TPN°6: Arreglos

Ejercicio 1

```
        #define N 10
        #define M 20

        int
        main (void)
        {
            int vectorA [ M * N];
1           int vectorB [ -10 ];
2           int vectorC [ 10.0 ];

3           vectorC [2.5] = 'a';
            vectorB [-1] = 5;
            vectorA ['0'] = 10;
            vectorC [vectorA[48]] = 5.5;
            vectorA [1000] = 0;
            vectorA [M * N] = 10;
            return 0;
        }
```

1) El tamaño de un arreglo no puede ser negativo.
2) El tamaño de un arreglo no puede ser un número real.
3) El subíndice de un arreglo no puede ser un número real.

El resto de las instrucciones no genera errores de compilación, aunque sí pueden provocar errores en tiempo de ejecución.

Ejercicio 2

```
#include <stdio.h>
#include <math.h>
#include "getnum.h"

#define SIMPLIFICAR 1
#define SUMAR 2
#define SALIR 3

/* Función que lee una fracción de la entrada estándar */
void leerFrac(int * num, int * den);

/* Simplificación de la fracción representada por num y den */
void simplFrac(int * num, int * den);

/* Calcula la suma de dos fracciones representadas por num1 / den1, num2 / den2
 * Deja el resultado en numS / denS
 */
void sumarFrac(int num1, int num2, int den1, int den2, int* mumS, int* denS);

/* Devuelve el máximo común divisor de dos números */
int dcm (int num1, int num2);

/* Imprime una fracción */
void imprimeFrac(int num, int den);
```

```
/* Menú de opciones */
int menu(void);

int
main(void)
{
      int opcion;
      int num1, num2, den1, den2, numS, denS;

      do
      {
            opcion=menu();

            switch(opcion)
            {
            case SIMPLIFICAR:
                  printf("Ingrese fraccion a simplificar\n");
                  leerFrac(&num1, &den1);
                  simplFrac(&num1, &den1);
                  printf("Fraccion simplificada: ");
                  imprimeFrac(num1, den1);
                  break;
            case SUMAR:
                  printf("Ingrese la primera fraccion\n");
                  leerFrac(&num1, &den1);
                  printf("Ingrese la segunda fraccion\n");
                  leerFrac(&num2, &den2);
                  sumarFrac(num1, num2, den1, den2, &numS, &denS);
                  printf("Resultado: ");
                  imprimeFrac(numS, denS);
                  break;
            case SALIR: break;
            default: printf("\nOpción invalida"); break;
            }
      }
      while (opcion != 3);

      return 0;
}

int
menu(void)
{
      int opcion;

      printf("\n1 - Simplificar una fracción");
      printf("\n2 - Sumar dos fracciones");
      printf("\n3 - Terminar");

      opcion = getint("\nElija una opción:");

      return opcion;
}

void
leerFrac(int * num, int * den)
{
      /* Leer numerador */
      *num = getint("Ingrese numerador: ");

      /* Leer denominador */
      while ( (*den = getint("Denominador: ")) == 0 )
            printf("El denominador no puede ser cero.\n");
      return;
}
```

El denominador no puede ser cero.

```
void
simplFrac(int * num, int * den)
{
      int valor;

      /* Divide el numerador y el denominador por el mcd */
      if (abs(( valor = dcm(*num, *den))) != 1)
      {
            *num /= valor;
            *den /= valor;
      }
}

void
sumarFrac(int num1, int num2, int den1, int den2, int* numS, int* denS)
{
      /* Calcula la suma */
      *denS = den1 * den2;
      *numS = *denS / den1 * num1 + *denS / den2 * num2;

      /* Simplifica la fracción de respuesta */
      simplFrac(numS, denS);
}

int
dcm ( int num1, int num2)
{
      int auxi ;
      auxi = num1;
      while (auxi!=0)
      {
            num1 = num2;
            num2 = auxi;
            auxi = num1 % num2 ;
      }
      return num2;
}

void
imprimeFrac(int num, int den)
{

      printf("%s%d", (num*den >= 0)?"":"-", abs(num));
      if (abs(den)!=1)
            printf("/%d",abs(den));
      putchar('\n');
}
```

Ejercicio 3

```
#define EPSILON 0.000001

/* Calcula la máxima diferencia entre dos valores consecutivos de un arreglo de reales */
double
maxDiferencia(const double arreglo[])
{
    double dif= 0.0;
    int j;
    if ( fabs(arreglo[0]) > EPSILON )
           for (j=1; fabs(arreglo[j]) > EPSILON; j++)
                 if (fabs(arreglo[j] - arreglo[j-1]) > dif )
                       dif = fabs(arreglo[j] - arreglo[j-1]);

    return dif;
}
```

Otra versión correcta:

```
#define EPSILON 0.000001

double
maxDiferencia(const double arreglo[])
{
      double dif=0, difAux;
      int j=0;

      if ( fabs(arreglo[0]) > EPSILON )
            while (fabs(arreglo[++j]) > EPSILON)
                  if ( (difAux = fabs(arreglo[j] - arreglo[j-1])) > dif)
                        dif = difAux;
     return dif;
}
```

Se calcula una sola vez el valor absoluto de la diferencia.

¿Es correcto el siguiente código?

```
double
maxDiferencia(const double arreglo[])
{
      double dif;
      int j;

      for (dif=0, j=1; fabs(arreglo[j-1]) > EPSILON; j++)
            if (fabs(arreglo[j] - arreglo[j-1]) > diferencia )
                  dif = fabs(arreglo[j] - arreglo[j-1]);

      return dif;
}
```

Ejercicio 4

Proponemos primero una solución que, si bien funciona, no es aceptable por ser ineficiente.

```
int
eliminaRepetidos(const int original[], int dimOrig, int resultado[])
{
      int dimResult;
      int i,j, noEsta;

      /* Para cada elemento del vector original recorrer los elementos ya
      ** insertados en el vector final, si no estaba entonces agregarlo.
      */
      for (i=0,dimResult=0; i < dimOrig; i++)
      {
            noEsta = 1;
            for (j=0; j < dimResult; j++)
                  if (original[i] == resultado[j])
                        noEsta = 0;
            if (noEsta)
                  resultado[dimResult++] = original[i];
      }
      return dimResult;
}
```

- **¿Por qué no es necesario validar que dimOrig sea mayor a cero?**
- **¿Por qué no se copia el primer elemento fuera del ciclo?**

Solución eficiente

```
int
eliminaRepetidos( const int original[], int dimOrig, int resultado[])
{
      int dimResult;
      int i,j;
      int repetido;

      /* Para cada elemento del vector original recorrer los elementos ya
      ** insertados en el vector final, si no estaba entonces agregarlo.
      */
      for (i=0,dimResult=0; i < dimOrig; i++)
      {
            repetido = 0;
            for (j=0; j < dimResult && !repetido ; j++)
                  if (original[i] == resultado[j])
                        repetido = 1;
            if (!repetido)
                  resultado[dimResult++] = original[i];
      }
      return dimResult;
}
```

Detenemos el ciclo apenas se detecta que el elemento está repetido.

Ejercicio 5

Obviamente, la solución propuesta para el ejercicio anterior también funciona en el caso de vectores ordenados, pero sería ineficiente utilizarla para los mismos. Una solución correcta es comparar cada elemento con el que lo antecede y realizar la copia solo si son distintos.

```
int
eliminaRepOrden( const int original[], int dimOrig, int resultado[])
{
      int dimResult=0;
      int i;

      if (dimOrig > 0)
      {
            resultado[0] = original[0];
            for (i=1,dimResult=1; i < dimOrig; i++)
                  if (original[i] != original[i-1])
                        resultado[dimResult++] = original[i];
      }
      return dimResult;
}
```

Otra versión, también correcta: comparar contra el último elemento copiado.

```
int
eliminaRepOrden( const int original[], int dimOrig, int resultado[])
{
      int dimResult=0;
      int i;

      if (dimOrig > 0)
      {
            resultado[0] = original[0];
            for (i=1,dimResult=1; i < dimOrig; i++)
                  if (original[i] != resultado[dimResult-1])
                        resultado[dimResult++] = original[i];
      }
      return dimResult;
}
```

Ejercicio 6

Resolvemos este problema eligiendo para cada elemento del vector otro elemento al azar y se los intercambia. Otra aproximación, también correcta, sería elegir pares de elementos al azar e intercambiarlos.

```
void
desordenaArreglo( int arreglo[], int dim)
{
      int i, j, aux;

      randomize();

      for (i=0; i<dim; i++)
      {
            j = randInt(0, dim-1);
            aux = arreglo[i];
            arreglo[i] = arreglo[j];
            arreglo[j] = aux;
      }
      return;
}
```

Ejercicio 7

¿Por qué esta solución no es aceptable?

```
void
unirArreglos(const int ar1[], const int ar2[], int resultado[])
{
        int i, k, l, dimResultado;

        /* Copiamos el primer arrreglo */
        for(dimResultado=0; (resultado[dimResultado] = ar1[dimResultado]) != -1;
            dimResultado++)
                ;

        /* Insertamos cada elemento de ar2, si no estaba */
        for(i=0; ar2[i] != -1; i++)
        {
                for(k=0; resultado[k] != -1 && resultado[k] < ar2[i]; k++)
                        ;
                if(resultado[k] != ar2[i])
                {
                      /* Corremos todos los elementos para hacer lugar */
                      dimResultado++;
                      for(l=dimResultado; l != k; l--)
                             resultado[l] = resultado[l-1];

                      resultado[k] = ar2[i];
                }
        }
}
```

La unión se realiza en forma completa en un solo ciclo. Otra opción, también aceptable, sería hacer el ciclo mientras haya datos en ambos arreglos (terminar el ciclo si no hay más datos en uno de ellos), y luego de ese ciclo copiar directamente los elementos restantes del otro arreglo. Tener en cuenta que ambas soluciones recorren una sola vez cada arreglo.

```
#define MAXIMO 20
#define V_FINAL -1
enum { IGUAL, MENOR, MAYOR };
void unirArreglos ( const int ar1[], const int ar2[], int resultado[]);

/* Dados dos elementos de la lista, devuelve si el primero es menor, igual o mayor al
** segundo, considerando que V_FINAL indica fin de datos, y por lo tanto es el "máximo"
*/
int compara ( int clave1, int clave2);

void
unirArreglos ( const int ar1[], const int ar2[], int resultado[])
{
      int i=0, j=0;    /* índices del primer y segundo arreglo */
      int k=0;         /* índice de la unión ( arreglo resultado ) */

      while (( ar1[i]!=V_FINAL || ar2[j]!=V_FINAL ) && k < MAXIMO -1)
      {
            switch ( compara(ar1[i], ar2[j] ))
            {
            case IGUAL:
                  resultado[k] = ar1[i++];
                  j++;
                  break;
            case MENOR:
                  resultado[k] = ar1[i++];
                  break;
            case MAYOR:
                  resultado[k] = ar2[j++];
            }
            k++;
      }
      resultado[k] = V_FINAL;
}

int
compara ( int clave1, int clave2)
{
      if ( clave1 == clave2 )
            return IGUAL;
      if ( clave1 == V_FINAL )
            return MAYOR;
      if ( clave2 == V_FINAL )
            return MENOR;
      if ( clave1 > clave2 )
            return MAYOR ;
      return MENOR;
}
```

Ejercicio 8

```
#include <stdio.h>

#define MAXIMO 20
#define V_FINAL -1
enum { IGUAL, MENOR, MAYOR};

void unirArreglos ( const int arreglo1[], const int arreglo2[], int resultado[]);

/* La misma función de comparación del ejercicio anterior */
int compara ( int clave1, int clave2);

void
unirArreglos ( const int arreglo1[], const int arreglo2[], int resultado[])
{
      int i=0, j=0;         /* índices del primer y segundo arreglo */
      int k=0;              /* índice de la unión ( arreglo resultado ) */

      while (( arreglo1[i]!=V_FINAL || arreglo2[j]!=V_FINAL ) && k < MAXIMO -1)
      {
            /* Saltear elementos repetidos en arreglo1 */
            if ( arreglo1[i] != V_FINAL )
                  while ( arreglo1[i]==arreglo1[i+1] )
                        i++;

            /* Saltear elementos repetidos en arreglo2 */
            if ( arreglo2[j] != V_FINAL )
                  while ( arreglo2[j]==arreglo2[j+1] )
                        j++;
            switch ( compara(arreglo1[i], arreglo2[j] ))
            {
            case IGUAL:
                  resultado[k] = arreglo1[i++];
                  j++;
                  break;
            case MENOR:
                  resultado[k] = arreglo1[i++];
                  break;
            case MAYOR:
                  resultado[k] = arreglo2[j++];
            }
            k++;
      }

      resultado[k] = V_FINAL;
}
```

Ejercicio 9

Prototipos de las funciones:

```
/* Calcula la desviacion estandar */
double desv( char vector[], int cantnum );

/* nibbleh. Dado un char devuelve el nibble alto */
char nibbleh ( char num );

/* nibblel. Dado un char devuelve el nibble bajo */
char nibblel ( char num );

/* Recibe el vector y la cantidad de numeros. Devuelve el promedio */
double promedio ( const char vector[] , int cantnum );

/* Calcula la sumatoria de la desviacion estandar */
double sumatoria ( double promedio , const char vector[], int cantnum);
```

Presentamos primero una versión de la función promedio que funciona únicamente cuando la cantidad de elementos del vector es par.

```
double
promedio( const char vector[] , int cantnum )
{
      int i;
      double suma=0;

      for ( i=0 ; i < cantnum/2 ; i++ )
            suma += nibbleh( vector[i] ) + nibblel ( vector[i] ) ;

      return suma/cantnum;
}
```

```
double
desv( const char vector[] , int cantnum )
{
      double prom, sumat;

      prom=promedio( vector , cantnum );
      sumat=sumatoria( prom , vector , cantnum );

      return sqrt(sumat/cantnum);
}


double
promedio( const char vector[] , int cantnum )
{
      int i=0, cantAux;
      double suma=0;

      cantAux=cantnum;
      while ( cantAux-- > 0)
      {
            /* Tomamos el nibble alto */
            suma += nibbleh( vector[i] );

            /* Si corresponde tomamos el nibble bajo */
            if ( cantAux-- )
                  suma += nibblel ( vector[i] );
            i++;
      }
      return suma / cantnum;
}
```

```
double
sumatoria ( double promedio , const char vector[], int cantnum )
{
      int i=0;
      double acum=0;

      while ( cantnum-- > 0)
      {
            acum+=pow( nibbleh( vector [i] ) - promedio, 2);

            if ( cantnum-- )
                  acum+=pow( nibblel( vector [i] ) - promedio, 2);
            i++;
      }

      return acum;
}

char nibbleh ( char num )
{
      return nibblel ( num>>=4 ) ;
}

char nibblel ( char num )
{
      return num & 0x0F;
}
```

Ejercicio 10

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "random.h"

#define FILAS 20
#define COLUMNAS 60

/* Generar una matriz con valores al azar */
void generaMatriz(int cielo[][COLUMNAS], int filas, int columnas);

/* Dada una matriz que representa una zona estelar,
** grafica las estrellas en salida estandar */
void graficar (const int cielo[][COLUMNAS], int filas, int columnas);

int
main (void)
{
      int cielo[FILAS][COLUMNAS];

      /* Setear la semilla de números aleatorios */
      randomize();

      generaMatriz(cielo, FILAS, COLUMNAS);
      graficar(cielo, FILAS, COLUMNAS);
      return 0;
}


void
generaMatriz(int cielo[][COLUMNAS], int filas, int columnas)
{
      int i,j;

      /* Generar una matriz en forma aleatoria */
      for (i=0; i< filas; i++)
            for (j=0; j < columnas; j++)
                  cielo[i][j] = randInt(0, 20);
}


void
graficar (const int cielo[][COLUMNAS], int filas, int columnas)
{
      int i,j,k,l,suma;

      for (i=1; i < filas-1; i++)
      {
            printf("\n ");
            for (j=1; j < columnas-1; j++)
            {
                  suma = 0;
                  for (k=-1; k<=1; k++)
                        for (l=-1; l<=1; l++)
                              suma += cielo[i+k][j+l];
                  if (suma / 9 > 10 )
                        putchar('*');
                  else
                        putchar(' ');
            }
      }
}
```

Ejercicio 11

La siguiente implementación funciona correctamente, pero a la misma se le pueden hacer algunas críticas. Recomendamos analizar el código y descubrir qué es lo que está mal antes de leer nuestros comentarios. Esta solución fue propuesta por un alumno.

```
void
ordenaMatriz (int matriz[][COLS], int fils, int cols, int colOrden )
{
    int i, j;

    for ( i = fils - 1; i > 0 && !ordenado ( matriz, fils, colOrden ); i-- )
       for ( j = 0; j < i; j++ )
            if ( matriz[j][colOrden-1] > matriz[j+1][colOrden-1] )
                swapFil ( matriz, cols, j, j+1 );

    return;
}

int ordenado ( int matriz[][COLS], int fils, int colOrden )
{
    int i, flag = 1;

    for ( i = 0; i < fils && flag; i++ )
        if ( matriz[i][colOrden - 1] > matriz[i+1][colOrden - 1] )
            flag = 0;

    return flag;
}

void swapFil ( int matriz[][COLS], int cols, int fila1, int fila2 )
{
    int i, aux;

    for ( i = 0; i < cols; i++ )
    {
        aux = matriz[fila1][i];
        matriz[fila1][i] = matriz[fila2][i];
        matriz[fila2][i] = aux;
    }

    return;
}
```

1) En la función ordenaMatriz, no es eficiente preguntar en cada paso si está ordenado o no ya que se vuelve a recorrer el vector. Teniendo en cuenta que se aplica burbujeo, el criterio sería no seguir el ciclo si en un paso no hubo que intercambiar filas.
2) La función swapFil intercambia filas, por lo que podría ser genérica. Así como está escrita sirve únicamente para intercambiar filas de una matriz de COLS columnas. Teniendo en cuenta que cada fila es un vector, se podría hacer una función que reciba dos vectores, la dimensión de ambos e intercambie sus elementos.

A continuación las soluciones propuestas por la Cátedra

```
/* Función que intercambia los valores de dos filas */
void
intercambiaFilas(int fila1[],int fila2[], int col)
{
      int i,aux;

      for (i=0; i<col; i++)/* Recorre todas las columnas */
      {
            aux = fila1[i];
            fila1[i] = fila2[i];
            fila2[i] = aux;
      }
      return;
}

/* Ordena por el método de burbujeo */
void
ordenaMatriz (int matriz[][MAXCOL], int fil, int col, int colOrd)
{
      int i,j;

      colOrd--;  /* Resto uno porque la 1era. columna debe ser la 0 */
      for (i=0; i < fil-1 ; i++)
            for (j=0; j<fil-1-i; j++) /* Cada ciclo interno tiene una vuelta menos */
                  if (matriz[j][colOrd]>matriz[j+1][colOrd])
                        intercambiaFilas(matriz[j],matriz[j+1], col);
      return;
}
```

La siguiente es otra versión de la función OrdenaMatriz que es más eficiente que la anterior. No utiliza el método de burbujeo sino que recorre todas las filas de la matriz buscando la fila que debería quedar en esa posición, evitando mover tantos elementos en memoria.

```
void
ordenaMatriz (int matriz[][MAXCOL], int fil, int col, int colOrd)
{
      int i,j, menor = 0;

      colOrd--;
      for (i=0; i < fil-1 ; i++)  /* Recorro las filas comenzando por la 0 */
      {
            menor = i;   /* Busco la menor, que es la que debería quedar allí */
            for (j= menor+1; j<fil; j++)
                  if (matriz[j][colOrd] < matriz[menor][colOrd])
                        menor = j;

            if (i != menor)   /* Si la menor no está bien ubicada, la ubico */
                  intercambiaFilas(matriz[i],matriz[menor], col);
      }
      return;
}
```

Ejercicio 12

```
void
traspuesta( int matriz[][DIMMAX], int dim)
{
      int i,j,aux;

      for (i=0; i<dim; i++)
            for (j=0; j<i; j++)
            {
                  aux = matriz[i][j];
                  matriz[i][j] = matriz[j][i];
                  matriz[j][i] = aux;
            }
      return;
}
```

Ejercicio 13

Si en el enunciado no hubiera pedido restricción en cuanto a la cantidad de ciclos anidados, la solución bien podría ser la siguiente.

```
void
productoMat( const int m1[][MAX], const int m2[][MAX], int m3[][MAX], dim)
{
      int i,j,k;

      for (i = 0; i < dim; i++)
            for (j = 0; j < dim; j++)
                  for( k = 0, m3[i][j] = 0; k < dim; k++)
                        m3[i][j] += m1[i][k] * m2[k][j];

}
```

Una forma de modularizarlo es separar la multiplicación de una fila por una columna.

```
int
filaPorCol(const int fila[], const int mat2[][MAX], int col, int dim)
{
      int i;
      int total = 0;

      for(i=0; i<dim; i++)
            total += fila[i]* mat2[i][col];
      return total;
}

void
productoMat( const int m1[][MAX], const int m2[][MAX], int m3[][MAX], int dim)
{
      int i,j;

      for (i=0; i<dim; i++)
            for (j=0; j<dim; j++)
                  m3[i][j] = filaPorCol(m1[i], m2, j, dim);

      return;
}
```

Otra forma de modularizar: Separando la multiplicación de un vector por una matriz y dejando el resultado en un tercer vector.

```
void
multVector( const int vec[], const int matriz[][MAX], int vecFinal[], int dim)
{
     int j,k;
     for (j = 0; j < dim; j++)
          for( k = 0, vecFinal[j] = 0; k < dim; k++)
               vecFinal[j] += vec[k] * matriz[k][j];
}

void
productoMat( const int m1[][MAX], const int m2[][MAX], int m3[][MAX], int dim)
{
     int i;
     for (i=0; i < dim; i++)
          multVector(m1[i], m2, m3[i], dim);
}
```

Ejercicio 14

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

typedef unsigned int uInt ;

#define TRUE 1
#define FALSE !TRUE

/* Recibe un string con el formato dd/mm/yyyy y retorna por parámetro el día, mes
** y año de la misma. En caso de que el formato sea inválido
** o la fecha incorrecta, retorna FALSE y no altera los parámetros
*/
int valorFecha( const char * fecha, uInt *dia, uInt *mes, uInt * anio);

 /* Devuelve TRUE si el año es bisiesto y FALSE en caso contrario */
int esBisiesto(uInt anio);

/* Función auxiliar que verifica que un string sea de la forma dd/mm/yyyy */
static int
valFormato( const char * fecha)
{
     int resp = TRUE, i;

     /* Validamos primero la longitud y los separadores */
     resp = (fecha[2] == '/' && fecha[5] == '/' && strlen(fecha)==10 );

     /* Validamos ahora que el resto sean digitos */
     if ( resp )
          for (i=0 ; fecha[i] && resp; i++)
               if ( i != 2 && i !=5 && !isdigit(fecha[i]))
                    resp = FALSE;
     return resp;
}
```

```
int
esBisiesto(uInt anio)
{
      return ( (anio % 4 == 0 && anio % 100 != 0) || anio & 400 == 0 );
}

/* Funcion auxiliar que extrae el dia, mes y año de un string con el formato
** dd/mm/yyyy ya validado
*/
static int
extraeFecha( const char * fecha, uInt *dia, uInt *mes, uInt *anio)
{
   /* Usamos dos vectores auxiliares en los cuales se almacenan los días de cada mes.
   ** El primer vector para los años no bisiestos y el segundo para los años bisiestos.
   */
      static int diasMes[][12] = {{31,28,31,30,31,30,31,31,30,31,30,31},
                                  {31,29,31,30,31,30,31,31,30,31,30,31}};
      int fechaOK = FALSE;

      *dia = atoi(fecha);
      *mes = atoi(fecha + 3);
      *anio = atoi(fecha + 6);

      if ( *dia >0 && *mes >=1 && *mes <=12)
            fechaOK = *dia <= diasMes[esBisiesto(*anio)][*mes-1] ;

      return fechaOK;
}

int
valorFecha( const char * fecha, uInt *dia, uInt *mes, uInt * anio)
{
      int fechaOK;
      uInt lDia, lMes, lAnio;

      fechaOK = valFormato(fecha) && extraeFecha(fecha,&lDia,&lMes,&lAnio);

     /* Si la fecha es valida, actualizar los parametros de salida. Caso contrario
     ** quedan con los valores al momento de la invocación
     */
      if ( fechaOK)
      {
            *dia  = lDia;
            *mes  = lMes;
            *anio = lAnio;
      }
      return fechaOK;
}
```

Otra forma de implementar la función valFormato. En lugar de recorrer la cadena fecha para calcular su longitud y luego recorrerla para verificar los dígitos, se la recorre una sola vez, verificando al final del recorrido que su longitud sea igual a 10.

```
static int
valFormato( const char * fecha)
{
      int resp = TRUE, i;

      /* Validamos primero los separadores */
      resp = (fecha[2] == '/' && fecha[5] == '/' );

      /* Validamos ahora que el resto sean digitos y su longitud igual a 10 */
      if ( resp )
            for (i=0 ; i<10 && fecha[i] && resp; i++)
                  if ( i != 2 && i !=5 && !isdigit(fecha[i]))
                        resp = FALSE;

      return resp && (i==10);
}
```

Ejercicio 15

Una posible solución sería implementar el siguiente algoritmo: buscar un blanco, si a continuación también hay blancos correr todos los caracteres hacia la izquierda, seguir buscando secuencias repetidas mientras no se termine el string. Obviamente esta solución es inaceptable pues recorre demasiadas veces el mismo string.

En esta versión utilizamos un vector auxiliar, asumiendo que el arreglo de entrada tiene a lo sumo 200 caracteres. Se copian todos los caracteres, excepto los blancos que estén de más, al vector auxiliar y luego se copia el vector auxiliar sobre el vector original.

```
void
eliminaBlancos (char cadena[])
{
      unsigned int i,j;
      char aux[200], esBlanco = 0;  /* indica si el caracter anterior era un blanco */

      /* Recorrer los string con los índices:
      ** i: índice de cadena para recorrer todos los caracteres hasta el final
      ** j: índice del vector auxiliar, al cual se van copiando los caracteres.
      ** Este índice no se incrementa cuando hay blancos repetidos
      */
      for (i = j = 0; cadena[i] ; i++)
      {
            /* Si el caracter es un blanco y el anterior también lo era,
            ** no incrementar el índice actual */
            if ( cadena[i] == ' ' )
            {
                  if ( ! esBlanco )
                        aux[j++] = cadena[i];
                  esBlanco = 1;
            }
            else
            {
                  esBlanco = 0;
                  aux[j++] = cadena[i];
            }
      }
      /* Colocar el 0 de fin de string */
      aux[j] = 0;

      /* Copiar ahora sobre el vector original hasta encontrar el cero final */
      for ( i=0; cadena[i] = aux[i]; i++);
}
```

El código anterior presenta varios problemas, en caso de que el arreglo de entrada tenga más de 200 caracteres el programa pisa memoria, además se puede apreciar fácilmente que el arreglo auxiliar es totalmente innecesario, ya que la copia se puede hacer directamente sobre el vector original. La siguiente implementación corrige todos esos errores.

```
void
eliminaBlancos (char cadena[])
{
      unsigned int i,j;
      char esBlanco = 0;    /* indica si el caracter anterior era un blanco */

      /* Recorrer el string con dos índices:
      ** i: índice "original", para recorrer todos los caracteres hasta el final
      ** j: índice "actual", al cual se van copiando los caracteres. Este índice no
      **    se incrementa cuando hay blancos repetidos
      */
      for (i = j = 0; cadena[i] ; i++)
      {
            /* Si el caracter es un blanco y el anterior también lo era,
            ** no incrementar el índice actual */
            if ( cadena[i] == ' ' )
```

```
            {
                  if ( ! esBlanco )
                  {
                        cadena[j++] = cadena[i];
                        esBlanco = 1;
                  }
            }
            else
            {
                  esBlanco = 0;
                  cadena[j++] = cadena[i];
            }
      }
      /* Colocar el 0 de fin de string en el índice actual */
      cadena[j] = 0;
}
```

Por último, la siguiente versión se comporta de la misma forma con menos instrucciones ya que prescinde del flag ***esBlanco***.

```
void
eliminaBlancos (char cadena[])
{
      unsigned int i,j;

      /* Recorrer el string con dos índices:
      ** i: índice "original", para recorrer todos los caracteres hasta el final
      ** j: índice "actual", al cual se van copiando los caracteres. Este índice no
      **    se incrementa cuando hay blancos repetidos
      */
      for (i = j = 0; cadena[i] ; i++)
      {
            /* Si es el 1er. caracter ó no hay dos blancos seguidos, se copia */
            if ( i==0 || !(cadena[i] == ' ' && cadena[i-1] == ' '))
                  cadena[j++] = cadena[i];
      }

      /* Colocar el 0 de fin de string en el índice actual */
      cadena[j] = 0;
      return 0;
}
```

Para pensar: ¿Por qué falla la siguiente invocación?

```
int
main(void)
{
      char * cadena = "Hola    .";
      eliminaBlancos(cadena);
      return 0;
}
```

Ejercicio 16

El vector de respuesta tiene una dimensión máxima de dimMax, por lo tanto se le pueden copiar hasta dimMax-1 caracteres, dejando un lugar para el cero final.

```
int
copiaSubVector(const char * arregloIn, char * arregloOut, int desde, int hasta,
                                                                    int dimMax)
{
      int i, dim=0;

      /* verificar que 'desde' esté dentro del vector */
      if (strlen(arregloIn) <= desde )
            return 0;

      if ( desde < 0 || hasta < desde || dimMax <= 0 )
            return 0;

      for (i=desde; dim < dimMax-1 && i<=hasta && arregloIn[i]!= 0; i++)
            arregloOut[dim++] = arregloIn[i];

      arregloOut[dim] = 0;

      return dim;
}
```

En el caso anterior, verifica primero si el argumento "desde" se encuentra dentro del vector y luego verifica si es negativo o mayor a "hasta", por lo tanto recorre todo el string incluso si los parámetros son inválidos. Esto se arregla en la siguiente versión.

```
int
copiaSubVector(const char * arregloIn, char * arregloOut, int desde, int hasta,
                                                                    int dimMax)
{
      int i, dim=0;

      if ( desde < 0 || hasta < desde || dimMax <= 0 )
            return 0;

      /* verificar que 'desde' esté dentro del vector */
      if (strlen(arregloIn) <= desde )
            return 0;

      for (i=desde; dim < dimMax-1 && i<=hasta && arregloIn[i]!= 0; i++)
            arregloOut[dim++] = arregloIn[i];

      arregloOut[dim] = 0;

      return dim;
}
```

Una versión más eficiente, sin recorrer inicialmente ***todo*** el string para calcular su longitud.

```
int
copiaSubVector(const char * arregloIn, char * arregloOut, int desde, int hasta,
                                                                    int dimMax)
{
      int i, dim=0;

      if ( desde < 0 || hasta < desde || dimMax <= 0 )
            return 0;

      /* verificar que 'desde' esté dentro del vector */
      for(i=0; i<desde && arreglo[i]; i++)
            ;
      if ( i < desde )
            return 0;

      for ( ; dim < dimMax-1 && i<=hasta && arregloIn[i]!= 0; i++)
            arregloOut[dim++] = arregloIn[i];

      arregloOut[dim] = 0;

      return dim;
}
```