

Sesión 07: Memoria dinámica

Programación para Sistemas

Ángel Herranz

2019-2020

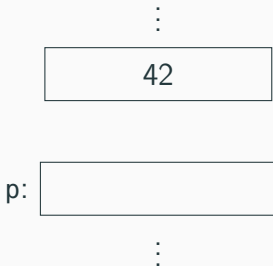
Universidad Politécnica de Madrid

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**:

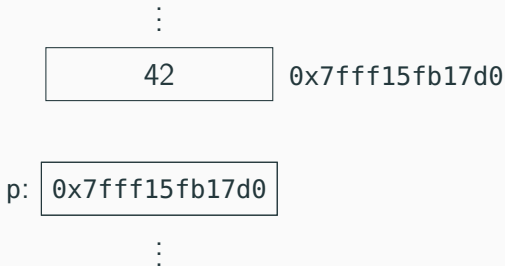


Recordatorio: punteros i

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

`int *p;`

- p es un **puntero a entero**:

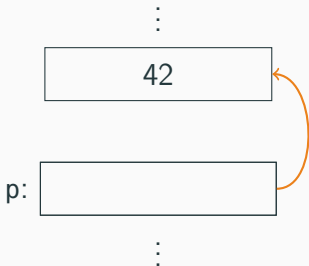


Recordatorio: punteros i

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

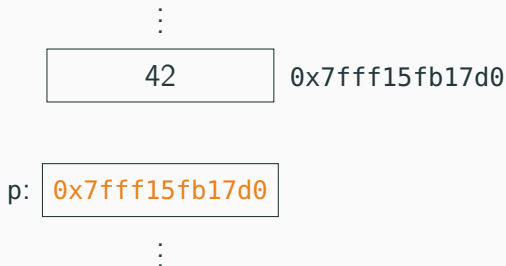
`int *p;`

- 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...

⋮

x:	42
----	----

⋮

`0x7fff15fb17d0`

Recordatorio: direcciones

- Es posible conocer la **dirección de memoria** de algo en C:
 $\&e$
- Por ejemplo, $\&x$ es la dirección... **0x7fff15fb17d0**

⋮

x:	42
----	----

⋮

0x7fff15fb17d0

Recordatorio: direcciones

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

`&e`

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

⋮

x:	42
----	----

⋮

`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

$$[[p]] = [[a]]$$

$$[[p]] = [[\&a[0]]]$$

$$[[*p]] = [[a[0]]]$$

$$[[p+i]] = [[\&a[i]]]$$

$$[[*(p+i)]] = [[a[i]]]$$

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

`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 estándar¹ (**#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*

¹man 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 */
```




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

while (n) Ordenar

2
7
1
3
1
4
2
0

Entrada

1
7

1
2
4

Salida

¿Problema?

liberar la memoria solicitada una vez usada

- En la biblioteca estándar² (**#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.*

²man 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).

¿Cuánta memoria puedes pedir?

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

- 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));
```

³Relax: 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));
```

Incluso³

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

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