

Sesión 05: Arrays y Strings

Hoja de problemas


Programación para Sistemas


Ángel Herranz


aherranz@fi.upm.es

Universidad Politécnica de Madrid

Otoño 2018

 **Ejercicio 1.** Repasa las transparencias de clase. No dejes de transcribir y realizar todos los programas sugeridos en las mismas.

 **Ejercicio 2.** Para realizar los siguientes ejercicios vamos a utilizar la función *inversa* a `printf`. Se llama `scanf` y en vez de imprimir datos en la salida estándar lee datos de la entrada estándar. Puedes probar tú mismo el manual de `scanf`: `man 3 scanf` o incluso puedes echar un ojo al libro K&R (sección 7.4 de la segunda edición en inglés).

 **Ejercicio 3.** Aunque para entender bien el funcionamiento de la familia de funciones `scanf` es necesario saber un poquito de punteros, vamos a usarla con el siguiente patrón:

```
1 #include <stdio.h>
2
3 int main() {
4     int x;
5     int leidos;
6     leidos = scanf("%i", &x);
7     while (leidos == 1) {
8         printf("%i\n", 2*x);
9         leidos = scanf("%i", &x);
10    }
11    return 0;
12 }
```

La línea 6 intenta leer **un** entero (eso se indica con "`%i`") de la entrada estándar y lo deja en la variable `x` (eso se indica con la expresión `&x`). Si en la entrada estándar se lee un entero la función `scanf` devuelve **1** que se asigna a `leidos`, si lo que hay en la entrada estándar es algo diferente a un entero entonces devuelve un **0** que se asigna a `leidos`.

Ese programa lee enteros en la entrada estándar y los multiplica por dos antes de imprimirlo en la salida estándar. El bucle para cuando lo que se lee de la entrada estándar no es un entero. Compila y ejecuta para que lo puedas ver.

Se puede ver cómo se aprovecha que la función `scanf` devuelve el número de argumentos que encajan con el formato, que dicho número se aprovecha directamente como condición del bucle: si distinto de 0.

```

1 #include <stdio.h>
2
3 int main() {
4     int x;
5     while (scanf("%d", &x))
6         printf("%d\n", 2*x);
7     return 0;
8 }

```

La verdad es que un programador de C pura zepa jamás escribiría el código anterior, cualquier pensaría que lo ha hecho un programador de Java. Veamos un programar equivalente en un estilo más C:

- 📄 **Ejercicio 4.** Con lo que has aprendido en el ejercicio anterior y en las transparencias, escribe un programa que lea enteros y los almacene en un array para imprimirlos en el orden inverso. Tu programa admitirá un máximo de M enteros (pongamos 1000). Si en la entrada estándar hay más de M enteros sólo se invertirán los M primeros. En el momento en el que haya algo diferente a un entero, el programar invertirá todos los enteros leídos hasta ese momento.

Para no tener que teclear números desde el teclado os sugerimos que pongáis los números en un fichero, digamos `numeros.txt`:

```

1
8
4000
2
23
51
87
fin

```

Si suponemos que has llamado a tu programa `invertir_enteros`, esperamos que este sea el comportamiento:

```

$ ./invertir_enteros < numeros.txt
87
51
23
2
4000
8
1
$ |

```

- 📄 **Ejercicio 5.** En las transparencias hemos visto cómo podemos calcular la longitud de un array usando esta expresión:

`sizeof(a)/sizeof(a[0])`

- ⚠ **Escribe una macro (`#define`) que realice ese trabajo dado un array cualquiera¹.
Recuerda que dicha técnica no es aplicable cuando el array no es de longitud fija.**

- 📄 **Ejercicio 6.** Escribir una función que ordena un array de enteros entre dos posiciones. La *cabecera* de dicha función será:

```
void ordenar(int a[], int min, int max);
```

- 📄 **Ejercicio 7.** Utiliza la función anterior para implementar un programa que lea enteros de la entrada estándar y los imprima en la salida estándar ordenados. Tu programa admitirá un máximo de M enteros (pongamos 1000). Si en la entrada estándar hay más de M enteros sólo se ordenarán los M primeros. En el momento en el que haya algo diferente a un entero, el programar ordenará todos los enteros leídos hasta ese momento.

Si suponemos que has llamado a tu programa `ordenar_enteros`, esperamos que este sea el comportamiento²:

```
$ ./ordenar_enteros < numeros.txt
1
2
8
23
51
87
4000
$ |
```

- 📄 **Ejercicio 8.** Puedes intentar varios algoritmos de ordenación (*buble sort*, *insert sort*, *merge sort*, y *quick sort* por ejemplo) y probar el tiempo que tarda en ejecutar cada uno de ellos.

Para ello puedes generar 1000 números aleatorios y almacenarlos en el fichero `numeros.txt` usando este mandato de Bash:

```
for i in $(seq 1000); do echo $RANDOM >> numeros.txt; done
```

Con este otro mandato de Bash puedes comprobar el tiempo de ejecución:

```
time ./ordenar_enteros < numeros.txt
```

- 🔊 **Ejercicio 9.** Escribe y prueba una función que intercambie dos números enteros. Su *cabecera*:

```
void intercambiar(int x, int y);
```

Si la usamos en el siguiente contexto:

¹Mira la sección 4.11.2 del K&R: *Macro Substitution*

²El fichero `numeros.txt` es el mismo que el del ejercicio 4

```
int x = 42, y = 27;
printf("Antes de intercambiar: (%i, %i)\n", x, y);
intercambiar(x,y);
printf("Despues de intercambiar: (%i, %i)\n", x, y);
```

Lo que se espera en la salida estándar es:

```
Antes de intercambiar: (42, 27)
Despues de intercambiar: (27, 42)
```

- 💬 **Ejercicio 10.** ¿Se puede combinar una inicialización de un array con el hecho de que sea un array de longitud fija? ¿Se puede hacer esto en C?

```
int a[21] = {1, 2, ,3};
```

¿Qué significa? ¿Cuál es su longitud? ¿Cuántos enteros caben?

- 💬 **Ejercicio 11.** Sorprendentemente C admite hacer declaraciones de argumentos de tipo array con longitud fija. ¿Qué significado tiene?

- 💬 **Ejercicio 12.** ¿Puedes almacenar un string de longitud 5 (como por ejemplo "adios") en un array de longitud 4? ¿Y en uno de longitud 5? ¿Y en uno de longitud 6? ¿Y en uno de longitud 10? ¿Entiendes la diferencia entre la longitud de un string y la del array que lo contiene?

- 📖 **Ejercicio 13.** Escribe un programa que lea de la entrada estándar dos matrices y las multiplique. Puedes asumir que las matrices están codificadas de la siguiente forma en la entrada estándar:

- Un entero positivo m que indica el número de filas de la primera matriz.
- Un entero positivo n que indica el número de columnas de la primera matriz (que coincide con el número de filas de la segunda).
- Un entero positivo p que indica el número de columnas de la segunda matriz.
- $m \times n$ floats: los n primeros son la primera fila de la primera matriz, los n segundos la segunda fila, etc.
- $n \times p$ floats: los p primeros son la primera fila de la segunda matriz, los p segundos la segunda fila, etc.

Se puede asumir que m , n y p están entre 1 y 1000. El resultado serán $m \times p$ floats donde los p primeros son la primera fila de la matriz multiplicación, los p segundos la segunda fila, etc.

Para no tener que teclear las matrices desde el teclado os sugerimos que las pongais en un fichero, digamos `matrices.txt`:

```
3
2
3
1 2
-1 0
```

```
-3 -1  
2 0 1  
-5 2 3
```

Si suponemos que has llamado a tu programa `multiplicar_matrices`, esperamos que este sea el comportamiento:

```
$ ./multiplicar_matrices < matrices.txt  
-8 4 7  
-2 0 -1  
-1 -2 -6  
$ |
```