Tema 3. Punteros y Arrays

FUNDAMENTOS DE PROGRAMACIÓN II

Profesor: Baldomero Imbernón Tudela

Escuela Politécnica Superior
Grado en Ingeniería Informática

Contenidos

- Aritmética de punteros
- Punteros a punteros
- Uso de punteros para el procesamiento de arrays
- Uso del nombre de array como puntero
- Arrays como argumentos
- Punteros y arrays multidimensionales
- Gestión dinámica de memoria

Justificación

- C tiene dos formas de procesar arrays:
  - Indexación, como en la expresión **`a[i];`**
  - Punteros.
- En C existe una íntima relación entre **arrays** y **punteros**.
  - Es crítico entender esta relación.
  - La principal razón del uso de punteros para manejar arrays es la eficiencia, sin embargo, ya no es tan importante debido a la gran mejora de los compiladores.

Contenidos

- **Aritmética de punteros**
- Punteros a punteros
- Uso de punteros para el procesamiento de arrays
- Uso del nombre de array como puntero
- Arrays como argumentos
- Punteros y arrays multidimensionales
- Gestión dinámica de memoria

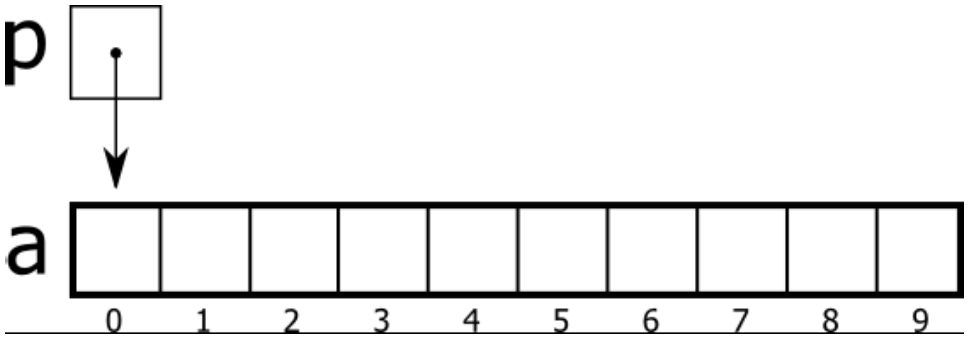
Aritmética de punteros (I)

- Un puntero puede apuntar a un elemento en concreto dentro de un array:

```
int a[10], *p;
```

- Se puede hacer que el puntero p apunte al elemento a[0] del array así:

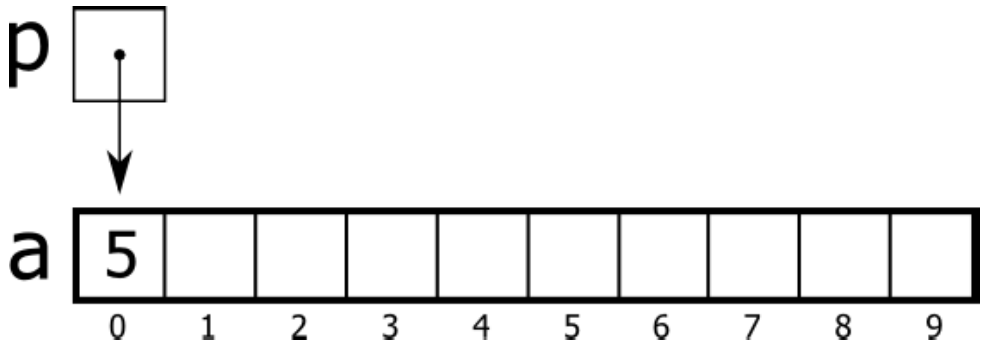
```
p = &a[0];
```



Aritmética de punteros (II)

- Ahora se puede acceder al elemento a[0] a través de *p, se podría almacenar un valor así:

```
*p = 5;
```



- Una vez que p apunta dentro del array, a través de operaciones aritméticas de punteros se puede hacer que p apunte a otros elementos del array.
- Un puntero aumenta o decrementa según el tamaño del tipo referenciado.

Aritmética de punteros (III)

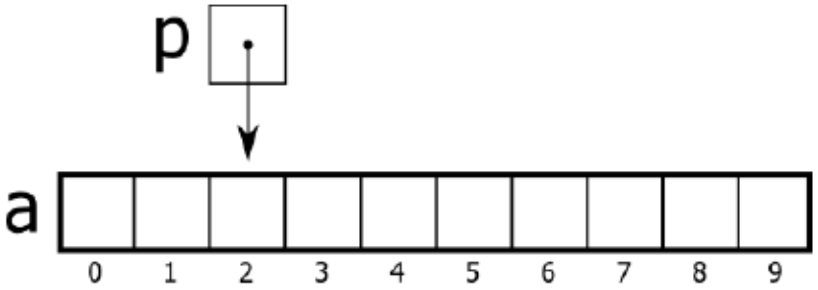
- Las operaciones que se pueden realizar con punteros son:
  - Sumar un entero a un puntero.
  - Restar un entero de un puntero.
  - Sustraer un puntero de otro.
- Supongamos las siguientes declaraciones:

```
int a[10], *p, *q, i;
```

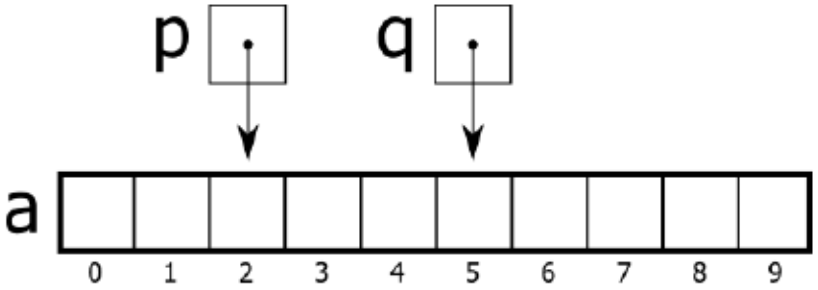
- Si p apunta al elemento **`a[i]`**, **`p + j`** apuntará al elemento **`a[i+j]`**, siendo **`j`** un entero.

Aritmética de punteros: Suma

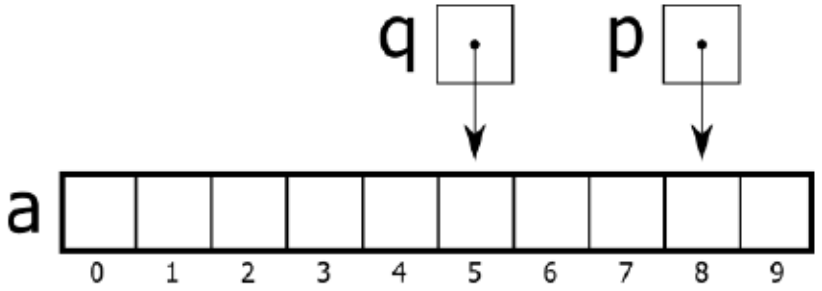
```
p = &a[2];
```




```
q = p + 3;
```

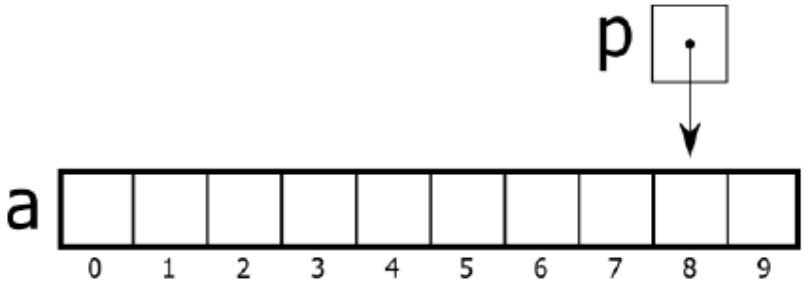



```
p += 6;
```

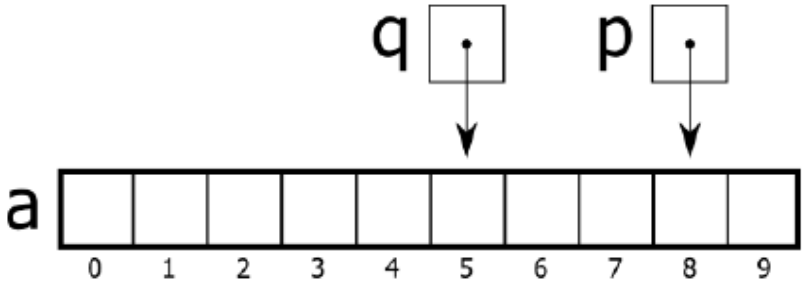


Aritmética de punteros: Resta (I)

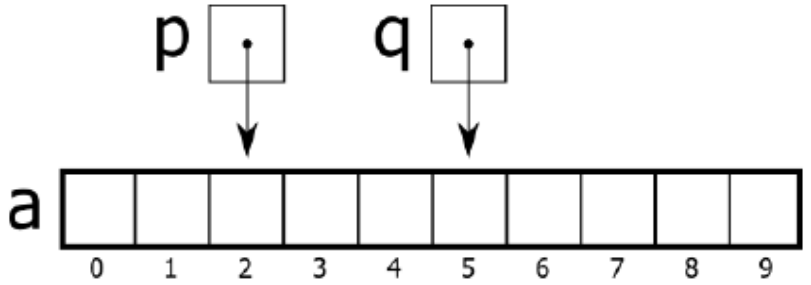
```
p = &a[8];
```




```
q = p - 3;
```




```
p -= 6;
```

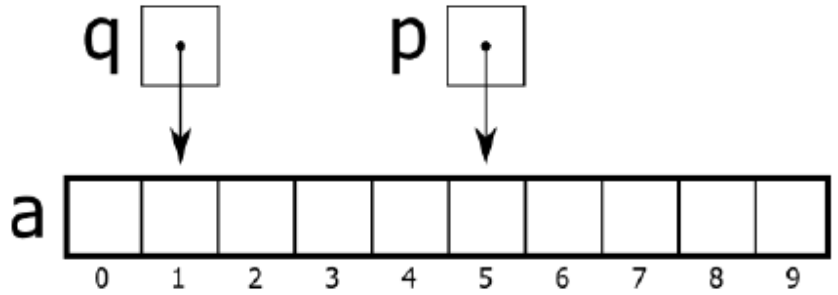


Aritmética de punteros: Resta (II)

- Cuando se restan dos punteros, el resultado es la distancia, en números de elementos del array, entre los dos punteros.
  - Si **p** apunta a **a[i]**, y **q** lo hace a **a[j]**, **p - q** será igual a **i - j**
  - Si uno de los punteros no apunta a algún elemento del array, o los dos punteros no son del mismo tipo, el comportamiento será **indefinido**.

```
p = &a[5];
q = &a[1];

i = p - q;     // i vale 4
i = q - p;     // i vale -4
```



Comparación de punteros

- Se pueden comparar punteros usando los operadores relacionales y de igualdad.
  - <, <=, > y >=
    - Esto solo tiene sentido si los punteros apuntan a elementos del mismo array.
  - == y !=
- Ejemplo:

```
p = &a[5];
q = &a[1];
```

`p <= q` tomará el valor 0 (FALSO).

`p >= q` tomará el valor 1 (VERDADERO).

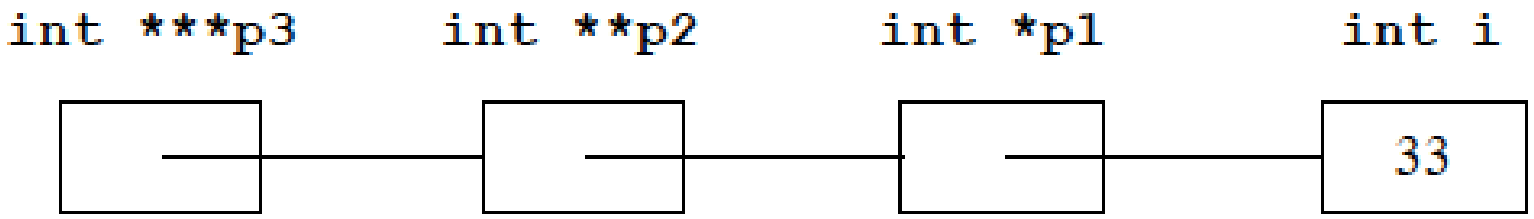
Contenidos

- Aritmética de punteros
- **Punteros a punteros**
- Uso de punteros para el procesamiento de arrays
- Uso del nombre de array como puntero
- Arrays como argumentos
- Punteros y arrays multidimensionales
- Gestión dinámica de memoria

Puntero a puntero

- Un puntero puede referenciar a otro puntero
- Por ejemplo:

```
int i= 33;
int *p1= &i;
int **p2= &p1;
int ***p3= &p2;
```



- Esta característica es útil para la definición de arrays dinámicos multidimensionales.

Contenidos

- Aritmética de punteros
- Punteros a punteros
- **Uso de punteros para el procesamiento de arrays**
- Uso del nombre de array como puntero
- Arrays como argumentos
- Punteros y arrays multidimensionales
- Gestión dinámica de memoria

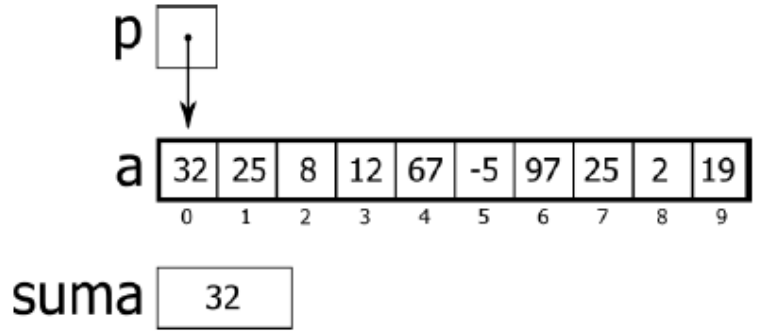
Uso de punteros para el procesamiento de arrays (I)

- La aritmética de punteros permite recorrer un array a través de incrementos repetidos sobre un puntero.
  - El siguiente ejemplo parte de un puntero que inicialmente apunta al elemento a[0]. En cada iteración el puntero se va incrementando y va apuntando al siguiente elemento.
  - El bucle termina al llegar al último elemento:

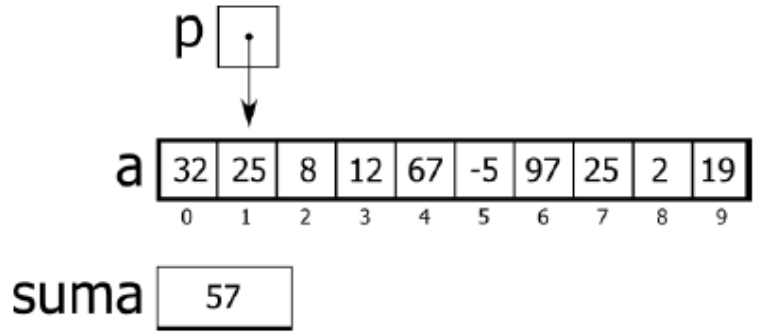
```
#define N 10
...
int a[N], sum, *p;
...
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

Punteros y arrays (II)

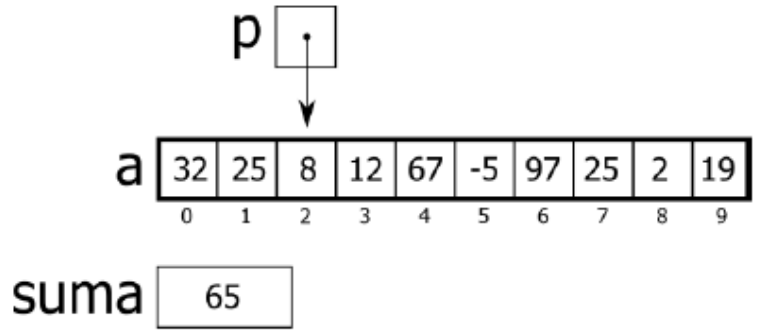
- Al final de la primera iteración:



- Al final de la segunda...



- Al final de la tercera...



Combinando * y ++ (I)

- Algunos programadores de C, a menudo utilizan juntos el operador * y ++.
- El siguiente código ...

```
a[i++] = 32;
```

  - ... almacena 32 en la posición a[i] e incrementa i.
- Si p fuera un puntero que apuntara al elemento a[i], el anterior código se podría haber escrito así:

```
*p++ = 32;
```

  - Como **el operador ++ tiene precedencia sobre el operador ***, el compilador ve la anterior sentencia como:

```
*(p++) = 32;
```

  - El valor de p++ es p, ya que estamos utilizando la versión postfija de ++, el valor de p no se incrementa hasta que p es evaluado. El valor de *(p++) es *p

Combinando * y ++ (II)

- *p++ no es la única forma legal de combiar * y ++:
- La expresión (*p)++ devuelve el valor al que apunta p y acto seguido lo incrementa. El puntero p permanece inalterado.

Expresión	**Significado**
*p++ ó *(p++)	Valor de *p antes del incremento, posterior incremento de p.
(*p)++	Valor de *p antes del incremento, posterior incremento de *p.
*++p ó *(++p)	Incremento de p, valor de *p después del incremento.
++*p ó ++(*p)	Incremento de *p, valor de *p después del incremento.

- Estas cuatro combinaciones suelen aparecer en el programas escritos en C; aunque la más frecuente es *p++, muy últil para la escritura de bucles.

Combinando * y ++ (III)

- En vez de escribir el siguiente bucle para sumar los elementos de un array:

```
for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

- ... se podría haber escrito este otro:

```
p = &a[0];
while (p < &a[N])
   sum += *p++;
```

- El funcionamiento del operador -- en combinación con el operador * es análogo.

Contenidos

- Aritmética de punteros
- Punteros a punteros
- Uso de punteros para el procesamiento de arrays
- **Uso del nombre de array como puntero**
- Arrays como argumentos
- Punteros y arrays multidimensionales
- Gestión dinámica de memoria

Nombres de arrays y punteros (I)

- La aritmética de punteros no es la única interconexión entre arrays y punteros:

El nombre de un array puede utilizarse como un puntero al primer elemento del array.

- Esta relación simplifica la aritmética de punteros y hace a punteros y arrays unas herramientas más versátiles.

Nombres de arrays y punteros (II)

- Supongamos que hemos declarado un array a así:

```
int a [10];
```

- Utilizando **a** como un puntero al primer elemento, podemos modificar a[0] así:

```
*a = 7; // almacena el valor 7 en a[0]
```

- Podemos modificar a[1] a través del puntero a + 1:

```
*(a + 1) = 15; // guarda 15 en a[1]
```

- En general, tanto **a+i** como **&a[i]** representan lo mismo, ambos son punteros al elemento **i** del array **a**:
  - **a+i** es equivalente a **&a[i]**
  - ***(a+i)** es equivalente a **a[i]**

Nombres de arrays y punteros (III)

- El anterior bucle:

```
for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

- ... puede ser sustituido por este otro:

```
for (p = a; p < a + N; p++)
    sum += *p;
```

- Lo que **no se puede hacer es intentar incrementar el puntero que representa el array**...

```
a++;    // MAL
```

... para hacerlo, debemos antes realizar una copia del puntero:

```
p = a;
p++;
```

Contenidos

- Aritmética de punteros
- Punteros a punteros
- Uso de punteros para el procesamiento de arrays
- Uso del nombre de array como puntero
- **Arrays como argumentos**
- Punteros y arrays multidimensionales
- Gestión dinámica de memoria

Arrays como argumentos (I)

- Cuando se pasa un array como argumento, el nombre del array siempre es tratado como un puntero.
- El paso de arrays en C siempre es por referencia.
  - **Nunca por valor**, **nunca se realiza una copia del array.**
- El tiempo requerido para un array como parámetro a una función NO DEPENDE del tamaño del array. No hay penalización para pasar arrays grandes, ya que únicamente se pasa un puntero, no hay copia del array.
- Como parámetro, un array también se puede declarar como puntero. La siguientes declaraciones son equivalentes:

```
int encontrar_maximo(int a[], int n);
int encontrar_maximo(int *a, int n);
```

... para el compilador ambas declaraciones son idénticas.

Arrays como argumentos (II)

- Consecuencias (cont.):
  - A una función que tenga un array como parámetro se le puede pasar una parte del array.
  - La siguiente llamada a la función anterior examinaría 5 elementos a partir del elemento b[5]:

```
max = encontrar_maximo(&b[5], 5);
```

Uso de punteros como nombres de arrays

- Si el nombre de un array se puede utilizar como un puntero... ¿se puede **utilizar un puntero como si fuera un array e indexarlo de la misma forma**?
  - Sí.
- Ejemplo:

```
#define N 10
...
int a[N], i, sum = 0, *p = a;
...
for (i = 0; i < N; i++)
    sum += p[i];
```

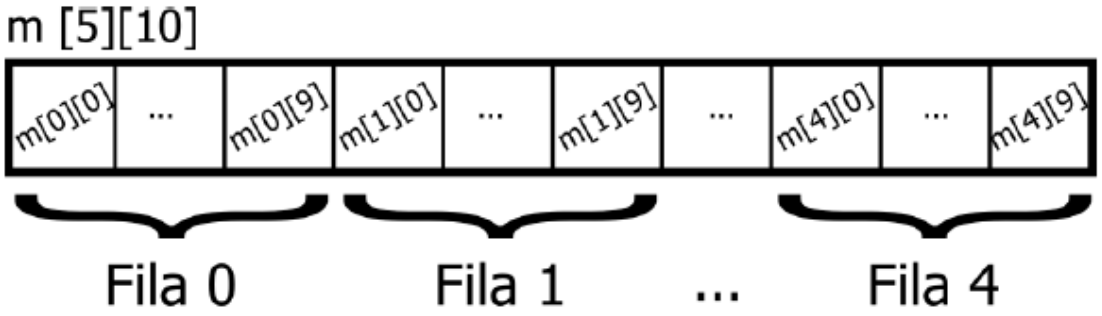
  - Como se verá más adelante, este mecanismo es bastante potente.

Contenidos

- Aritmética de punteros
- Punteros a punteros
- Uso de punteros para el procesamiento de arrays
- Uso del nombre de array como puntero
- Arrays como argumentos
- **Punteros y arrays multidimensionales**
- Gestión dinámica de memoria

Punteros y arrays multidimensionales (I)

- De la misma forma que un puntero puede apuntar a cualquier elemento de un array unidimensional, también puede hacerlo dentro de un array multidimensional.
  - En el caso de array bidimensionales, hay que recordar que C los almacena en memoria de forma secuencial por filas:



Punteros y arrays multidimensionales (II)

- El siguiente código inicializa un array bidimensional:

```
int a[FILAS][COLUMNAS];
...
for (filas = 0; filas < FILAS; filas++)
  for (columnas = 0; columnas < COLUMNAS; columnas++)
     a[filas][columnas] = 0;
...
```

- Pero **a** también se puede ver como un array unidimensional de enteros, tal como está almacenado, de esta forma, se puede recorrer con un puntero así:

```
int *p;
...
for (p=&a[0][0]; p<=&a[FILAS -1][COLUMNAS -1]; p++)
    *p = 0;
```

Punteros y arrays multidimensionales (III)

```
int *p;
...
for (p=&a[0][0]; p<=&a[FILAS -1][COLUMNAS -1]; p++)
    *p = 0;
```

- El bucle se inicia con p apuntando al elemento a[0][0]
- Sucesivas iteraciones hacen que p vaya apuntando a a[0][1], a[0][2], a[0][3], etc.
- Cuando p alcanza a[0][COL -1] (último alemento de a[0]), p vuelve a incrementarse para apuntar a a[1][0], primer elemento de a[1].
- El proceso continúa hasta que p alcanza a[FILAS -1][COLUMNAS -1], último elemento del array.
- No es muy usual recorrer un array multidimensional así, la ganancia que se obtenía en rendimiento antes, ya no se consigue con los compiladores actuales.

Contenidos

- Aritmética de punteros
- Punteros a punteros
- Uso de punteros para el procesamiento de arrays
- Uso del nombre de array como puntero
- Arrays como argumentos
- Punteros y arrays multidimensionales
- **Gestión dinámica de memoria**

Memoria dinámica

- Hasta ahora todos los programas han manejado **memoria estática**, es decir, memoria reservada en tiempo de compilación.
- Sin embargo, la potencia de C reside en el uso de **memoria dinámica**, es decir, memoria reservada en tiempo de ejecución.
- El uso de memoria dinámica debe hacerse con cuidado
  - Toda la memoria reservada debe liberarse antes de finalizar el programa.
  - Existen otros programa (p.ej. Java) donde la liberación es automática.
- El manejo de memoria dinámica se hace usando punteros.

```
int *arrayDimanico;
float **matrizDinamica= NULL;
```

- Herramientas necesarias:
  - Reserva de memoria:
  - Liberar memoria
  - Controlar si un puntero apunta a memoria reservada dinámicamente o no.

Reserva de memoria: **`malloc`**

```
void * malloc (unsigned bytes)
```

- Reserva *tamano* bytes de memoria y devuelve un puntero a la zona reservada.
- Ejemplo uso:

```
arrayDinamico = (int *) malloc (100*sizeof(int));
```

- Devuelve **NULL** si no se ha podido reservar esa cantidad.
- No se inicializa la memoria (**`calloc`** sí lo hace).

Redimensionamiento de memoria: **realloc**

```
void * realloc(void * ptr, int bytes)
```

- Reserva *tamano* bytes de memoria y devuelve un puntero a la zona reservada.
- Ejemplo uso:

```
arrayDinamico = (int *) realloc
(arrayDinamico,100*sizeof(int));
```

- Devuelve NULL si no se ha podido reservar esa cantidad.

Liberar memoria: `free`

```
void free(void *ptr)
```

- Libera una zona de memoria reservada previamente.
- No usar si ptr ==NULL.

Ejemplo: Array dinámico

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

int main(void){
  int * a = NULL;
  int talla, i;
  int * p;

   printf ("Numero de elementos: "); scanf ("%d", &talla);
   a = malloc( talla * sizeof(int) );
   for (i=0, p=a; i<talla; i++, p++)
       *p = i;
   free(a);
   a = NULL;
   return 0;
}
```

El efecto del bucle es inicializar el vector con la secuencia 0, 1, 2. . . El puntero **p** empieza apuntando a donde **a**, o sea, al principio del vector. Con cada autoincremento, **p++**, pasa a apuntar a la siguiente celda. Y la sentencia ***p = i** asigna al lugar apuntado por **p** el valor **i**.

Ejemplo: Matriz dinámica

```
#define <stdio.h>
#define <stdlib.h>
int main(void){
    float ** m = NULL;
    int filas, columnas;
    printf ("Filas: "); scanf ("%d", &filas);
    printf ("Columnas: "); scanf ("%d", &columnas);
    /* reserva de memoria */
    m = malloc(filas * sizeof(float *));
    for (i=0; i<filas; i++)
        m[i] = malloc(columnas * sizeof(float));
    /* trabajo con m[i][j] */
        ...
    /* liberacion de memoria */
    for (i=0; i<filas; i++)
        free(m[i]);
    free(m);
    m = NULL;
    return 0;
}
```

Ejemplo: matriz dinámica de tamaño variable

```
#include <stdlib.h>

int main(void){
   int * a;
   // Se pide espacio para 10 enteros.
   a = malloc(10 * sizeof(int));
   // Ahora se amplía para que quepan 20.
   a = realloc(a, 20 * sizeof(int));
   // Y ahora se reduce a s´olo 5 (los 5 primeros).
   a = realloc(a, 5 * sizeof(int));
   free(a);

   return 0;
}
```

Bibliografía

- King, K.N. **C Programming. A modern approach.** 2ªed. Ed. W.W. Norton & Company. Inc. 2008. Chapter 12.
- Khamtane Ashok**. Programming in C**. Ed. Pearson. 2012.
- Ferraris Llanos, R. D. **Fundamentos de la Informática y Programación en C**. Ed. Paraninfo. 2010.