Sesión 9: Memoria dinámica

Programación para Sistemas

Ángel Herranz

Curso 2023-2024

Recordatorio *arrays*

- Variable global o local, se necesita la longitud y debe ser constante:

$$T\ a[N];$$
$$T\ a[] = \{\ e_0,\ e_1,\ \ldots,\ e_{n-1}\ \};$$

- Asignación prohibida: $a = b$ ⛔
- Sin longitud: **sizeof**(a)/**sizeof**$(a[0])$ no funciona en general
- Argumento array: C sólo pasa la dirección de memoria del primer elemento

```
void f(T a[]) { ... }
```

- Por convención se pasa la longitud como argumento

```
void f(T a[], size_t n) { ... }
```

Recordatorio *strings*

- C no tiene *strings*, los *strings* en C son arrays de **char**

```
char s[] = "mundo";
```

- Por convención: los *strings* son NULL-*terminated* (equivalente a lo anterior)

```
char s[] = {'m', 'u', 'n', 'd', 'o', '\0'};
```

- Por eso `sizeof(s)/sizeof(s[0]) == 6` y `strlen(s) == 5`

⏭ La forma habitual de escribir el tipo es

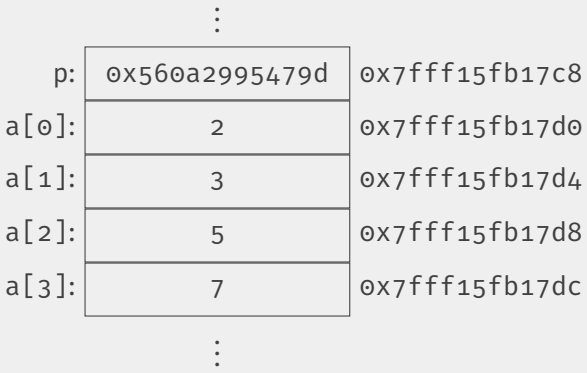
```
char *s ≡ char s[]
```

⚠

En el capítulo de hoy...

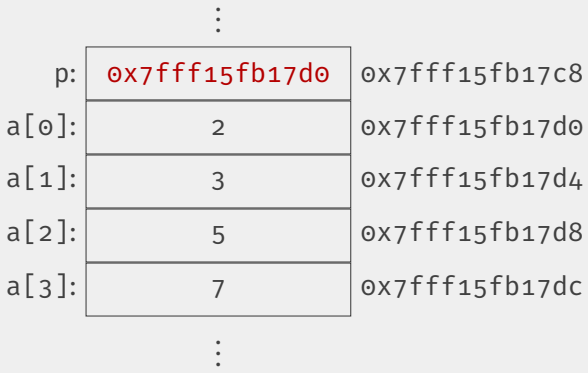
Estrecha relación entre punteros y arrays I

```
int *p;
int a[] = ...;
```



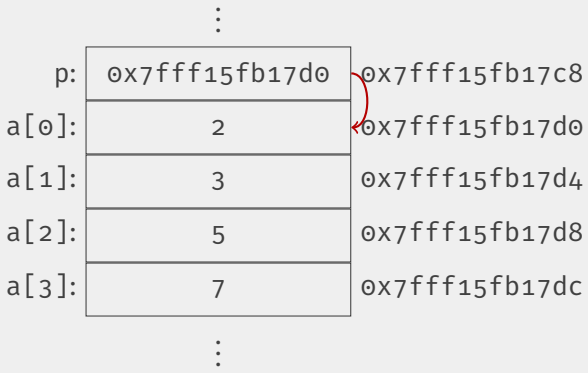
Estrecha relación entre punteros y arrays I

```
int *p;
int a[] = ...;
p = a;
```



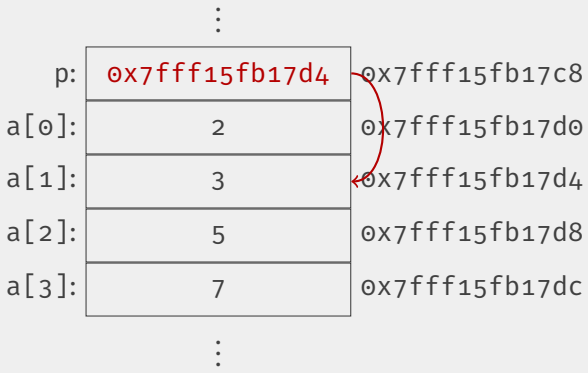
Estrecha relación entre punteros y arrays I

```
int *p;
int a[] = ...;
p = a;
assert(*p == a[0]);
```



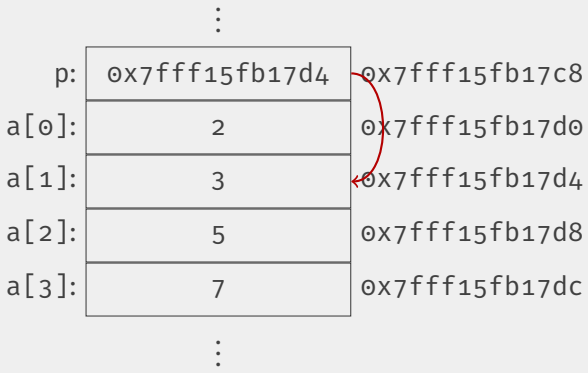
Estrecha relación entre punteros y arrays I

```
int *p;
int a[] = ...;
p = a;
assert(*p == a[0]);
p = p + 1;
```



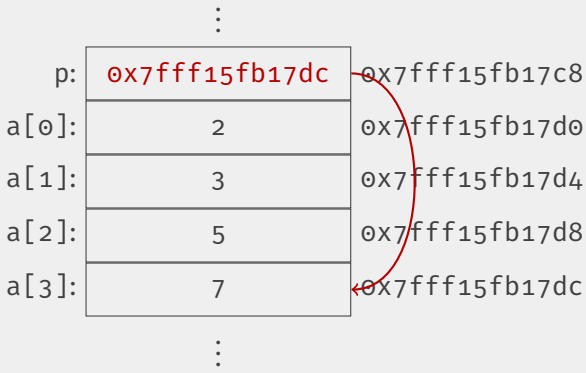
Estrecha relación entre punteros y arrays I

```
int *p;
int a[] = ...;
p = a;
assert(*p == a[0]);
p = p + 1;
assert(*p == a[1]);
```



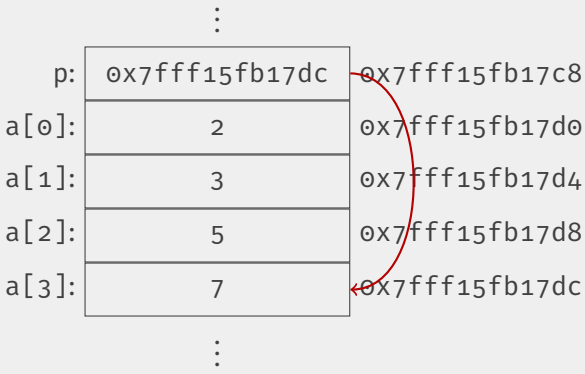
Estrecha relación entre punteros y arrays I

```
int *p;
int a[] = ...;
p = a;
assert(*p == a[0]);
p = p + 1;
assert(*p == a[1]);
p = p + 2;
```



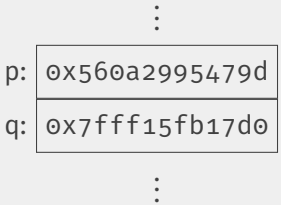
Estrecha relación entre punteros y arrays I

```
int *p;
int a[] = ...;
p = a;
assert(*p == a[0]);
p = p + 1;
assert(*p == a[1]);
p = p + 2;
assert(*p == a[3]);
```



Aritmética de punteros

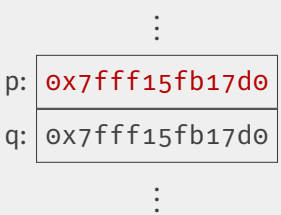
```
int *p;
long long int *q;
```



Aritmética de punteros

```
int *p;
long long int *q;
p = (int *)q;
```

← *¡sólo didáctico!*

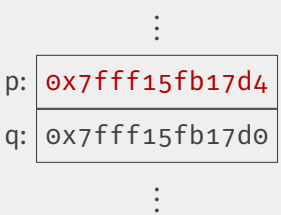


⚠ Este ejemplo es meramente didáctico, no usar ese tipo de asignaciones

Aritmética de punteros

```
int *p;
long long int *q;
p = (int *)q;
p++;
```

← *¡sólo didáctico!*

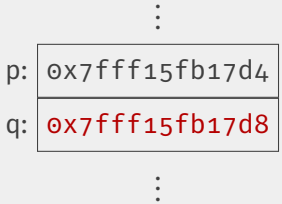


⚠ Este ejemplo es meramente didáctico, no usar ese tipo de asignaciones

Aritmética de punteros

```
int *p;
long long int *q;
p = (int *)q;
p++;
q++;
```

← *¡sólo didáctico!*

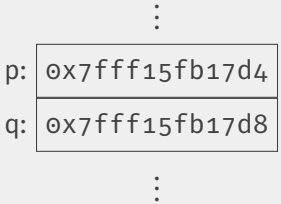


⚠ Este ejemplo es meramente didáctico, no usar ese tipo de asignaciones

Aritmética de punteros

```
int *p;
long long int *q;
p = (int *)q;
p++;
q++;
```

← *¡sólo didáctico!*



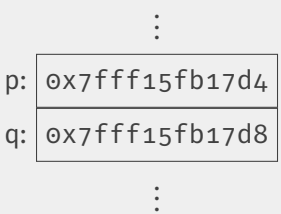
¿Ves la diferencia? ¿A qué se debe?

⚠ Este ejemplo es meramente didáctico, no usar ese tipo de asignaciones

Aritmética de punteros

```
int *p;
long long int *q;
p = (int *)q;
p++;
q++;
```

← *¡sólo didáctico!*



¿Ves la diferencia? ¿A qué se debe?
Imprimir los valores de los punteros 5'

Este ejemplo es meramente didáctico, no usar ese tipo de asignaciones

Actividad en el aula: www.wooclap.com

Código: IDIYRB

Estrecha relación entre punteros y arrays II

- Asumiendo el siguiente contexto...

```
T a[] = ...;
T *p = a;
```

- Tenemos las siguientes verdades

Estrecha relación entre punteros y arrays II

- Asumiendo el siguiente contexto...

```
T a[] = ...;
T *p = a;
```

- Tenemos las siguientes verdades

$$
\begin{aligned}
p &= a \\
p &= \&a[0] \\
*p &= a[0] \\
p+i &= \&a[i] \\
*(p+i) &= a[i]
\end{aligned}
$$

La densidad de información en las transparencias anteriores es enorme pero...

La densidad de información en las transparencias anteriores es enorme pero...

es imposible programar en C si no las entiendes

La densidad de información en las transparencias anteriores es enorme pero…

es imposible programar en C si no las entiendes

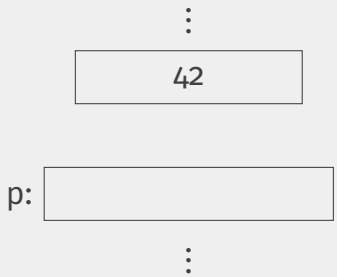
hoja de ejercicios

Recordatorio: punteros I

- Punteros: expresiones que representan direcciones de memoria y que apuntan a datos de tipo *T*:

```
int *p;
```

- p es un puntero a entero:

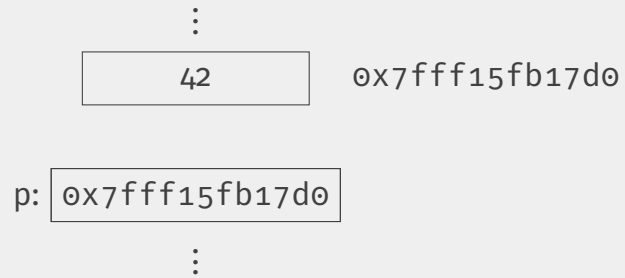


Recordatorio: punteros I

- Punteros: expresiones que representan direcciones de memoria y que apuntan a datos de tipo *T*:

```
int *p;
```

- p es un puntero a entero:

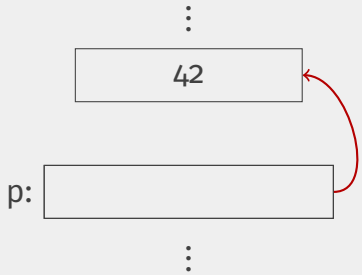


Recordatorio: punteros I

- Punteros: expresiones que representan direcciones de memoria y que apuntan a datos de tipo *T*:

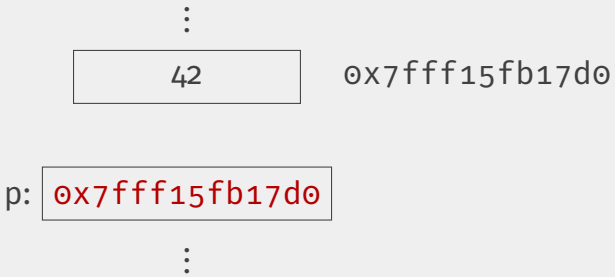
```
int *p;
```

- p es un puntero a entero:



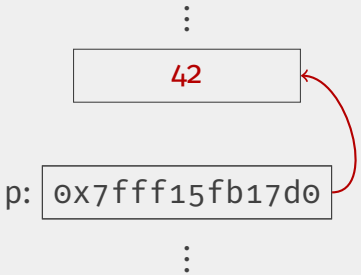
Recordatorio: punteros II

¿Qué es p?



Recordatorio: punteros II

¿Qué es p?
¿Qué es *p?



Recordatorio: direcciones

- Es posible conocer la dirección de memoria de algo en C:

  $\&e$

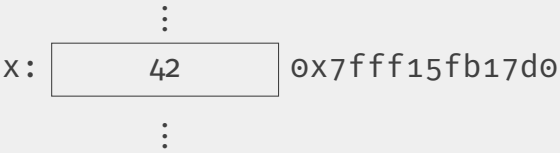
- Por ejemplo, `&x` es la dirección…

Recordatorio: direcciones

- Es posible conocer la dirección de memoria de algo en C:

  $\&e$

- Por ejemplo, `&x` es la dirección...

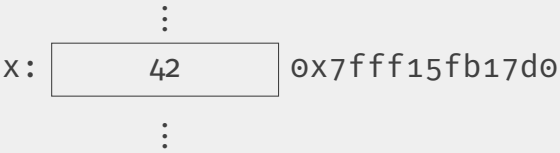


Recordatorio: direcciones

- Es posible conocer la dirección de memoria de algo en C:

  $\&e$

- Por ejemplo, `&x` es la dirección…`0x7fff15fb17d0`

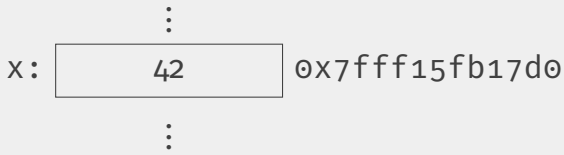


Recordatorio: direcciones

- Es posible conocer la dirección de memoria de algo en C:

  $\&e$

- Por ejemplo, `&x` es la dirección…`0x7fff15fb17d0`



- Por compatibilidad de tipos:

```
p = &x;
```

Punteros y arrays

- Asumiendo el siguiente contexto...

```
T a[];
T *p = a;
```

- Tenemos las siguientes verdades

Punteros y arrays

- Asumiendo el siguiente contexto...

```
T a[];
T *p = a;
```

- Tenemos las siguientes verdades

$$p = a$$
$$p = \&a[0]$$
$$*p = a[0]$$
$$p+i = \&a[i]$$
$$*(p+i) = a[i]$$

Además, en el capítulo de hoy...

`malloc`

`free`

Ordenar enteros

- Escribe un programa que ordene enteros de menor a mayor
- La entrada estándar tiene
  - Un entero positivo *n* en la primera línea
  - *n* enteros en las *n* siguientes líneas
- La salida de tu programa tiene, los *n* enteros después de la primera línea ordenados de menor a mayor

Ordenar enteros

- Escribe un programa que ordene enteros de menor a mayor
- La entrada estándar tiene
  - Un entero positivo *n* en la primera línea
  - *n* enteros en las *n* siguientes líneas
- La salida de tu programa tiene, los *n* enteros después de la primera línea ordenados de menor a mayor

Evita consumir más memoria de la necesaria

Ejemplo

Entrada	Salida
3 1 4 2	1 2 4

scanf = printf^{-1}

- scanf ya visto en clase y en ejercicios

```
int i, n, *datos;
...
scanf("%d", &n);
```

scanf = printf^{-1}

- `scanf` ya visto en clase y en ejercicios

```
int i, n, *datos;
...
scanf("%d", &n);
```

- El operador `&` se puede aplicar a cualquier *lvalue*

```
for (i = 0; i < n; i++)
  scanf("%d", &datos[i]);
```

scanf = printf^{-1}

- scanf ya visto en clase y en ejercicios

```
int i, n, *datos;
...
scanf("%d", &n);
```

- El operador & se puede aplicar a cualquier *lvalue*

```
for (i = 0; i < n; i++)
  scanf("%d", &datos[i]);
```

- Pero siempre podemos usar aritmética de punteros

```
for (i = 0; i < n; i++)
  scanf("%d", datos+i);
```

Bubble

```
for (i = 0 ; i < n - 1; i++)
  for (j = 0 ; j < n - i - 1; j++)
    if (datos[j] > datos[j + 1])
      intercambiar(&datos[j], &datos[j+1]);
```

¿Memoria suficiente?

- Hasta ahora sólo podíamos hacer esto

```
#define MAX 1000000

...

int datos[MAX]
```

- Pero…si hay menos de 1000000, desperdiciamos memoria
- Y si hay más de 1000000, tenemos un problema

Solicitud de memoria en tiempo de ejecución

- En lugar de establecer la memoria en tiempo de compilación debemos hacerlo en tiempo de ejecución
- Nada de automatismo en C: solicitud al sistema operativo
- En la biblioteca estándar[1] (**#include** <stdlib.h>)

```
void *malloc(size_t size);
```

- *The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. On error, these functions return NULL*

[1] man 3 malloc

¿Cuánta memoria hay que pedir en bytes?

```
int n;
int *datos;

scanf("%d", &n);

datos =          malloc(        ?        );

/* datos es un puntero a un bloque de
   memoria en el que caben n enteros,
   manejable como un array */
```

¿Cuánta memoria hay que pedir en bytes?

```
int n;
int *datos;

scanf("%d", &n);

datos =     ?   malloc(n * sizeof(int));

/* datos es un puntero a un bloque de
   memoria en el que caben n enteros,
   manejable como un array */
```

¿Cuánta memoria hay que pedir en bytes?

```
int n;
int *datos;

scanf("%d", &n);

datos = (int *) malloc(n * sizeof(int));

/* datos es un puntero a un bloque de
   memoria en el que caben n enteros,
   manejable como un array */
```

A PROGRAMAR

15’

Ordenar grupos de enteros

- Escribe un programa que ordene enteros de menor a mayor
- La entrada estándar tiene ¡de forma repetida!
  - Un entero positivo *n* en la primera línea
  - *n* enteros en las *n* siguientes líneas
- La salida de tu programa tiene, por cada entrada, los *n* enteros después de la primera línea ordenados de menor a mayor

Ordenar grupos de enteros

- Escribe un programa que ordene enteros de menor a mayor
- La entrada estándar tiene ¡de forma repetida!
  - Un entero positivo *n* en la primera línea
  - *n* enteros en las *n* siguientes líneas
- La salida de tu programa tiene, por cada entrada, los *n* enteros después de la primera línea ordenados de menor a mayor

Evita consumir más memoria de la necesaria

while (n) Ordenar

```
2
7
1
3
1
4
2
0
```

Entrada

```
1
7

1
2
4
```

Salida

¿Problema?

Responsabilidad: `free`

liberar la memoria solicitada una vez usada

- En la biblioteca estándar[2] (**`#include <stdlib.h>`**)

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

- *The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(). Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.*

[2] `man 3 malloc`

Liberar despues de cada ordenación

```
while(n) {
  /* Solicitar memoria */
  datos = (int *) malloc(n * sizeof(int));
  /* Leer enteros y ordenarlos */
  ...
  /* Imprimir el array ya ordenado */
  ...
  /* Liberar memoria */
  free(datos);
  /* Leer siguiente n */
  scanf("%d", &n);
}
```

Memory leaks

Cuando se nos olvida liberar memoria

Memory leaks

Cuando se nos olvida liberar memoria

Segmentation fault

Cuando se nos olvida solicitar memoria

Memory leaks

Cuando se nos olvida liberar memoria

Segmentation fault

Cuando se nos olvida solicitar memoria
o usamos más allá de la solicitada

Memory leaks

Cuando se nos olvida liberar memoria

Segmentation fault

Cuando se nos olvida solicitar memoria
o usamos más allá de la solicitada

Comportamiento indefinido

Cuando liberamos memoria no solicitada

¿Cuánta memoria puedes pedir?

- Escribe un programa que escriba el tamaño de memoria máximo que puedes solicitar (en bytes).

5’

¿Cuánta memoria puedes pedir?

- Escribe un programa que escriba el tamaño de memoria máximo que puedes solicitar (en bytes).

5’

- Idea:

1, 2, 4, 8, 16, 32, 64, 128, 256,
192, 224,
208,
200, 204
¡Bingo!

malloc ES APLICABLE A CUALQUIER TIPO

```
char *s = (char *) malloc(N * sizeof(char));
double *reales = (double *) malloc(N * sizeof(double));
```

[3] Relax: espero que podamos entender la sintaxis para declarar la variable `vectores` al final de la asignatura

malloc ES APLICABLE A CUALQUIER TIPO

```
char *s = (char *) malloc(N * sizeof(char));
double *reales = (double *) malloc(N * sizeof(double));
```

Incluso[3]

```
char **cadenas =
  (char **) malloc(N * sizeof(char *));
int (*vectores)[10] =
  (int (*)[10]) malloc(N * sizeof(int [10]));
```

[3] Relax: espero que podamos entender la sintaxis para declarar la variable `vectores` al final de la asignatura

malloc's *FRIENDS*: man 3 malloc

```
MALLOC(3)                Linux Programmer's Manual               MALLOC(3)

NAME
       malloc,  free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);

DESCRIPTION
       The  malloc()  function  allocates  size  bytes  and  returns a
       ...
```