


Estructuras de datos dinámicas III

1ºDAM IoT

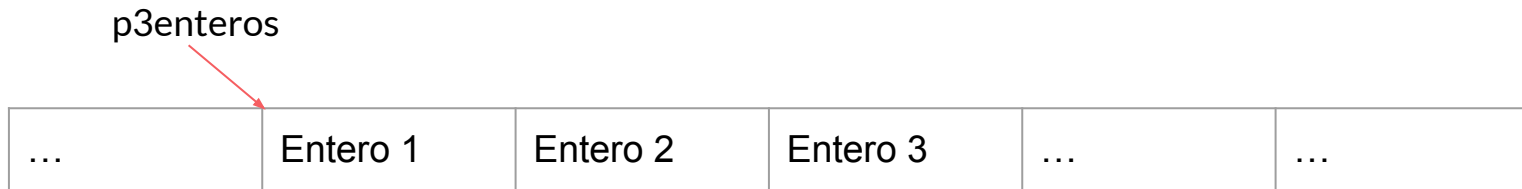
Reserva de memoria múltiple

Si queremos reservar espacio para 3 enteros que se guardan consecutivos en memoria utilizamos también la función `malloc()`.

```
int *p3enteros;  
p3enteros = (int *)malloc(sizeof(int) * 3);
```



El puntero quedaría apuntando a la primera de todas (la de la dirección más baja) pero es posible acceder fácilmente al resto, como si de un array se tratara.



Reserva de memoria múltiple

Ejemplo: Podemos solicitar al usuario el tamaño del array y crearlo dinámicamente.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int dimension;
    printf("Dime el tamaño del array: ");
    scanf("%d", &dimension);
    int *p3enteros;
    p3enteros = (int *) malloc(sizeof(int)*dimension);
    int i;
    //p3enteros apunta al primer elemento
    for(i=0;i<dimension;i++){
        p3enteros[i]= i+1; //inicializamos
    }
    //imprimimos el contenido
    for(i=0;i<dimension;i++){
        printf("%d\n", p3enteros[i]);
    }
    return 0;
}
```

```
Dime el tamaño del array: 10
1
2
3
4
5
6
7
8
9
10
```

Reserva de memoria múltiple

Podemos interpretar los apuntadores de C como un tipo particular de array dinámico (array porque almacenan más de un valor del mismo tipo bajo el mismo identificador de variable, y dinámico porque hacemos la reserva de memoria en tiempo de ejecución, con la cantidad elegida en tiempo de ejecución).

Es importante reiterar que:

- Las direcciones serán consecutivas, una justo a continuación de la otra.
- El apuntador siempre apunta a la dirección con número más bajo.

Incrementar/decrementar valor puntero

Si lo que queremos es incrementar el valor apuntado por el puntero entonces podemos hacer lo siguiente:

```
(*puntero)++;
```

```
(*puntero)--;
```

Incrementar / decrementar valor puntero

```
Dirección de memoria puntero1 00A61678
Dirección de memoria puntero2 00313000
Valor apuntado puntero1 10888240
Valor apuntado puntero1 0
Dirección de memoria puntero1 00A61678
Dirección de memoria puntero2 00313000
Valor apuntado puntero1 5
Valor apuntado puntero1 0
Dirección de memoria puntero1 00A61678
Dirección de memoria puntero2 00A61678
Valor apuntado puntero1 5
Valor apuntado puntero1 5
Valor apuntado puntero1 6
Valor apuntado puntero1 6
Valor apuntado puntero1 7
Valor apuntado puntero1 7
```

```
int *puntero1 = (int *)malloc(sizeof(int));
int *puntero2;
//La dirección de puntero2 es inválida, sin inicializar
printf("Dirección de memoria puntero1 %p\n", puntero1);
printf("Dirección de memoria puntero2 %p\n", puntero2);
printf("Valor apuntado puntero1 %d\n", *puntero1);
printf("Valor apuntado puntero1 %d\n", *puntero2);

*puntero1 = 5;
printf("Dirección de memoria puntero1 %p\n", puntero1);
printf("Dirección de memoria puntero2 %p\n", puntero2);
printf("Valor apuntado puntero1 %d\n", *puntero1);
printf("Valor apuntado puntero1 %d\n", *puntero2);

puntero2 = puntero1;
//Ahora los dos apuntan a la misma dirección
printf("Dirección de memoria puntero1 %p\n", puntero1);
printf("Dirección de memoria puntero2 %p\n", puntero2);
//Y los dos tienen el mismo valor
printf("Valor apuntado puntero1 %d\n", *puntero1);
printf("Valor apuntado puntero1 %d\n", *puntero2);

//Incrementamos el valor apuntado por puntero1
(*puntero1)++;
printf("Valor apuntado puntero1 %d\n", *puntero1);
printf("Valor apuntado puntero1 %d\n", *puntero2);

//Incrementamos el valor apuntado por puntero2
(*puntero2)++;
printf("Valor apuntado puntero1 %d\n", *puntero1);
printf("Valor apuntado puntero1 %d\n", *puntero2);
```

Nota sobre free()

Si dos punteros apuntan a la misma zona reservada dinámicamente (reserva que se ha hecho UNA vez). Basta con realizar free() de uno de ellos (y no de los dos) para que la memoria se libere.

Es lógico que debe haber un y sólo un free() por cada malloc() hecho.

Nota sobre free()

En el ejemplo anterior puntero1 y puntero2 apuntan a la misma dirección de memoria. Sólo sería necesario hacer un free de uno de los punteros para liberar el espacio de memoria que reservamos.

```
int *puntero1 = (int *)malloc(sizeof(int));
int *puntero2;
*puntero1 = 5;
puntero2 = puntero1;
(*puntero1)++;
(*puntero2)++;
printf("ANTES de liberar memoria\n");
printf("Dirección de memoria puntero1 %p\n", puntero1);
printf("Dirección de memoria puntero2 %p\n", puntero2);
printf("Valor apuntado puntero1 %d\n", *puntero1);
printf("Valor apuntado puntero2 %d\n", *puntero2);

free(puntero1);
printf("DESPUÉS de liberar memoria\n");
printf("Dirección de memoria puntero1 %p\n", puntero1);
printf("Dirección de memoria puntero2 %p\n", puntero2);
printf("Valor apuntado puntero1 %d\n", *puntero1);
printf("Valor apuntado puntero2 %d\n", *puntero2);
```

```
Dirección de memoria puntero1 00C21678
Dirección de memoria puntero2 00C21678
Valor apuntado puntero1 7
Valor apuntado puntero2 7
DESPUÉS de liberar memoria
Dirección de memoria puntero1 00C21678
Dirección de memoria puntero2 00C21678
Valor apuntado puntero1 12723248
Valor apuntado puntero2 12723248
```

Observa que seguimos apuntando a la misma dirección de memoria pero ya no es nuestra.

Nota sobre free()

Una buena práctica es inicializar con NULL un puntero o asignarle NULL después de liberar el espacio de un puntero. Cada vez que se use un puntero debería comprobarse si es NULL.

```
free(puntero1);
puntero1 = NULL;
puntero2 = NULL;

if(puntero1!=NULL && puntero2!=NULL){
    printf("DESPUÉS de liberar memoria\n");
    printf("Dirección de memoria puntero1 %p\n", puntero1);
    printf("Dirección de memoria puntero2 %p\n", puntero2);
    printf("Valor apuntado puntero1 %d\n", *puntero1);
    printf("Valor apuntado puntero2 %d\n", *puntero2);
}
```

Aritmética de punteros

```
int *p;  
int *q;  
p = (int *)malloc(sizeof(int) * 5);  
q = p + 1;
```

El resultado de esta operación aritmética es la dirección de memoria del siguiente valor apuntado por “p”.



Aritmética de punteros

```
int *p;  
int *q;  
p = (int *)malloc(sizeof(int) * 5);  
q = p + 1;  
  
*(q - 1) = 10; /* equivale a *p = 10; */  
*(p + 1) = 15; /* equivale a *q = 15; */
```



Aritmética de punteros

```
int *p;  
int *q;  
p = (int *)malloc(sizeof(int) * 5);  
q = p + 1;  
  
*(q - 1) = 10; /* equivale a *p = 10; */  
*(p + 1) = 15; /* equivale a *q = 15; */  
  
*(p + 2) = 20;  
*(q + 2) = 25;  
*(q + 3) = 30;
```

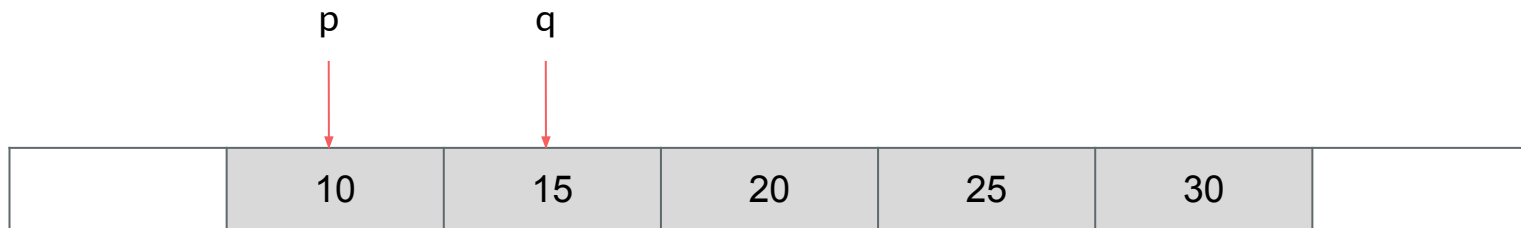
¿Dónde se guardan los valores de las 3 últimas instrucciones?



Aritmética de punteros

```
int *p;  
int *q;  
p = (int *)malloc(sizeof(int) * 5);  
q = p + 1;  
  
*(q - 1) = 10; /* equivale a *p = 10; */  
*(p + 1) = 15; /* equivale a *q = 15; */  
  
*(p + 2) = 20;  
*(q + 2) = 25;  
*(q + 3) = 30;
```

¿Dónde se guardan los valores de las 3 últimas instrucciones?



Aritmética de punteros

Ejercicio: Reserva espacio para 5 números enteros utilizando un puntero. Asigna los valores del 1 al 5 al espacio reservado. Utiliza un bucle for para recorrer las direcciones de memoria y asigna con el puntero el valor correspondiente. Libera al terminar el espacio de memoria del free.

Aritmética de punteros

Ejercicio: Reserva espacio para 5 números enteros utilizando un puntero. Asigna los valores del 1 al 5 al espacio reservado. Utiliza un bucle for para recorrer las direcciones de memoria y asigna con el puntero el valor correspondiente.

```
int *p;  
p = (int *)malloc(sizeof(int) * 5);  
for (int i = 0; i < 5; i++)  
{  
    *(p + i) = i + 1;  
}
```

Nota sobre free()

En el siguiente caso no podemos hacer `free(p2)` porque `p2` no apunta a la misma dirección que nos devolvió el `malloc()`.

```
int *p1 = (int *)malloc(sizeof(int) * 10); // reserva 10 ints
int *p2 = p1 + 1;                        // p2 apunta a los 9 últimos ints reservados
```

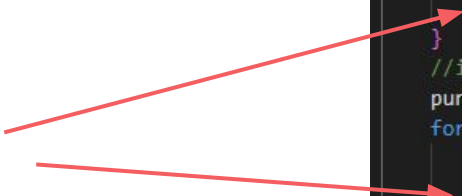
Sin embargo este sí sería válido porque cuando se libera `p2` si que apunta a la dirección original devuelta por `malloc()`.

```
int *p1 = (int *)malloc(sizeof(int) * 10); // reserva 10 ints
int *p2 = NULL;
p1++; /* p1 pasa a apuntar al segundo de los 10 ints reservados */
/* a partir de este momento "free(p1)" no sería válido */
p2 = p1 - 1; /* p2 pasa a apuntar al primero de los 10 */
free(p2);
```


Incrementar/decrementar valor puntero

Ejemplo: Volviendo al ejemplo del principio podemos utilizar un puntero que apunte al inicio del array y vamos incrementando su valor recorriendo el array.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int dimension;
    printf("Dime el tamaño del array: ");
    scanf("%d", &dimension);
    int *p3enteros;
    p3enteros = (int *) malloc(sizeof(int)*dimension);
    int i;
    int *puntero;
    puntero = p3enteros; //puntero apunta a la dirección de p3enteros
    for(i=0;i<dimension;i++){
        *puntero= i+1; //inicializamos
        puntero++; //Incrementamos en 1
    }
    //imprimimos el contenido
    puntero = p3enteros;
    for(i=0;i<dimension;i++){
        printf("%d\n", *puntero);
        puntero++; //Incrementamos en 1
    }
    return 0;
}
```



Redimensionar realloc

Podemos redimensionar el tamaño de un array. Es lo bueno de utilizar punteros. Al inicializar un programa puede ser que necesitemos un array de tamaño 5 pero si más adelante necesitamos que sea de 10 podemos solicitar más espacio.

La función **realloc** funciona exactamente igual que la función **malloc**.

Ejemplo:

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

int main() {

    int *miArray = (int *)malloc(5 * sizeof(int)); // Array de tamaño 5
    for (int i = 0; i < 5; i++) { // Inicializar el array con algunos valores
        miArray[i] = i * 2;
    }
    for (int i = 0; i < 5; i++) {
        printf("Posición: %d %d \n", i, miArray[i]); // Imprimir los valores del array
    }
    // Redimensionar el array a 10 enteros utilizando realloc
    miArray = (int *)realloc(miArray, 10 * sizeof(int));
    // Inicializar los nuevos elementos del array
    for (int i = 5; i < 10; i++) {
        miArray[i] = i * 2;
    }
    // Imprimir los valores del array después de redimensionar
    for (int i = 0; i < 10; i++) {
        printf("Posición: %d %d \n", i, miArray[i]);
    }
    // Liberar la memoria asignada dinámicamente utilizando free
    free(miArray);
    return 0;
}
```

Redimensionar

Utilizaremos otras formas para redimensionar según las necesidades del usuario.

Realizaremos diferentes implementaciones según necesitemos insertar o borrar nuestra estructura de datos dinámica.

Las estructuras dinámicas que veremos serán:

- Listas
- Pilas
- Colas

En estas estructuras dinámicas guardaremos los datos englobados dentro de una estructura como las que hemos visto hasta ahora.

Punteros a registros (struct)

Un puntero puede apuntar también a tipos de datos más complejos.

Imagina una estructura que guarda diferentes pero hasta que no se ha avanzado en el programa no sabemos si necesitamos guardar dichos datos o no. Sería mucho desperdicio de espacio...

```
struct Medicion
{
    int id;
    double utmx;
    double utmy;
    float altitud;
    byte estado;
};
```

Punteros a registros (struct)

Podríamos declarar un puntero a la estructura de modo que solo reservamos espacio en la memoria cuando lo necesitamos.

```
struct Medicion *m1;
/* ... */
/* Si necesitamos registrar una medición */
m1 = (struct Medicion *)malloc(sizeof(struct Medicion));
(*m1).id = 1;
(*m1).utm_x = 40.012345;
(*m1).utm_y = 3.012345;
(*m1).altitud = 112;
(*m1).estado = 1;
/* Operaciones con m1 */
/* ... */
/* Hemos terminado con m1, liberamos espacio */
free(m1);
m1 = NULL;
```

```
struct Medicion *m1;
/* ... */
/* Si necesitamos registrar una medición */
m1 = (struct Medicion *)malloc(sizeof(struct Medicion));
m1->id = 1;
m1->utm_x = 40.012345;
m1->utm_y = 3.012345;
m1->altitud = 112;
m1->estado = 1;
/* Operaciones con m1 */
/* ... */
/* Hemos terminado con m1, liberamos espacio */
free(m1);
m1 = NULL;
```