RELACIÓN DE PROBLEMAS I. Punteros

1. Describir la salida de los siguientes programas:

a)

```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p;

  a = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

b)

```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p;

  *p = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

c)

```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p = &a;

  *p = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

d)

```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p = &a, **p2 = &p;

  **p2 = *p + (**p2 / a);
  *p = a+1;
  a = **p2 / 2;
  cout << "a es igual a: " << a << endl;
  return 0;

}
```

2. Represente gráficamente la disposición en memoria de las variables del programa mostrado en la figura 20, e indique lo que escribe la última sentencia de salida.

3. Declare una variable `v` como un vector de 1000 enteros. Escriba un trozo de código que recorra el vector y modifique todos los enteros *negativos* cambiándolos de signo.

   No se permite usar el operador `[]`, es decir, el recorrido se efecuará usando aritmética de punteros y el bucle se controlará mediante un contador entero.

   Nota: Para inicializar aleatoriamente el vector con valores enteros entre -50 y 50, por ejemplo, puede emplearse el siguiente fragmento de código (o empleando la clase que genera números aleatorios - *Fundamentos de Programación*):

```
1  #include <iostream>
2  using namespace std;
3
4  struct Celda {
5    int d;
6    Celda *p1, *p2, *p3;
7  };
8
9  int main (int argc, char *argv[])
10 {
11   Celda a, b, c, d;
12
13   a.d = b.d = c.d = d.d = 0;
14
15   a.p1 = &c;
16   c.p3 = &d;
17   a.p2 = a.p1->p3;
18   d.p1 = &b;
19   a.p3 = c.p3->p1;
20   a.p3->p2 = a.p1;
21   a.p1->p1 = &a;
22   a.p1->p3->p1->p2->p2 = c.p3->p1;
23   c.p1->p3->p1 = &b;
24   (*((*(c.p3->p1)).p2->p3)).p3 = a.p1->p3;
25   d.p2 = b.p2;
26   (*(a.p3->p1)).p2->p2->p3 = (*(a.p3->p2)).p3->p1->p2;
27
28   a.p1->p2->p2->p1->d = 5;
29   d.p1->p3->p1->p2->p1->p1->d = 7;
30   (*(d.p1->p3)).p3->d = 9;
31   c.p1->p2->p3->d = a.p1->p2->d - 2;
32   (*(c.p2->p1)).p2->d = 10;
33
34   cout << "a="<<a.d<<"  b="<<b.d<<"  c="<<c.d<<"  d="<<d.d<<endl;
35 }
```

Figura 20: Código asociado al problema 2

```
#include <cstdlib>
#include <ctime>
   ...
   const int MY_MAX_RAND = 50;  // Queremos valores -50<=n<=50
   time_t t;
   ...
   srand ((int) time(&t)); // Inicializa el generador con
                           // el reloj del sistema
   ...
   for int (i=0; i<1000; i++)
      v[i] = (rand() % ((2*MY_MAX_RAND)+1)) - MY_MAX_RAND;
```

Acerca de `srand()`, `rand()` y `time()`: `http://www.cplusplus.com`

4. Modifique el código del problema 3 para controlar el final del bucle con un puntero a la posición siguiente a la última.

5. Con estas declaraciones:

```
const int TOPE = 100;
float v1 [TOPE] = {2,3,8,22,44,88,99,100,101,255,665};
float v2 [TOPE] = {1,3,4,5,6,25,87,89,99,100,500,1000};
float res [2*TOPE];

int tam_v1=11, tam_v2=12;      // 0 <= tam_v1, tam_v2 < TOPE
int tam_res = tam_v1+tam_v2;   // 0 <= tam_res < 2*TOPE
```

   Escribir un trozo de código para mezclar, de manera *ordenada*, los valores de `v1` y `v2` en el vector `res`.

   Nota: Observad que `v1` y `v2` almacenan valores *ordenados* de menor a mayor.

   No se puede usar el operador `[]`, es decir, se debe resolver usando aritmética de punteros.

6. Consideremos un vector `v` de números reales de tamaño `TOPE`. Supongamos que se desea dividir el vector en dos secciones: la primera contendrá a todos los elementos menores o iguales al primero y la otra, los mayor.

   Para ello, proponemos un algoritmo que consiste en:

   - Colocamos un puntero al principio del vector y lo adelantamos mientras el elemento apuntado sea menor o igual que el primero.
   - Colocamos un puntero al final del vector y lo atrasamos mientras el elemento apuntado sea mayor que el primero.
   - Si los punteros no se han cruzado, es que se han encontrado dos elementos "mal colocados". Los intercambiamos y volvemos a empezar.
   - Este algoritmo acabará cuando los dos punteros se crucen, habiendo quedado todos los elementos ordenados según el criterio inicial.

   Escriba un trozo de código que declare una constante (`TOPE`) con valor 20 y un vector de reales con ese tamaño, lo rellene con números aleatorios entre 0 y 100 y lo reorganice usando el algoritmo antes descrito.

7. Las cadenas de caracteres (tipo "C", o cadenas "clásicas") son una buena fuente para ejercitarse en el uso de punteros. Una cadena de este tipo almacena un número indeterminado de caracteres (para los ejercicios bastará un valor siempre menor que 100) delimitados al final por el *carácter nulo* (`'\0'`).

   Escriba un trozo de código que lea una cadena y localice la posición del primer *carácter espacio* (`' '`) en una cadena de caracteres "clásica". El programa debe indicar su posición (0: primer carácter, 1: segundo carácter, etc.).

Notas:

- La cadena debe recorrerse usando aritmética de punteros y sin usar ningún entero.
- Usar la función `getline()` para la lectura de la cadena (Cuidado: usar el método público de `istream` sobre `cin`,o sea `cin.getline()`). Ver `http://www.cplusplus.com/reference/iostream/istream/getline/`

8. Consideremos una cadena de caracteres "clásica". Escriba un trozo de código que lea una cadena y la imprima pero saltándose la primera palabra, *evitando escribirla carácter a carácter*.

   Considere que puede haber una o más palabras, y si hay más de una palabra, están separadas por espacios en blanco.

9. Considere una cadena de caracteres "clásica". Escriba la función `longitud_cadena`, que devuelva un *entero* cuyo valor indica la longitud de la cadena: el número de caracteres desde el inicio hasta el carácter nulo (no incluído).

   Nota: No se puede usar el operador `[]`, es decir, se debe resolver mediante aritmética de punteros.

10. Escriba una función a la que le damos una cadena de caracteres y calcule si ésta es un palíndromo.

    Nota: No se puede usar el operador `[]`, es decir, se debe resolver mediante aritmética de punteros.

11. Considere dos cadenas de caracteres "clásicas". Escriba la función `comparar_cadenas`, que devuelve un valor *entero* que se interpretará como sigue: si es *negativo*, la primera cadena es más "pequeña"; si es *positivo*, será más "grande"; y si es *cero*, las dos cadenas son "iguales".

    Nota: Emplead como criterio para determinar el orden el código ASCII de los caracteres que se están comparando.

12. Considere dos cadenas de caracteres "clásicas". Escriba la función `copiar_cadena`, que copiará una cadena de caracteres en otra. El resultado de la copia será el primer argumento de la función. La cadena original (segundo argumento) **no** se modifica.

    Nota: Se supone que hay suficiente memoria en la cadena de destino.

13. Considere dos cadenas de caracteres "clásicas". Escriba la función `encadenar_cadena`, que añadirá una cadena de caracteres al final de otra. El resultado se dejará en el primer argumento de la función. La cadena que se añade (segundo argumento) **no** se modifica.

    Nota: Se supone que hay suficiente memoria en la cadena de destino.

14. Escriba una función a la que le damos una cadena de caracteres, una posición de inicio `p` y una longitud `l` sobre esta cadena. Queremos obtener una *subcadena* de ésta, que comienza en `p` y que tiene longitud `l`.

    Notas:

    - Si la longitud es demasiado grande (se sale de la cadena original), se devolverá una cadena de menor tamaño (la que empieza en `p` y llega hasta el final de la cadena).
    - No se puede usar el operador `[]`, es decir, se debe resolver mediante aritmética de punteros.

15. Escriba una función a la que le damos una cadena de caracteres. Queremos obtener una nueva cadena, resultado de invertir la primera.

    Notas:

    - La cadena original **no** se modifica.
    - No se puede usar el operador `[]`, es decir, se debe resolver mediante aritmética de punteros.

16. Escribir una función que recibe un vector de números enteros y dos valores enteros que indican las posiciones de los extremos de un intervalo sobre ese vector, y devuelve un puntero al elemento mayor dentro de ese intervalo.

    La función tendrá como prototipo:

    ```
    int * PosMayor (int *pv, int izda, int dcha);
    ```

    donde `pv` contiene la dirección de memoria de una casilla del vector e `izda` y `dcha` son los extremos del intervalo entre los que se realiza la búsqueda del elemento mayor.

    Considere la siguiente declaracón:

    ```
    const int TOPE = 100;
    int vector [TOPE];
    ```

    Escriba un programa que rellene aleatoriamente el vector (completamente, o una parte) y que calcule el mayor valor entre dos posiciones:

    a) Si el programa se ejecuta sin argumentos, se rellenará completamente (`TOPE` casillas) y se calculará el mayor valor de todo vector (entre las casillas 0 y `TOPE`-1).
    b) Si el programa se ejecuta con un argumento (`n`), se rellenarán `n` casillas, y calculará el mayor valor entre ellas (entre las casillas 0 y `n`-1).
    c) Si el programa se ejecuta con dos argumentos (`n` y `d`), se rellenarán `n` casillas, y calculará el mayor valor entre las casillas 0 y `d`.
    d) Si el programa se ejecuta con tres argumentos (`n`, `i` y `d`), se rellenarán `n` casillas, y calculará el mayor valor entre las casillas `i` y `d`.
    e) Si el programa se ejecuta con más de tres argumentos, muestra un mensaje de error y no hace nada más.

    Nota: Modularice la solución con funciones.

17. Escriba la función

```
void Ordena (int *vec, int **ptr, int izda, int dcha);
```

que reorganiza los punteros de `ptr` de manera que recorriendo los elementos referenciados por esos punteros encontraríamos que están ordenados de manera creciente.

- Los elementos referenciados por los punteros son los elementos del vector `vec` que se encuentran entre las casillas `izda` y `dcha`.
- Los elementos de `vec` no se modifican.

En la figura 21 mostramos el resultado de la función cuando se "ordena" el vector `vec` entre las casillas 0 y 5.

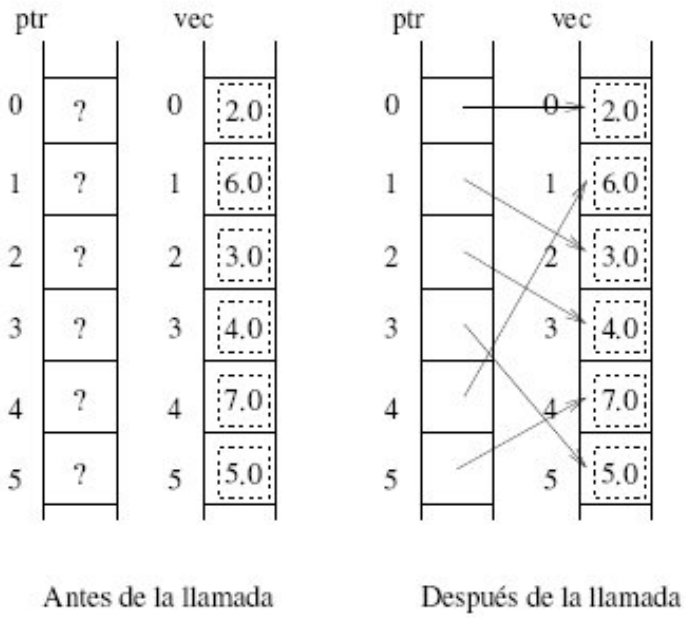

Figura 21: Resultado de reorganizar el vector de punteros

Observe que el vector de punteros debe ser un parámetro de la función, y estar reservado previamente a la llamada con un tamaño, al menos, igual al del vector.

```
const int TOPE = 50; // Capacidad

int   vec [TOPE];    // Array de datos
int * ptr [TOPE];    // Indice al array "vector"
```

Escriba un programa que haciendo uso de la función permita "ordenar" el vector entre dos posiciones:

- Si el programa se ejecuta sin argumentos "ordenará" todo vector.
- Si el programa se ejecuta con dos argumentos, "ordenará" el vector entre las dos posiciones dadas.
- En otro caso, muestra un mensaje de error y no hace nada más.

Nota: Iniciar aleatoriamente el vector `vec`.

Nota: Modularice la solución con funciones.

18. (*Este ejercicio está basado en el ejercicio 2 de la Relación de Problemas IV del Cuaderno de Prácticas de la asignatura* ***Fundamentos de Programación***)

    Se van a gestionar las calificaciones de una clase formada por un número indeterminado de alumnos, aunque no superior a cien. Se pretende calcular la nota media final de cada alumno en base a **cuatro** calificaciones parciales con diferente peso.

Paso 1. El programa leerá, en primer lugar, los cuatro pesos que se asignan a las calificaciones parciales (valores *reales* que indican porcentajes). Una vez leídos comprobará que efectivamente suman $\mathbf{100{,}0}$ y en el caso de que no lo fueran, se detendrá la ejecución del programa.

Paso 2. Leer para cada alumno: *apellidos*, *nombre*, y las cuatro *calificaciones*. Cada uno de los seis datos asociados a cada alumno se lee por separado[3]. Los datos se almacenan en memoria usando un *array*.

La lectura finaliza cuando se introduce el caracter `*` en la lectura de los apellidos de un alumno.

Paso 3. Calcular (y guardar) la nota media de cada alumno. Usar un *array* para guardar las calificaciones medias.

Paso 4. Finalmente, mostrar un listado en el que se detalla, para cada alumno: *apellidos y nombre* y *nota media*.

Para la resolución de este ejercicio proponemos usar el tipo `RegAlumno`:

```
struct RegAlumno {
    double notas[NUM_NOTAS];
    char   apellido_nombre[TAM_NOMBRE]; // Cadena "clásica"
};
```

(`NUM_NOTAS` será 4 según el enunciadpo, y `TAM_NOMBRE` será, por ejemplo, 80).

y las siguientes estructuras de datos para almanecenar y procesar los datos:

- `pesos`: array de exactamente `NUM_NOTAS` valores `double`: son los pesos de las calificaciones de los alumnos.

[3] en el caso de una lectura redirigida desde un fichero, cada dato estará en una línea diferente.

- `alumnos`: array de `TAMANIO` casillas (100, según el enunciado) de tipo `RegAlumno`. Se ocuparán `num_alumnos` casillas ($0 \leq$ `num_alumnos` $\leq$ `TAMANIO`)
- `calificacion_media`: array de `TAMANIO` valores `double`: las calificaciones medias de los alumnos. Se ocuparán `num_alumnos` casillas, el mismo número que en el array `alumnos`: son arrays **paralelos**.

Recomendación: Leer los datos utilizando la **redirección de entrada**. Usad para ello un fichero de texto como los disponibles en la página de la asignatura.

19. Modularizar con fuciones la solución desarrollada para el ejercicio 18 y ampliar su funcionalidad de manera que ahora:

    - Ordene (sentido creciente) el vector `alumnos` usando como clave la calificación media.
    - El programa se ejecute con argumentos desde la línea de órdenes de tal modo que:
      - Si se ejecuta sin argumentos, en el listado final (ordenado por calificación) muestra los datos de todos los alumnos.
      - Si se ejecuta con **dos** argumentos muestra los datos de los alumnos cuyas calificaciones están comprendidas (incluidas) entre los dos valores indicados.

    Escribir las siguientes funciones:

```
void LeePesos (double ponderaciones[], int num_pesos);
```

    Lee los pesos de las calificaciones y los guarda en el vector de pesos.

```
bool CorrectosPesos (double ponderaciones[], int num_pesos);
```

    Comprueba la validez de los pesos leidos: deben sumar 100.0.

```
void MuestraPesos (double ponderaciones[], int num_pesos);
```

    Muestra los pesos leídos para las ponderaciones de las calificaciones.

```
int  LeerAlumnos (RegAlumno datos[], int num_casillas);
```

    Lee los datos de los alumnos y los guarda en un vector.

```
void CalculaCalificacionesMedias (double medias[],
                      RegAlumno datos[], int num_alumnos,
                      double ponderaciones[], int num_notas);
```

    Calcula las calificaciones medias de los alumnos.

```
void MuestraVectorAlumnos (RegAlumno datos[],
                           double medias[], int num_alumnos);
```

Mostrar la lista de los alumnos con todas las calificaciones y su media.

```
void OrdenaVectorAlumnos (RegAlumno datos[],
                          double medias[], int num_alumnos);
```

Ordenar el vector de datos de alumnos en base a la nota media.

20. (*Este ejercicio está basado en el ejercicio 25 de la Relación de Problemas IV del Cuaderno de Prácticas de la asignatura* ***Fundamentos de Programación***)

    Se quiere monitorizar los datos de ventas semanales de una empresa que cuenta con varias sucursales. La empresa tiene 100 sucursales y dispone de un catálogo de 10 productos. Algunas sucursales no han vendido ningún producto, y algún producto no se ha vendido en ninguna sucursal.

    Cada operación de venta se registra con tres valores: el identificador de la sucursal (número entero, con valores desde 1 a 100), el código del producto (un carácter, con valores desde `a` hasta `j`) y el número de unidades vendidas (un entero).

    El programa debe leer un número *indeterminado* de datos de ventas: la lectura de datos finaliza cuando se encuentra el valor `-1` como código de sucursal. El programa sólo procesa datos de venta de las sucursales que hayan realizado operaciones de ventas, por lo que no todas las sucursales ni productos aparecerán.

    Recomendación: Leer los datos utilizando la **redirección de entrada**. Usad para ello un fichero de texto como los disponibles en la página de la asignatura.

    Después de leer los datos de ventas el programa mostrará:

    a) El número total de operaciones de venta.

    b) La sucursal que más unidades ha vendido y cuántas unidades.

    c) Listado en el que aparecerán: código de sucursal y número total de productos vendidos en la sucursal. Aparecerán únicamente las sucursales que hayan vendido algún producto.

    d) El número de sucursales que hayan vendido algún producto.

    e) El número total de unidades vendidas (calculado como suma de las ventas por sucursales).

    f) Producto más vendido y cuántas unidades.

    g) Listado en el que aparecerán: código de producto y número total de unidades vendidas. Aparecerán únicamente los productos que hayan tenido alguna venta.

    h) Cuántos tipos de producto han sido vendidos.

    i) El número total de unidades vendidas (calculado como suma de las ventas por producto).

    j) Tabla-resumen con toda la información. Por ejemplo:

```
    |     a     b     c     d     e     f     h
----------------------------------------------------
  1 |     2    14     2     0    20     0     0 |  38
  2 |    20    21     0     0     0     0     0 |  41
  3 |     0    11    49     5     0     0     0 |  65
  4 |     0     0     0    42    10     0     0 |  52
  5 |     3     0    10     0    20    23     4 |  60
  7 |     0    15     0    12     0     8     0 |  35
  9 |    10    44     0     0     0     0     0 |  54
 10 |     0     7    30     0     0     0     0 |  37
----------------------------------------------------
    |    35   112    91    59    50    31     4 | 382
```

Nota: Modularizar con fuciones la solución desarrollada, de manera que cada una de las tareas a realizar las lleve a cabo una función.

Para poder guardar en memoria la información requerida para los cálculos proponemos una matriz bidimensional `ventas` con tantas filas como número (máximo) de sucursales y columnas como número (máximo) de productos. La casilla (s, p) de esta matriz guardará en número total de unidades vendidas por la sucursal s del producto p (o el número de de unidades vendidas del producto p en la sucursal s).

No se conoce a priori el número de operaciones de venta. Tampoco se conoce el número de sucursales que se van a procesar, ni el código de éstas (algunas sucursales puede que no hayan realizado ventas). Se sabe que los códigos de sucursales son números entre 1 y 100.

Tampoco se conoce a priori los productos que se han vendido, ni el código de éstos (algunos productos puede que no hayan vendido). Se sabe que los códigos de producto son caracteres entre `'a'` y `'j'`.

Nota: Emplead datos `int` para los índices de las filas y `char` para las columnas, de manera que se accede a las ventas del producto `'b'` por la sucursal 3, por ejemplo, con la construcción: `ventas[3]['b']`.

Nota: Aconsejamos que una vez leídos los datos, y actualizada la matriz `ventas`:

a) Usad un vector, `ventas_sucursal`, con tantas casillas como filas tenga la matriz `ventas`. Guardará el número total de unidades vendidas por cada sucursal.

b) Usad un vector, `ventas_producto`, con tantas casillas como columnas tenga `ventas`. Guardará el número total de unidades vendidas de cada producto.