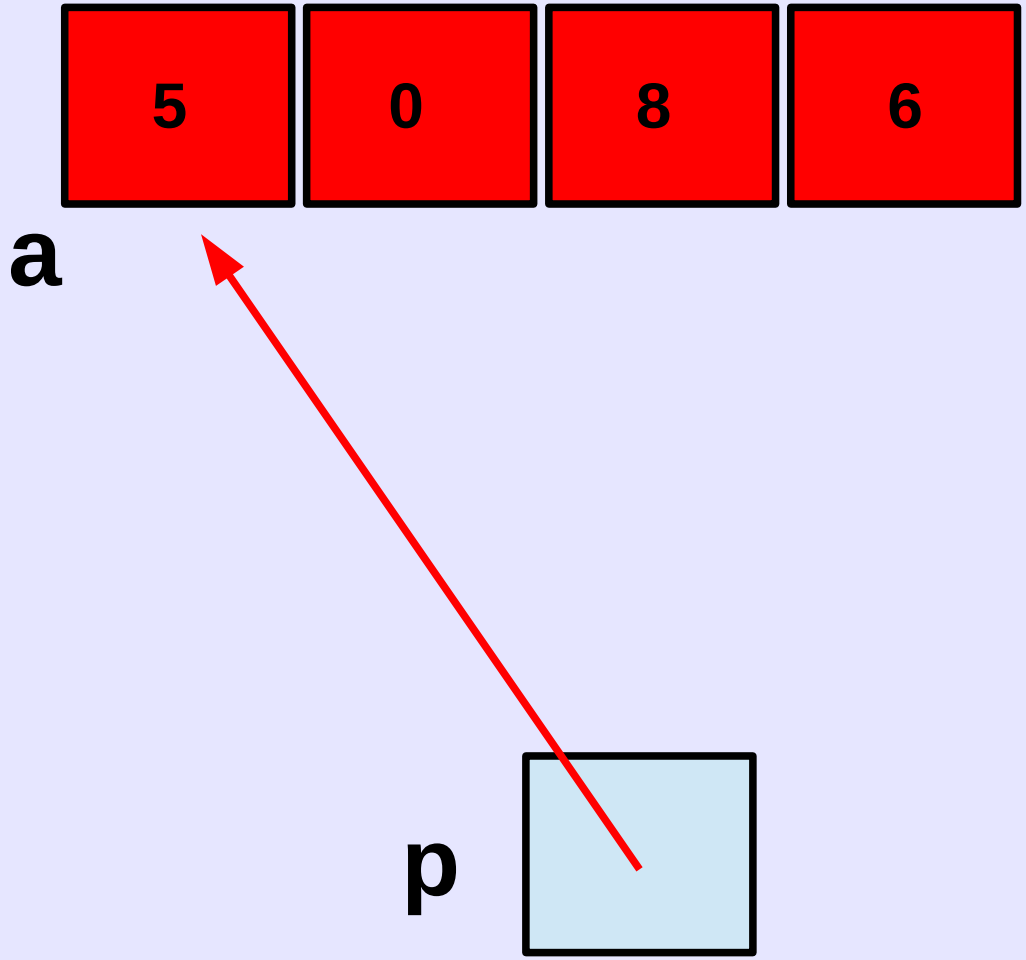
```
int a[4] = {1,3,8,6};

int *p = NULL;

p = a;

*p = 5;
```



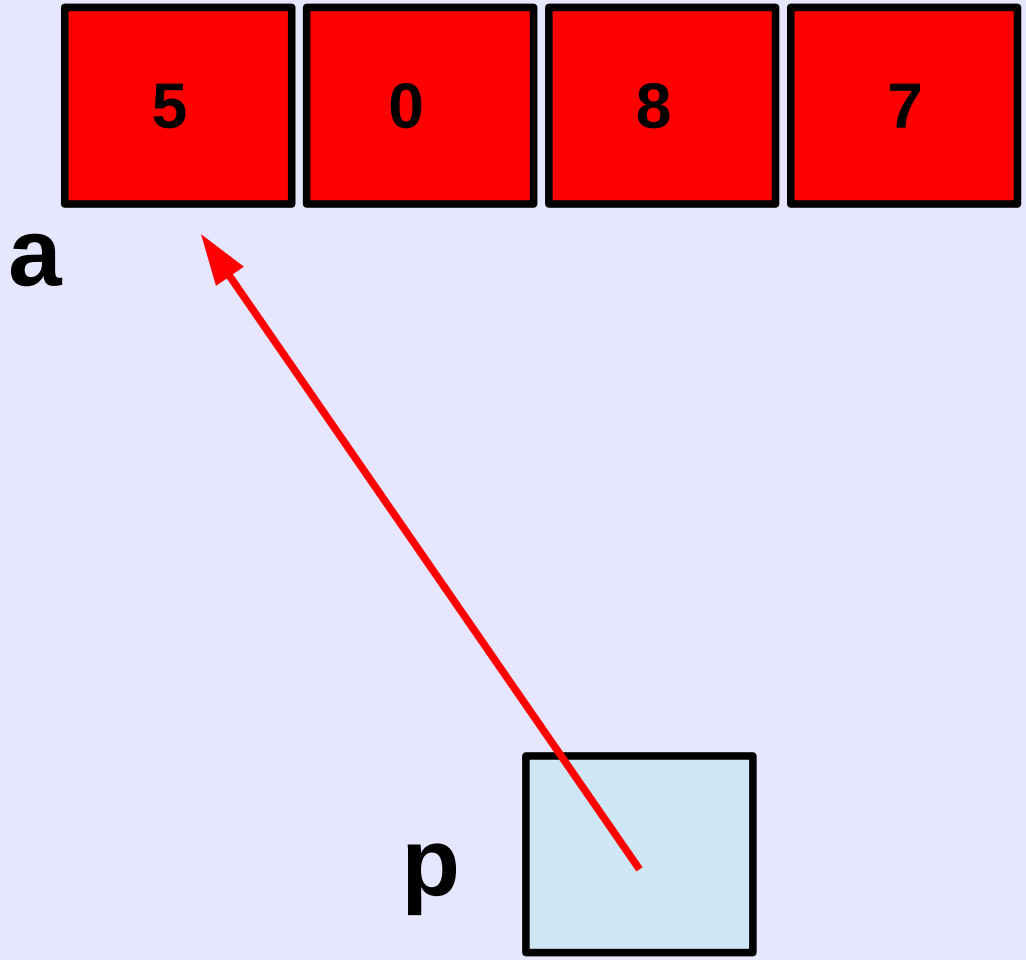
```
int a[4] = {1,3,8,6};

int *p = NULL;

p = a;

*p = 5;

*(p + 1) = 0;
```



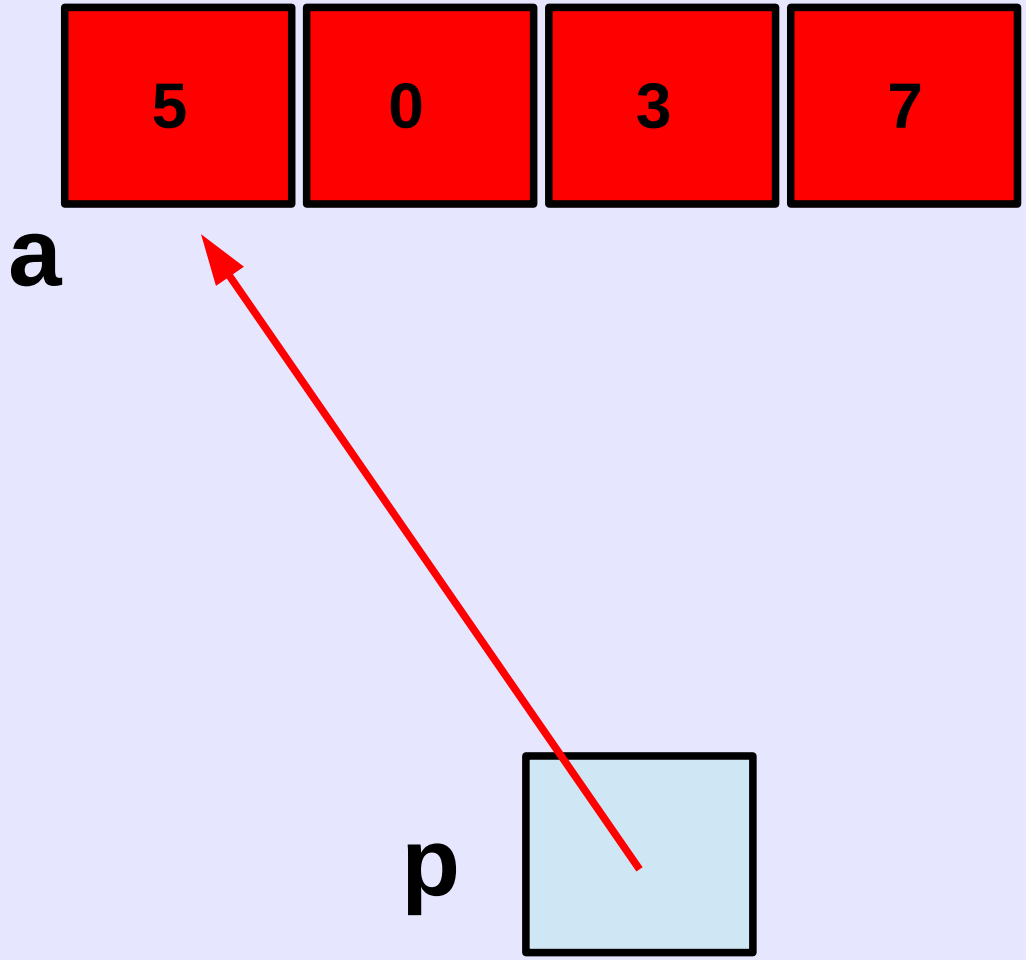
```
int a[4] = {1,3,8,6};

int *p = NULL;

p = a;

*p = 5;

*(p + 1) = 0;

*(p + 3) = 7;
```



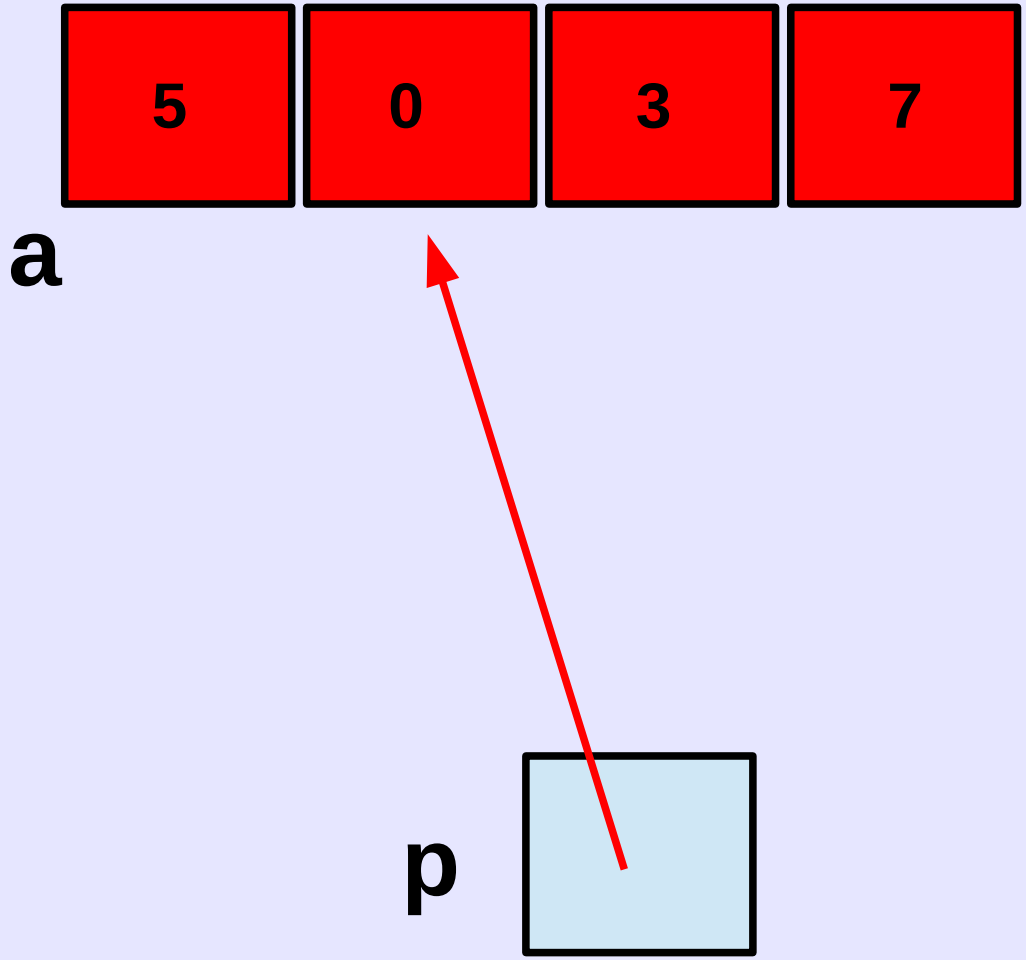
```
int a[4] = {1,3,8,6};

int *p = NULL;

p = a;

*p = 5;

*(p + 1) = 0;

*(p + 3) = 7;

p[2] = 3;
```



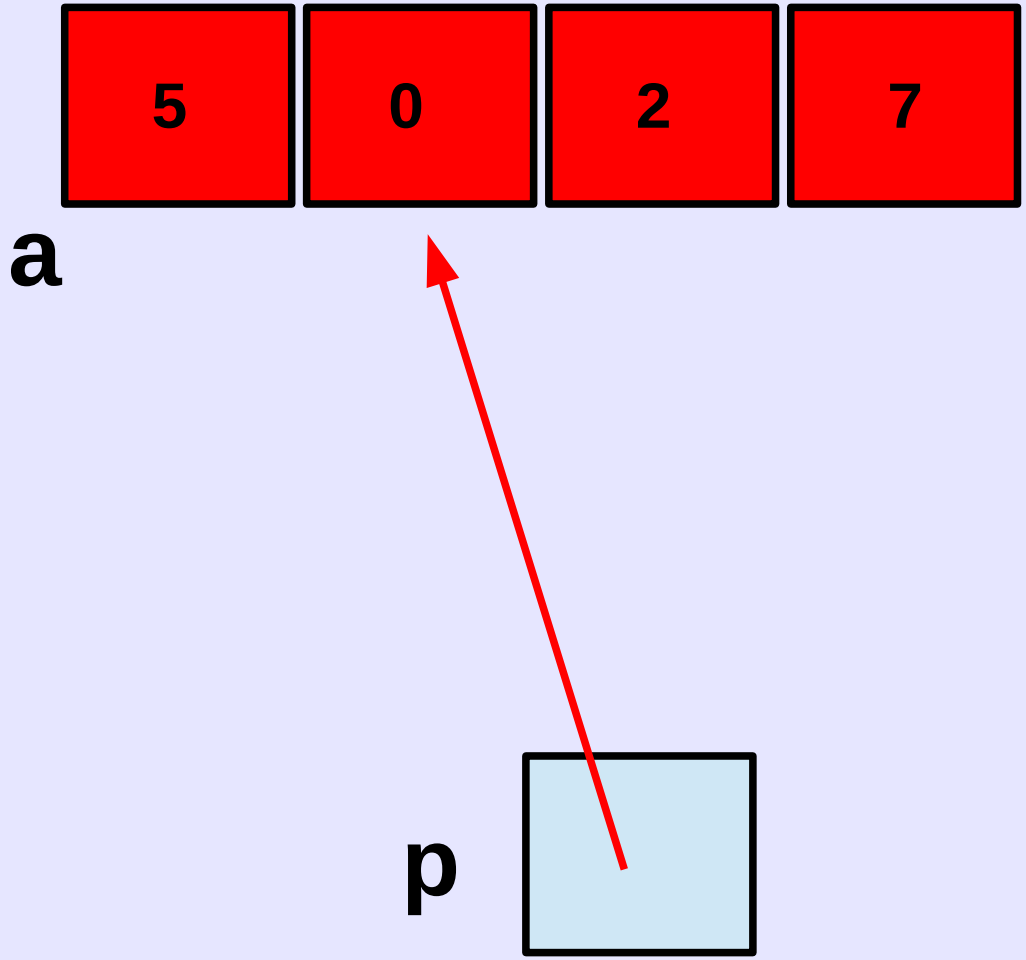
```
int a[4] = {1,3,8,6};

int *p = NULL;

p = a;

*p = 5;

*(p + 1) = 0;

*(p + 3) = 7;

p[2] = 3;

p = p + 1;
```



```
int a[4] = {1,3,8,6};

int *p = NULL;

p = a;

*p = 5;

*(p + 1) = 0;

*(p + 3) = 7;

p[2] = 3;

p = p + 1;

p[1] = 2;
```



```
int a[4] = {1,3,8,6};

int *p = NULL;

p = a;

*p = 5;

*(p + 1) = 0;

*(p + 3) = 7;

p[2] = 3;

p = p + 1;

p[1] = 2;
```



Checkpoint

Arreglos y punteros

int a[4] = {1,3,5,8}

En la asignación siguiente, el arreglo “decae” o “se convierte” a un puntero a su primer elemento.

int *p = a;

Un poco de aritmética de punteros

Nos aprovechamos que los arreglos siempre se alojan en memoria contigua (consecutiva)

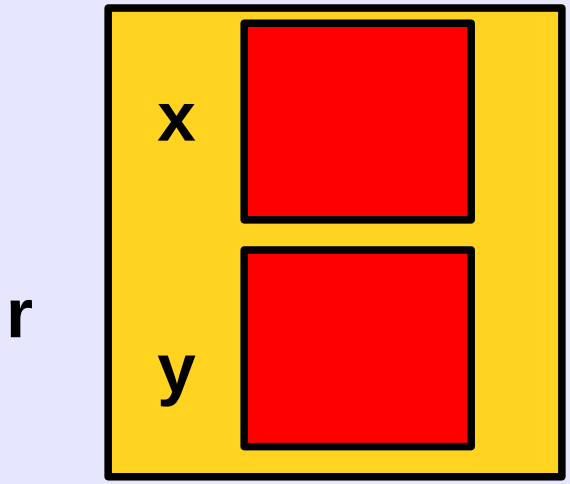
*(p + 1) = 4
“Seguir la flecha de p y moverse 1 lugar a la derecha”

Esa operación es equivalente a usar la notación de arreglos:
p[1] = 4;

p = p + 1;
“Desplazar la flecha 1 lugar a la derecha”

PARTE IV: Punteros y estructuras

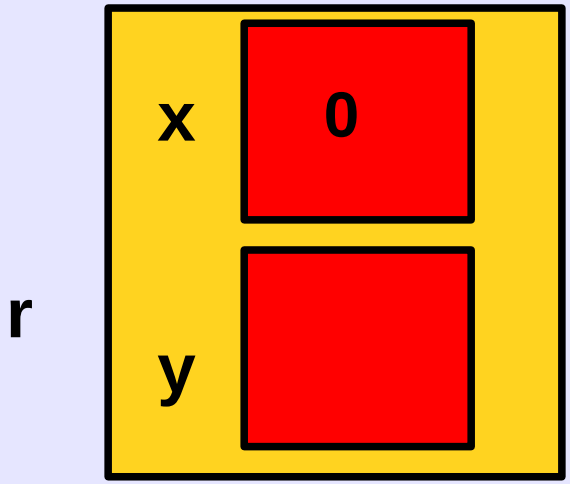
```
struct _punto {
    int x;
    Int y;
};
```




```
struct _punto r;
```

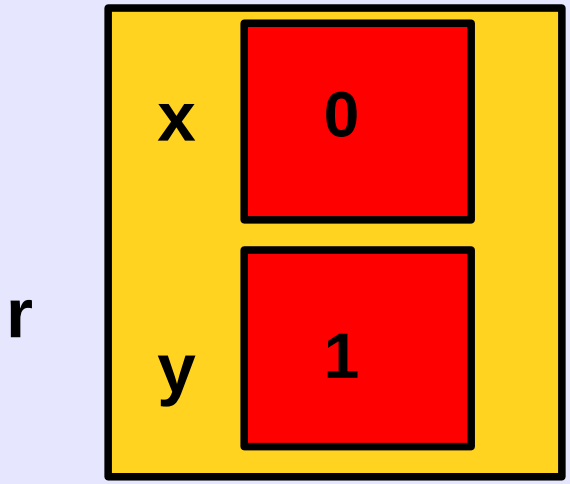
```
r.x = 0;
```

```
struct _punto {
    int x;
    Int y;
};
```



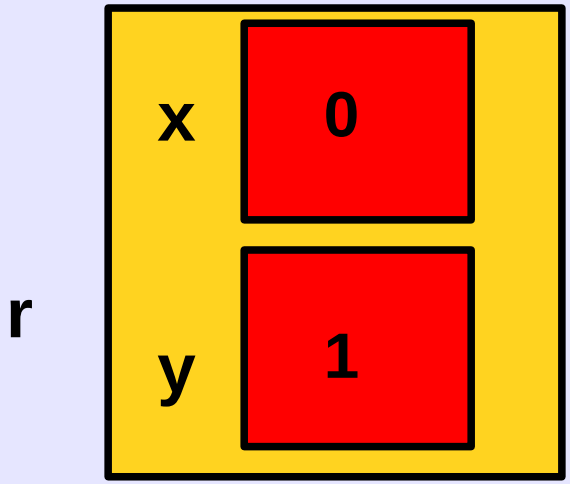

```
struct _punto r;

r.x = 0;

r.y  = 1;
```

```
struct _punto {
    int x;
    Int y;
};
```

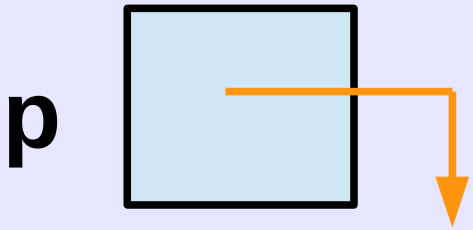



```
struct _punto r;

r.x = 0;

r.y  = 1;

struct _punto *p = NULL;
```

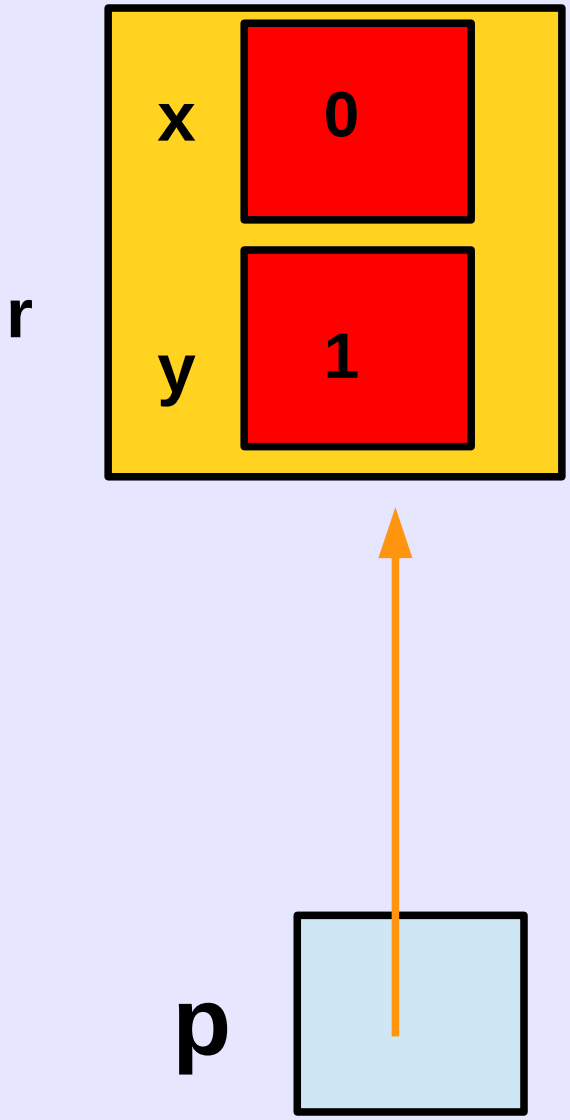
```
struct _punto {
    int x;
    Int y;
};
```




```
struct _punto r;

r.x = 0;

r.y  = 1;

struct _punto *p = NULL;

p = & r;
```

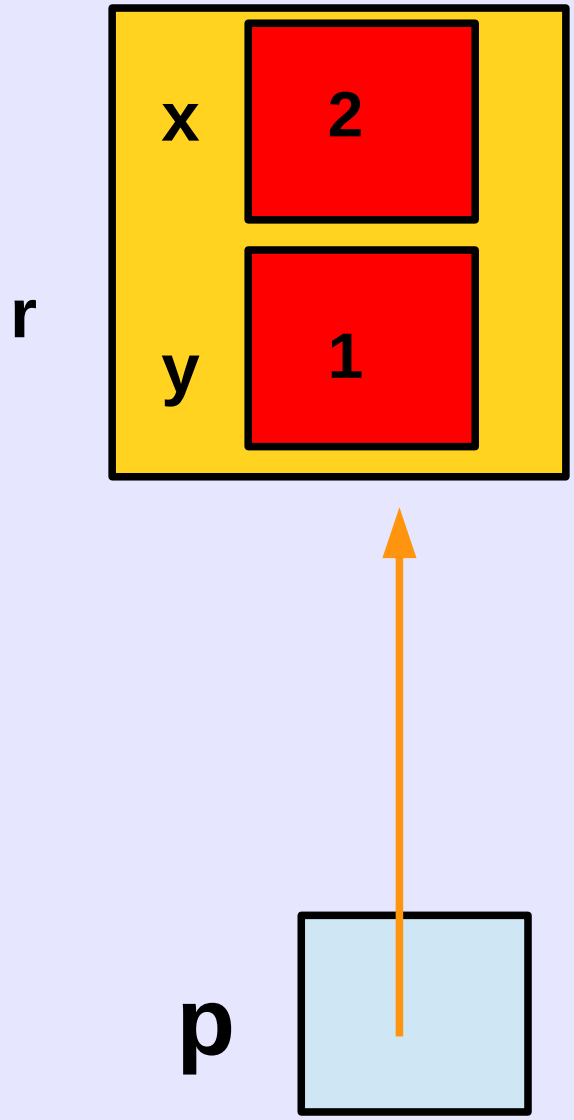


```
struct _punto {
    int x;
    Int y;
};
```



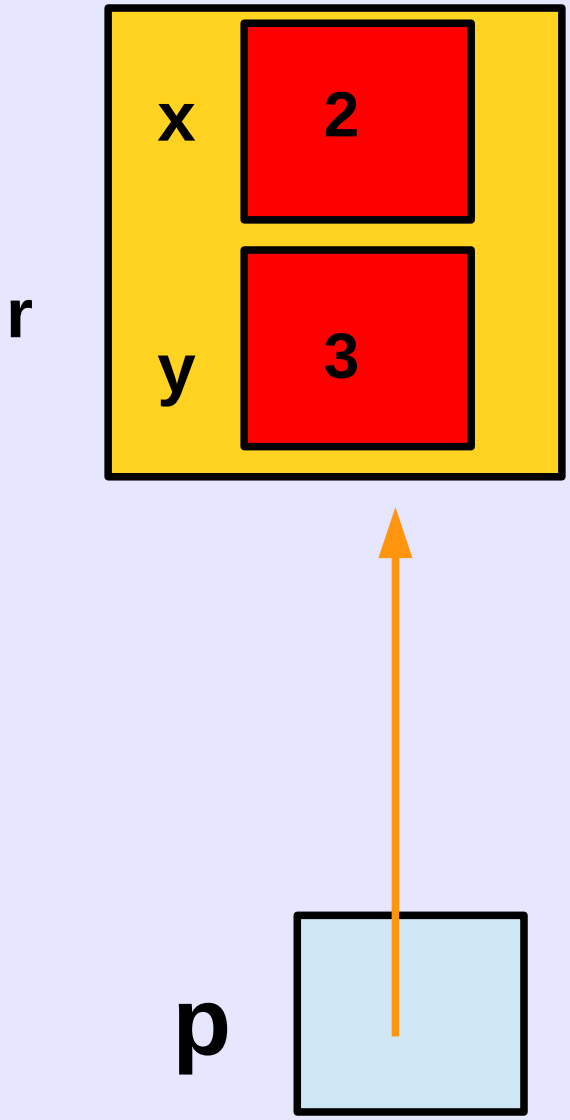

```
struct _punto r;

r.x = 0;

r.y  = 1;

struct _punto *p = NULL;

p = & r;

(*p).x = 2;
```

```
struct _punto {
    int x;
    Int y;
};
```



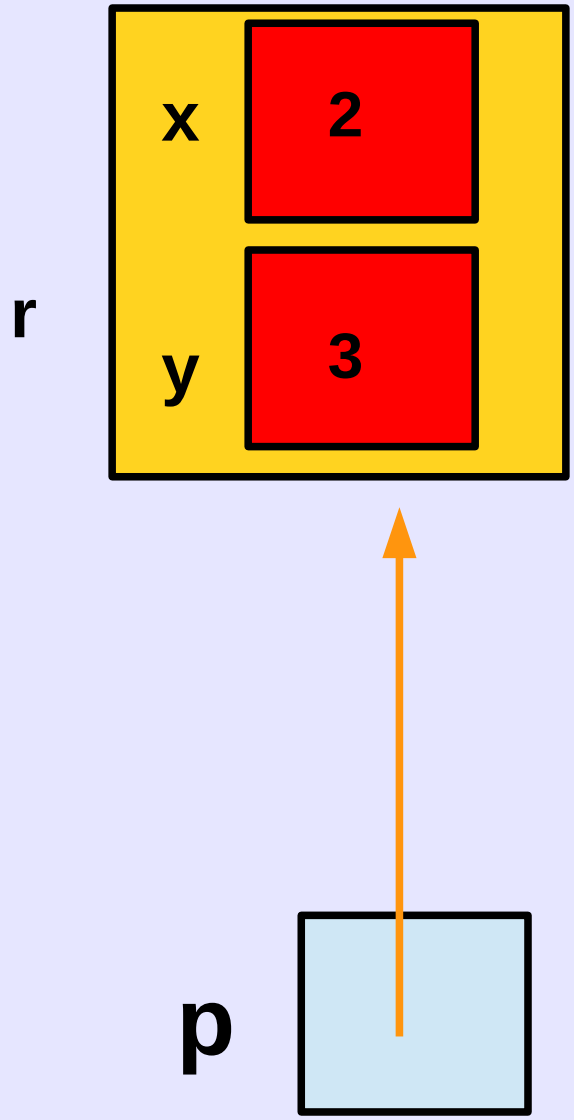

```
struct _punto r;

r.x = 0;

r.y  = 1;

struct _punto *p = NULL;

p = & r;

(*p).x = 2;

p->y = 3;
```

```
struct _punto {
    int x;
    Int y;
};
```

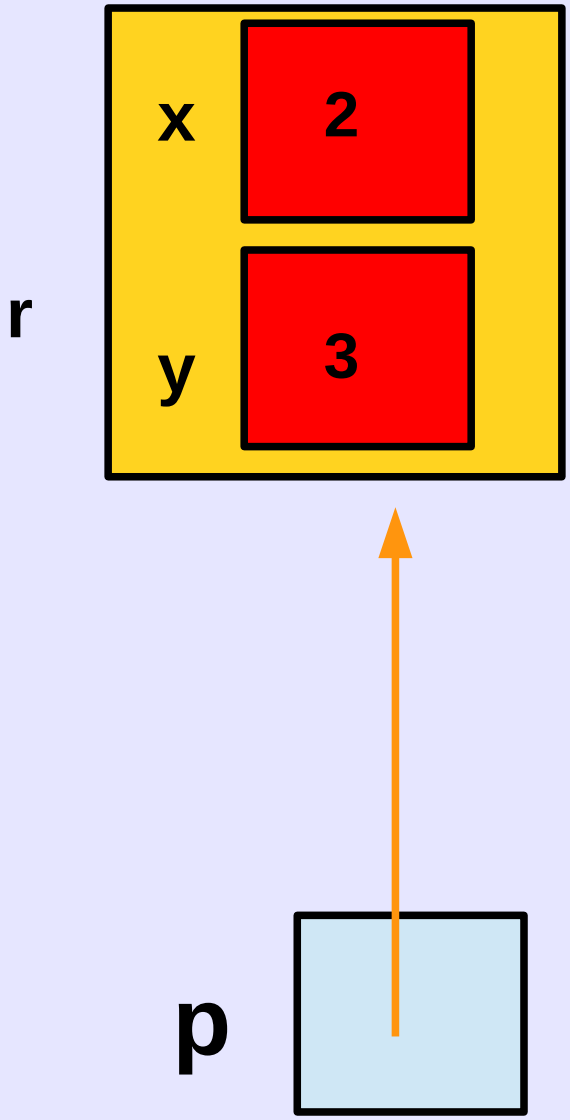



```
struct _punto r;

r.x = 0;

r.y  = 1;

struct _punto *p = NULL;

p = & r;

(*p).x = 2;

p->y = 3;
```

```
struct _punto {
    int x;
    Int y;
};

typedef struct _punto *punto_t;

struct _punto r;

r.x = 0;

r.y  = 1;

struct _punto *p = NULL;

p = & r;

(*p).x = 2;

p->y = 3;
```



```
struct _punto {
    int x;
    Int y;
};

typedef struct _punto *punto_t;

struct _punto r;

r.x = 0;

r.y  = 1;

punto_t p = NULL;

p = & r;

(*p).x = 2;

p->y = 3;
```



Checkpoint

Punteros y estructuras

El operador "->" sirve para acceder al campo de una estructura a partir de un puntero.

(*p).x es equivalente a p->x

De paso vimos...

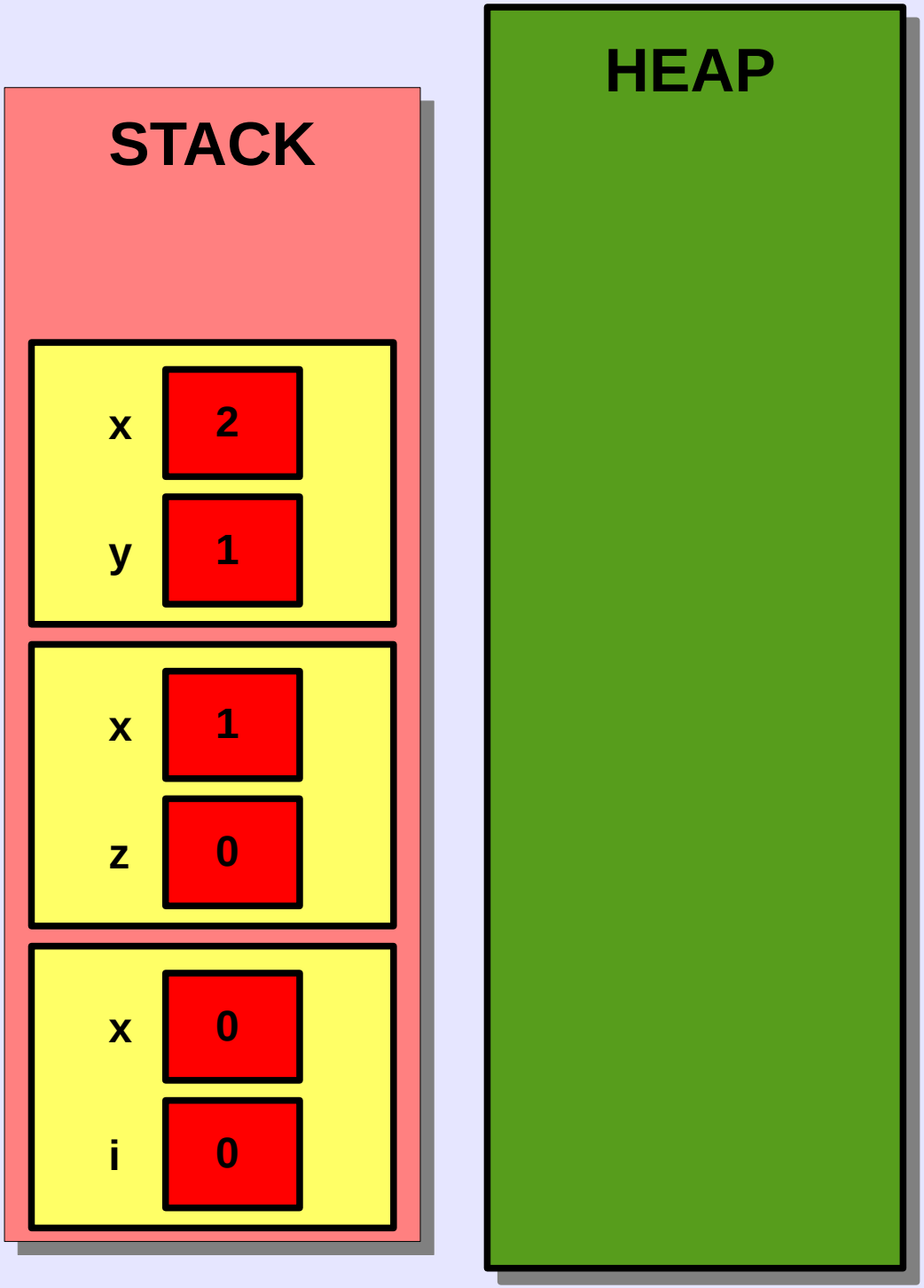
Sinónimos de tipos

typedef es útil para hacer sinónimos de tipos.

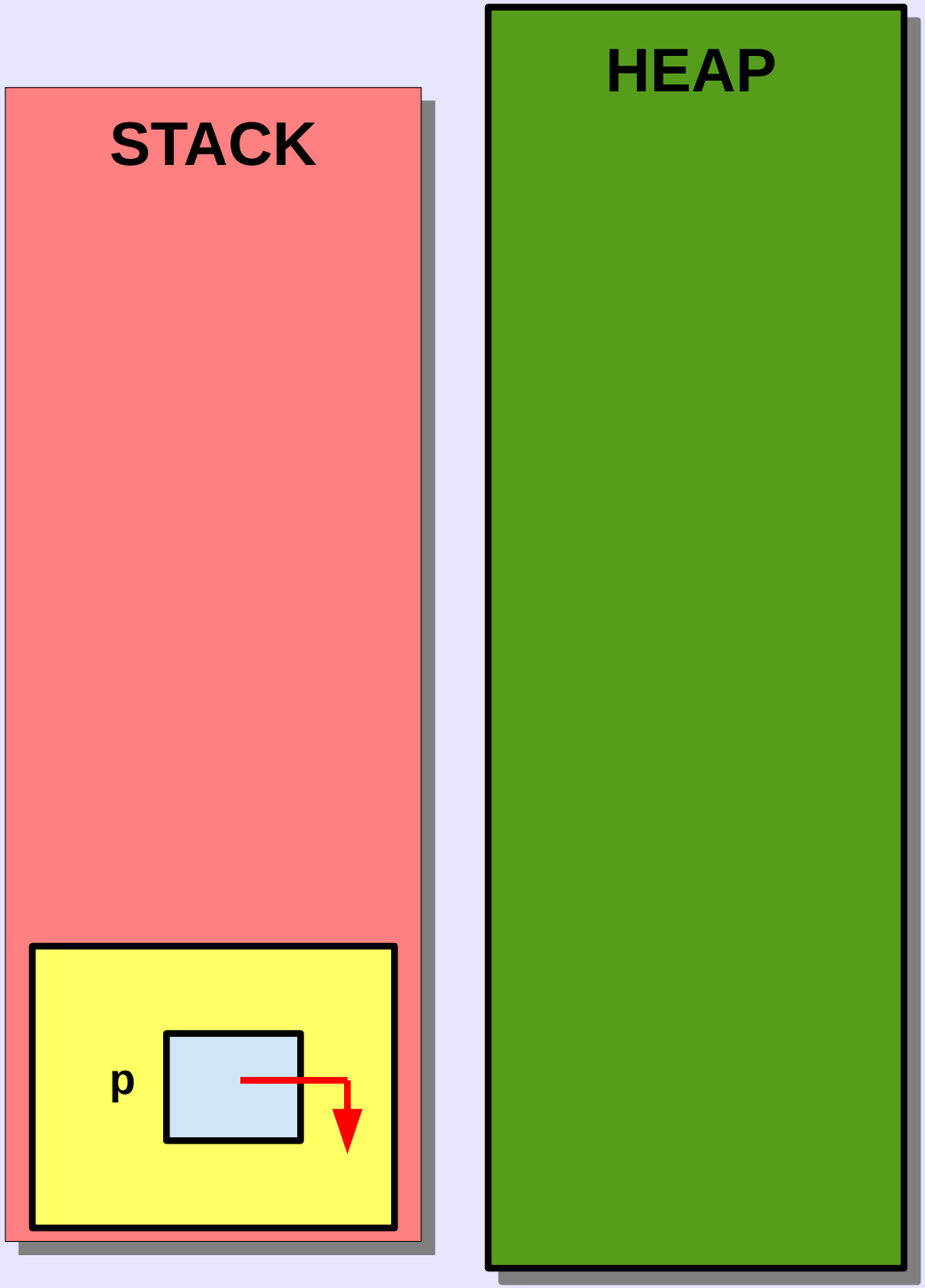
```
typedef int entero;
typedef char caracter;
typedef int *puntero_a_entero;

entero x = 0;
puntero_a_entero p = & x;
```

PARTE V: Memoria Dinámica

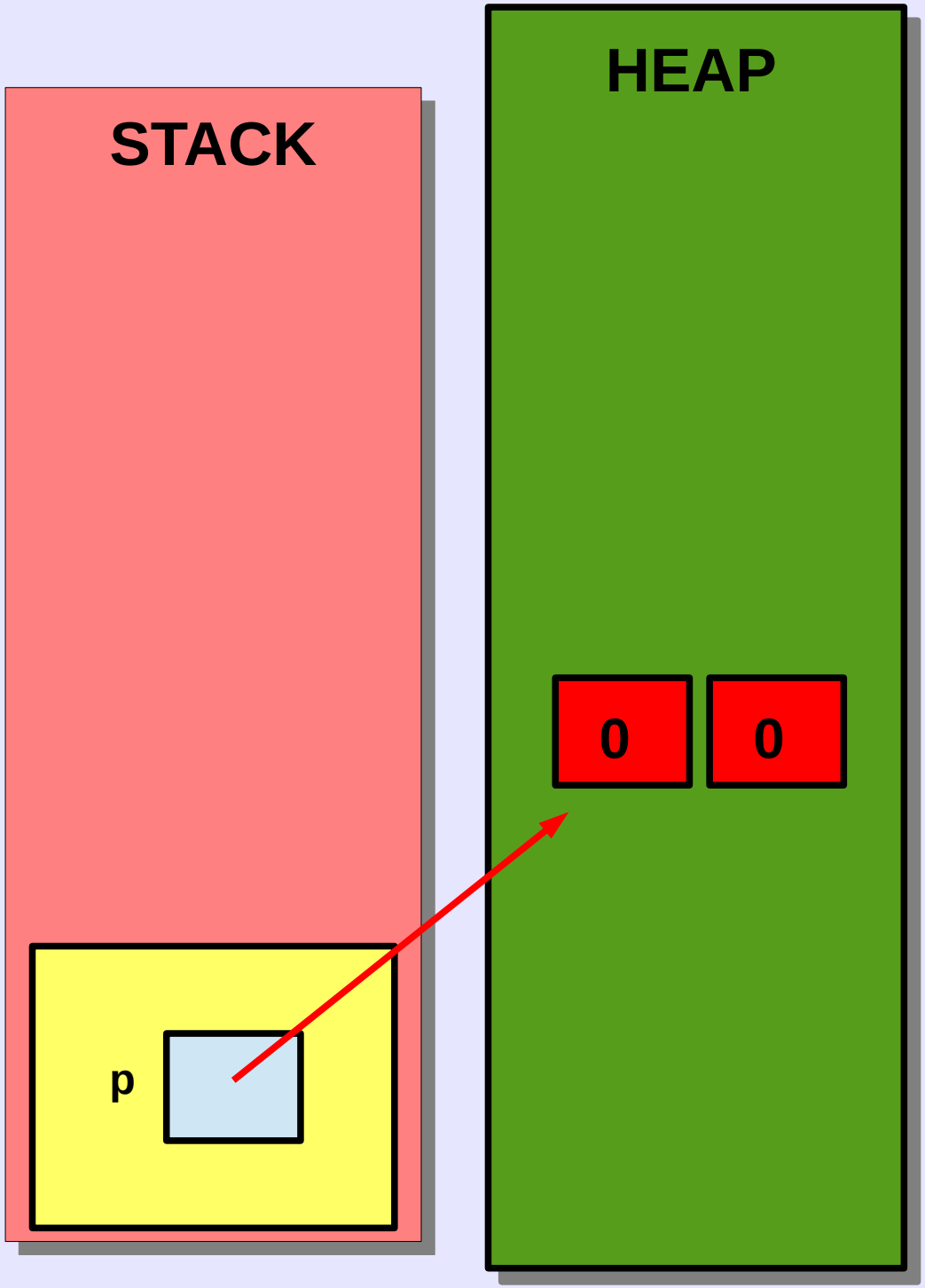
```
void g() {
    int x = 2;
    int y = 1;
    return;
}

void f() {
   int x = 1;
   int z = 0;
   g();
   return;
}

int main() {
   int x = 0;
   int i  = 0;
   f();
   return 0;
}
```



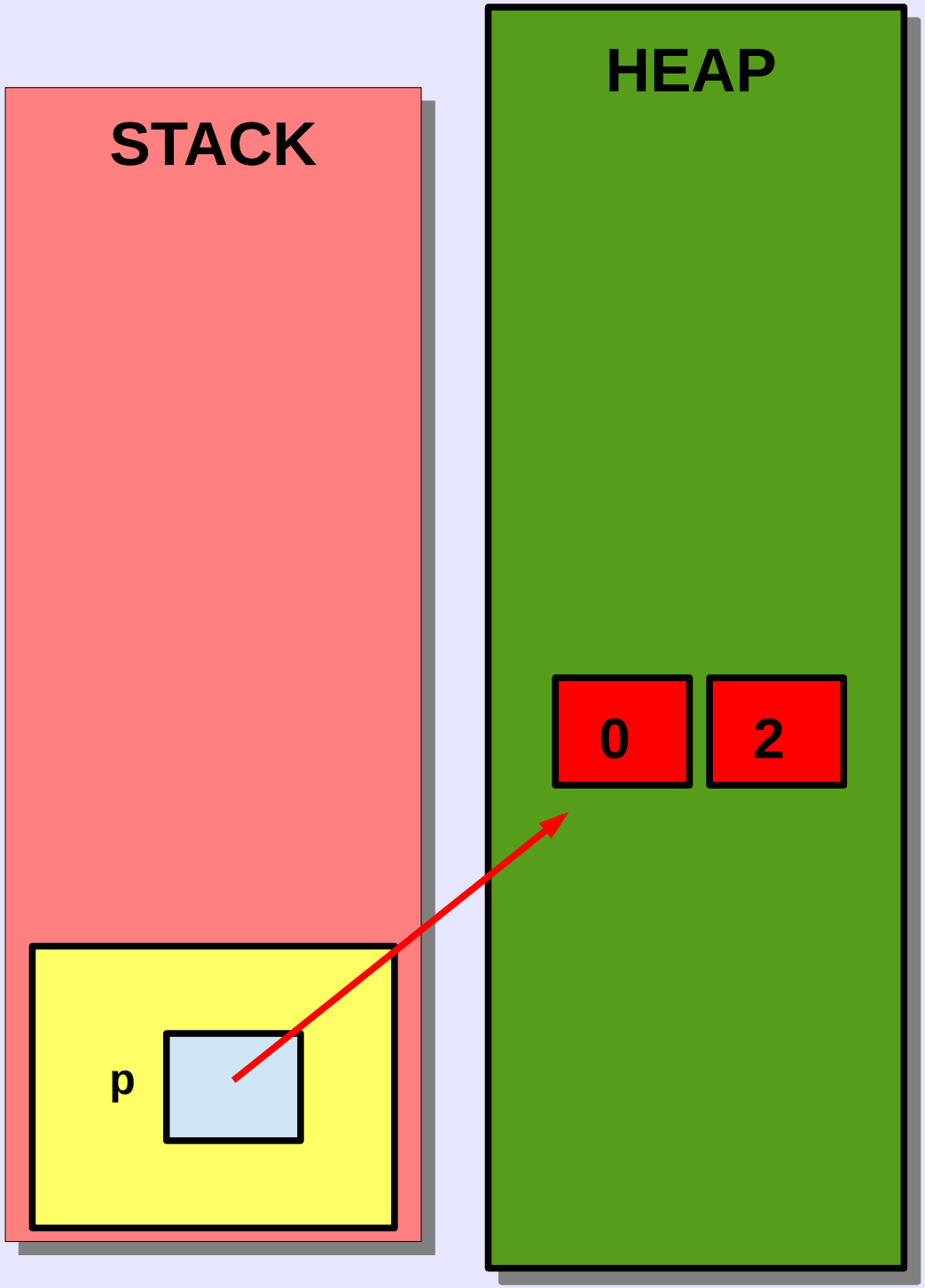
```
int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   p[1] = 2;
   free(p);
   p = NULL;
   return (0);
}
```



```
int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   p[1] = 2;
   free(p);
   p = NULL;
   return (0);
}
```

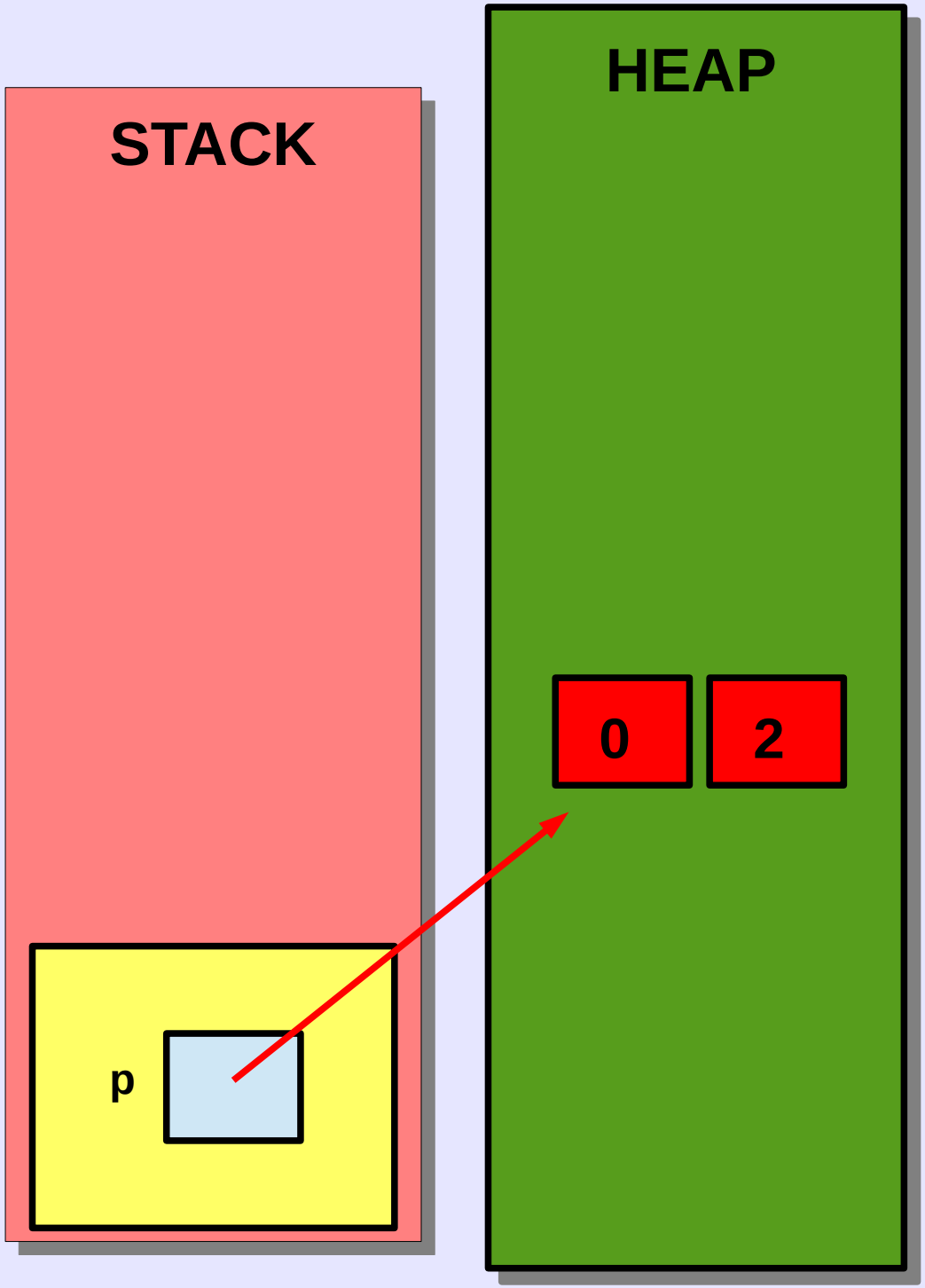


```
int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   p[1] = 2;
   free(p);
   p = NULL;
   return (0);
}
```

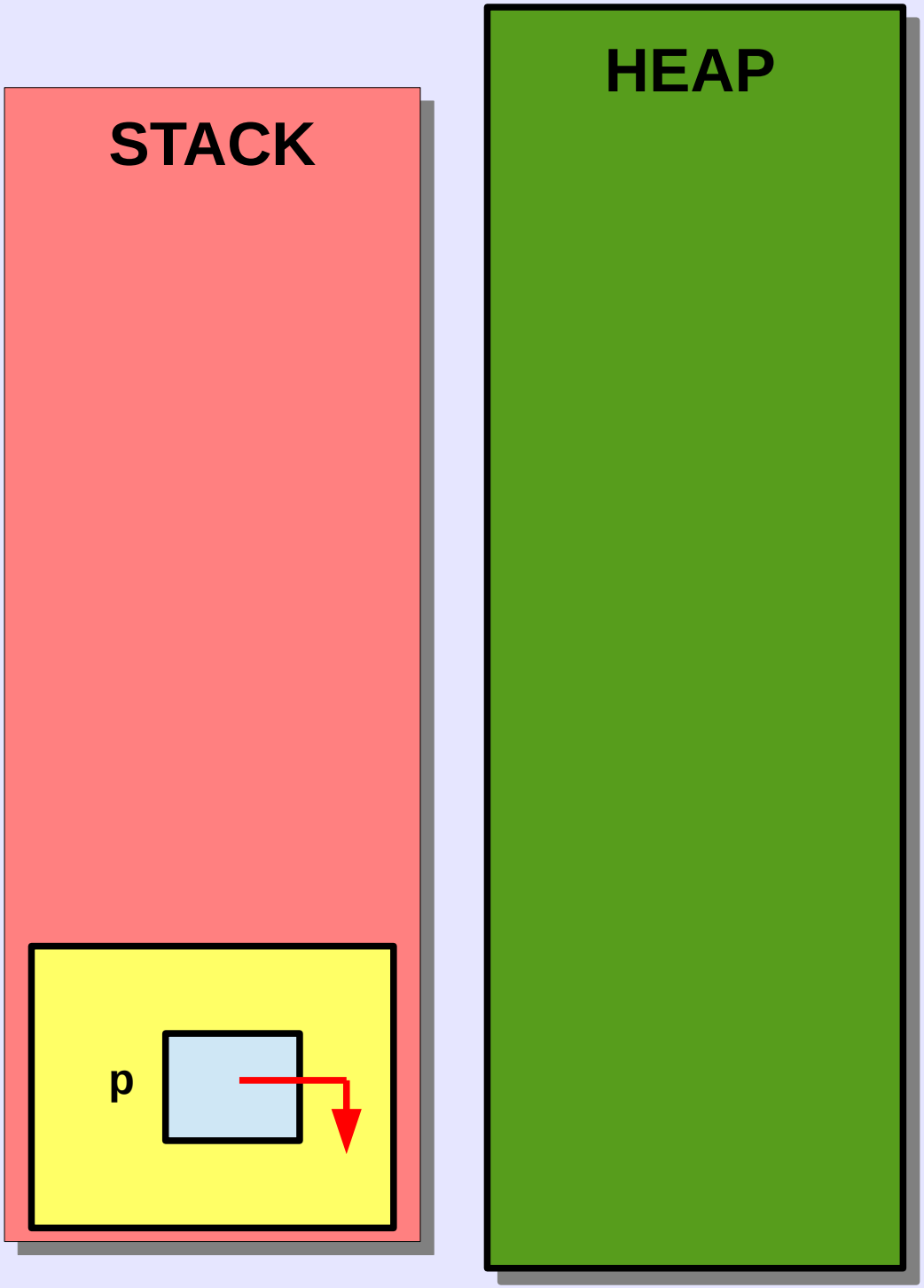


```
int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   p[1] = 2;
   free(p);
   p = NULL;
   return (0);
}
```

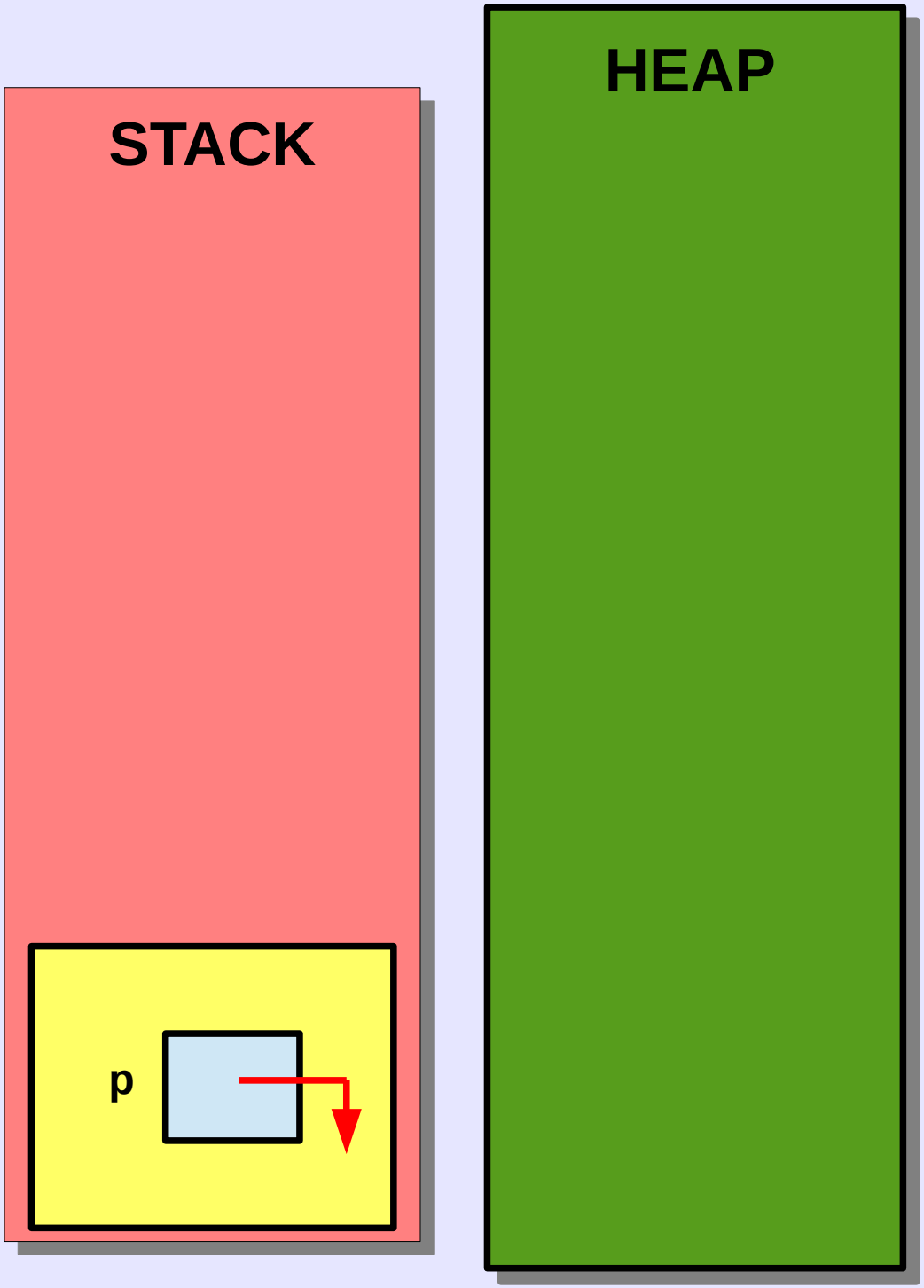
DANGLING POINTER



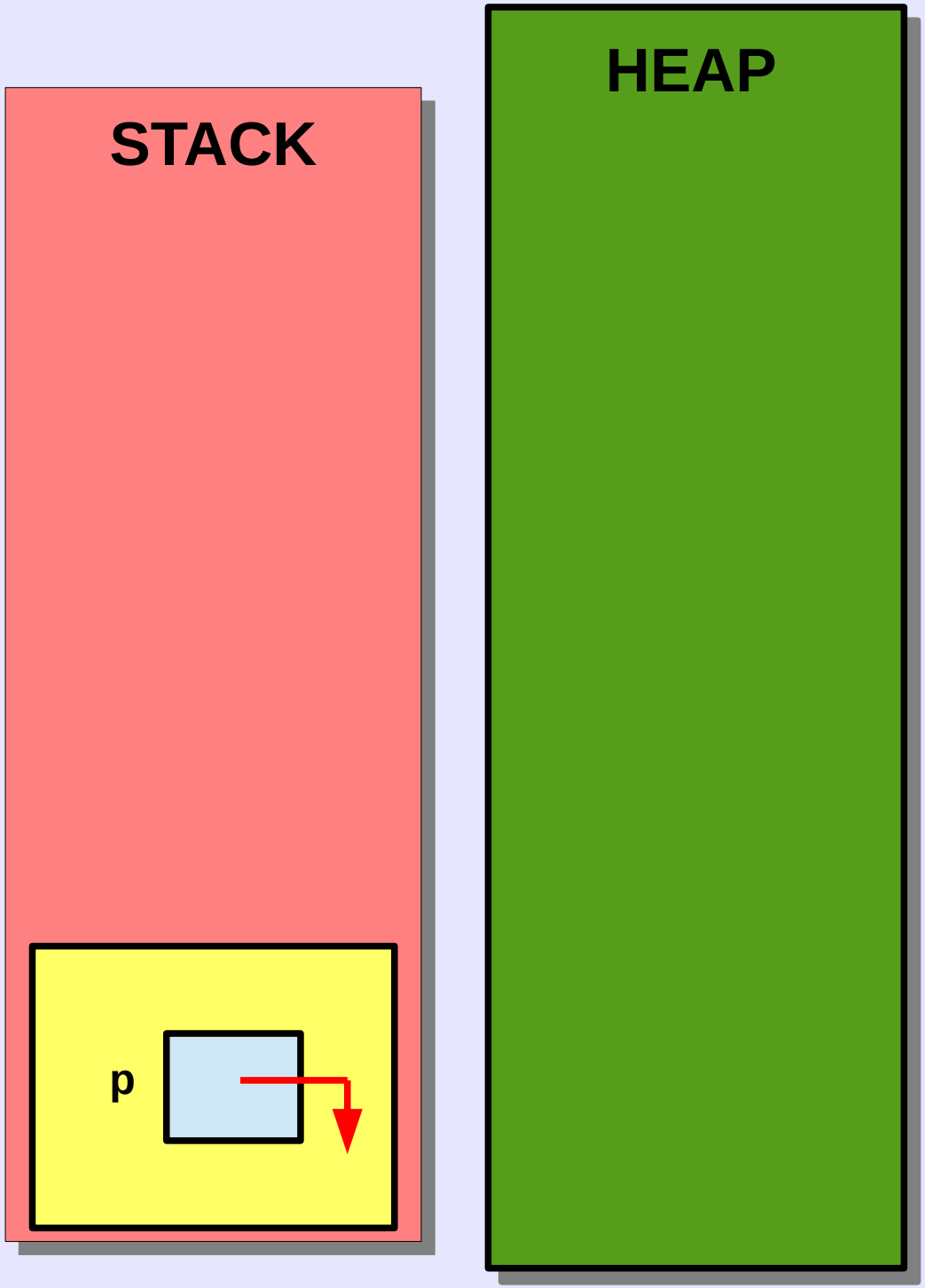
```
int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   p[1] = 2;
   free(p);
   p = NULL;
   return (0);
}
```



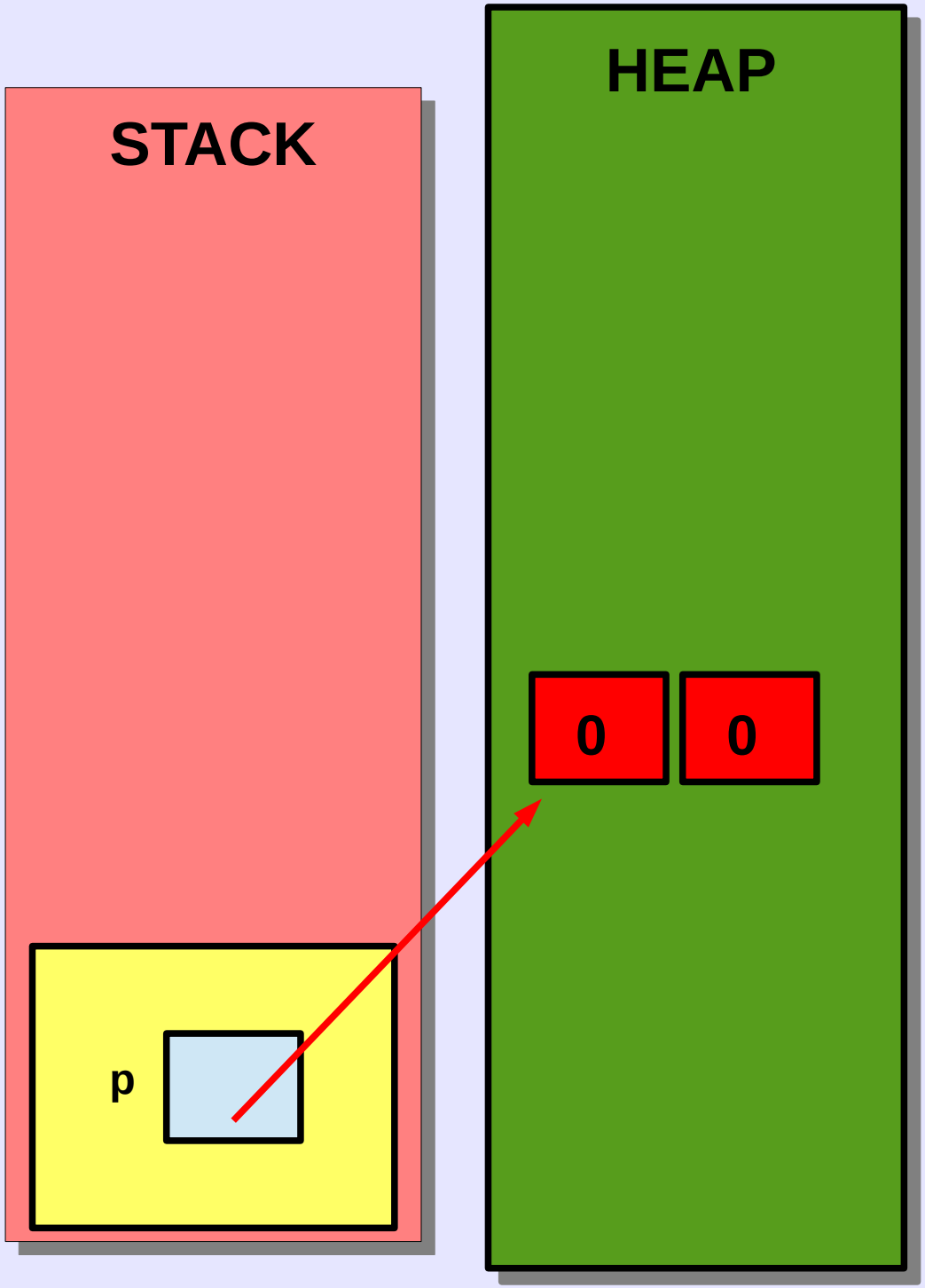
```
int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   p[1] = 2;
   free(p);
   p = NULL;
   return (0);
}
```



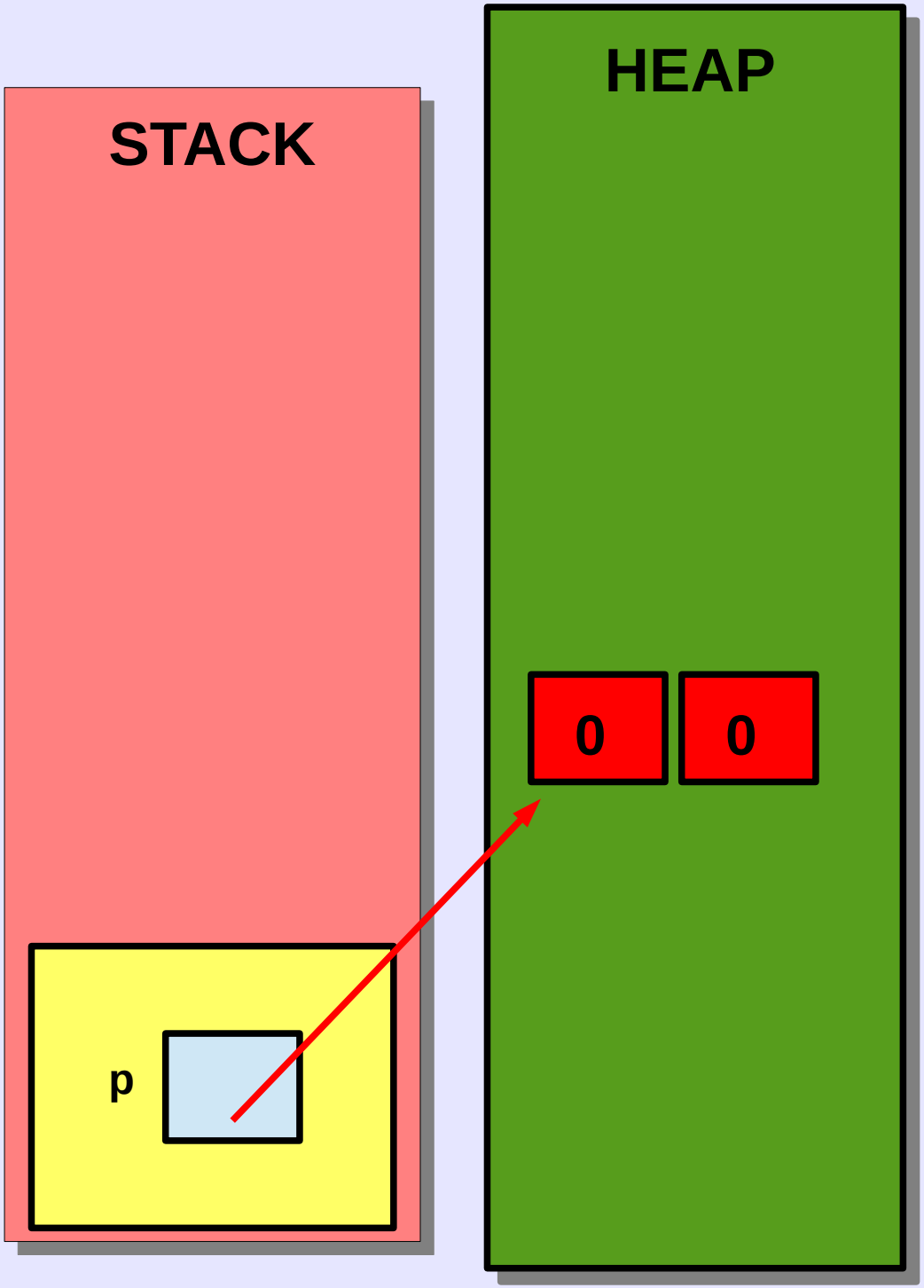
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



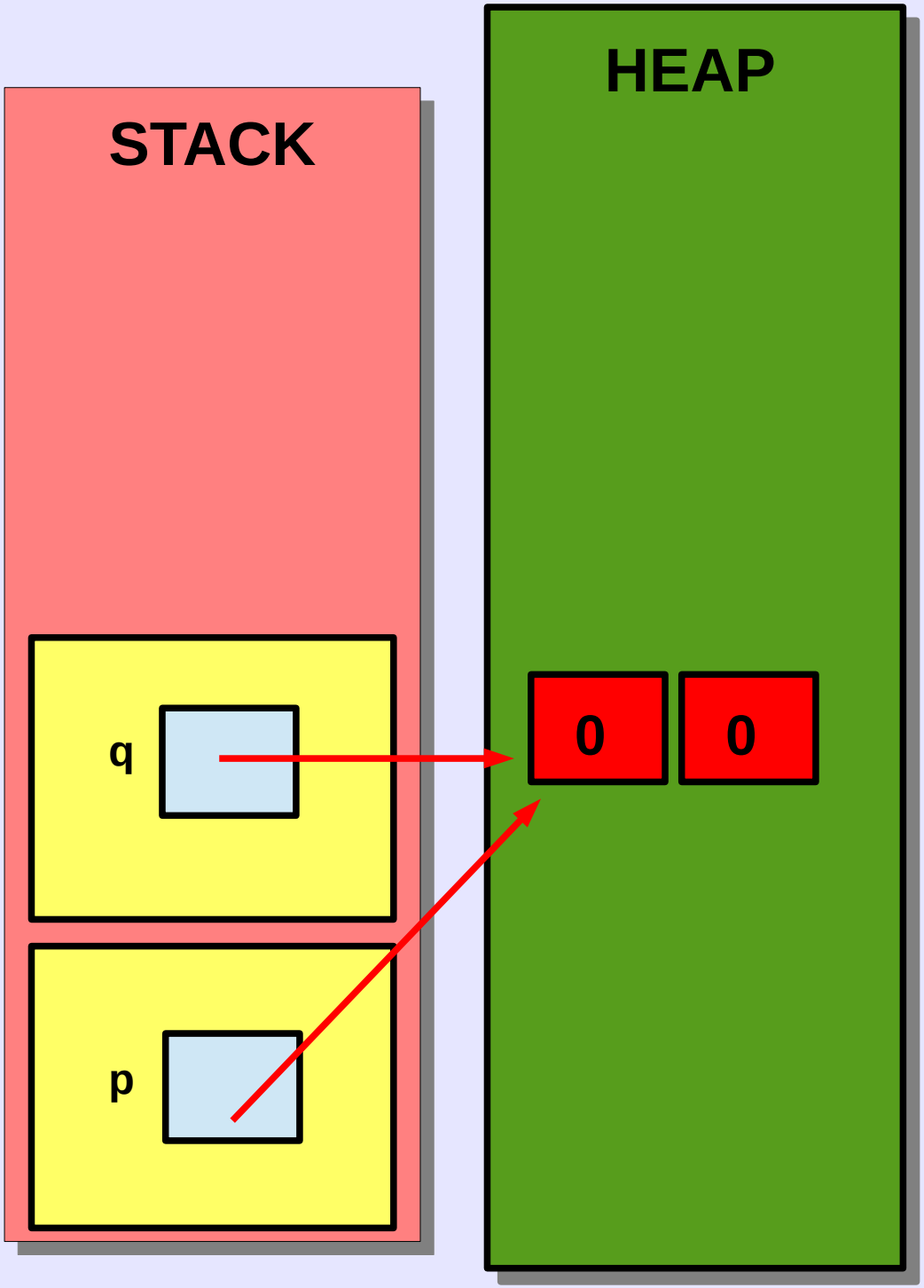
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



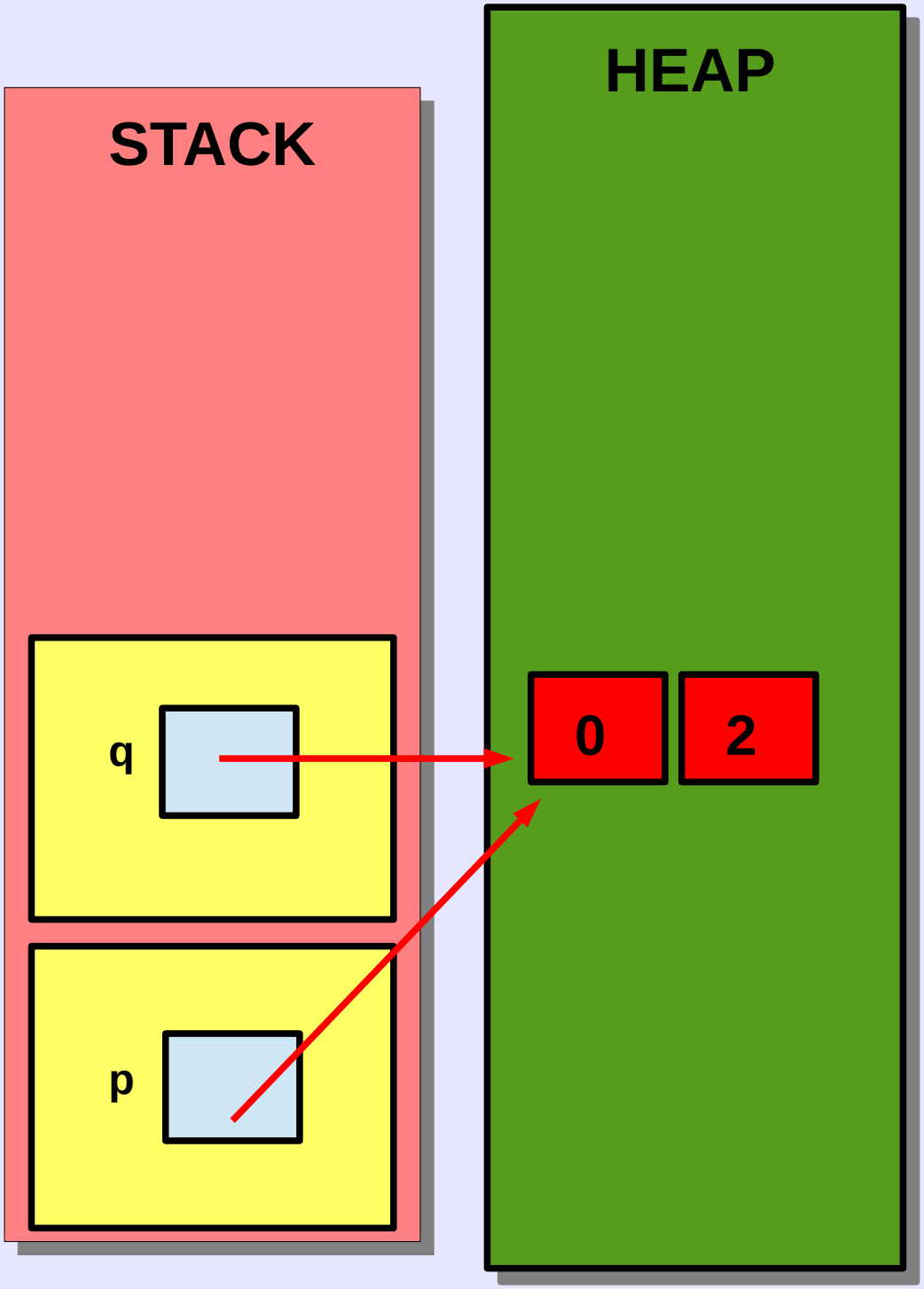
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



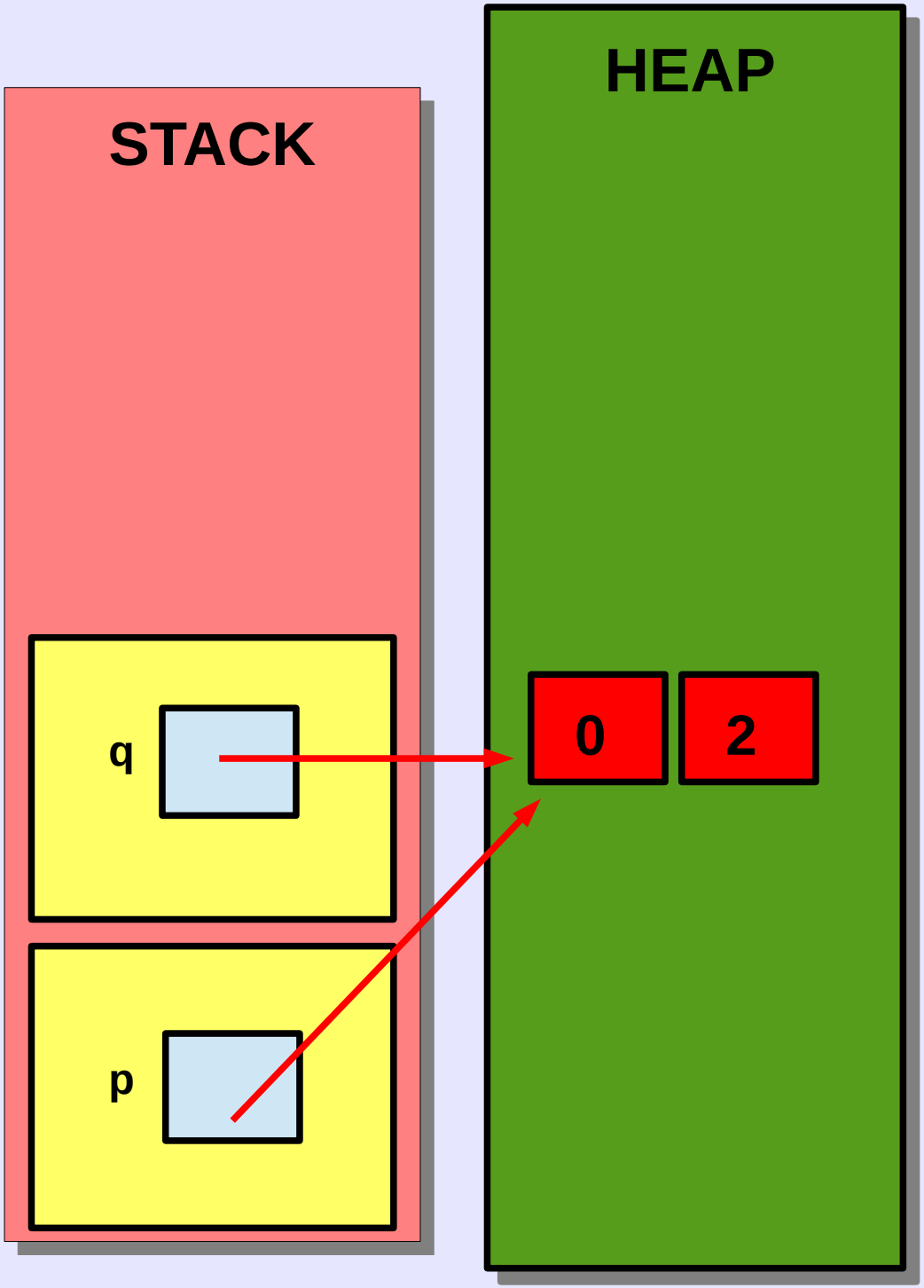
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



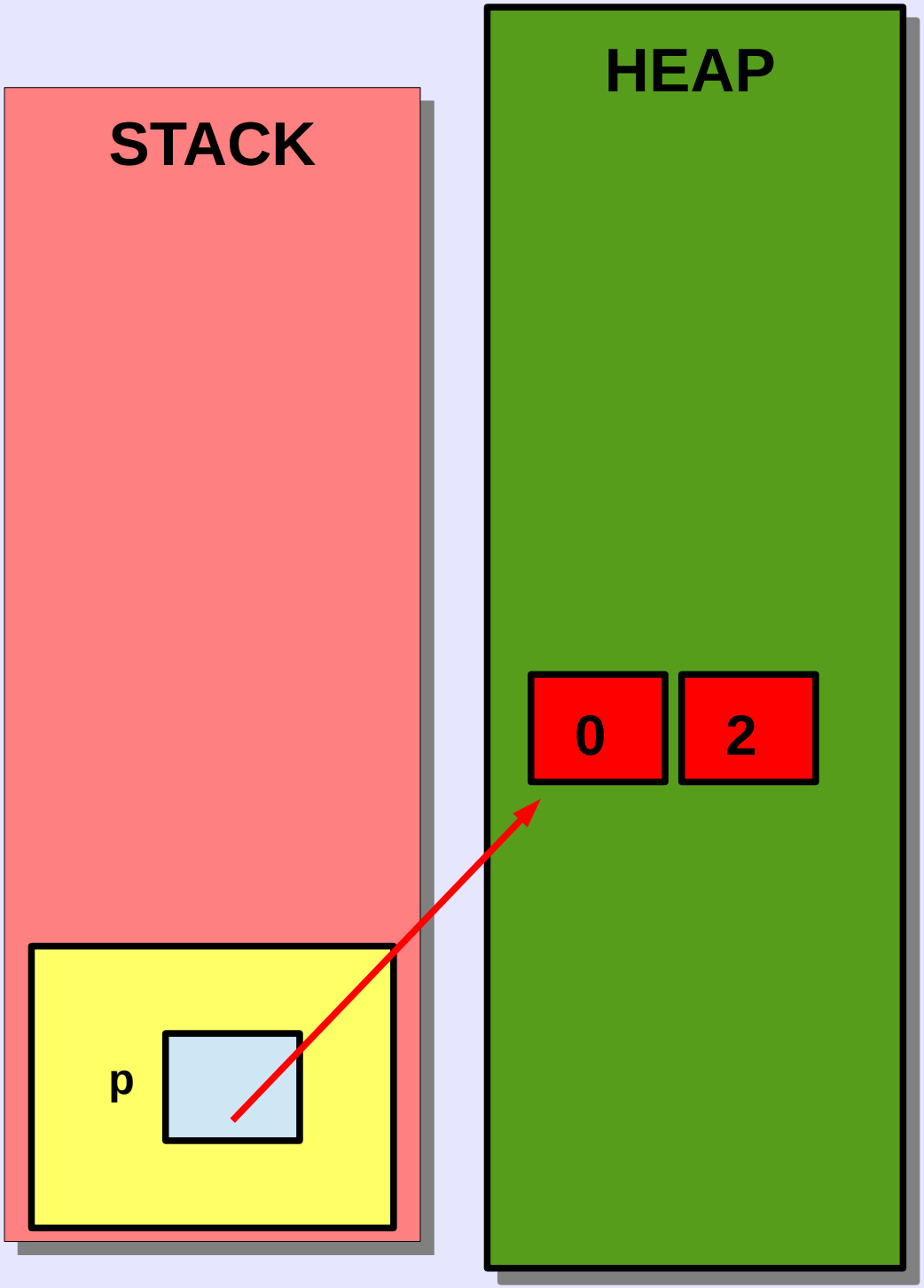
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



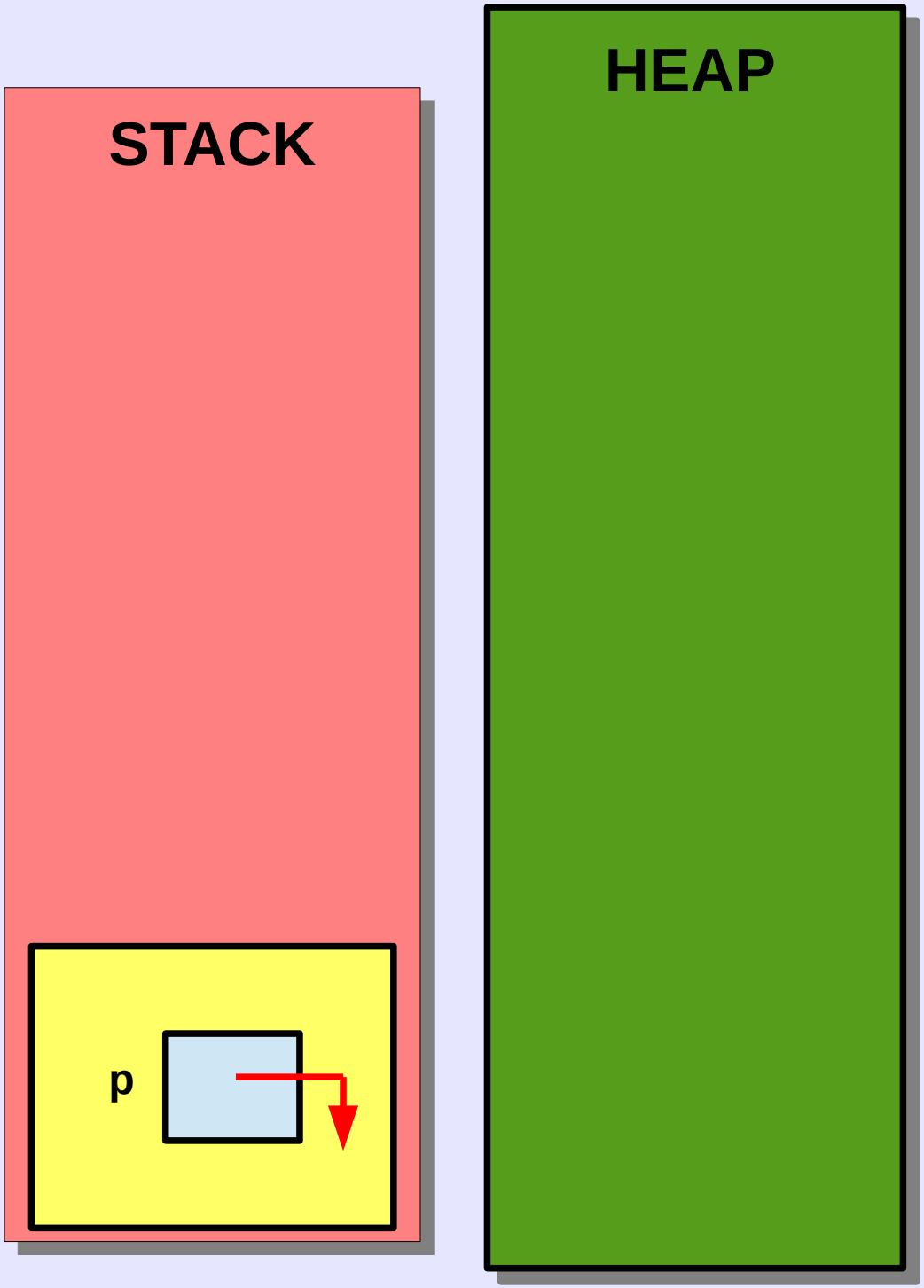
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



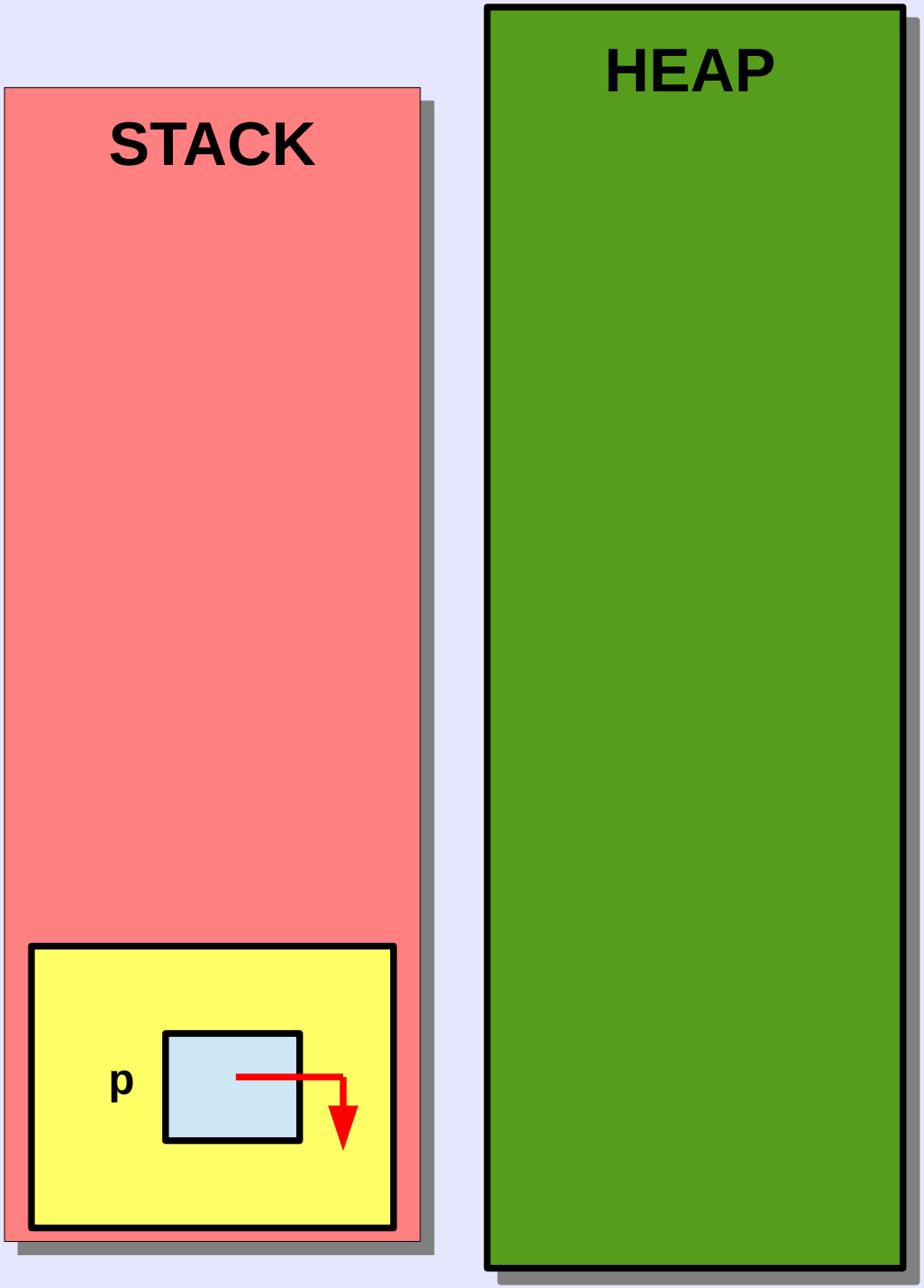
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



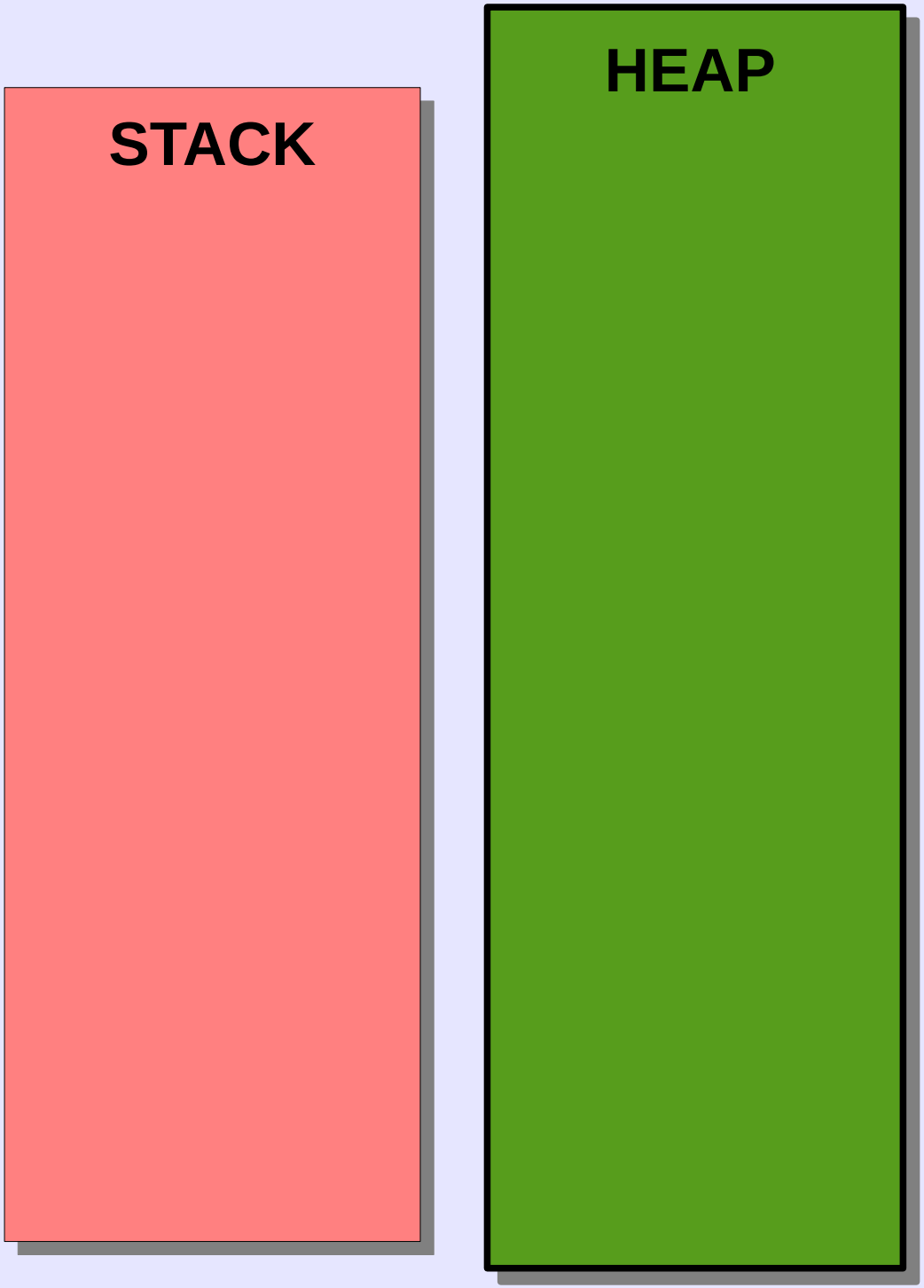
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



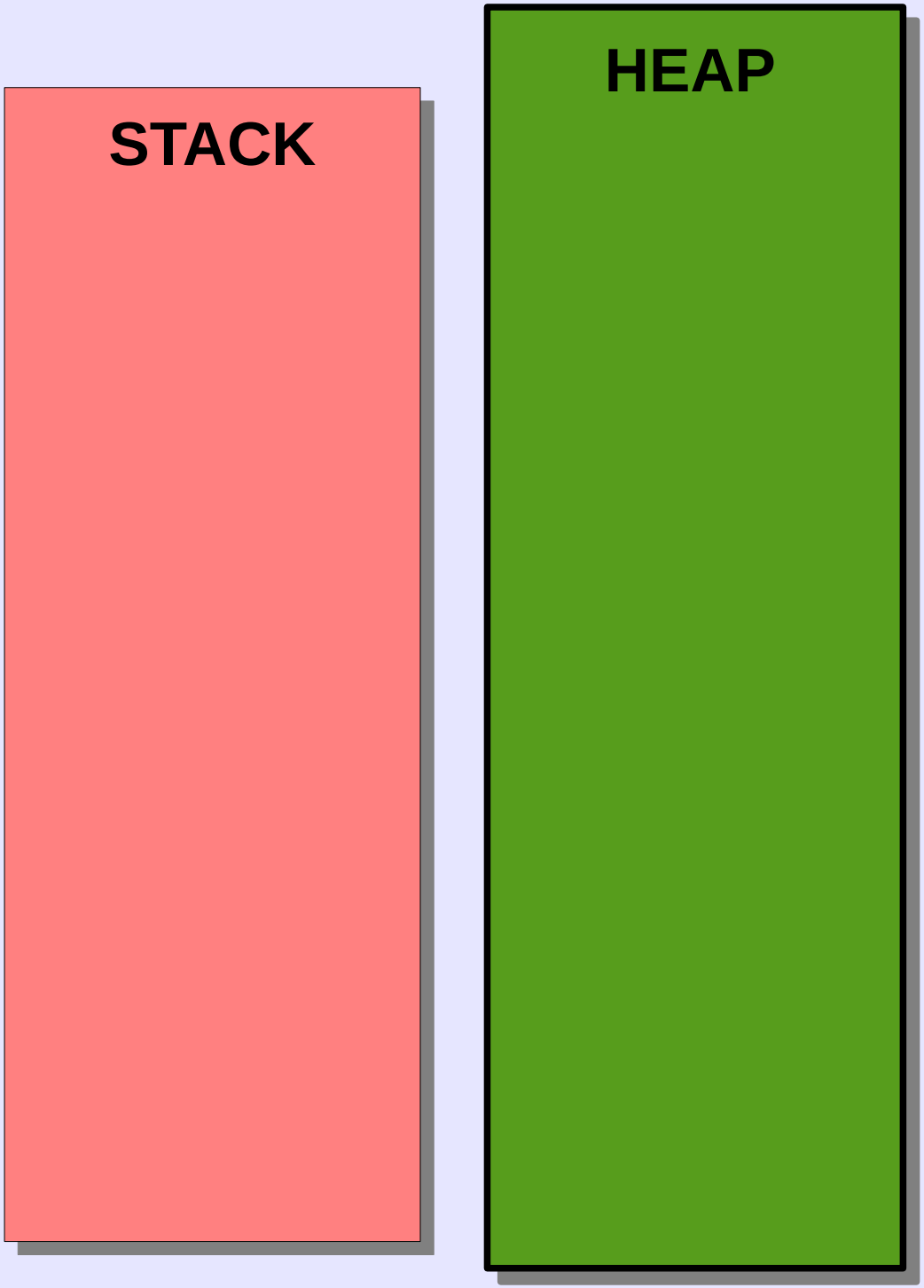
```
void modificar (int *q) {
    q[1] = 2;
    return;
}

int main() {
   int *p = NULL;
   p = calloc(2, sizeof(int));
   assert(p != NULL);
   modificar(p);
   free(p);
   p = NULL;
   return (0);
}
```



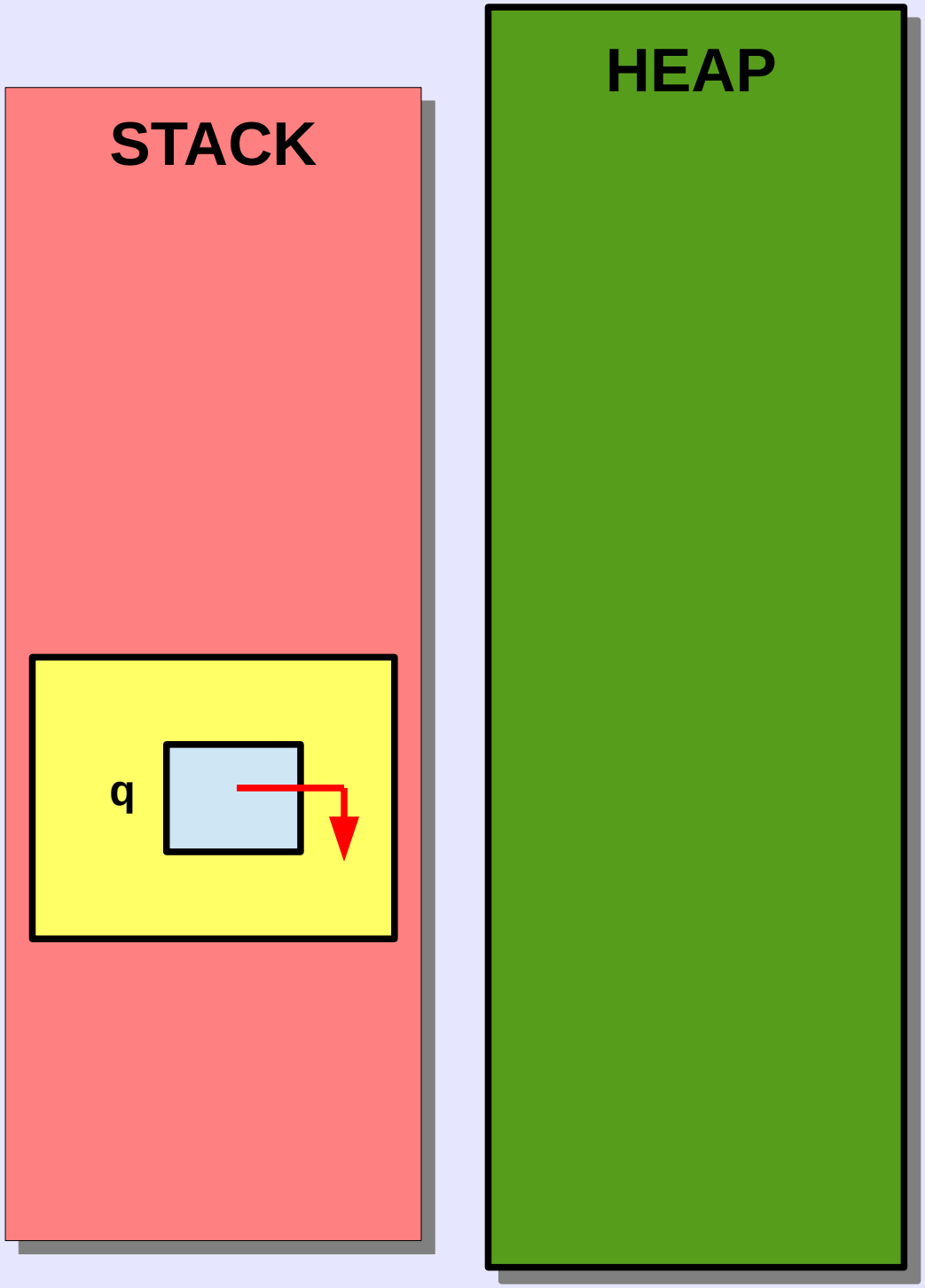
```
void amnesia() {
    int *q = NULL;
    q = calloc(1, sizeof(int));
    return;
}

int main() {
  amnesia();
  for (int i = 0; i<5; i++) {
     amnesia();
  }
  return (0);
}
```



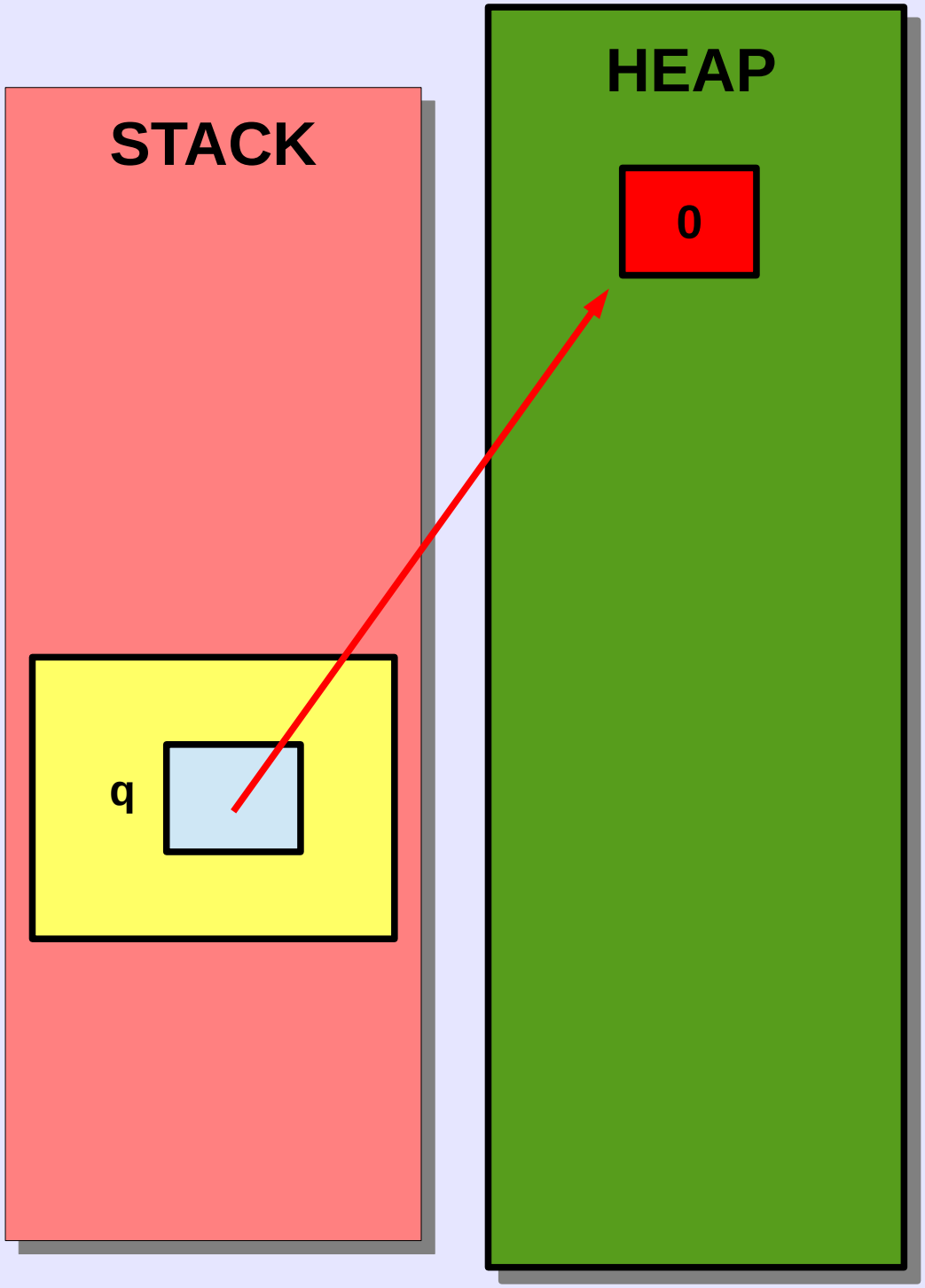
```
void amnesia() {
    int *q = NULL;
    q = calloc(1, sizeof(int));
    return;
}

int main() {
  amnesia();
  for (int i = 0; i<5; i++) {
     amnesia();
  }
  return (0);
}
```



```
void amnesia() {
    int *q = NULL;
    q = calloc(1, sizeof(int));
    return;
}

int main() {
  amnesia();
  for (int i = 0; i<5; i++) {
     amnesia();
  }
  return (0);
}
```

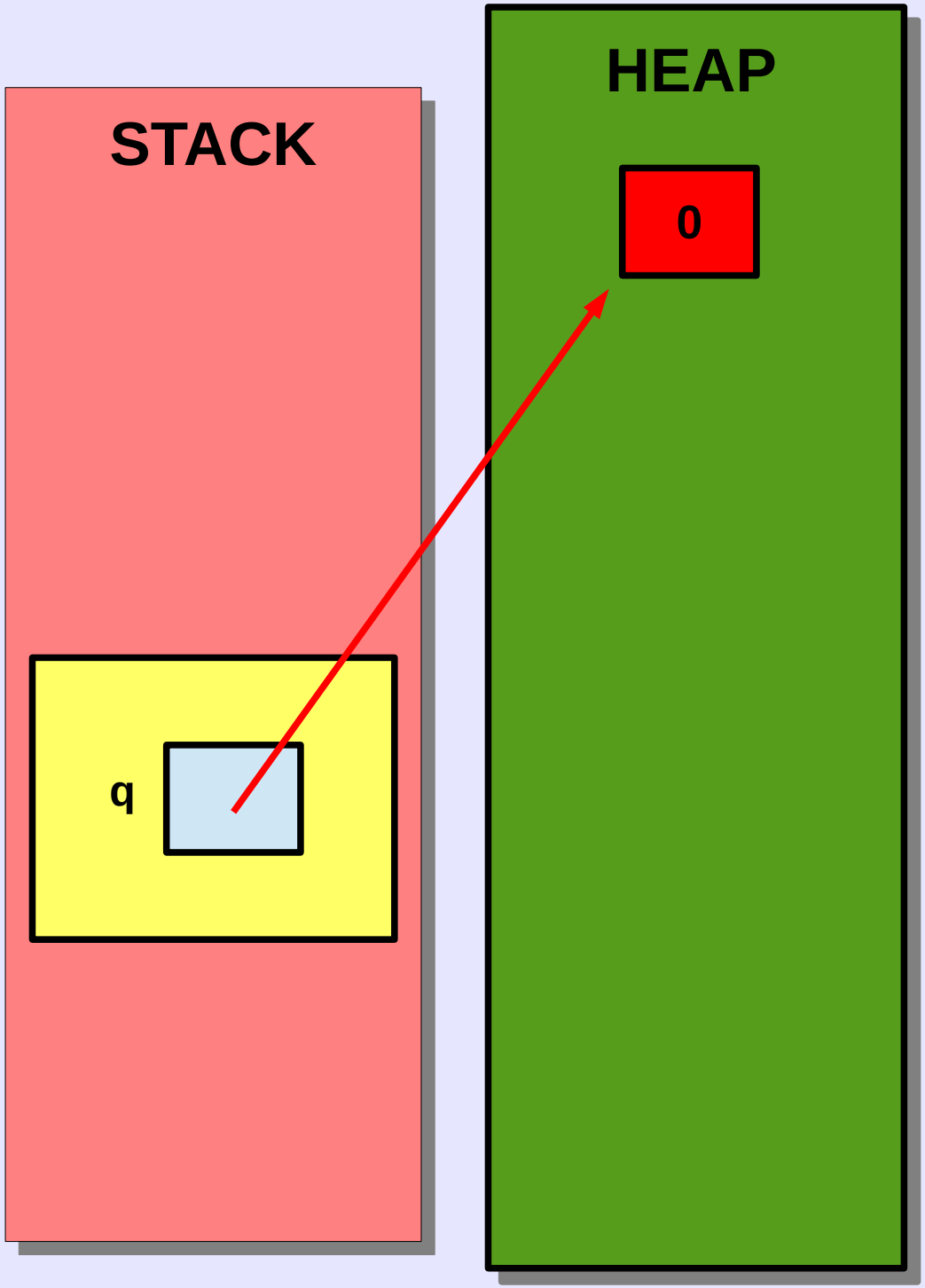


```
void amnesia() {
    int *q = NULL;
    q = calloc(1, sizeof(int));
    return;
}

int main() {
  amnesia();
  for (int i = 0; i<5; i++) {
     amnesia();
  }
  return (0);
}
```

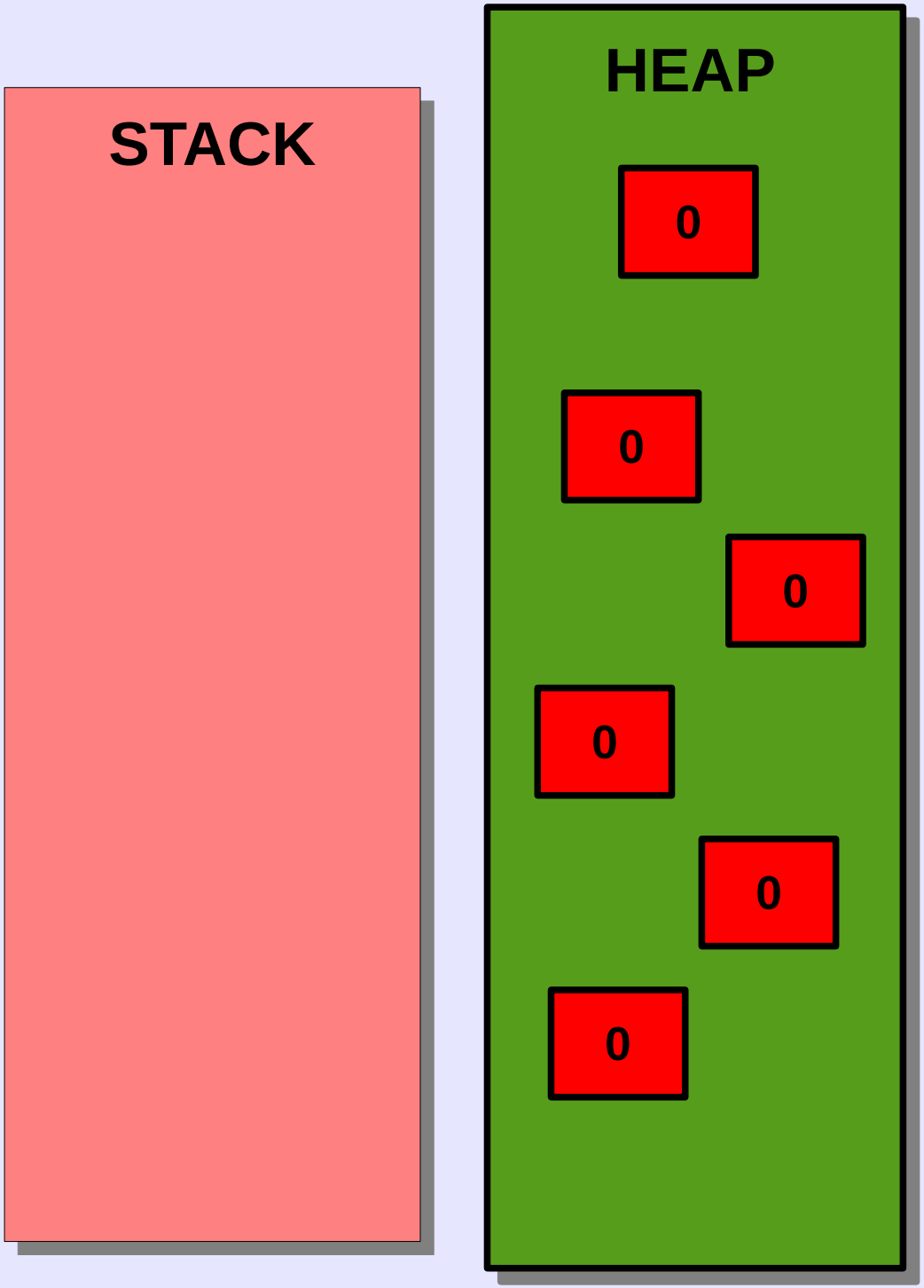


```
void amnesia() {
    int *q = NULL;
    q = calloc(1, sizeof(int));
    return;
}

int main() {
  amnesia();
  for (int i = 0; i<5; i++) {
     amnesia();
  }
  return (0);
}
```

MEMORY LEAK

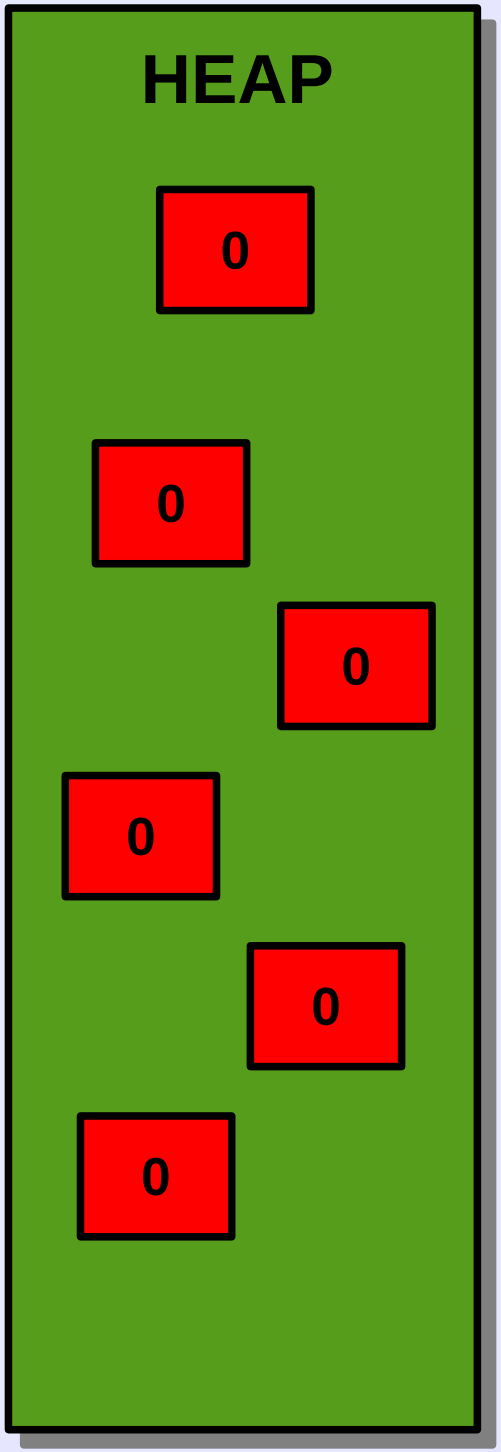


```
void amnesia() {
    int *q = NULL;
    q = calloc(1, sizeof(int));
    return;
}

int main() {
  amnesia();
  for (int i = 0; i<5; i++) {
      amnesia();
  }
  return (0);
}
```

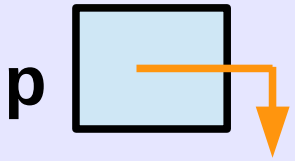


```
$>  gcc -g -o ejemplo ejemplo.c
$>  valgrind --show-reachable=yes --leak-check=full ./ejemplo
```

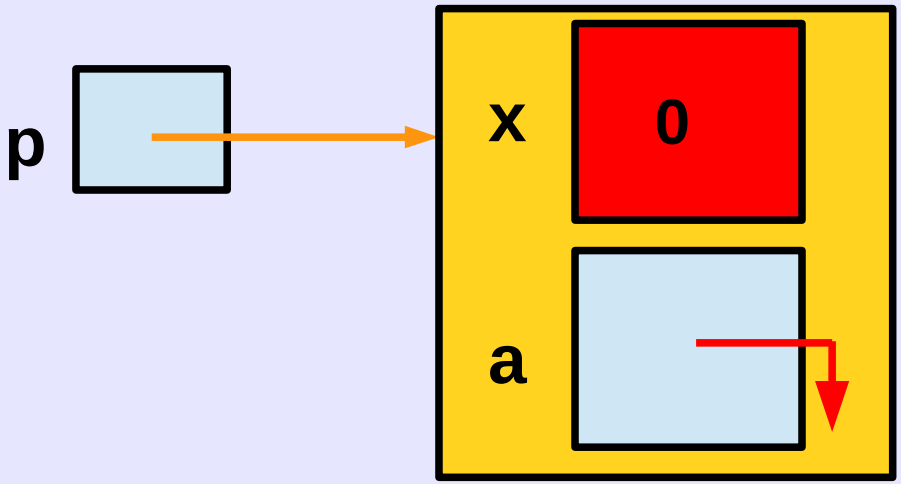
```
==8066== HEAP SUMMARY:
==8066==     in use at exit: 24 bytes in 6 blocks
==8066==   total heap usage: 6 allocs, 0 frees, 24 bytes allocated
==8066==
==8066== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==8066==    at 0x4C272B8: calloc (vg_replace_malloc.c:566)
==8066==    by 0x40052A: amnesia (ejemplo.c:6)
==8066==    by 0x400543: main (ejemplo.c:12)
==8066==
==8066== 20 bytes in 5 blocks are definitely lost in loss record 2 of 2
==8066==    at 0x4C272B8: calloc (vg_replace_malloc.c:566)
==8066==    by 0x40052A: amnesia (ejemplo.c:6)
==8066==    by 0x400556: main (ejemplo.c:14)
==8066==
==8066== LEAK SUMMARY:
==8066==    definitely lost: 24 bytes in 6 blocks
==8066==    indirectly lost: 0 bytes in 0 blocks
==8066==    possibly lost: 0 bytes in 0 blocks
==8066==    still reachable: 0 bytes in 0 blocks
==8066==    suppressed: 0 bytes in 0 blocks
```



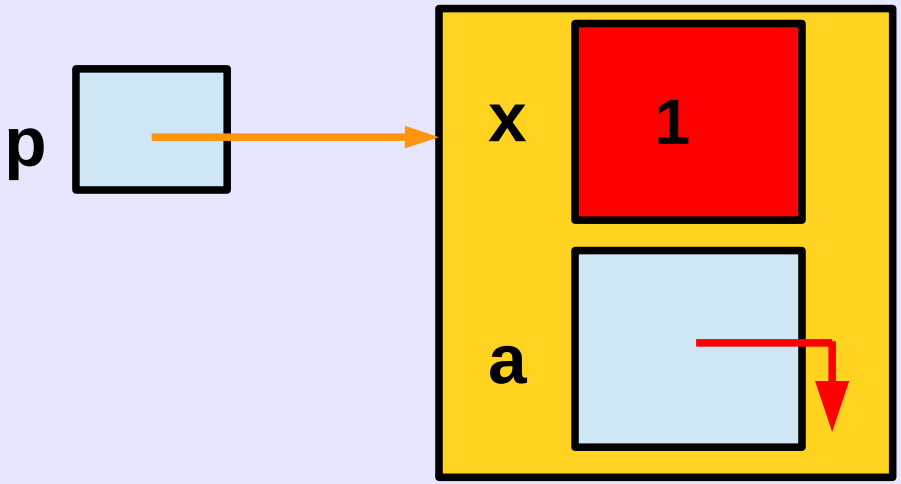
```
struct _data {
    int x;
    int *a;
};

struct _data *p = NULL;
p = calloc(1, sizeof(struct _data));
p->x = 1;
p->a = calloc(3, sizeof(int));

....

free(p->a);
p->a = NULL;
free(p);
p = NULL;
```



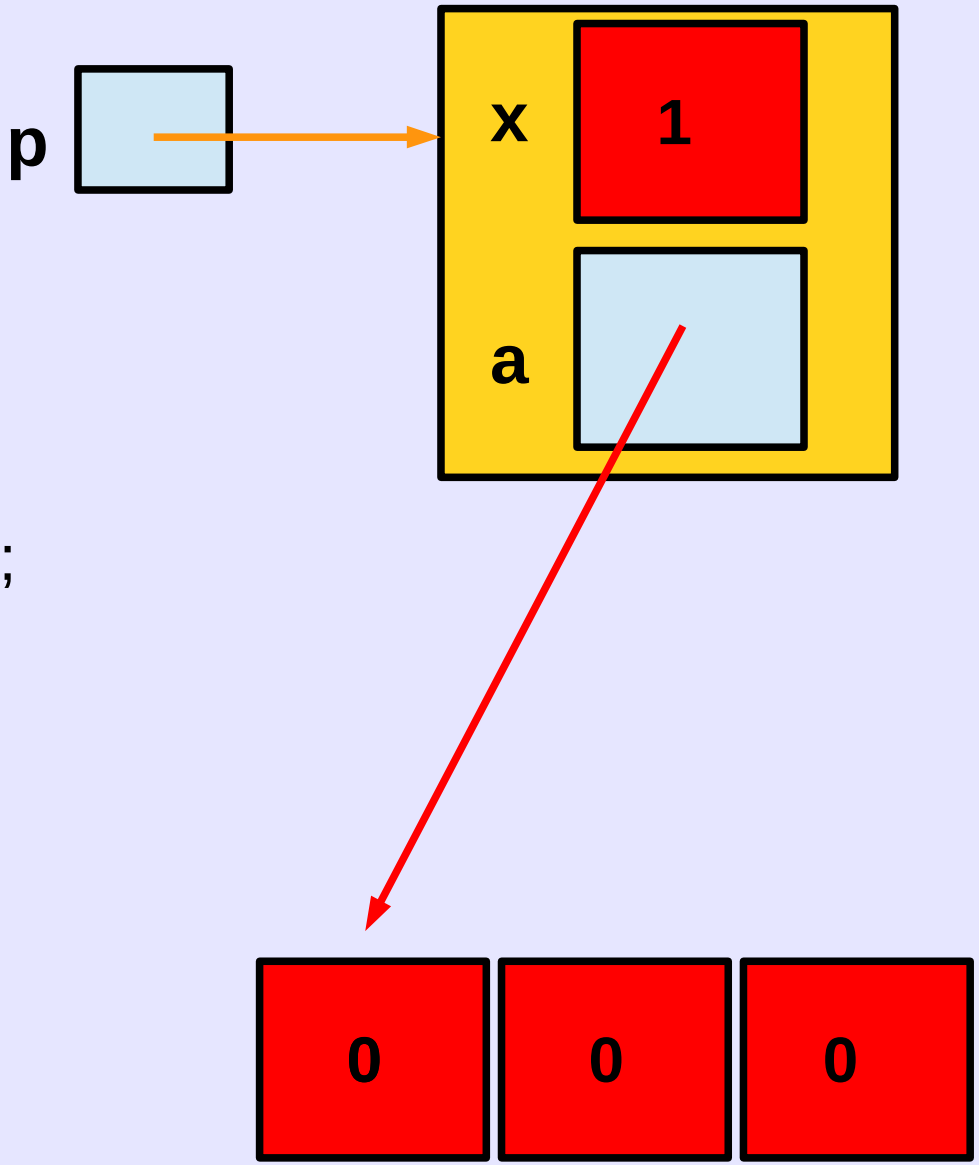
```
struct _data {
    int x;
    int *a;
};

struct _data *p = NULL;
p = calloc(1, sizeof(struct _data));
p->x = 1;
p->a = calloc(3, sizeof(int));

....

free(p->a);
p->a = NULL;
free(p);
p = NULL;
```



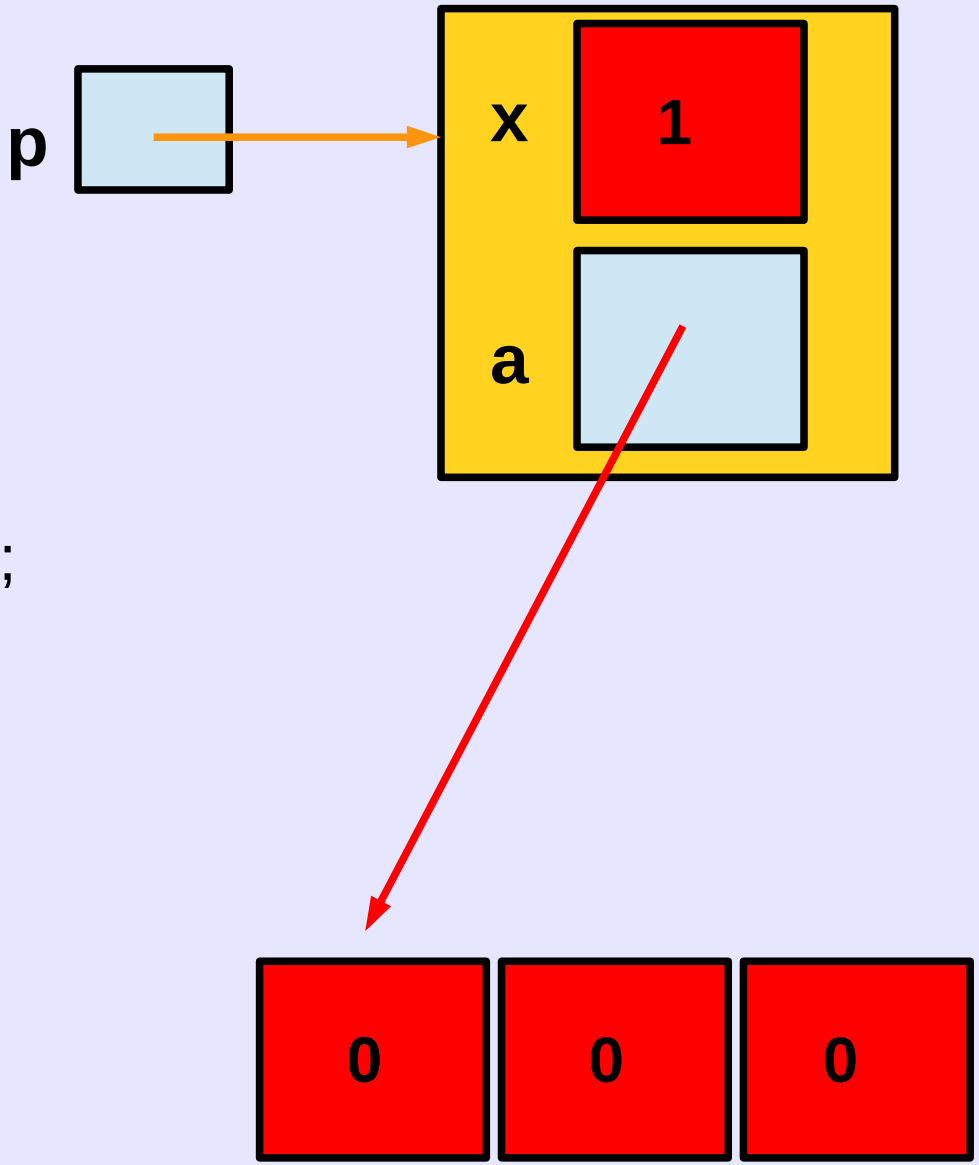
```
struct _data {
    int x;
    int *a;
};

struct _data *p = NULL;
p = calloc(1, sizeof(struct _data));
p->x = 1;
p->a = calloc(3, sizeof(int));

....

free(p->a);
p->a = NULL;
free(p);
p = NULL;
```



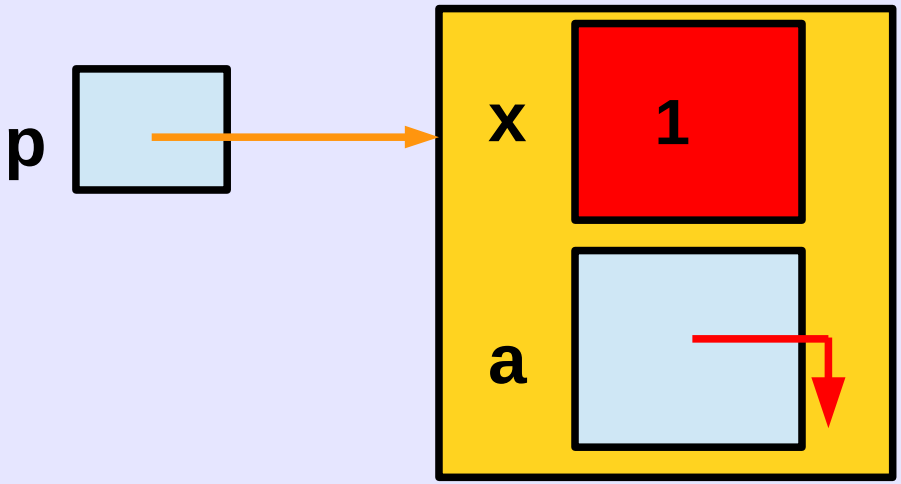
```
struct _data {
    int x;
    int *a;
};

struct _data *p = NULL;
p = calloc(1, sizeof(struct _data));
p->x = 1;
p->a = calloc(3, sizeof(int));

....

free(p->a);
p->a = NULL;
free(p);
p = NULL;
```



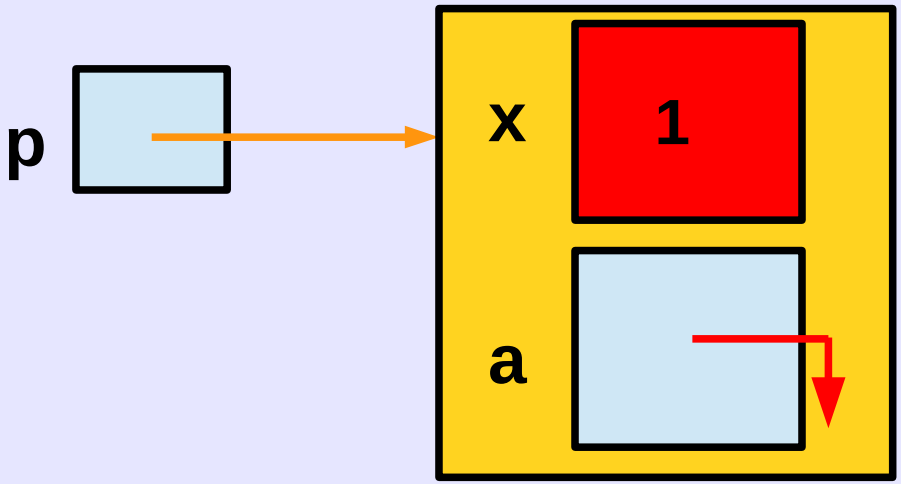
```
struct _data {
    int x;
    int *a;
};

struct _data *p = NULL;
p = calloc(1, sizeof(struct _data));
p->x = 1;
p->a = calloc(3, sizeof(int));

....

free(p->a);
p->a = NULL;
free(p);
p = NULL;
```



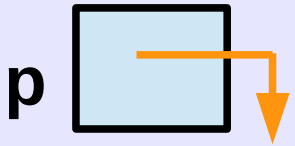
```
struct _data {
    int x;
    int *a;
};

struct _data *p = NULL;
p = calloc(1, sizeof(struct _data));
p->x = 1;
p->a = calloc(3, sizeof(int));

....

free(p->a);
p->a = NULL;
free(p);
p = NULL;
```



```
struct _data {
    int x;
    int *a;
};

struct _data *p = NULL;
p = calloc(1, sizeof(struct _data));
p->x = 1;
p->a = calloc(3, sizeof(int));

....

free(p->a);
p->a = NULL;
free(p);
p = NULL;
```



```
struct _data {
    int x;
    int *a;
};

struct _data *p = NULL;
p = calloc(1, sizeof(struct _data));
p->x = 1;
p->a = calloc(3, sizeof(int));

....

free(p->a);
p->a = NULL;
free(p);
p = NULL;
```



Checkpoint

Memoria estática:

Se ubica en la pila. Liberada por el sistema operativo.
Tiene tamaño limitado.

TIPS:
Consultar el tamaño máximo permitido de la pila usando “ulimit -s”
En GDB se puede analizar la pila actual usando el comando “frame”.

Memoria dinámica:

Se ubica en el heap. En C es liberada por el programador. En otros lenguajes como Java es liberada por el garbage collector (ver wikipedia).
No tiene límite (depende del SO).

CALLOC (<cantidad de bloques>, <tamaño de cada bloque>)
Crea una secuencia contigua de bloques de memoria y devuelve un puntero al bloque inicial. Inicializa todo el bloque nuevo con 0 (a nivel bit). Si no hay memoria en el sistema devuelve NULL. Incluir librería estándar para tenerla disponible:

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

Checkpoint

Dangling pointer (puntero colgante)

Puntero apuntando a una zona de memoria que ya fue liberada. Intentar dereferenciar un puntero colgante puede resultar en un error de violación de segmento.

free() sólo libera la memoria pero no modifica el puntero.

Memory leak (fuga de memoria)

Memoria del heap que nunca es liberada.

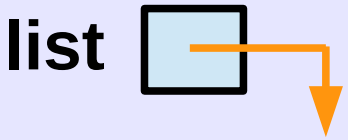
Una causa de memory leak es cuando queda memoria “suelta” en el heap a la cuál no apunta ningún puntero. El programador no tiene forma de liberarla pues se ha perdido el acceso a ella. Los programas con memory leaks se vuelven lentos con el tiempo, ya que el heap se llena de bloques de memoria sin uso.

Utilizar el comando valgrind para chequear fugas de memoria.

Violación de segmento

- Dereferenciar (*p) un puntero NULL o colgante.
- Hacer free(p) donde p es colgante.
- Pasarse de rango en un arreglo, pues se accede a memoria inválida.

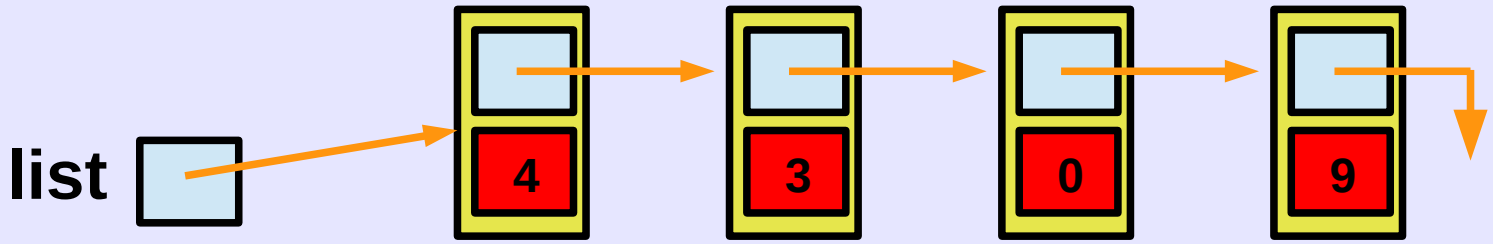
PARTE VI: Introducción a Listas Enlazadas




```
struct _node {
    struct _node *next;
    int elem;
};

typedef struct _node  *list_t;
```

La lista vacía [] es representada con un puntero nulo.

```
list_t list = NULL;
```

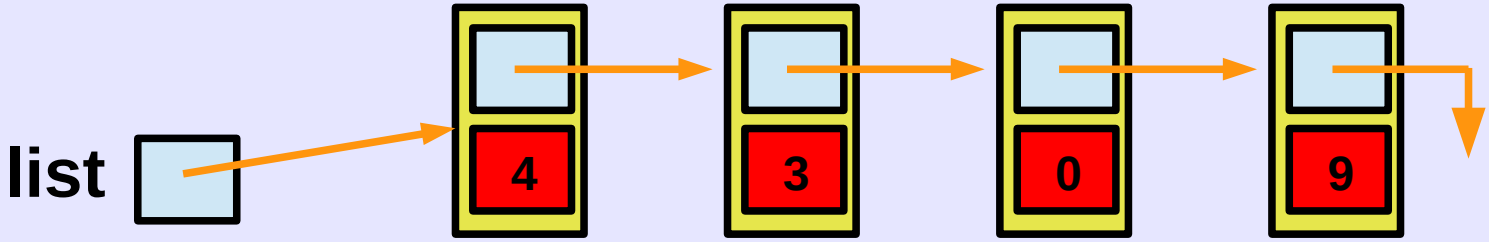



```
struct _node {
    struct _node *next;
    int elem;
};

typedef struct _node  *list_t;
```

La lista no vacía es representada como una “cadena” de nodos.

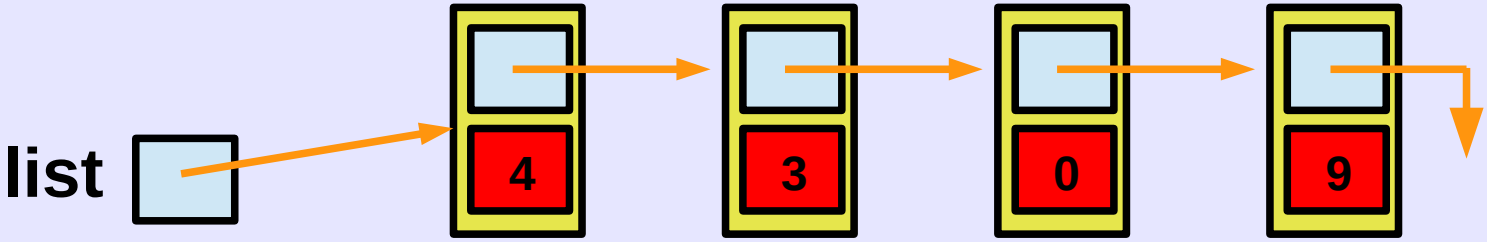
En este ejemplo usaremos listas de enteros.

[4, 3, 0, 9]



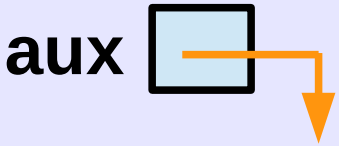
INSERCIÓN AL COMIENZO

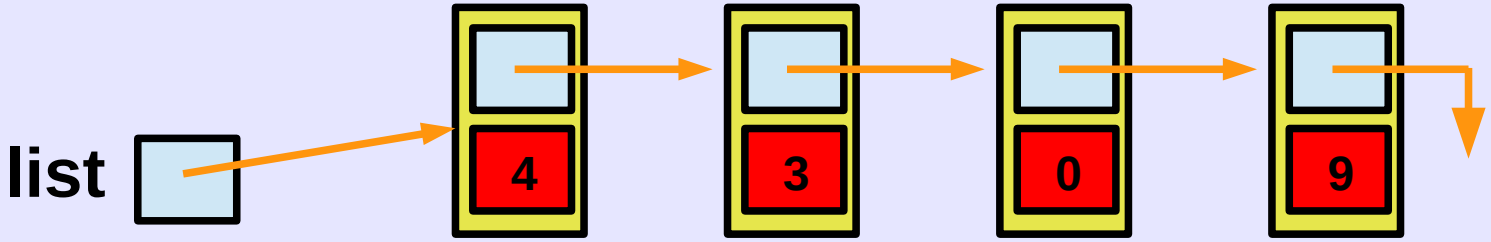
```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

aux->next = list;

list = aux;
```



INSERCIÓN AL COMIENZO

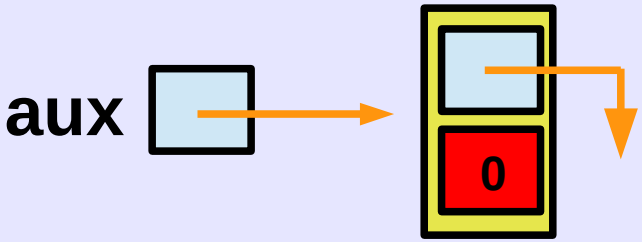
```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

aux->next = list;

list = aux;
```

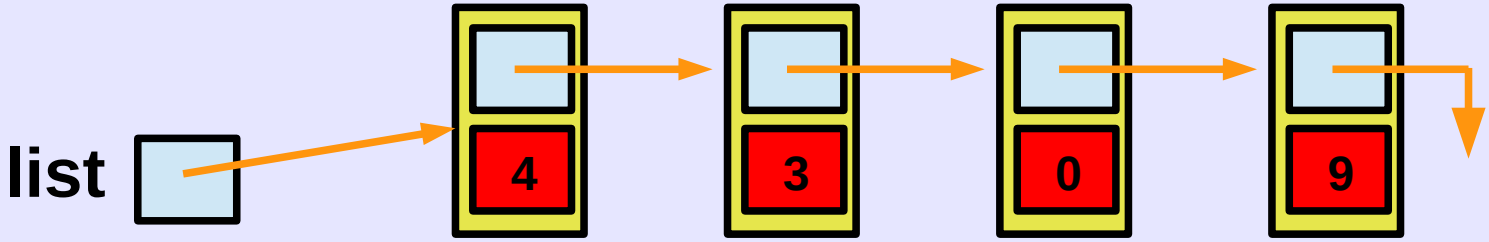




INSERCIÓN AL COMIENZO

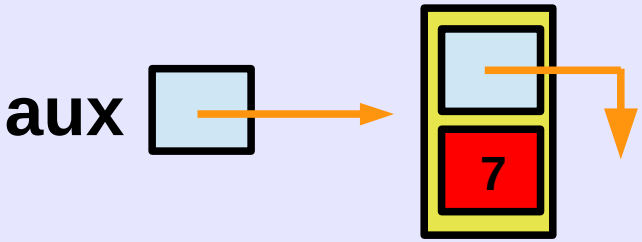
```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

aux->next = list;

list = aux;
```

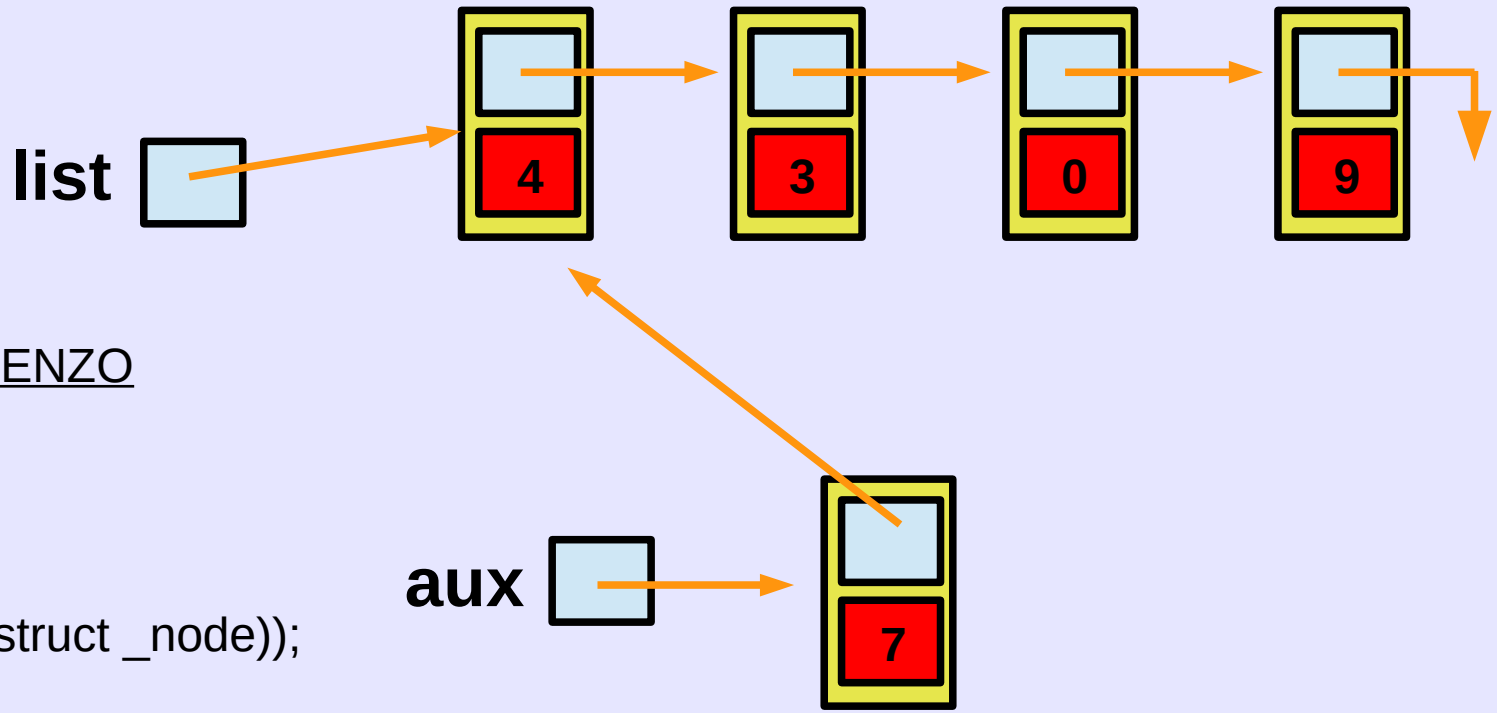




INSERCIÓN AL COMIENZO

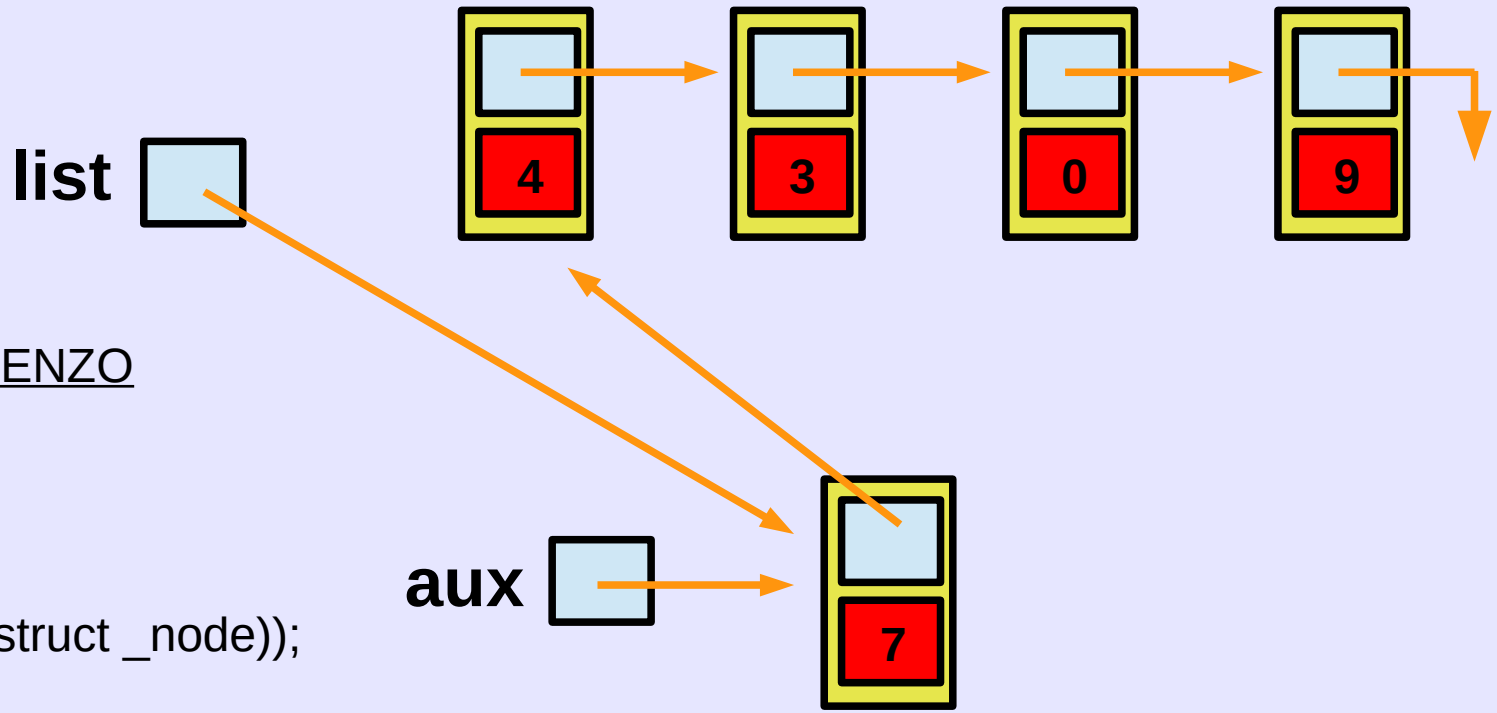
```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

aux->next = list;

list = aux;
```





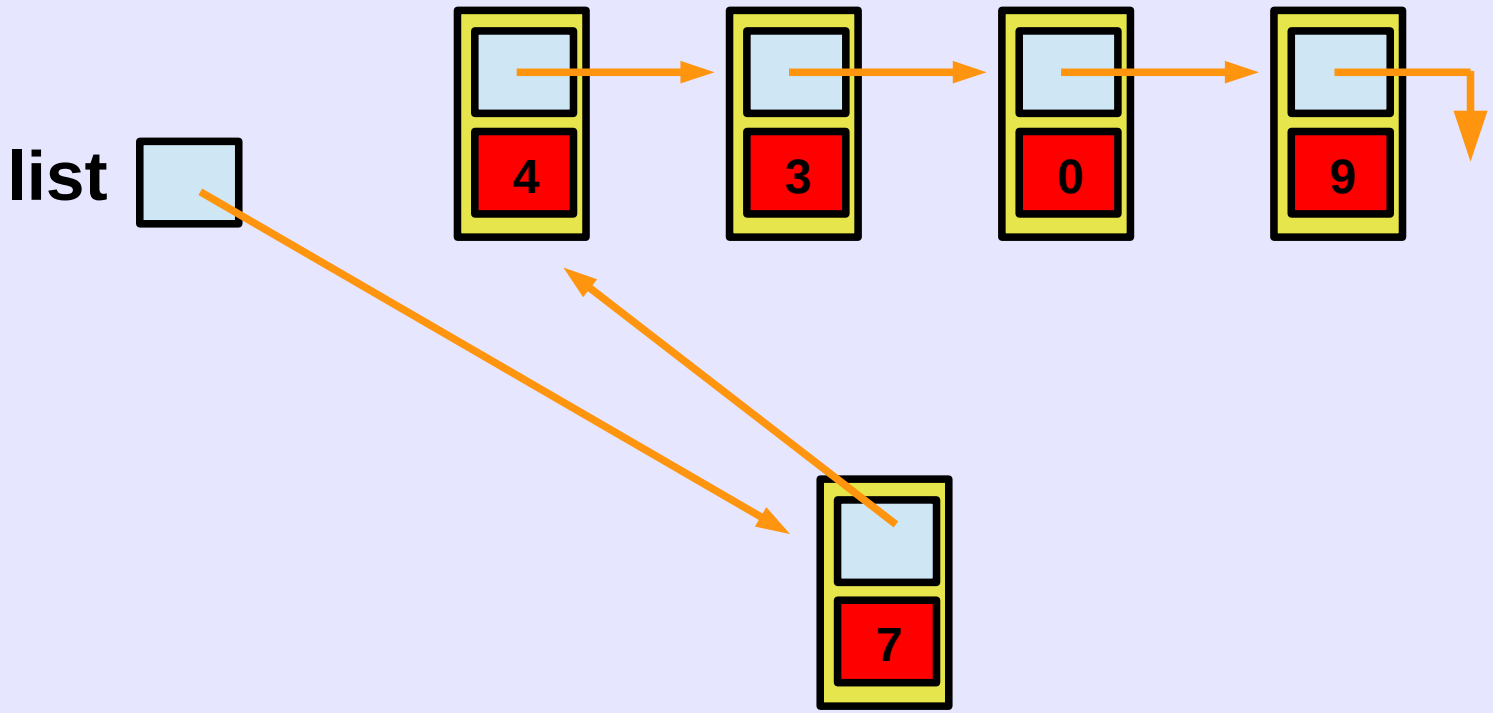
INSERCIÓN AL COMIENZO

```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

aux->next = list;

list = aux;
```

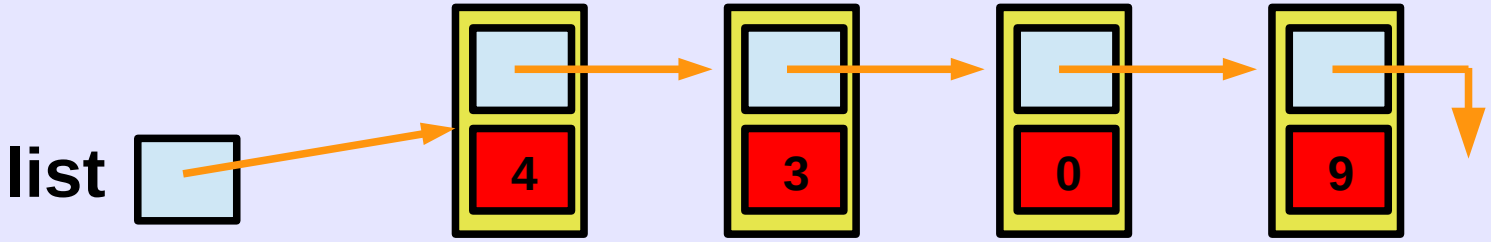


INSERCIÓN AL COMIENZO

```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

aux->next = list;

list = aux;
```



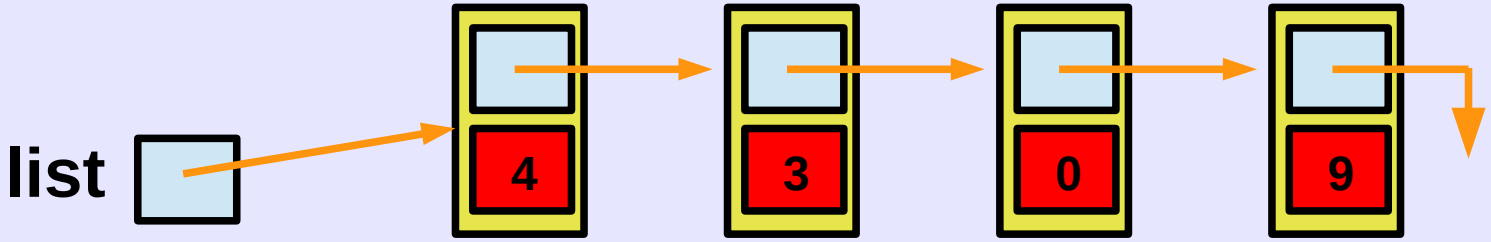
[7,4,3,0,9]



INSERCIÓN AL ÚLTIMO

(CASO LISTA NO VACÍA)

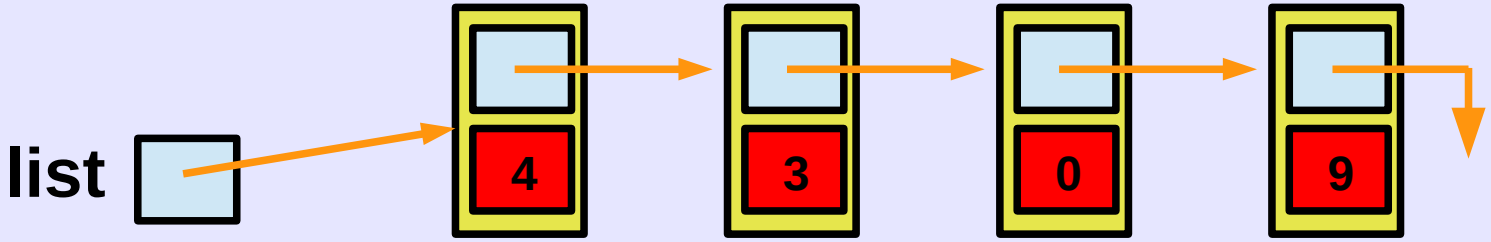
```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

list_t curr = list;

while (curr->next != NULL) {
    curr = curr->next;
}

curr->next = aux;
```



INSERCIÓN AL ÚLTIMO

(CASO LISTA NO VACÍA)

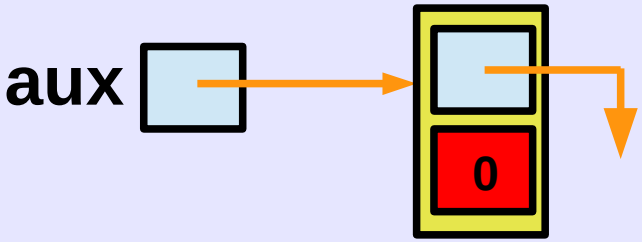
```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

list_t curr = list;

while (curr->next != NULL) {
    curr = curr->next;
}

curr->next = aux;
```

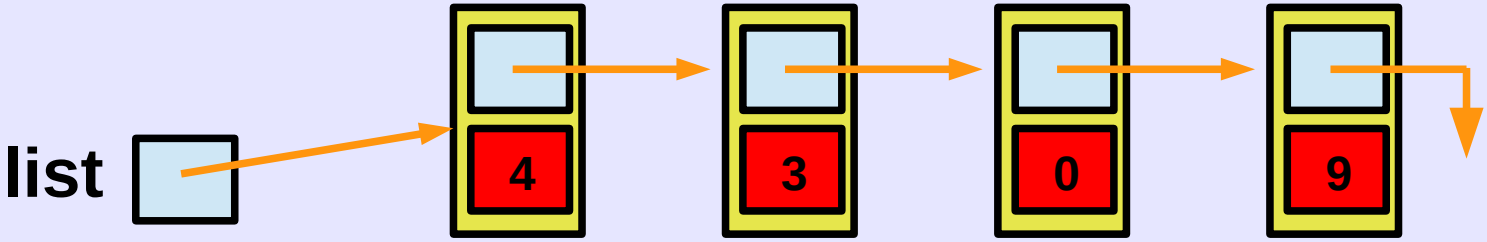


INSERCIÓN AL ÚLTIMO

(CASO LISTA NO VACÍA)

```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

list_t curr = list;

while (curr->next != NULL) {
    curr = curr->next;
}

curr->next = aux;
```

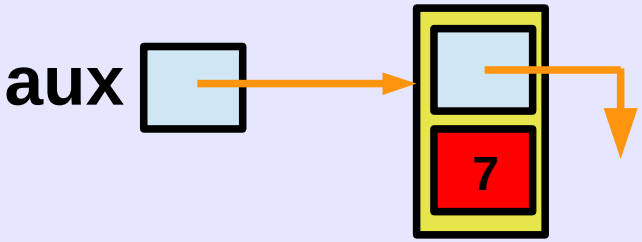




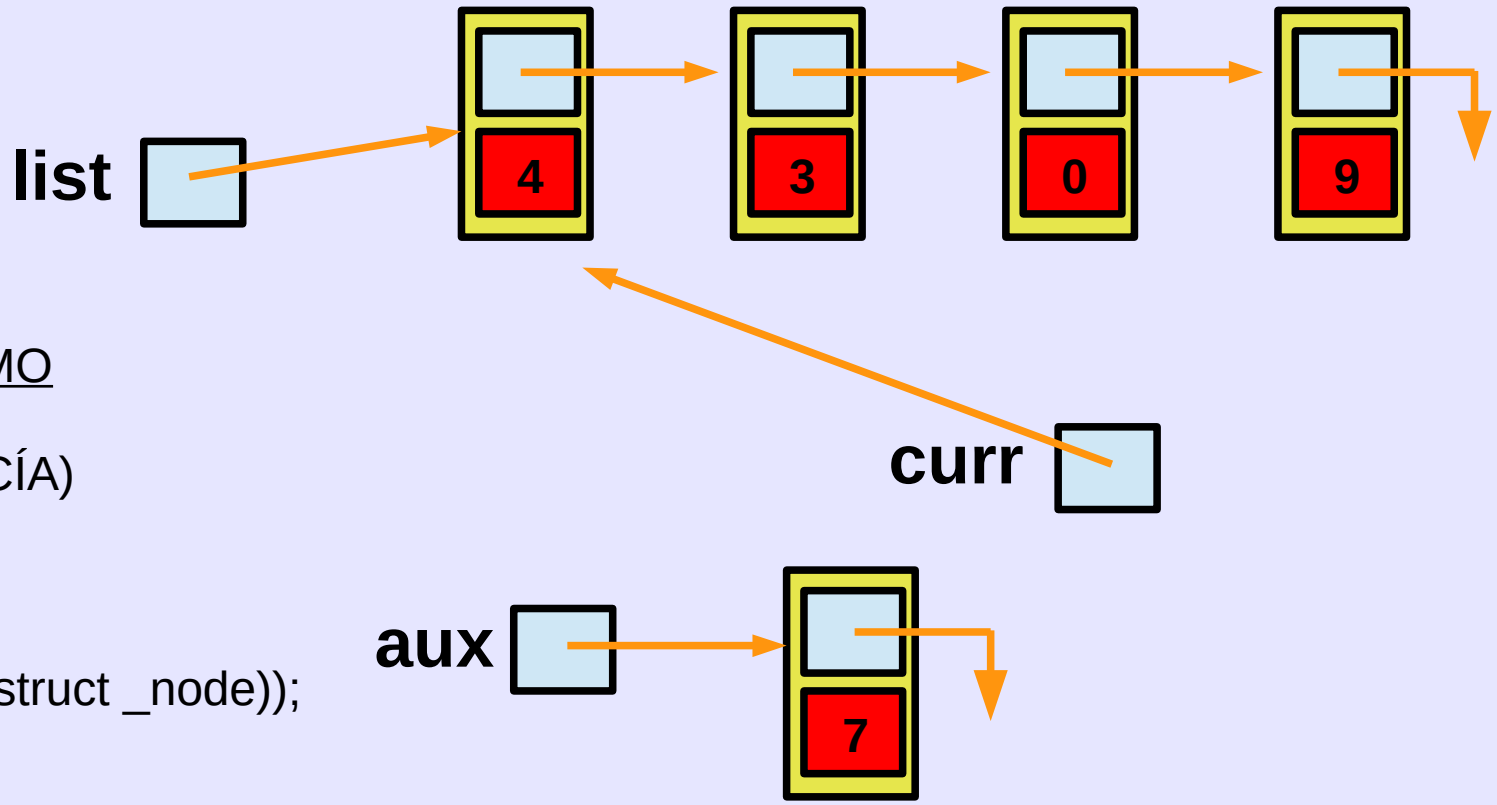
INSERCIÓN AL ÚLTIMO

(CASO LISTA NO VACÍA)

```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;
```



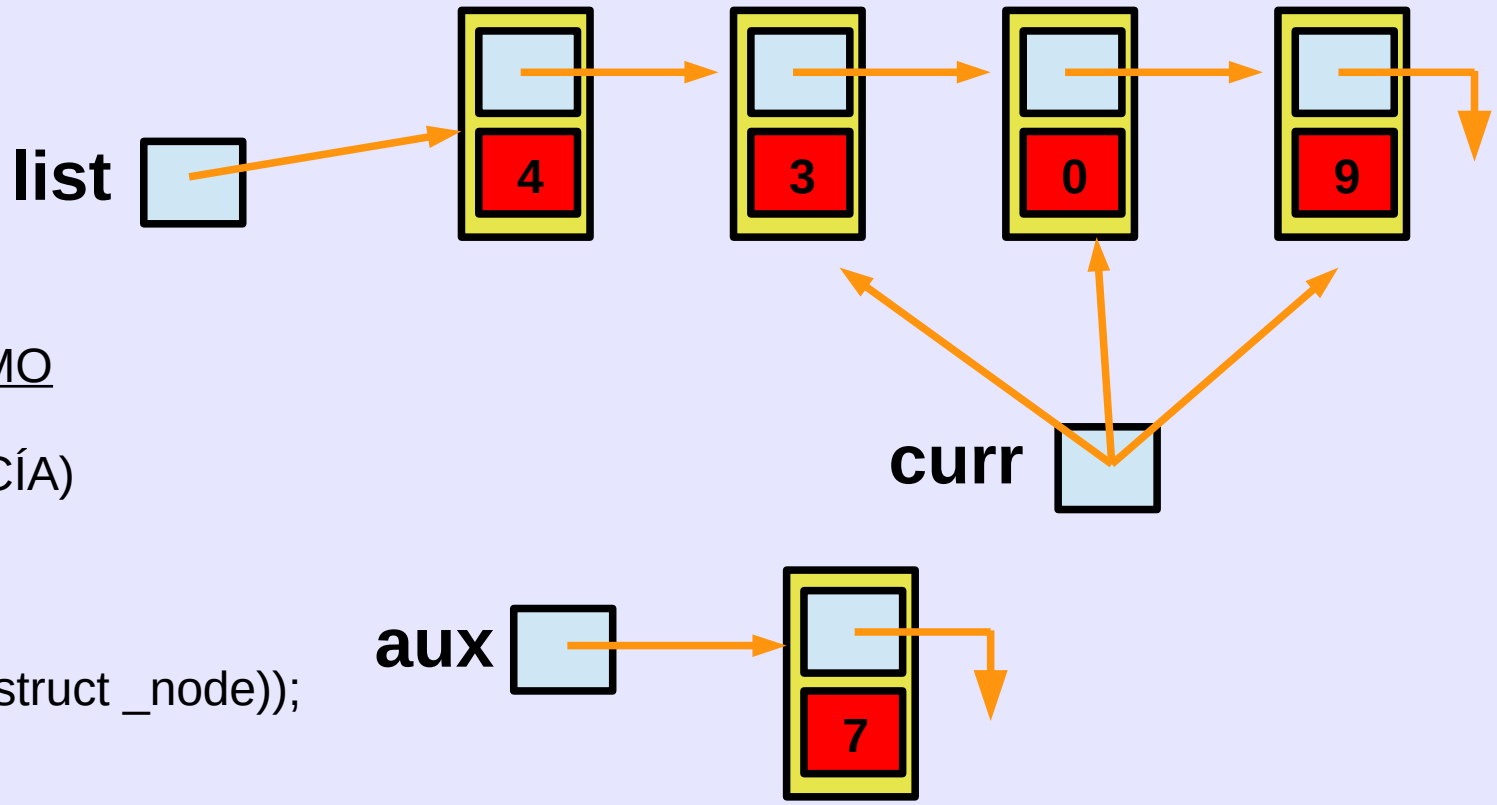

```
list_t curr = list;

while (curr->next != NULL) {
    curr = curr->next;
}

curr->next = aux;
```



INSERCIÓN AL ÚLTIMO

(CASO LISTA NO VACÍA)

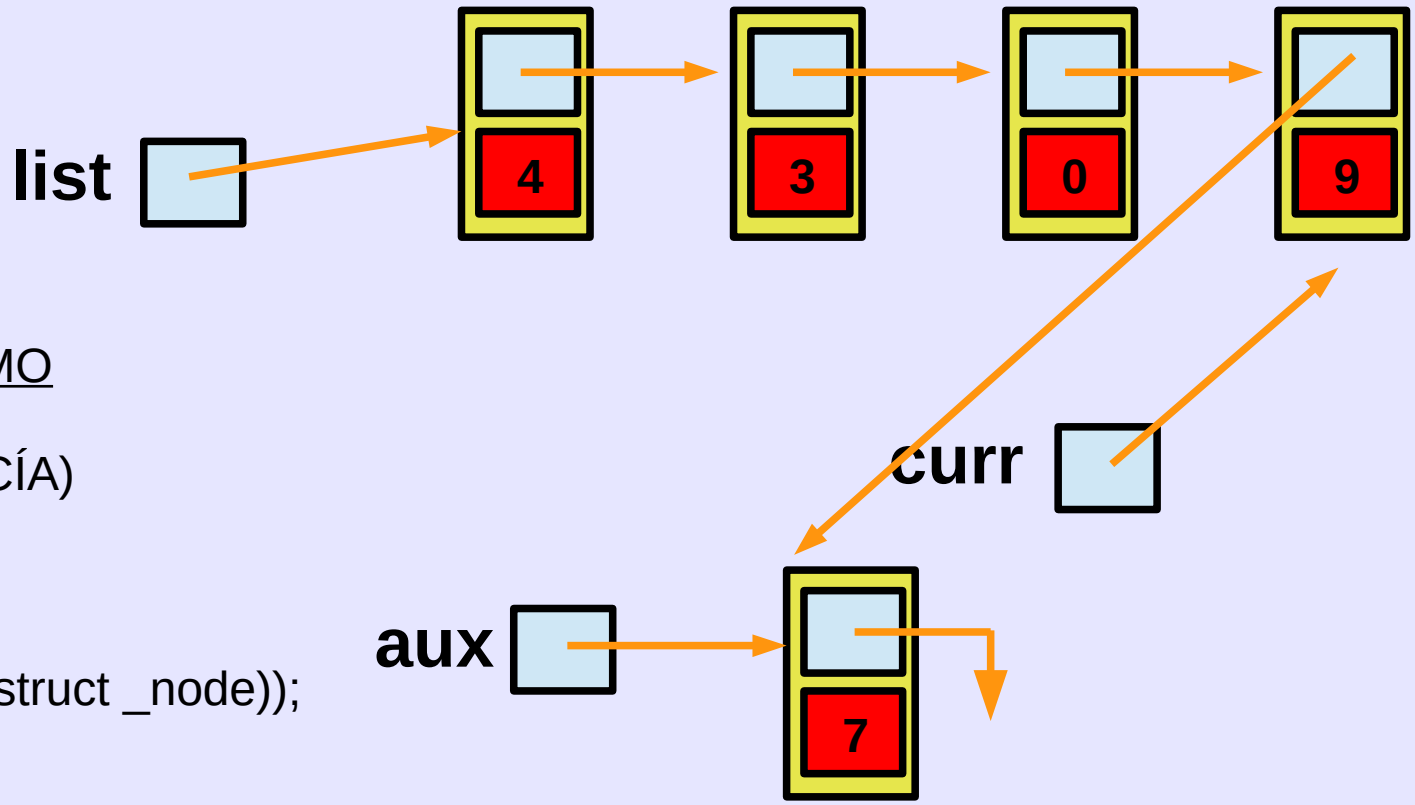
```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

list_t curr = list;

while (curr->next != NULL) {
    curr = curr->next;
}

curr->next = aux;
```



INSERCIÓN AL ÚLTIMO

(CASO LISTA NO VACÍA)

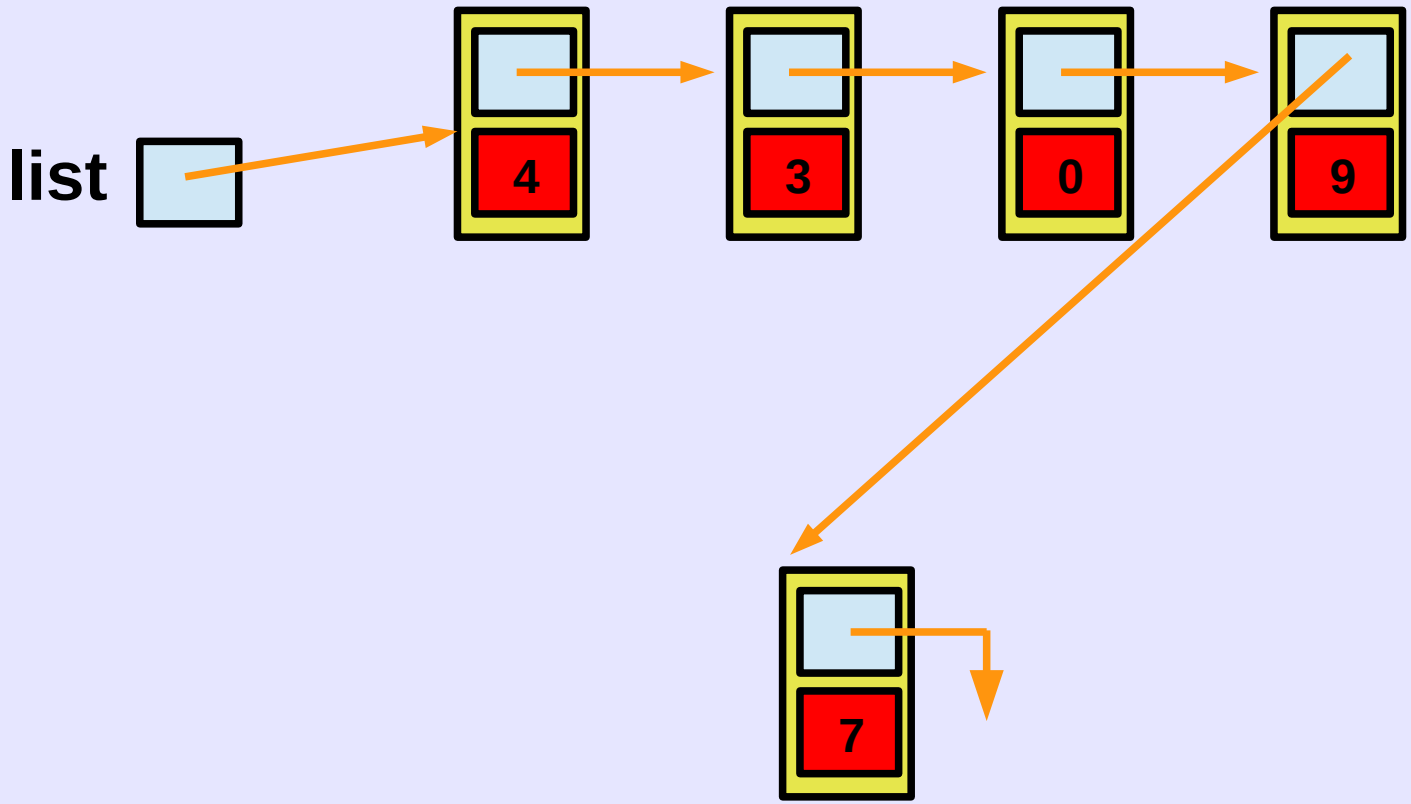
```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

list_t curr = list;

while (curr->next != NULL) {
    curr = curr->next;
}

curr->next = aux;
```



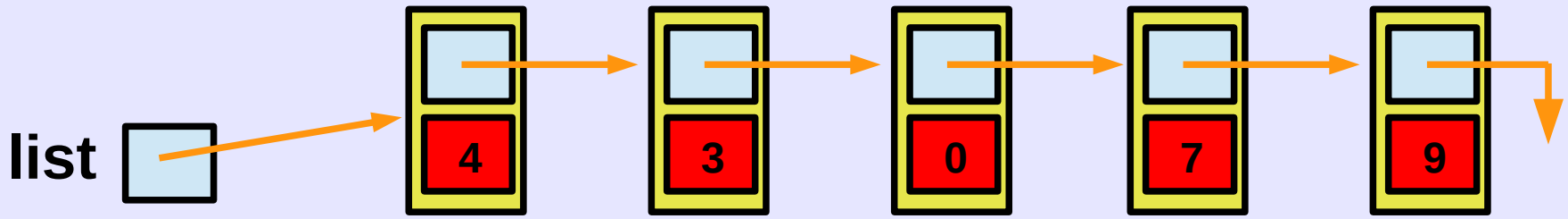
INSERCIÓN AL ÚLTIMO

(CASO LISTA NO VACÍA)

```
list_t aux = NULL;

aux = calloc(1, sizeof(struct _node));

aux->elem = 7;

list_t curr = list;

while (curr->next != NULL) {
    curr = curr->next;
}

curr->next = aux;
```

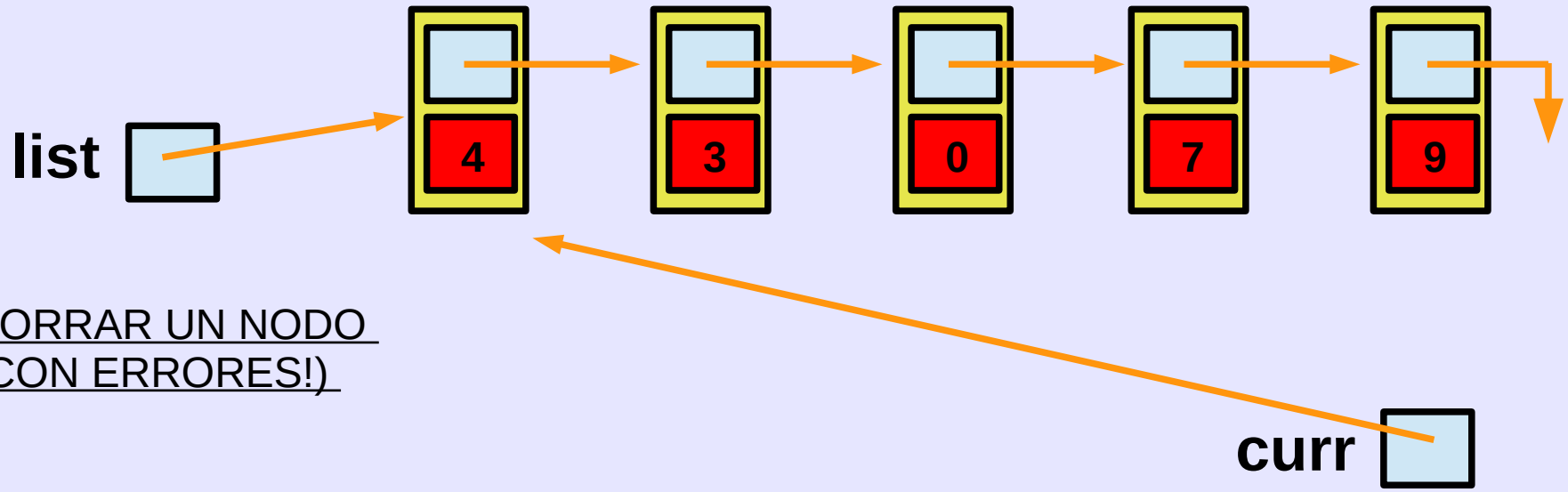


[4,3,0,9,7]



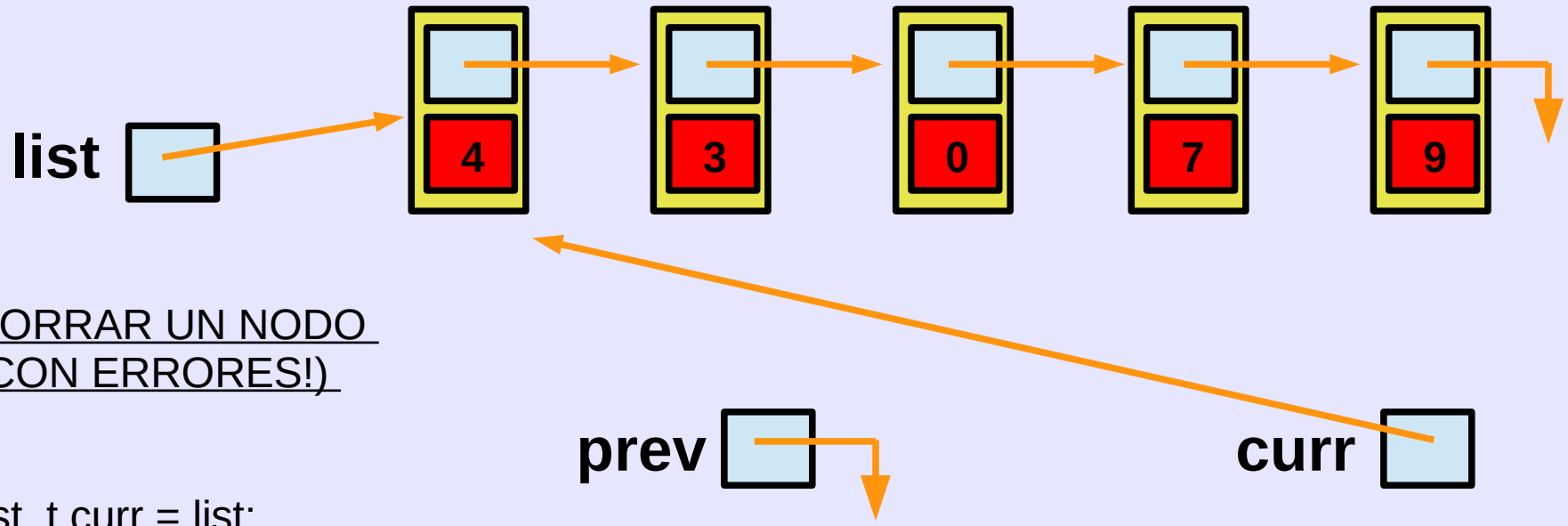
BORRAR UN NODO
(CON ERRORES!)

```
list_t curr = list;
list_t prev = NULL;

while (curr  != NULL && curr->elem != 7) {
    prev = curr;
    curr  = curr->next;
}

prev->next = curr->next;
free(curr);
curr = NULL;
```



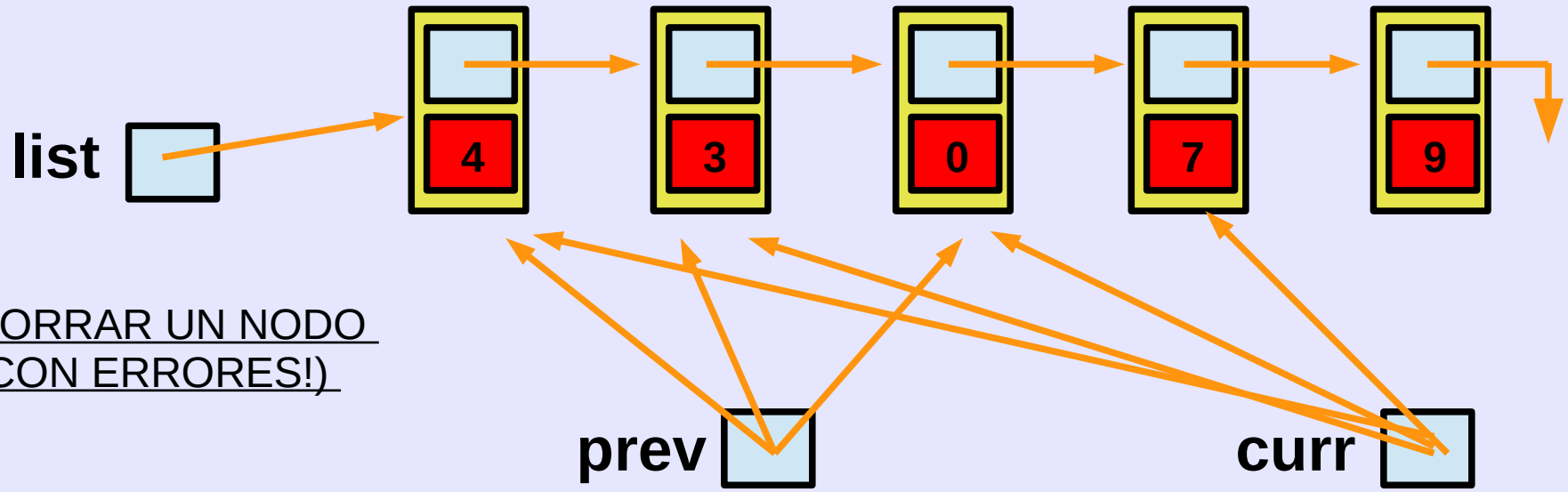
BORRAR UN NODO
(CON ERRORES!)

```
list_t curr = list;
list_t prev = NULL;

while (curr  != NULL && curr->elem != 7) {
    prev = curr;
    curr  = curr->next;
}

prev->next = curr->next;
free(curr);
curr = NULL;
```



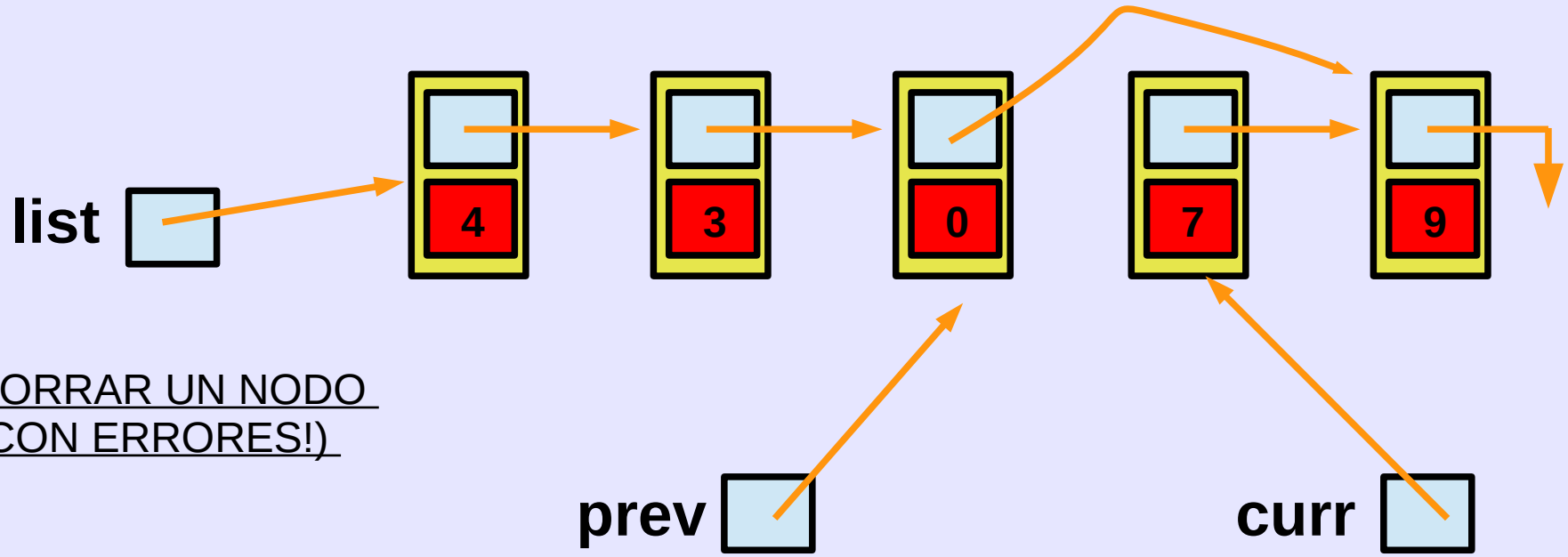
BORRAR UN NODO
(CON ERRORES!)

```
list_t curr = list;
list_t prev = NULL;

while (curr  != NULL && curr->elem != 7) {
    prev = curr;
    curr  = curr->next;
}

prev->next = curr->next;
free(curr);
curr = NULL;
```



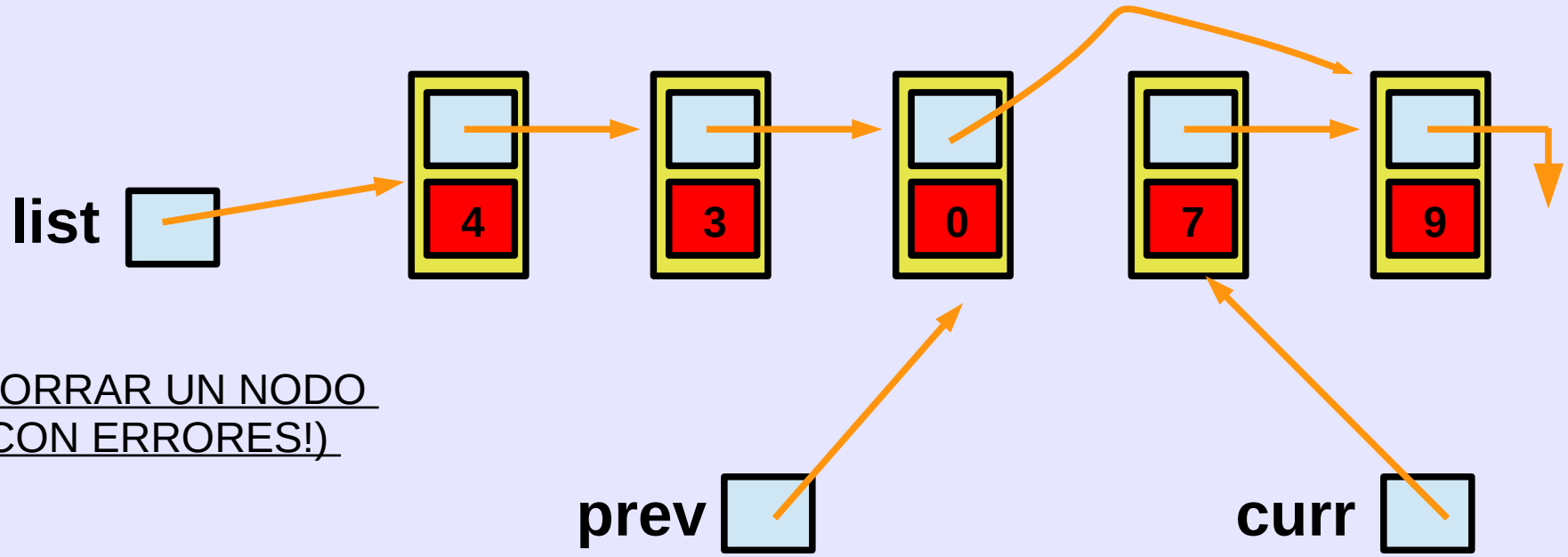
BORRAR UN NODO
(CON ERRORES!)

```
list_t curr = list;
list_t prev = NULL;

while (curr  != NULL && curr->elem != 7) {
    prev = curr;
    curr  = curr->next;
}

prev->next = curr->next;
free(curr);
curr = NULL;
```



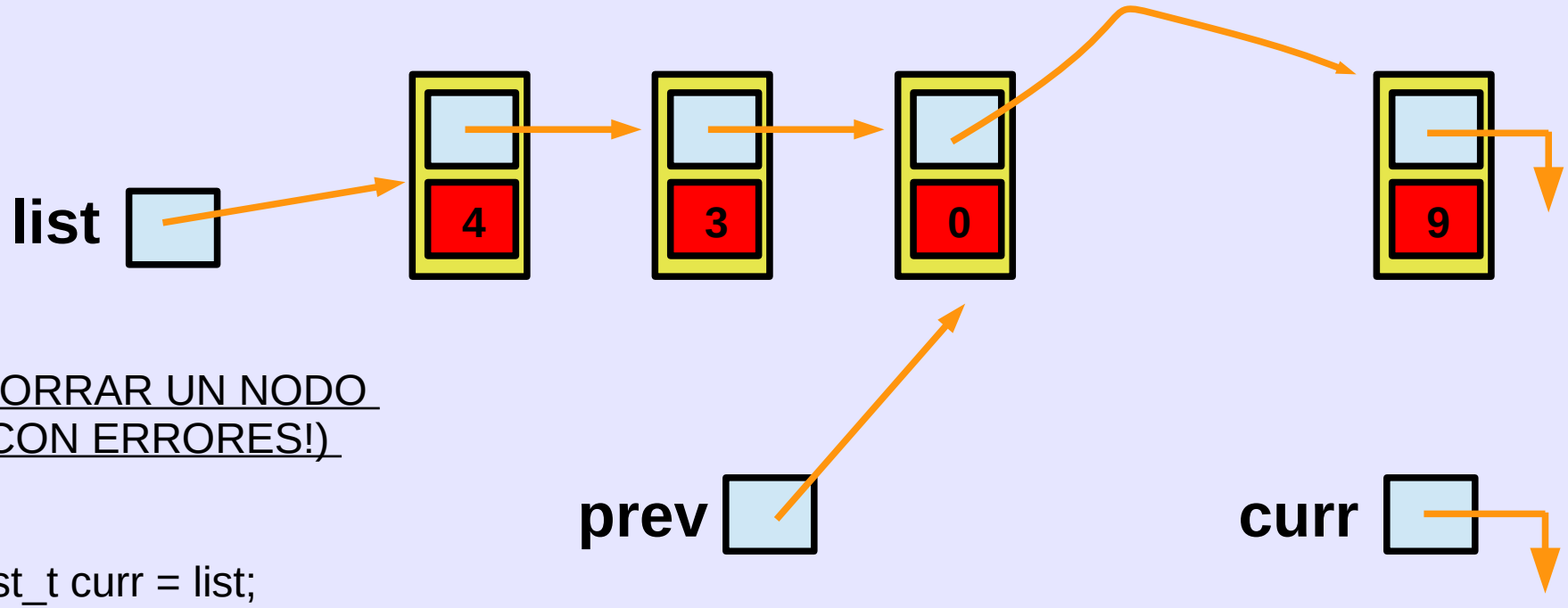
BORRAR UN NODO
(CON ERRORES!)

```
list_t curr = list;
list_t prev = NULL;

while (curr  != NULL && curr->elem != 7) {
    prev = curr;
    curr  = curr->next;
}

prev->next = curr->next;
free(curr);
curr = NULL;
```



BORRAR UN NODO
(CON ERRORES!)

```
list_t curr = list;
list_t prev = NULL;

while (curr  != NULL && curr->elem != 7) {
    prev = curr;
    curr  = curr->next;
}

prev->next = curr->next;
free(curr);
curr = NULL;
```



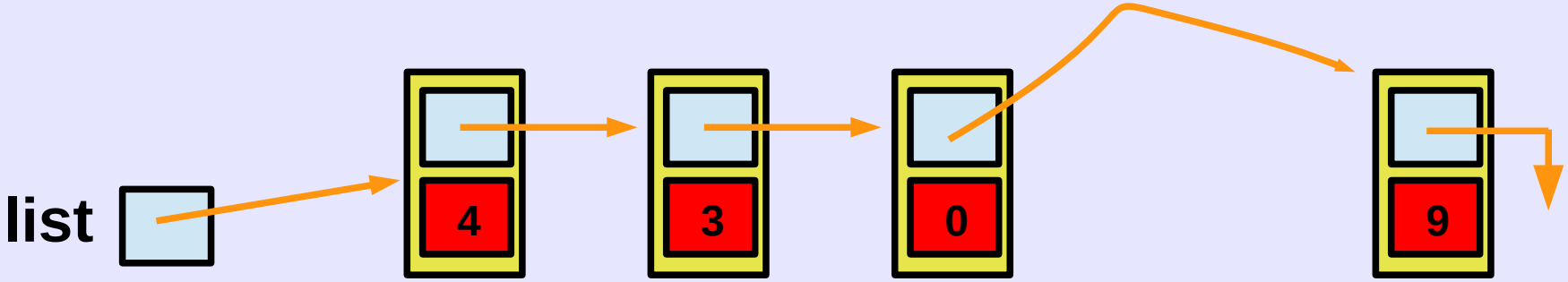
BORRAR UN NODO
(CON ERRORES!)

```
list_t curr = list;
list_t prev = NULL;

while (curr  != NULL && curr->elem != 7) {
    prev = curr;
    curr  = curr->next;
}

prev->next = curr->next;
free(curr);
curr = NULL;
```

Ejercicio: Arreglar el código.

No funciona cuando:

- La lista es vacía.
- El “7” no está en la lista.
- El “7” está en el primer nodo.



[4,3,0,9]

(to be continued...)