Metodología de la programación

5 de marzo de 2018

ugr | Universidad de Granada

Práctica 2: arrays, cadenas de caracteres y matrices

Curso 2017-2018

Índice

Objetivos

El objetivo de esta práctica es trabajar con los conceptos vistos en el primer tema de teoría. Para ello **debéis implementar 5 de los ejercicios propuestos en esta relación**. Para cada uno de ellos se ofrece un conjunto de pruebas concreto que el programa debe tratar de forma correcta para que se considere superado el ejercicio.

Se recuerda que el trabajo en estos ejercicios debe ser personal y que se recomienda la asistencia a clase de prácticas. La copia de código no aporta nada al aprendizaje y será considerada como un incumplimiento de las normas de la asignatura con las consecuencias que ello implica, según la normativa de la Universidad de Granada.

Para todos los ejercicios debes separar el código en módulos independientes: el que contiene la función (o funciones) se llamará **utilidades** y el que aporta el programa principal se llamará como el ejercicio que corresponda. Trabaja en un directorio (llamado como el ejercicio) en el que se creen los subdirectorios para archivos de cabecera, archivos de código objeto, ejecutable y código fuente. Proporciona un archivo **Makefile** que genere el ejecutable a partir del código fuente.

Para que quede más claro consideremos el caso del **ejercicio1**. Estará situado en un directorio llamado de la misma forma (**ejercicio1**) y que contendrá la siguiente estructura una vez esté completo y haya sido compilado y generado el ejecutable:

```
-- Makefile
-- bin
|   -- ejercicio1
-- include
|   -- utilidades.h
-- obj
|   -- ejercicio1.o
|   -- utilidades.o
-- src
|   -- ejercicio1.cpp
|   -- utilidades.cpp
```

Deben mantenerse estos nombres de carpetas para que el código pueda corregirse y compilarse de forma automática tras la entrega. También deben ajustarse los nombres de los archivos, funciones, clases,...., a las indicaciones dadas en los ejercicios. **NOTA: no se considerarán ejercicios en que no se cumplan estas normas. Tampoco aquellos en que se usen espacios en blanco en los nombres de los archivos o carpetas o caracteres especiales (como acentos)**.

En general, se proporcionará el código básico para probar la funcionalidad pedida, con los casos de prueba indicados. Debe observarse la forma de uso de las funciones y/o métodos en el programa principal para especificar de forma correcta los argumentos de las funciones. **NOTA: no se modificará el contenido de los archivos que se entregan con el programa principal**. Deben implementarse todas las funciones necesarias para que el código incluido en el programa principal se ejecute de forma correcta. Esto incluye, en algunos casos, una función que permita mostrar el contenido de un array.

Ejercicio 1

Implemente una función llamada **mezclar** que cumpla las siguientes indicaciones:

- recibe como argumentos (sólo se indican los argumentos esenciales; podéis agregar todos aquellos que consideréis oportunos):
  - dos arrays de bajo nivel, de tipo **double**, que contienen valores ordenados y sin repetir (aunque sí podría haber valores que aparezcan en ambos). La función debe combinar el contenido de estos arrays, de forma que el resultado sea una secuencia de valores ordenados y sin repetidos
  - un tercer array, resultado, donde se almacena el resultado del procesamiento de la función. Puede asumirse que este array está dimensionado para poder contener todos los valores necesarios
- la función devolverá un valor entero que indique el número de valores almacenados en el array resultante

Casos de prueba

Los casos de prueba a considerar son:

```
// Caso 1
double array1[]={1, 3, 5, 7};
```

```
double array2[]={2, 4, 4.3, 9};

// Caso 2
double array3[]={1, 3, 5};
double array4[]={2, 3.8, 4.3, 6.4, 9.3};

// Caso 3
double array5[]={5, 6.3, 7.5, 8.3, 9.2};
double array6[]={1.0, 3.4, 6.3};
```

La salida que debería obtenerse es:

```
Caso de prueba 1:
1 3 5 7
2 4 4.3 9
_____________________________________________
1 2 3 4 4.3 5 7 9
_____________________________________________

Caso de prueba 2:
1 3 5
2 3.8 4.3 6.4 9.3
_____________________________________________
1 2 3 3.8 4.3 5 6.4 9.3
_____________________________________________

Caso de prueba 3:
5 6.3 7.5 8.3 9.2
1 3.4 6.3
_____________________________________________
1 3.4 5 6.3 7.5 8.3 9.2
_____________________________________________
```

Ejercicio 2

Programe una función llamada **combinarSuma** que reciba como argumento dos vectores **C** de tipo **Valor** y produce como resultado otro vector del mismo tipo. La estructura básica de la clase **Valor** es la siguiente:

```
class Valor{
  public:
     int fila;
     int columna;
     double valor;
     ....................
};
```

Es decir, cada objeto de la clase **Valor** se usa para indicar una posición de una matriz, caracterizada por la fila y columna en que se encuentra y el valor almacenado en dicha posición. El resultado obtenido debe cumplir las siguientes características:

- si un objeto de la clase **Valor** aparece en un único vector de entrada, entonces se almacena directamente en el vector de salida, en la posición que le corresponda
- si en ambos vectores hay dos objetos de clase **Valor** que se refieren a una misma posición (tienen el mismo valor para **fila** y **columna**), entonces el objeto a almacenar en el vector de salida tendrá el mismo valor para estos dos datos miembro, pero el valor para el dato miembro **valor** se obtendrá de la suma de este dato en ambos objetos
- pueden usarse funciones auxiliares si se considera oportuno

Como ejemplo, imaginemos que en el primer array aparece el objeto con valores **fila=1, columna=1, valor=0.5** y en el segundo el objeto con contenido **fila=1, columna=1, valor=7.3**. Entonces el objeto a almacenar en el array resultante sería **fila=1, columna=1, valor=7.8**.

Se proporciona la implementación de la clase **Valor**, incluida en el archivo **Valor.h**.

Casos de prueba

Los casos de prueba a considerar son los siguientes:

```
// Caso 1
Valor array1[]={Valor(1, 1, 0.7), Valor(1, 3, 9.4), Valor(2, 1, 8.3), Valor(3, 1,
     6.45), Valor(3, 3, 2.75)};
Valor array2[]={Valor(0, 0, 10), Valor(1, 1, 0.3), Valor(1, 2, 4.3), Valor(2, 2,
    1.5), Valor(3, 2, 0.25)};

// Caso 2
Valor array3[] = {Valor(0, 0 , 9.3), Valor(1 , 3 , 5.4), Valor(2, 3, 8.3), Valor
    (3, 1, 6.45)} ;
Valor array4[] = {Valor(2, 2, 0.85) , Valor(2, 3, 3.8) , Valor(3, 1, 4.3) , Valor
    (3, 2, 6.4), Valor(3, 3, 9.3)};

// Caso 3
Valor array5[] = {Valor(0, 0 , 5.2), Valor(1 , 1 , 6.3), Valor(2, 2, 7.5), Valor
    (2, 3, 8.3), Valor(3, 3, 9.2)} ;
Valor array6[] = {Valor(0, 0, 1.0) , Valor(1, 1, 3.4) , Valor(3, 3, 6.3)};
```

Los resultados que deben obtenerse son:

```
------------------- caso prueba 1 -------------------
{1, 1, 0.7} -- {1, 3, 9.4} -- {2, 1, 8.3} -- {3, 1, 6.45} -- {3, 3, 2.75}
{0, 0, 10} -- {1, 1, 0.3} -- {1, 2, 4.3} -- {2, 2, 1.5} -- {3, 2, 0.25}
-----------------------------------------------------
{1, 1, 1} -- {1, 3, 9.4} -- {2, 1, 8.3} -- {3, 1, 6.45} -- {3, 3, 2.75} --
    {0, 0, 10} -- {1, 2, 4.3} -- {2, 2, 1.5} -- {3, 2, 0.25}

------------------- caso prueba 2 -------------------
{0, 0, 9.3} -- {1, 3, 5.4} -- {2, 3, 8.3} -- {3, 1, 6.45}
{2, 2, 0.85} -- {2, 3, 3.8} -- {3, 1, 4.3} -- {3, 2, 6.4} -- {3, 3, 9.3}
-----------------------------------------------------
{0, 0, 9.3} -- {1, 3, 5.4} -- {2, 3, 12.1} -- {3, 1, 10.75} -- {2, 2, 0.85}
    -- {3, 2, 6.4} -- {3, 3, 9.3}

------------------- caso prueba 3 -------------------
{0, 0, 5.2} -- {1, 1, 6.3} -- {2, 2, 7.5} -- {2, 3, 8.3} -- {3, 3, 9.2}
{0, 0, 1} -- {1, 1, 3.4} -- {3, 3, 6.3}
-----------------------------------------------------
{0, 0, 6.2} -- {1, 1, 9.7} -- {2, 2, 7.5} -- {2, 3, 8.3} -- {3, 3, 15.5}
```

Ejercicio 3

Haga un programa que obtenga la mayor secuencia monótona creciente de un array de enteros, guardándola en otro array (mediante una función). Se recomienda hacer uso del paso por referencia.

Casos de prueba

```
// Caso 1
int array1[]={7, 89, 75, 5, 8, 89, 97};

// Caso 2
int array2[]={7, 6, 5, 4, 3, 2, 1};

// Caso 3
int array3[]={1, 2, 3, 4, 5, 1, 2};

// Caso 4
int array4[]={1, 2, 1, 2, 3, 4, 5};

// Caso 5
int array5[]={1, 2, 3, 4, 5, 6, 7};
```

Salida esperada:

```
---------------------- caso prueba 1 ----------------------------
7 89 75 5 8 89 97
5 8 89 97
-----------------------------------------------------------------

---------------------- caso prueba 2 ----------------------------
7 6 5 4 3 2 1
7
-----------------------------------------------------------------

---------------------- caso prueba 3 ----------------------------
1 2 3 4 5 1 2
1 2 3 4 5
-----------------------------------------------------------------

---------------------- caso prueba 4 ----------------------------
1 2 1 2 3 4 5
1 2 3 4 5
-----------------------------------------------------------------

---------------------- caso prueba 5 ----------------------------
1 2 3 4 5 6 7
1 2 3 4 5 6 7
-----------------------------------------------------------------
```

Ejercicio 4

Implemente una función que reciba como entrada dos cadenas de caracteres estilo **C** y compruebe si la segunda cadena está contenida en la primera. En caso afirmativo, devolverá la posición en que se encuentra; en caso contrario devuelve -1.

Casos de prueba

```
// Caso 1
char cadena[] = "Hola, primera cadena de ejemplo";
char subcadena1[] = "no coincide";

// Caso 2: se prueba contra cadena (coincidencia inicio)
char subcadena2[] = "Hol";

// Caso 3: se prueba contra cadena (coincidencia final)
int array3[]="plo";

// Caso 4: se prueba contra cadena (coincidencia parte
// central)
int array4[]="cad";

// Caso 5: se prueba contra cadena (coincidencia parte
// central, pero no total)
int array4[]="cadenita";
```

Se debe obtener la salida indicada a continuación:

```
------------------------- caso prueba 1 -------------------------
no coincide - no es subcadena de - Hola, primera cadena de ejemplo
-----------------------------------------------------------------

------------------------- caso prueba 2 -------------------------
Hol - es subcadena de - Hola, primera cadena de ejemplo. Posicion: 0
-----------------------------------------------------------------

------------------------- caso prueba 3 -------------------------
plo - es subcadena de - Hola, primera cadena de ejemplo. Posicion: 28
-----------------------------------------------------------------

------------------------- caso prueba 4 -------------------------
cad - es subcadena de - Hola, primera cadena de ejemplo. Posicion: 14
-----------------------------------------------------------------

------------------------- caso prueba 5 -------------------------
cadenita - no es subcadena de - Hola, primera cadena de ejemplo
-----------------------------------------------------------------
```

Ejercicio 5

Implemente una función que inserte una cadena de caracteres dentro de otra cadena, en una determinada posición.

Casos de prueba

```
// Se declara la cadena base
char base[] = "cadena base para prueba de insercion";

// Se declara la cadena que se inserta
char aInsertar[] = "-agregado-";

// Caso 1: insercion en posicion 0 (inicio)

// Caso 2: insercion en posicion 36 (final)

// Caso 3: insercion en posicion 12

// Caso 4: insercion en posicion 50 (pos. no valida)
```

Se debe obtener la salida indicada a continuación:

```

------------------------- caso de prueba 1 -------------------------
Insercion en posicion: 0
Insercion posible: 1
-agregado-cadena base para prueba de insercion
--------------------------------------------------------------------
------------------------- caso de prueba 2 -------------------------
Insercion en posicion: 36
Insercion posible: 1
cadena base para prueba de insercion-agregado-
--------------------------------------------------------------------
------------------------- caso de prueba 3 -------------------------
Insercion en posicion: 12
Insercion posible: 1
cadena base -agregado-para prueba de insercion
--------------------------------------------------------------------
------------------------- caso de prueba 4 -------------------------
Insercion en posicion invalida: 50
Insercion posible: 0
--------------------------------------------------------------------
```

Ejercicio 6

Programe una función que realice el producto de dos matrices. Debe facilitarse a su vez una función que permita visualizar el contenido de una matriz para comprobar el correcto funcionamiento del código realizado.

Casos de prueba

```
// ------------------ CASO de prueba 1 ------------------
// matrices para el primer caso de prueba: matriz11 y matriz12
double matriz11[FIL][COL]={
        {1, 2, 3, 4},
        {5, 6, 7, 8},
```

```
        {9, 10, 11, 12}
};

double matriz12[FIL][COL]={
        {12, 11, 10},
        {9, 8, 7},
        {6, 5, 4},
        {3, 2, 1}
};

// ------------------ CASO de prueba 2 ------------------
// matrices para el primer caso de prueba: matriz21 y matriz22
double matriz21[FIL][COL]={
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12},
    {13, 14, 15, 16}
};

double matriz22[FIL][COL]={
    {16, 15, 14, 13},
    {12, 11, 10, 9},
    {8, 7, 6, 5},
    {4, 3, 2, 1}
};

// ------------------ CASO de prueba 3 ------------------
// matrices para el primer caso de prueba: matriz31 y matriz32
double matriz31[FIL][COL]={
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
};

double matriz32[FIL][COL]={
        {12, 11, 10},
        {9, 8, 7},
        {6, 5, 4}
};
```

Se debe obtener la salida indicada a continuación:

```
.............. caso 1 ..................
1 2 3 4
5 6 7 8
9 10 11 12

12 11 10
9 8 7
6 5 4
3 2 1
```

```
Operacion valida: 1
60 50 40
180 154 128
300 258 216

.............. caso 2 ..............
1 2 3 4
5 6 7 8
9 10 11 12

16 15 14
12 11 10
8 7 6
4 3 2

Operacion valida: 1
80 70 60 50
240 214 188 162
400 358 316 274
560 502 444 386

.............. caso 3 ..............
1 2 3 4
5 6 7 8
9 10 11 12

12 11 10
9 8 7
6 5 4

Operacion valida: 0
```

Ejercicio 7

Construid una función recursiva que calcule la descomposición factorial de un número entero y la devuelva en un array tipo **C**. La declaración de la función es la siguiente:

```
void descomponer(int numero, int factores[], int & numeroFactores)
```

Podéis usar las funciones auxiliares que consideréis pertinentes y organizar el código libremente.

Casos de prueba

Se propone el uso de los siguientes casos de prueba: número no primo con muchos factores (13860), número no primo con pocos factores (1989433), número primo (104729) y número 1. La salida a obtener debería ser similar a la indicada a continuación:

```
---------- Caso 1 ----------
Descomposicion de numero no primo: 13860
Numero de factores: 7
2 2 3 3 5 7 11
----------------------------------------
---------- Caso 2 ----------
Descomposicion de numero no primo: 1989433
```

```
Numero de factores: 2
19 104707
-----------------------------------------------------
---------------- Caso 3 -----------------------------
Descomposicion de numero primo: 104729
Numero de factores: 1
104729
-----------------------------------------------------
---------------- Caso 4 -----------------------------
Descomposicion de numero primo: 1
Numero de factores: 0
-----------------------------------------------------
```

Ejercicio 8

Supongamos que se eligen dos puntos de forma aleatoria, con distribución uniforme, en el cuadrado unitario $[0, 1]^2$. Escribid un programa que estime la longitud media esperada del segmento que une a ambos puntos. Para ello se seleccionarán muchos pares de puntos diferentes (100000, por ejemplo) y el programa ofrecerá como resultado la longitud media de las distancias calculadas para cada par de puntos. Los resultados de cada prueba (cada par de puntos produce una distancia) se almacenarán en un array.

La implementación se basará en usar una clase llamada **EstimadorDistancia** que contendrá los elementos necesarios para resolver este problema. Se ofrece la implementación base de la clase, dejando sin implementar las partes que aparecen con puntos suspensivos.

No se ofrecen en este caso casos de prueba, ya que la naturaleza aleatoria del programa producirá resultados diferentes cada vez que se ejecute el programa. Observad también que el programa principal para este ejercicio debe invocarse con un argumento, que permitirá especificar el número de muestras o pares de puntos a generar (se incluye la línea de comando necesario para ejecutar desde linux; el ejemplo ejecutaría el programa de forma que la distancia media obtenida se ofreciese como la estimación de la distancia entre 10000 pares de puntos):

```
./ejercicio8 10000
```