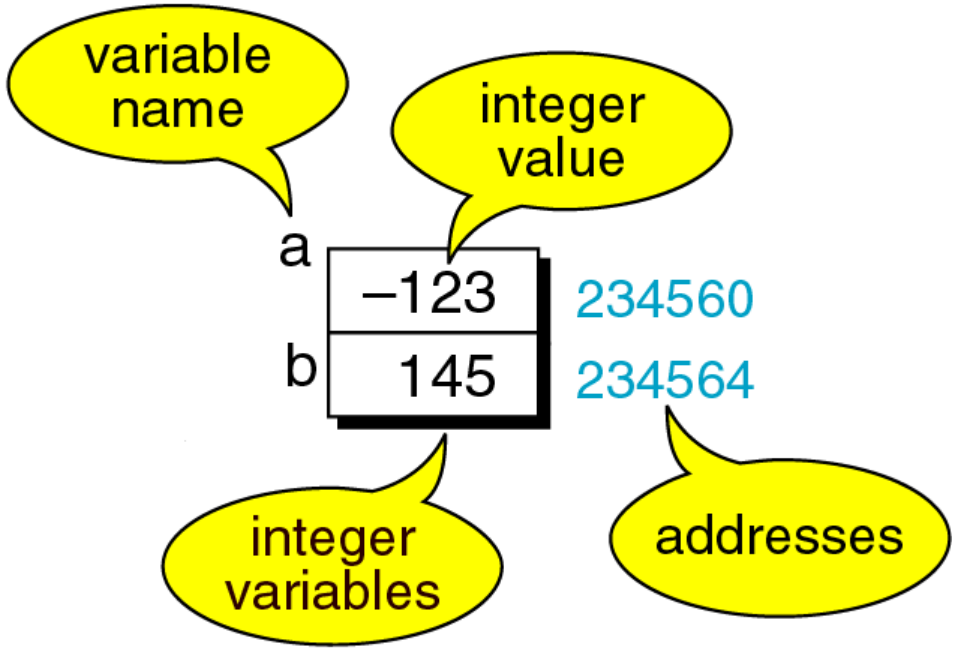
Punteros

Índice

- Variables y direcciones de memoria.
- Punteros – definición, declaración e inicialización.
- Punteros –declaración, asignación y dereferencia.
- Puntero nulo, tipo void.
- Aritmética y comparación de punteros.
- Arrays y punteros.
- Argumentos de main.
- Punteros a estructuras.
- Asignación dinámica de memoria.
- Estructuras autoreferenciadas.

Variables y direcciones de memoria

- Las variables definidas hasta ahora (char, int, float) sólo almacenan *valores* que se modifican mediante el operador asignación. El nombre de la variable representa una *posición de memoria* del ordenador.

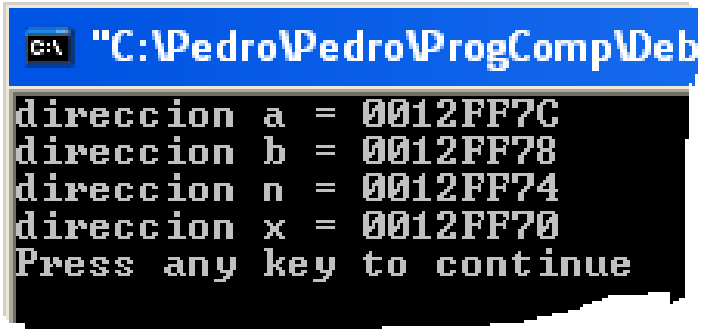


Punteros

- Un *puntero* es un tipo de variable especial que contiene la *dirección de memoria* de otra variable.
- La utilización de punteros permite:
  - Definir arrays de tamaño variable.
  - Que las funciones devuelvan más de un valor.
  - Definir estructuras de datos complejas (p.e. listas, árboles)
- Si una variable **p** contiene la dirección de otra variable **q**, entonces se dice que **p** apunta a **q.**
- Si **q** es una variable en la dirección 100 de la memoria, entonces **p** tendrá el valor 100 (dirección de q).

Punteros – ejemplo operador &

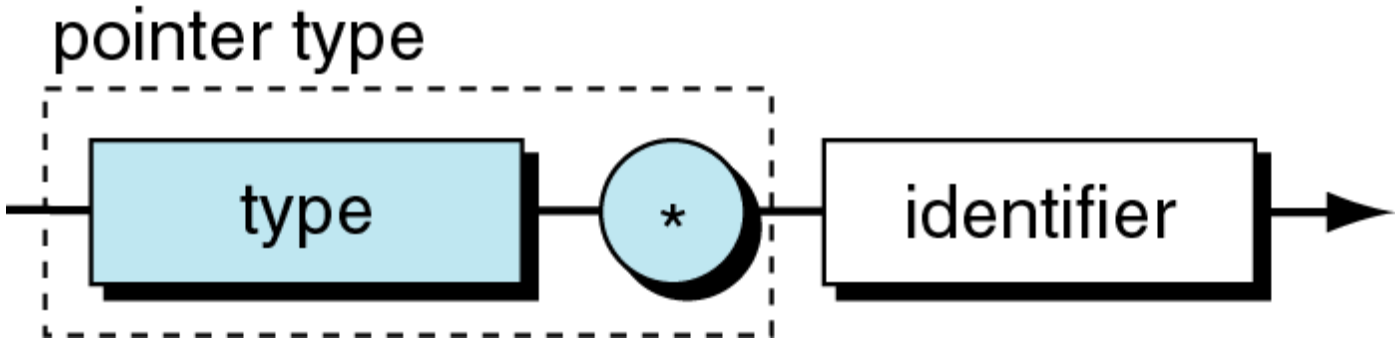
```
/*****************************************************************\
* Programa: pr_oper&.c                                            *
* Descripción: Prog. que imprime direcciones de memoria con &     *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

int main()
{
  char a;
  char b;
  int  n;
  float x;

  printf("direccion a = %p\n", &a);
  printf("direccion b = %p\n", &b);
  printf("direccion n = %p\n", &n);
  printf("direccion x = %p\n", &x);

  return 0;
}
```



Punteros - declaración

- Las variables puntero se declaran usando el operador *indirección* *, llamado también operador *dereferencia*.
- Sintaxis:

```
tipo_dato *nombre_name;
```



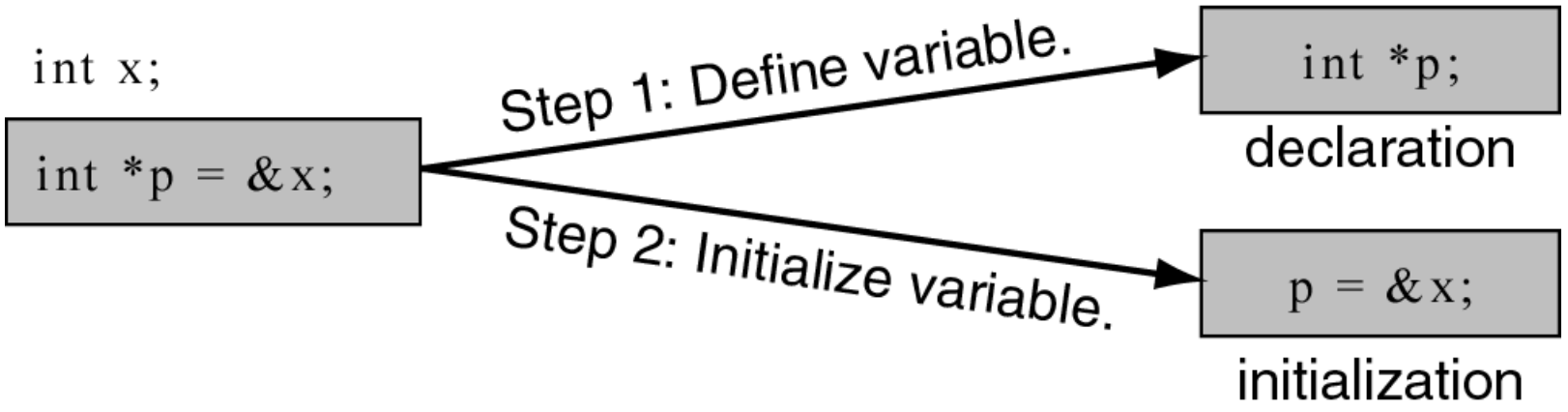
Punteros – operadores & y *

- El operador unario & se utiliza para obtener la *dirección* de memoria de cualquier variable del programa.
  - Si `x` es una variable: **`&x`** es la dirección de memoria donde está almacenada la variable.
- El operador unario *, llamado operador indirección, permite acceder al *valor* por medio del puntero.
  - Si `px` es un puntero (dirección): **`*px`** es el contenido del puntero (el valor almacenado en la dirección).

Punteros – declaración e inicialización

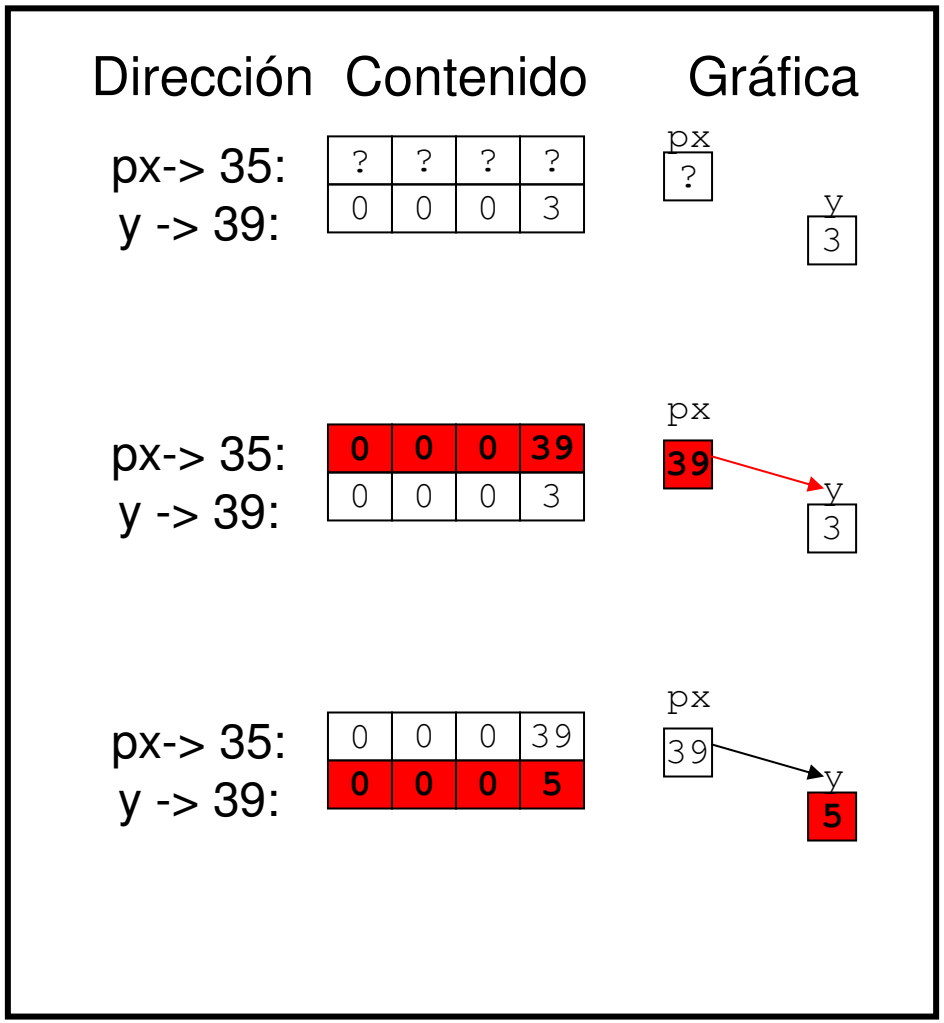
- Se puede declarar e inicializar un puntero en la misma línea.
- Ej:

```
int x;
int *p = &x;
```



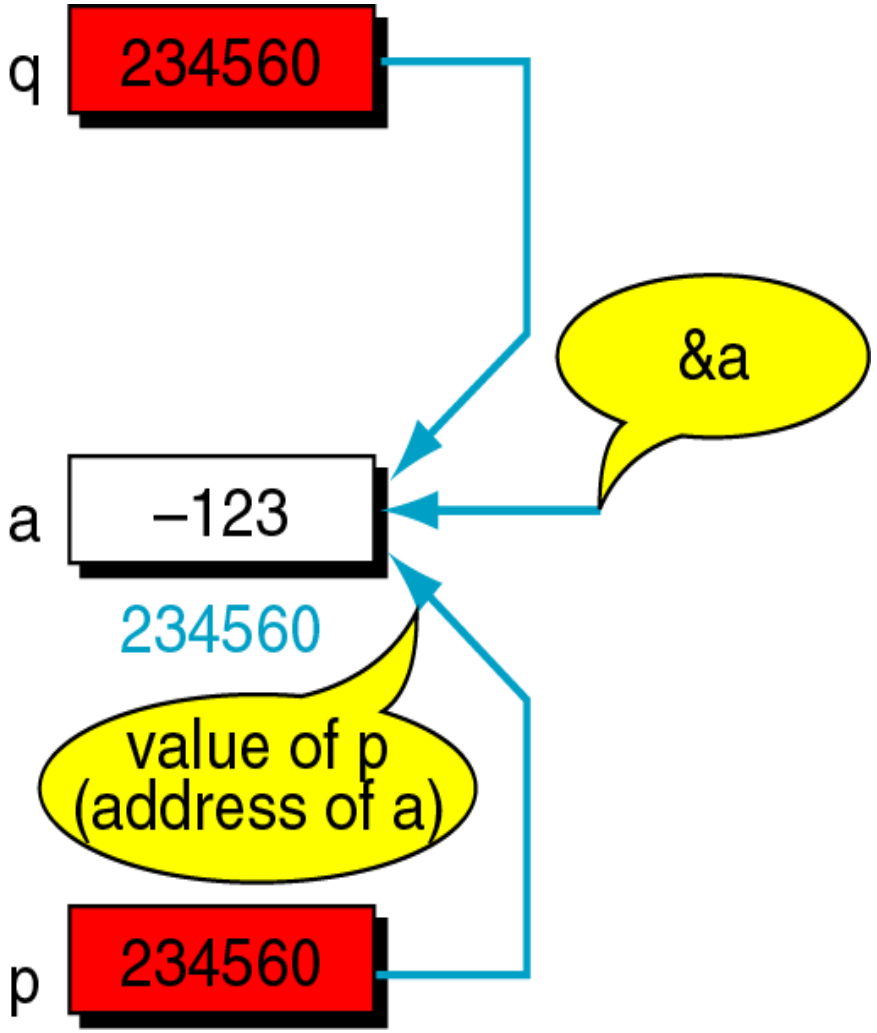
Punteros – declaración, asignación y dereferencia

```
int main()
{
  int *px, y = 3;

  px = &y;
  /* px apunta a y */

  *px = 5;
  /* y vale 5 */
}
```



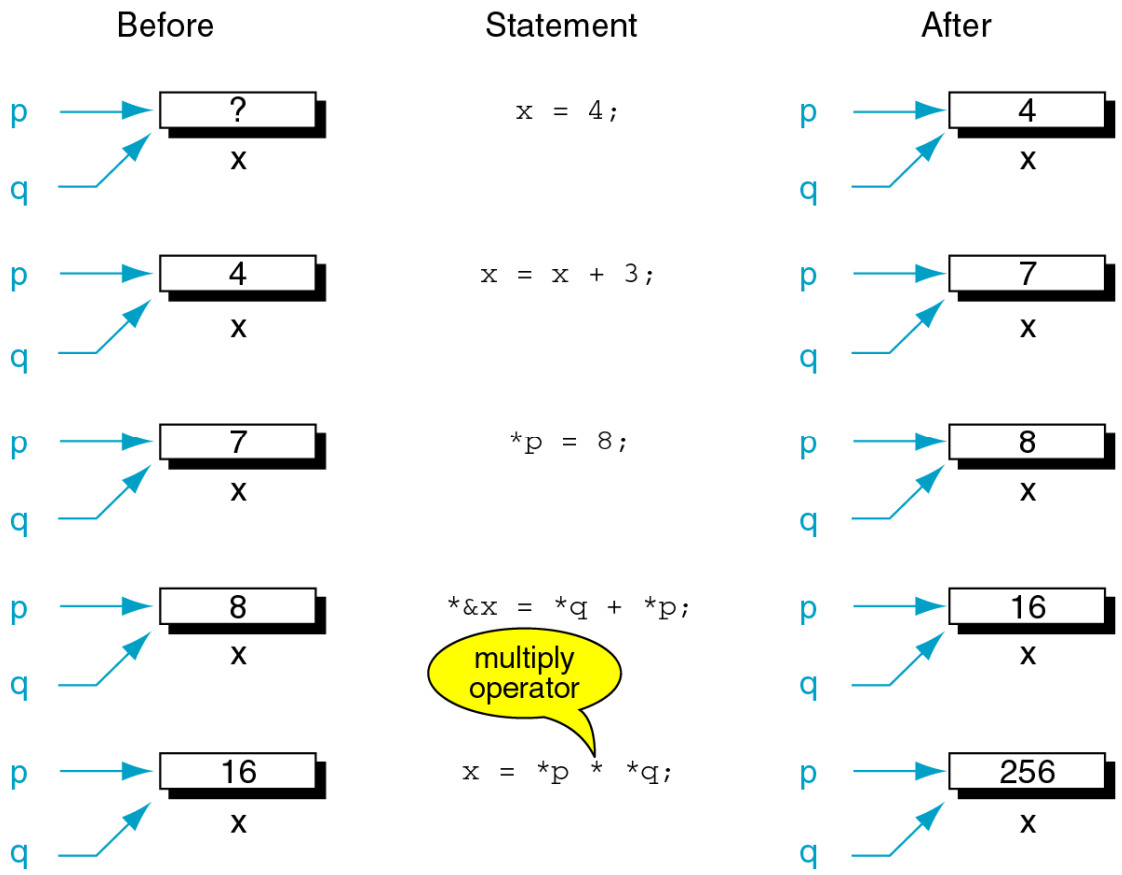
Punteros – declaración, asignación y dereferencia

```
int main()
{
   int *p, *q;
   int a = -123;
   int b;

   p = &a;
   q = p;
   printf("%d %d",*p, *q);
   b = *p;
   printf("%d %d",b, a);
}
```



Punteros – declaración, asignación y dereferencia

```
int main()
{
   int *p, *q;
   int x;

   p = q = &x;
   x = 4;

   x = x + 3;

   *p = 8;

   *&x = *p + *q;

   x = *p * *q;
}
```



Puntero nulo - NULL

- No tiene sentido asignar un valor entero a una variable puntero.
- Una excepción es la asignación de 0, que se utiliza para indicar condiciones especiales (dirección nula de terminación).
- Un puntero cuyo valor es la dirección nula no apunta a nada que puede accederse.
- Es recomendable usar para tal caso la constante **NULL** definida en el cabecero **stdio.h**.

Punteros - void

La declaración de punteros genéricos a direcciones se asocian al tipo `void`.

Declarar una variable (que no sea un puntero) de tipo `void` no tiene sentido.

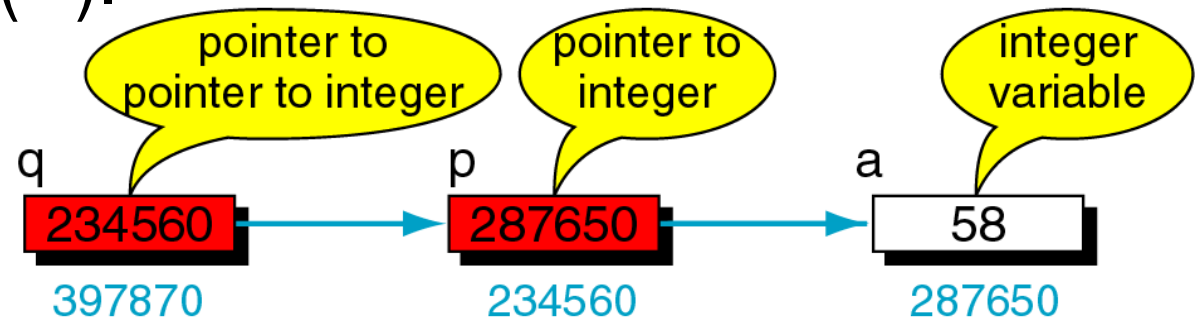
Ejemplo:

```
void *px,v;   /* La variable v
                 está mal
                 declarada */
```

Puntero a puntero

- Un puntero almacena la direccion de un objeto, que puede ser a su vez otro puntero.
- La notacion de puntero a puntero requiere de un doble asterisco (**).

```
int a = 58;
int *p = &a;
int **q = &p;
```




```
printf("%3d", a);
printf("%3d", *p);
printf("%3d", **q);
```

Aritmética y comparación de punteros

Las operaciones soportadas sobre punteros son:

- Suma y resta de valores enteros (+, -, ++ y --).
  - sólo se puede usar enteros en las operaciones. La aritmética se realiza en relación al tipo base. Ejemplo: p++;
    - si p está definido como int *p, p se incrementará en 4 bytes.
    - si p está definido como double *p, p se incrementará en 8 bytes.
    - Cuando se aplica a punteros, ++ significa poner el puntero apuntando al siguiente objeto.
- Comparación y relación (<, >, <=, >=, ==, != y comparación con **`NULL`**).

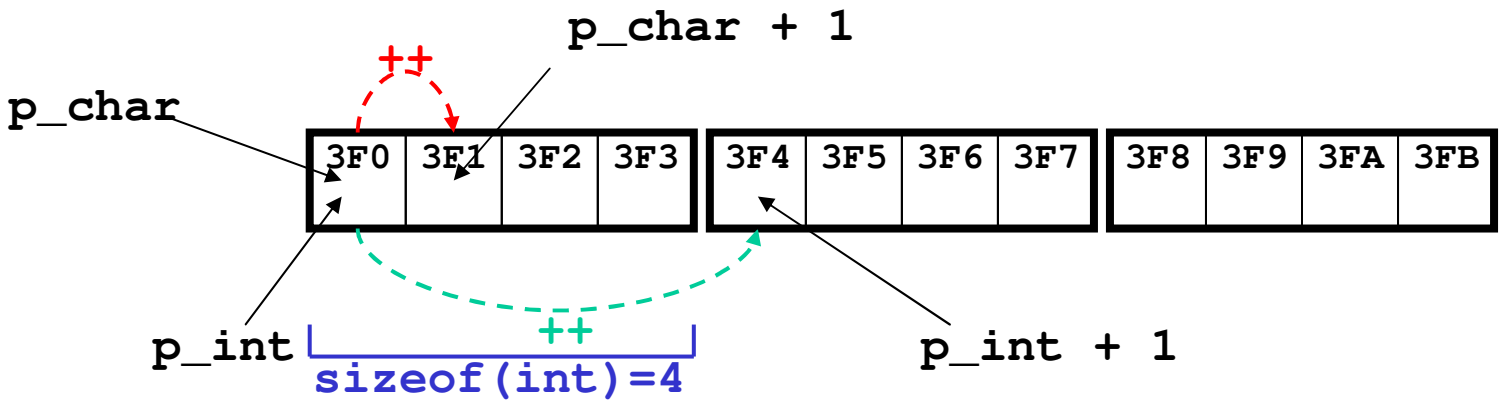
  (p == NULL)

  (p == q) equivale a (*p == *q)

Aritmética de Punteros

Las operaciones de suma o resta sobre punteros modifican el valor del dependiendo del tipo del puntero:

```
int*   p_int; char* p_char; p_int=p_char;
p_int++;  /* Suma sizeof(int) */
p_char++; /* Suma sizeof(char) */
```

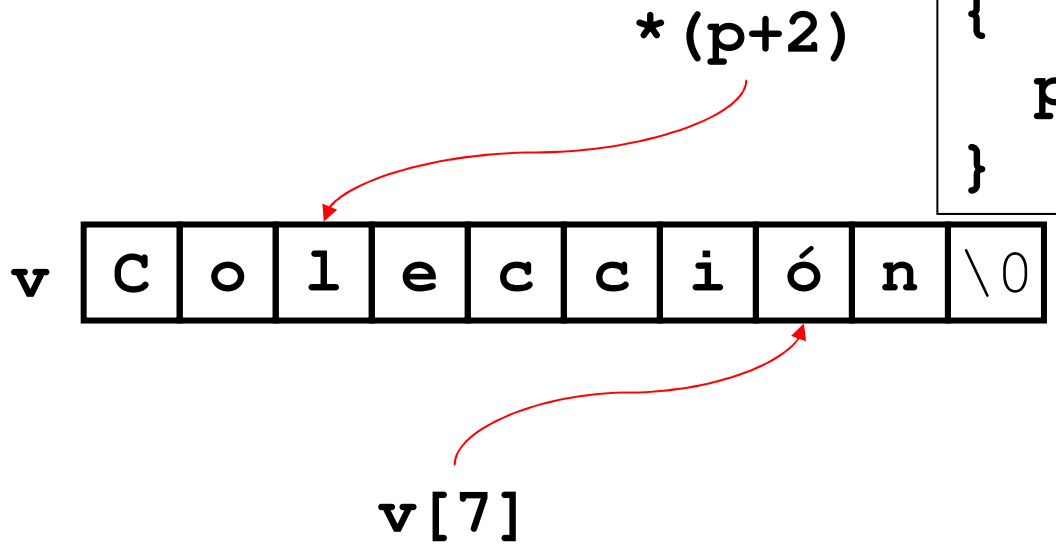


Aritmética de Punteros

- Las variables de tipo puntero soportan ciertas operaciones aritméticas.

```
char v[]="Colección";
char *p=v;
```

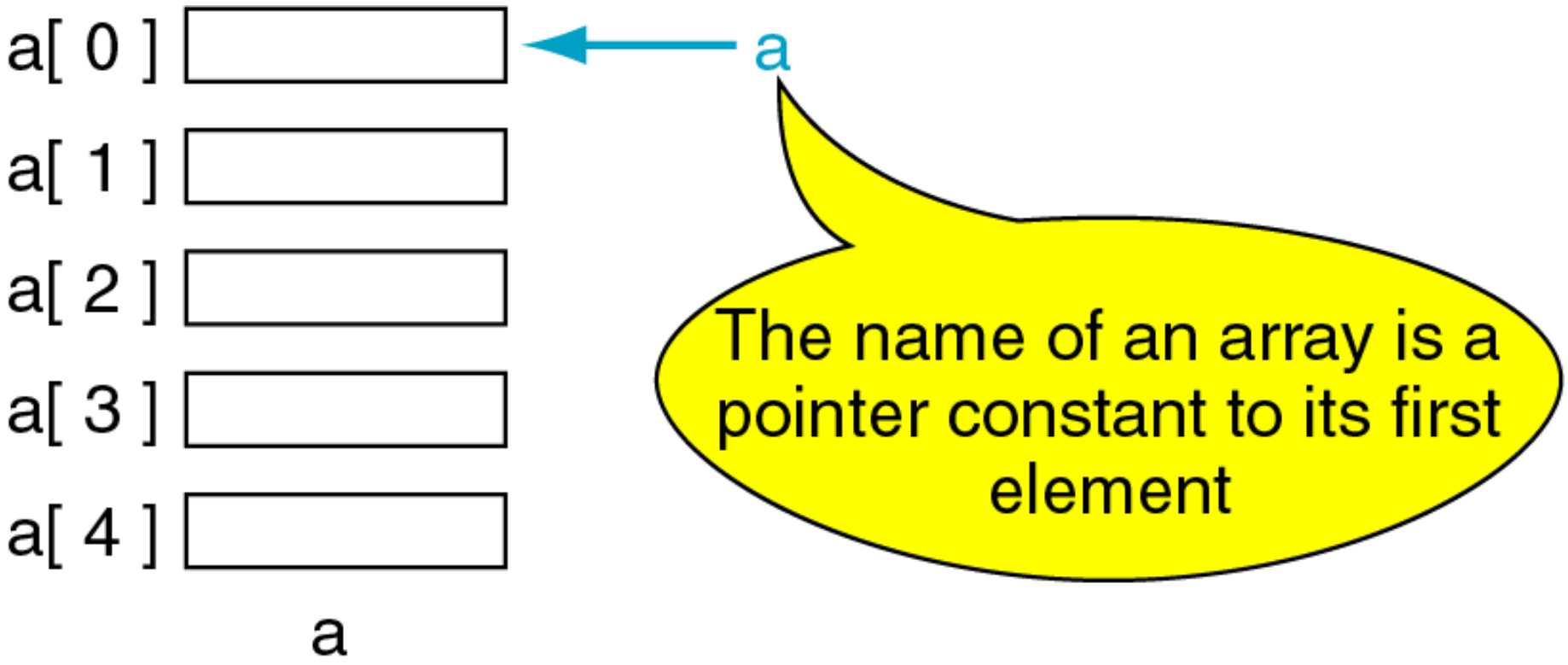
```
for(p=v; *p; p++)
{
  printf("%c",*p)
}
```



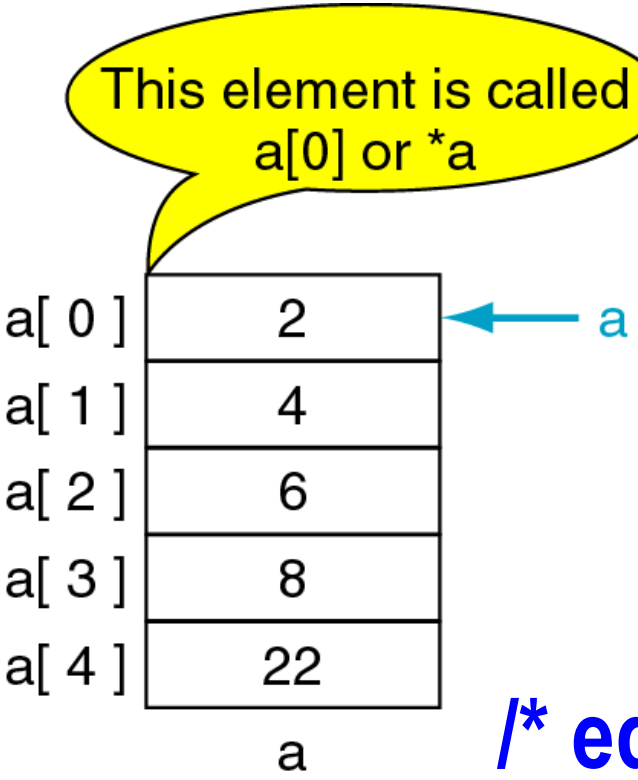
Arrays y punteros

- En C existe una relación estrecha entre arrays y punteros.
- Las variables arrays y punteros son esencialmente lo mismo en C, excepto que:
  - Los punteros son *variables*, por tanto puede modificarse la posición a la que apunta.
  - Las variables arrays son fijas o *constantes*, en tanto que sólo pueden apuntar a un array.
- Cualquier operación que puede realizarse usando índices de un array puede realizarse usando punteros.

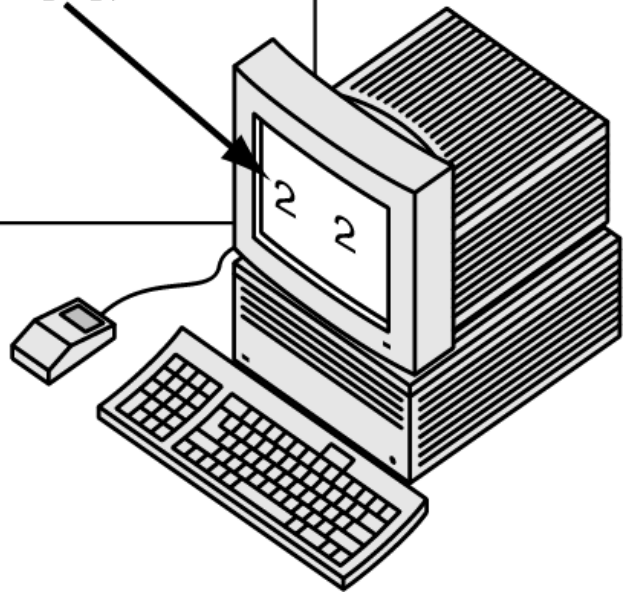
Arrays y punteros



Equivalencias entre arrays y punteros

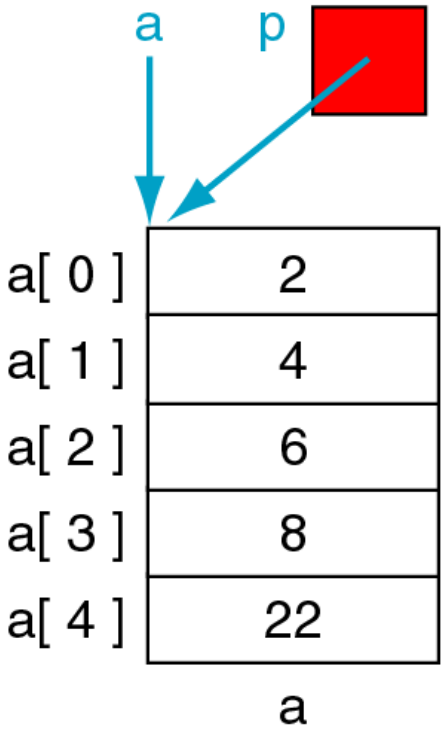



```
#include <stdio.h>
int main (void)
{
  int a[5] = {2,4,6,8,22};
  printf("%d %d", *a, a[0]);

  return 0;
} /* main */
```

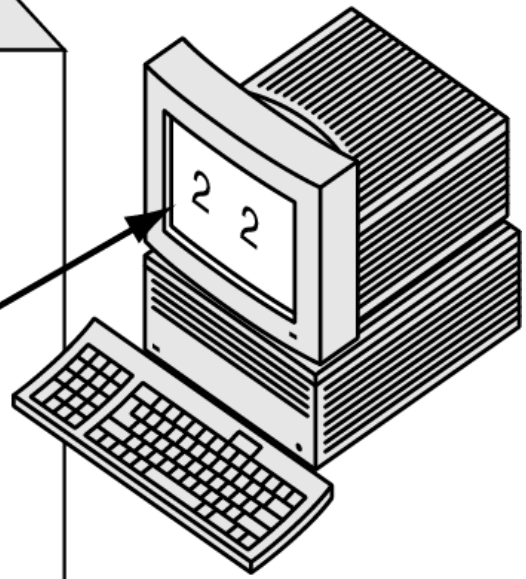



```
/* equivalencias */
ptr = array_a;
ptr = &array_a[0];
```

Equivalencias entre arrays y punteros

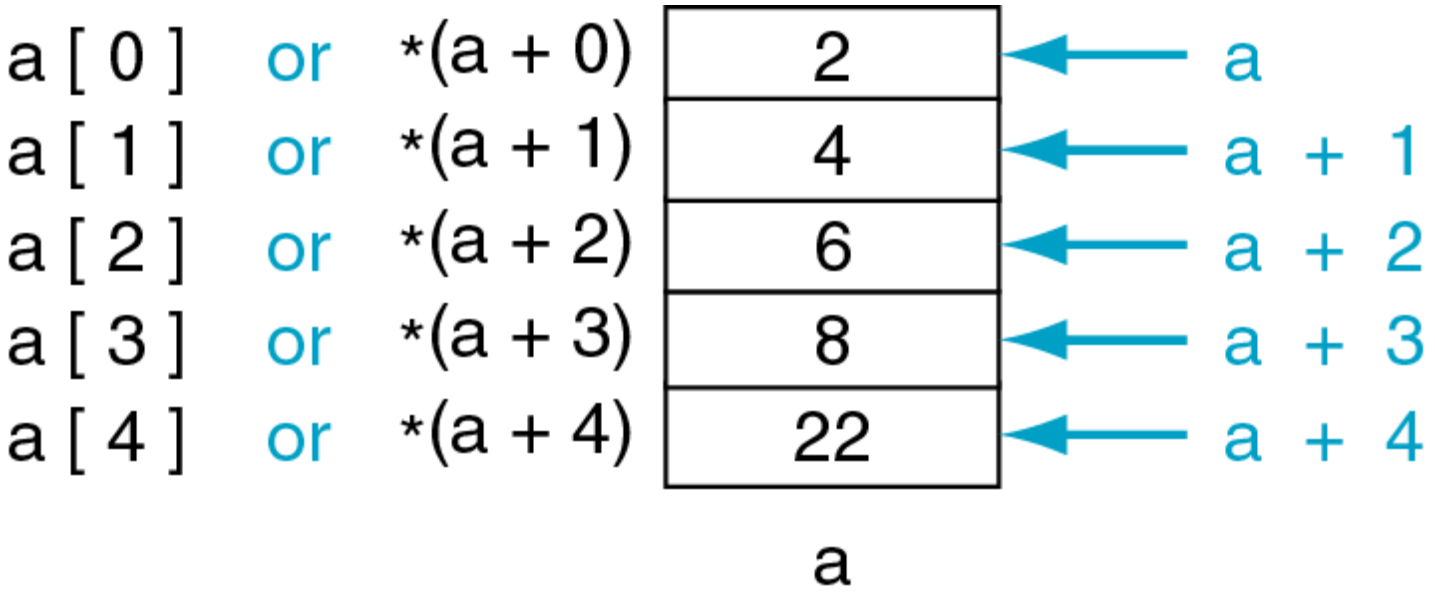



```
#include <stdio.h>
int main (void)
{
int a[5] = {2, 4, 6, 8, 22};
int *p   =  a;
int  i   = 0;
  ...
  printf("%d %d\n", a[i], *p);
  ...
  return 0;
} /* main */
```



Equivalencias entre arrays y punteros

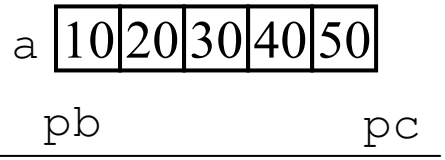
- Una referencia a a[i] también puede escribirse como *(a+i) .
- C convierte la indexación de un array a la notación de puntero durante la compilación.



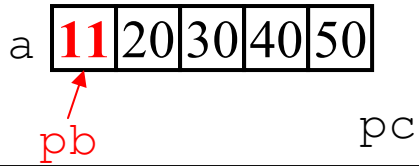
Equivalencias entre arrays y punteros

El identificador de una variable array tiene el valor de la dirección de comienzo del mismo. Por lo tanto, su valor puede usarse como un puntero.

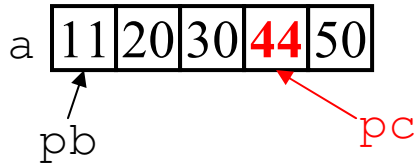
```
int *pb,*pc;
int a[5]={10,20,30,40,50};
```




```
pb=a;
*pb=11;
```




```
pc=&a[3];
*pc=44;
```



Punteros y arrays multidimensionales

- Un array bidimensional es una colección (array) de arrays unidimensionales.
- Se puede definir un array bidimensional como un puntero a un grupo de arrays unidimensionales, así:

  tipo_dato (*ptvar)[ind2]; ≡ tipo_dato array[ind1][ind2];

  ```
  int (*x)[20]; ≡ int x[10][20];
  ```

- Generalizando para arrays multidimensionales:

  tipo_dato (*ptvar)[ind2] [ind3] [ind4]…[ind*n*];

```
float (*t)[20][30]; ≡ float t[10][20][30];
```

Punteros y arrays multidimensionales

- Para acceder a un elemento individual de un array multidimensional se usa la aplicación repetida del operador indirección.
- Suponiendo que x es un array bidimensional de 10 filas y 20 columnas, para acceder al elemento 2, 5:

```
x[2][5] ≡ *(*(x + 2) + 5)
```

- Generalizando para arrays multidimensionales:

tipo_dato (*ptvar)[ind2] [ind3] [ind4]…[ind*n*]

float (*t)[20][30]; ≡ float t[10][20][30];

Arrays de punteros

- Otra forma de expresar un array multidimensional es mediante arrays de punteros.

  tipo_dato *array[ind1]; ≡ tipo_dato array[ind1][ind2];

- Generalizando para arrays multidimensionales:

  tipo_dato *array [ind1] [ind2] [ind3]…[ind*n-1*];

  tipo_dato array [ind1] [ind2] [ind3]…[ind*n*];

- Para acceder a un elemento individual:

  Si x es un array de int con 10 filas y 20 columnas.

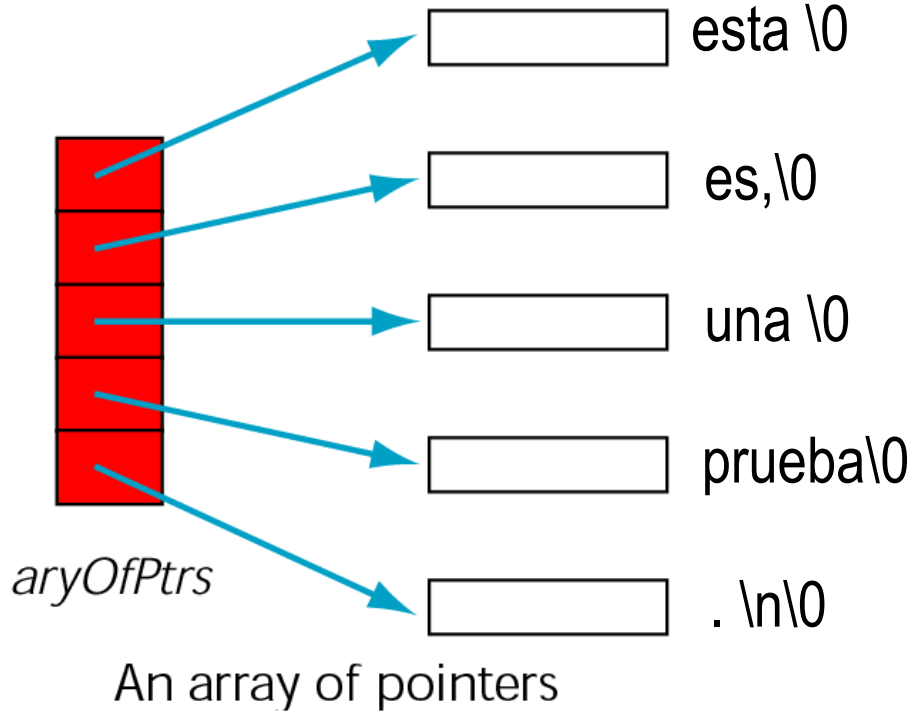
  int *x[10];

  x[2][5] ≡ *(x[2] + 5)

Arrays de punteros

- Son muy usados para almacenar cadenas de caracteres. No se declara el tamaño de la cadena.

```
char *palabras[ ] = { "esta ”,“es,”,“ una ","prueba”,”. \n"};
```



palabras[2][1] = ‘n’

((palabras+1)+2) = ‘,’

Argumentos de `main`

- La función principal **`main`** puede recibir argumentos que permiten acceder a los parámetros con los que es llamado el ejecutable.
- Dentro del paréntesis de main se especifican dos argumentos:
  - `int argc` : Número de parámetros.
  - `char* argv[]` : Parámetros del ejecutable.

  con lo que la definición de main queda así:

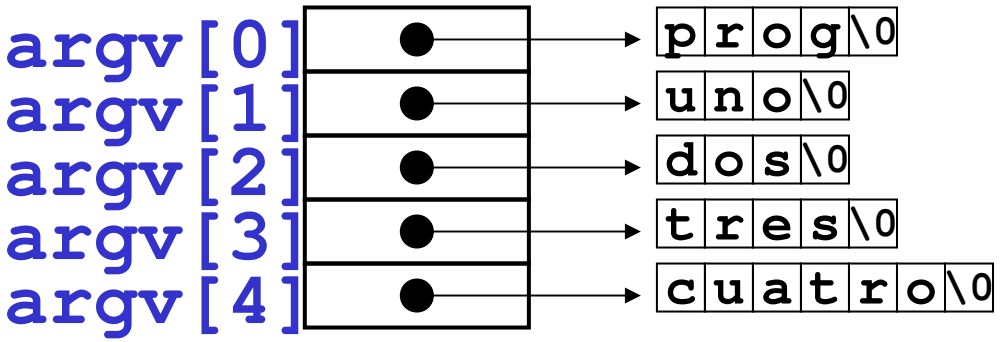
```
int main(int argc, char* argv[])
```

Argumentos de `main`

```
> cl prog.c -o prog.exe
$ prog uno dos tres cuatro
```

```
int main(int argc, char* argv[])
```

`argc=5` (incluye el comando)



Ejemplo: imprime argumentos de main

```
/*****************************************************************\
* Programa: arg_main.c                                          *
* Descripción: Prog. que imprime el comando y los argumentos de main     *
* Uso: >progcomp arg1 arg2 arg3                                 *
* Autor: Pedro Corcuera                                         *
* Revisión: 1.0 2/02/2008                                       *
\*****************************************************************/
#include <stdio.h>

int main(int argc, char *argv[])
{
   int i=0;
   printf("Ejecutable: %s \tNumero de argumentos: %d\n",argv[0], argc);

   for(i = 1; i < argc;i++)
        printf("Argumento[%d]: %s\n",i,argv[i]);

   return 0;
}
```

Punteros a estructuras

- Son muy frecuentes en C. Se declaran de la misma forma que un puntero a una variable ordinaria. Ej.:

  struct alumno *palumno;

  ALUMNO *palumno; /* usando typedef */

- Para acceder a un campo de la estructura:

  prom_nota = (*palumno).nota

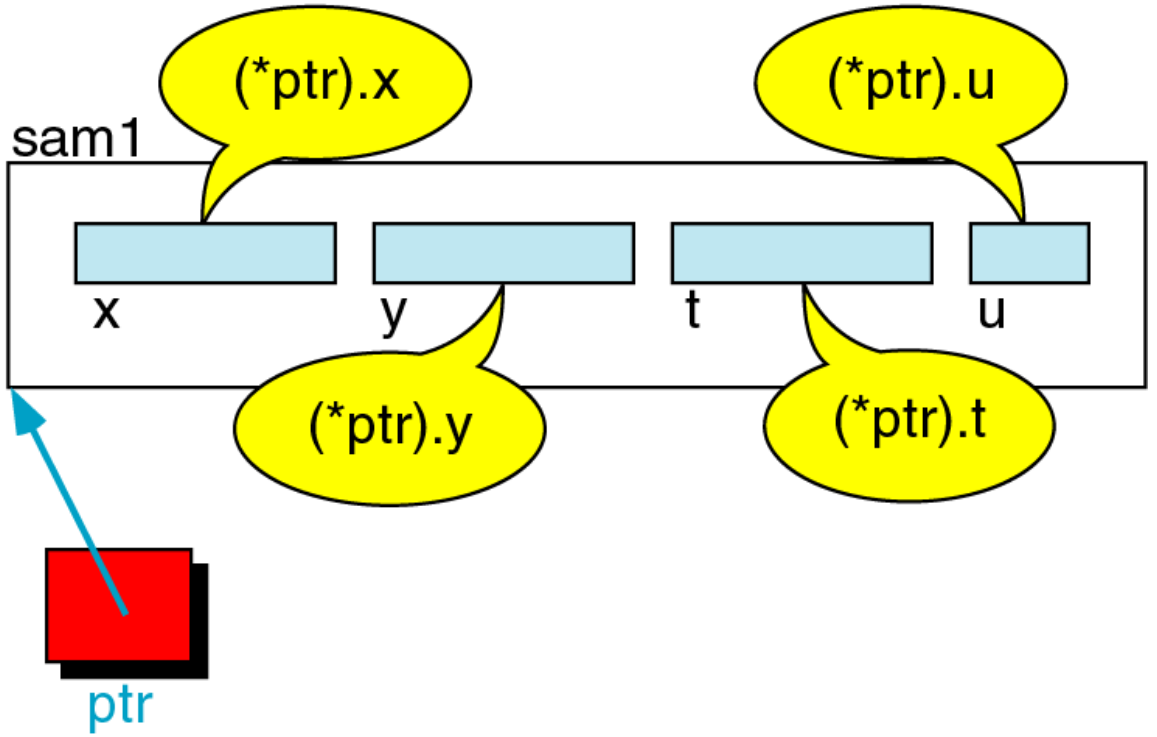
  los paréntesis son estrictamente necesarios.

  Hay un operador (**->**) alternativo para este propósito:

  palumno -> nota

Punteros a estructuras

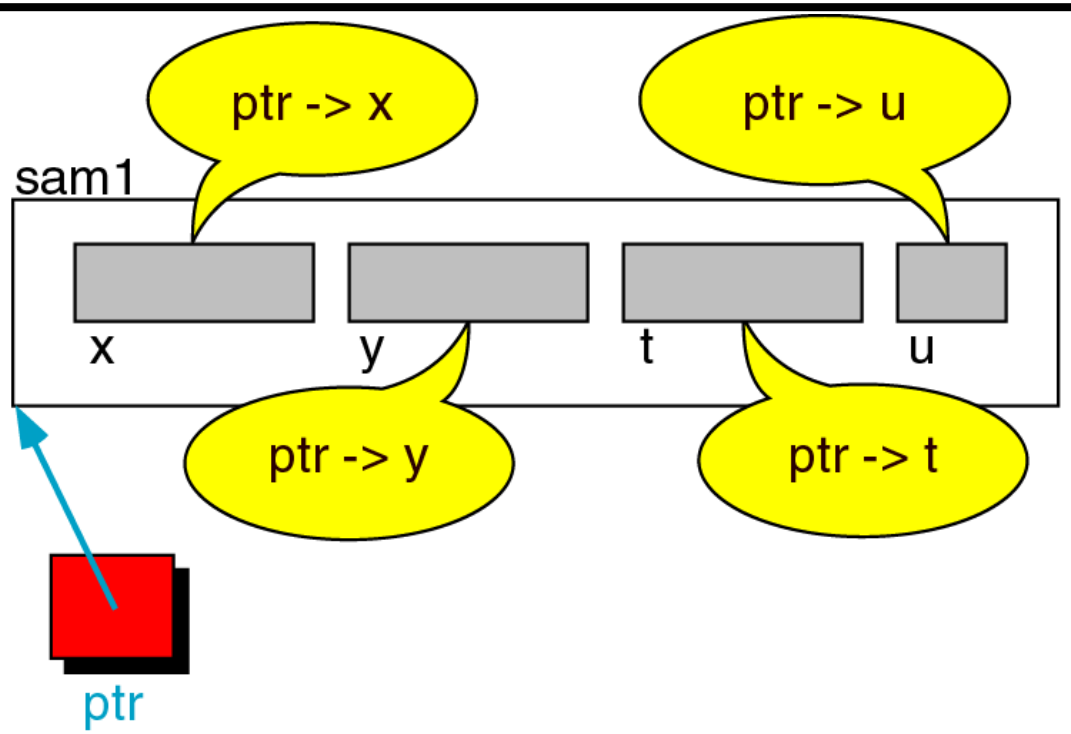
```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
   ...
   SAMPLE   sam1;
   SAMPLE  *ptr;
   ...
   ptr = &sam1;
   ...
```

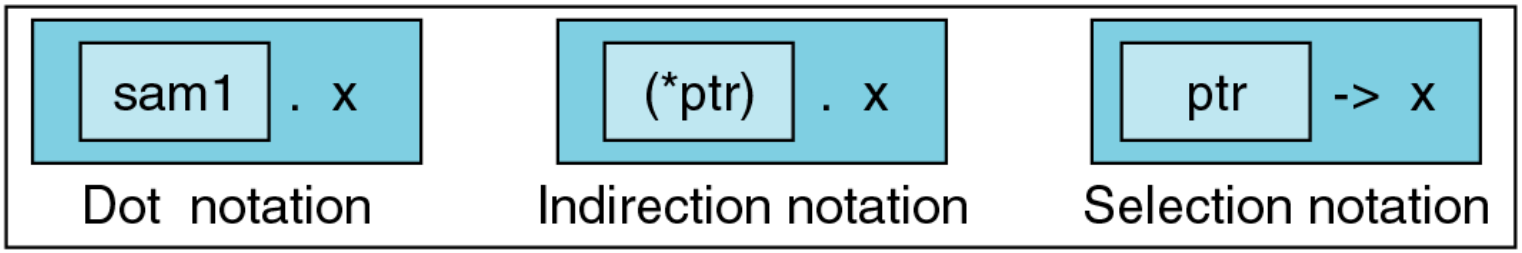


Two ways to reference x

Punteros a estructuras

```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
  ...
  SAMPLE  sam1;
  SAMPLE  *ptr;
  ...
  ptr = &sam1;
  ...
```



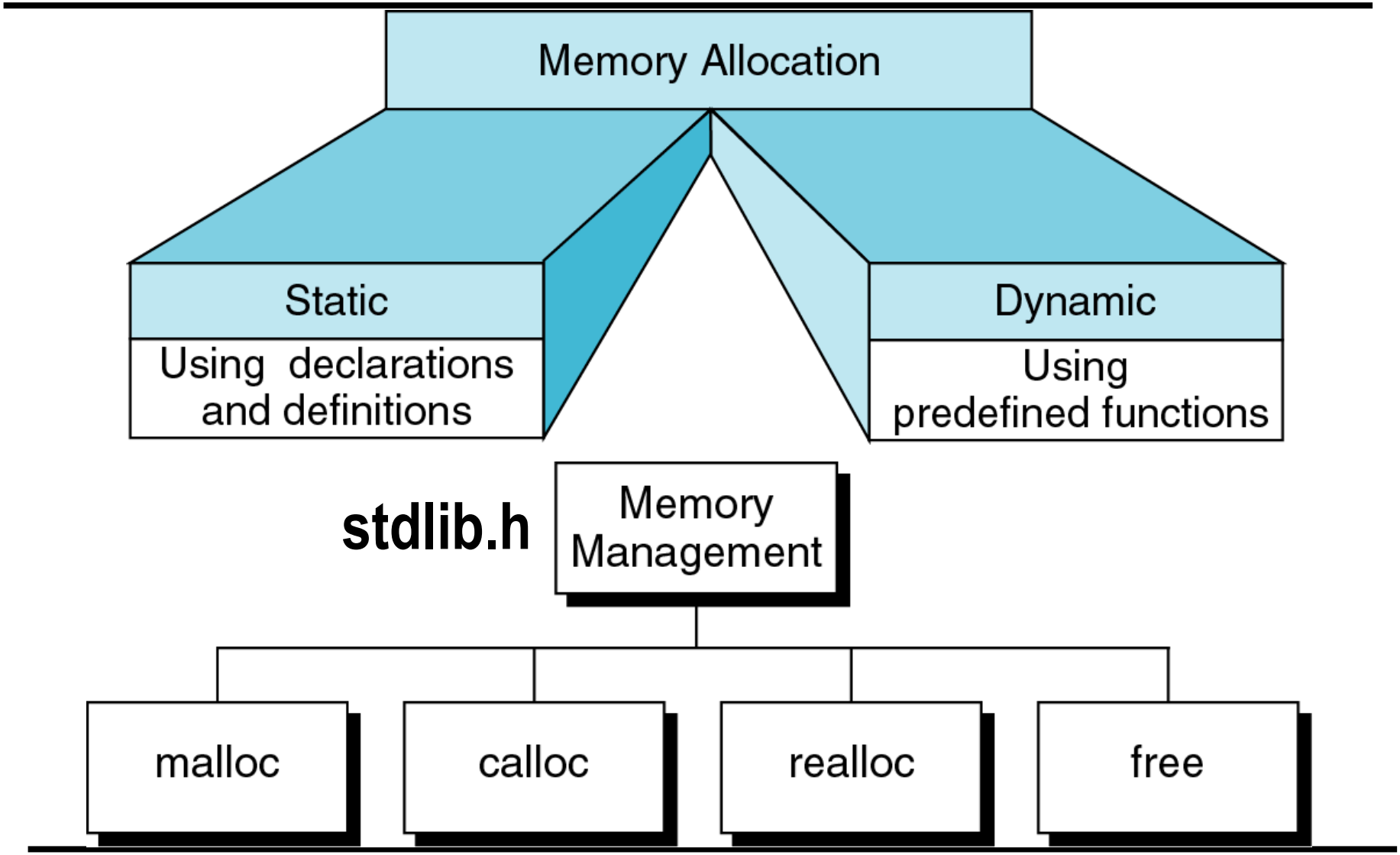


three ways to reference the field **x**

Asignación dinámica de memoria

- Además de la reserva de espacio estática (cuando se declara una variable), es posible reservar memoria de forma dinámica.
- Hay funciones de gestión de memoria dinámica (stdlib.h):
  - **void *malloc(size_t)**: Reserva memoria dinámica.
  - **void *calloc(size_t)**: Reserva memoria dinámica.
  - **void *realloc(void *,size_t)**: Ajusta el espacio de memoria dinámica.
  - **free(void *)**: Libera memoria dinámica.

Memoria dinámica

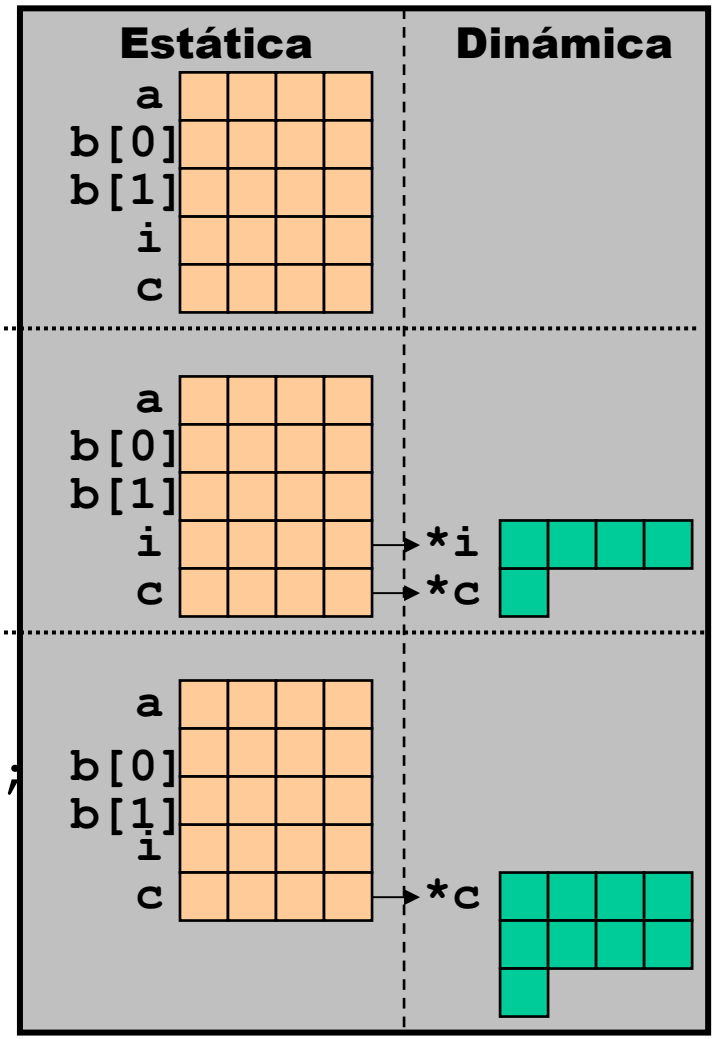


Memoria Dinámica - uso

```
#include <stdlib.h>
int     a,b[2];
int *i;
char *c;
```

```
i=(int *)malloc(sizeof(int));
c=(char *)malloc(sizeof(char));
```

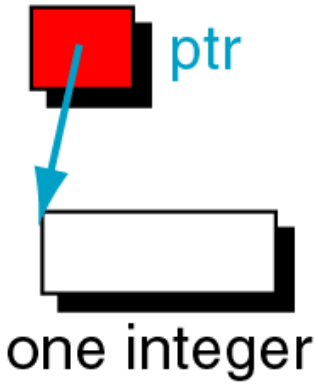
```
free(i);
c=(char *)realloc(c,sizeof(char)*9);
```



Memoria Dinámica – uso malloc

`void *malloc(size_t tamaño_bloque)`

devuelve NULL si no hay suficiente memoria

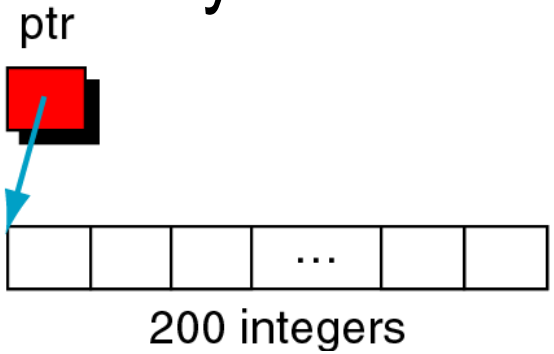



```
ptr = (int *)malloc(sizeof(int));
if ( ptr == NULL)
  /* No hay memoria */
  printf ("ERROR. No hay memoria\n");
else
  …. /* Memoria disponible */
```

Memoria Dinámica – uso calloc

`void *calloc(size_t nelem, size_t elsize)`

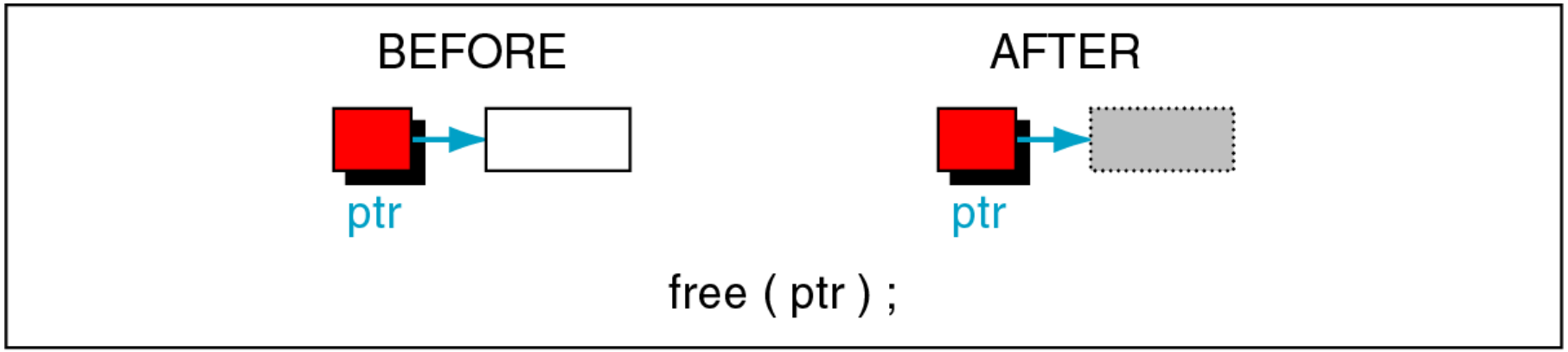
devuelve NULL si no hay suficiente memoria

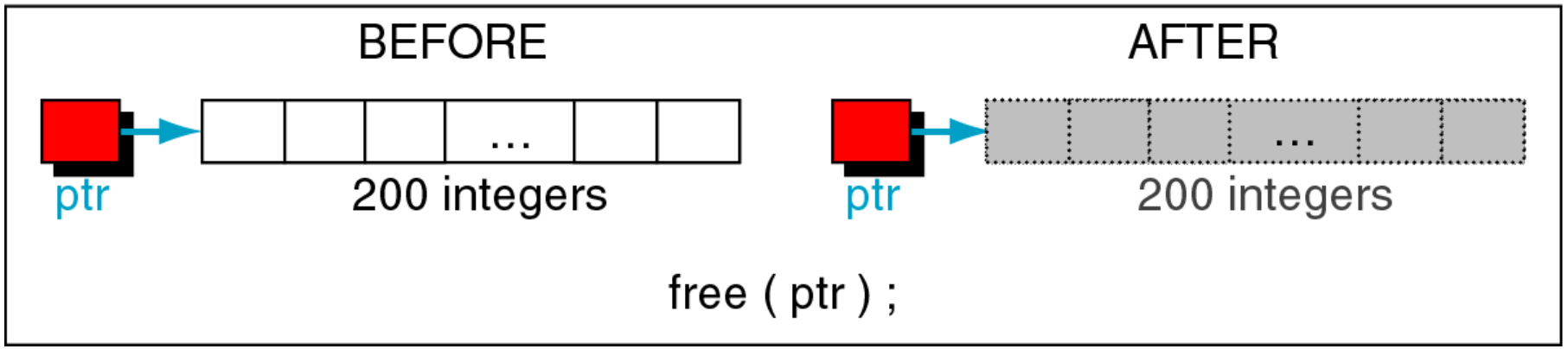



```
ptr = (int *)calloc(200, sizeof(int));
if ( ptr == NULL)
  /* No hay memoria */
  printf ("ERROR. No hay memoria\n");
else
  …. /* Memoria disponible */
```

Memoria Dinámica – uso free

```
void free(void *puntero_al_bloque)
```





Ejemplo: array dinámico

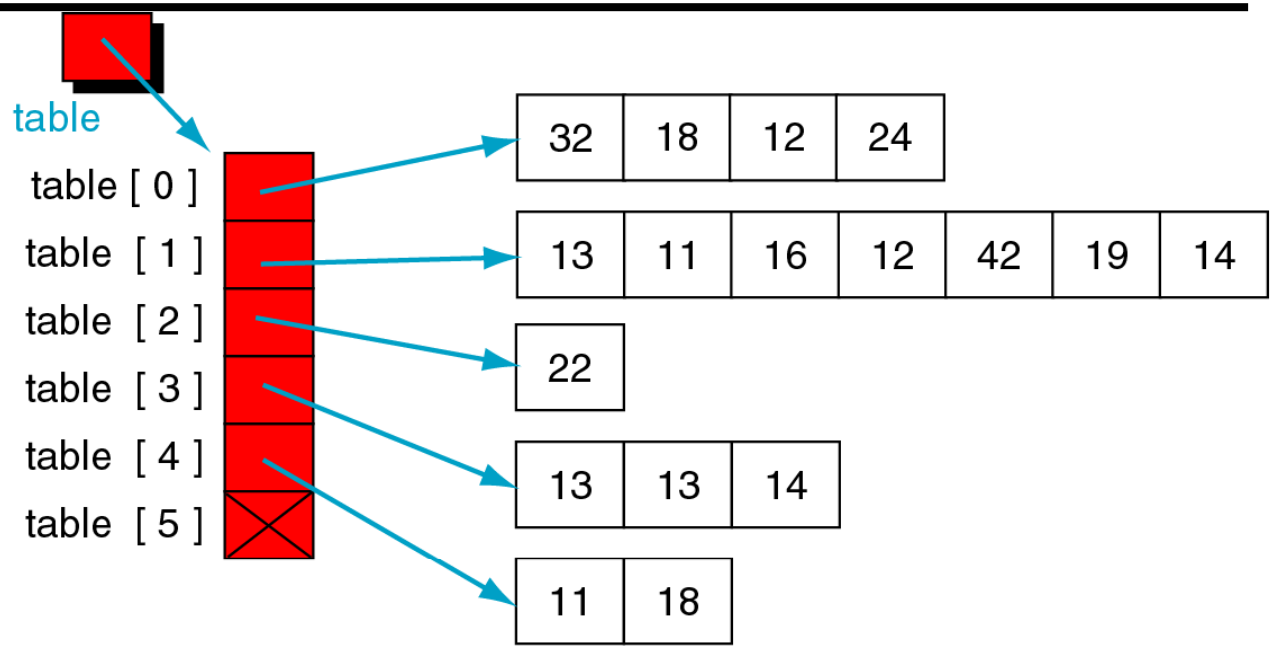
```
/*****************************************************************\
* Programa: array_dinamico.c                                      *
* Descripción: Prog. que crea una array dinamico y genera otro con los   *
*              elementos pares del primer array y lo imprime      *
\*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int dim_usu;    /* dimension del vector del usuario */
    int dim_par;    /* dimensión del vector de elementos pares */
    int n;          /* indice para los for */
    int m;          /* indice para recorrer arrya de pares */
    int *pvec_usu; /* puntero al vector introducido por el usuario */
    int *pvec_par; /* puntero al vector elementos pares (dinamico) */

    printf("Introduzca la dimension del vector: ");
    scanf(" %d", &dim_usu);

    pvec_usu = (int *) calloc( dim_usu, sizeof(int)); /*Asignar memoria vect. usuario
    */
    /*pvec_usu = (int *) malloc( dim_usu*sizeof(int));*/
    if (pvec_usu == NULL) {  /* si no hay memoria */
          printf("Error: no hay memoria para un vector de %d elementos\n", dim_usu);
    }
```

Ejemplo: array dinámico

```
    else
    {
          for (n = 0; n < dim_usu; n++) /* pedir elementos del vector */
          { printf("Elemento %d = ", n); scanf("%d", &(pvec_usu[n]));}
          dim_par = 0;
          for (n = 0; n < dim_usu; n++)
             if ((pvec_usu[n] % 2) == 0) dim_par++;
          pvec_par = (int *) calloc( dim_par, sizeof(int));
          if (pvec_par == NULL) {  /* si no hay memoria */
             printf("Error: no hay memoria para un vector de %d elementos\n",
    dim_par);
          }
          else
          { /* se copian los elementos pares */
                    m = 0;
                    for (n = 0; n < dim_usu ; n++)
                        if ((pvec_usu[n] % 2) == 0) { pvec_par[m] = pvec_usu[n]; m++;}
                    printf("\n----------------------------\n");
                    for (n = 0; n < dim_par ; n++)
                              printf("Elemento par %d = %d \n", n, pvec_par[n]);
          }
          free (pvec_par);
    }
    free (pvec_usu);
}
```

Matriz con número de columnas diferentes

- Se representa como un array de arrays (puntero a puntero) dinámico.

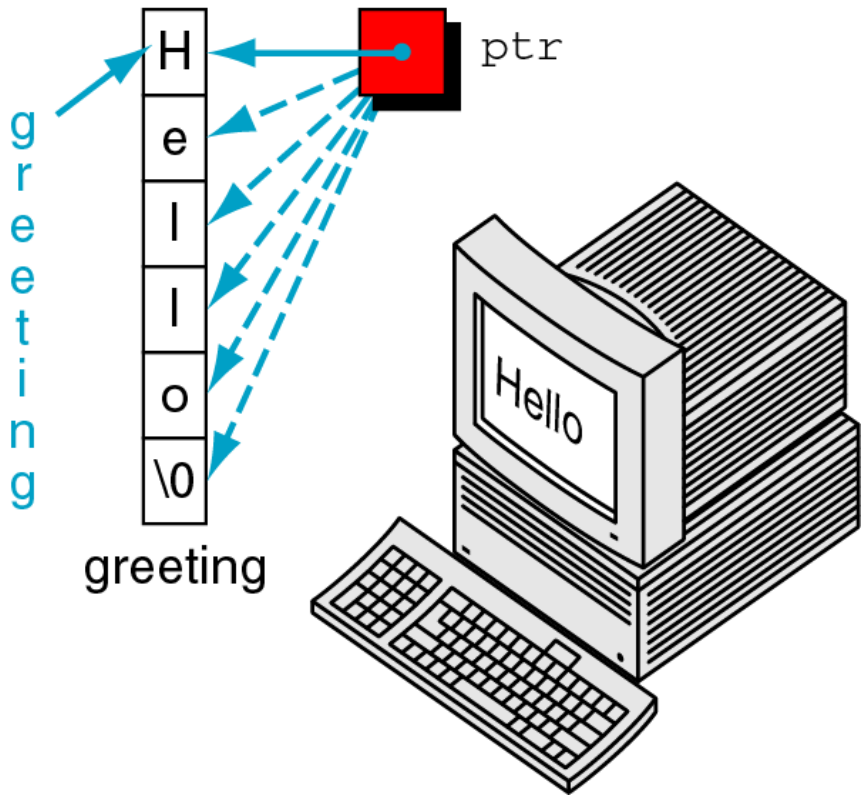



```
table = (int **)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

Punteros a cadenas de caracteres

- Utiles en recorridos de cadenas de caracteres.

```
{ /* Printing Strings */
  char  greeting[] = "Hello";
  char *ptr;

  ptr = greeting;
  while (*ptr != '\0')
      {

      printf( "%c", *ptr);
      ptr++;
     } /* while */
  print("\n");
} /* Printing Strings */
```



Estructuras autoreferenciadas

- Para construir estructuras de datos más potentes es necesario incluir *dentro* de una estructura un campo que sea un puntero a la misma estructura.

```
struct alumno
{
   char dni[10];
   char nombre[100];
   struct fecha fnac;
   float notas[10];
   struct alumno *palumno;
};
```