# ***TP N°6: Arreglos***

***Ejercicio 1***



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***

Para poder separar el backend del front, tanto simplificar como sumar deben modificar las fracciones recibidas, y no imprimir. Para ello almacenamos una fracción en un vector de dos posiciones, donde la primera posición es el numerador y la segunda el denominador. También se puede hacer usando punteros, que es la forma correcta de hacerlo.

***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;

}

¿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 (resuelto en videos)***

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;

}

Aunque también se puede hacer mejor, modularizando. Ver código en las soluciones.

***Ejercicio 5 (resuelto en videos)***

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 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);

El código lo tienen en el drive.***Ejercicio 11 (resuelto en video)***

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 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( int m1[][MAX], 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[], 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( int m1[][MAX], 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[], 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( int m1[][MAX], 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***

Una buena práctica consiste en modularizar y hacer primero una función que indique si todos los elementos de un vector están incluidos en otro. Luego se necesita, en el peor de los casos, invocar a esa función dos veces para lograr la funcionalidad pedida.

int

**contiene**(const int \* v1, int dim1, const int \* v2, int dim2)

{

if(contieneAux(v1, dim1, v2, dim2))

return 1;

if(contieneAux(v2, dim2, v1, dim1))

return 2;

return 0;

}

int

**contieneAux**(const int \* v1, int dim1, const int \* v2, int dim2)

{

int i, j, encontrado;

for(i = 0, encontrado = 1; i < dim1 && encontrado; i++){

for(j = 0, encontrado = 0; j < dim2 && !encontrado; j++){

if(v1[i] == v2[j]) {

encontrado = 1;

}

}

}

return encontrado;

}

Otra opción consiste en modularizar la función **contieneAux** y escribir una función auxiliar **pertenece** para el segundo ciclo *for*.

int

**contiene**(const int \* v1, int dim1, const int \* v2, int dim2)

{

if(contieneAux(v1, dim1, v2, dim2))

return 1;

if(contieneAux(v2, dim2, v1, dim1))

return 2;

return 0;

}

int

**contieneAux**(const int \* v1, int dim1, const int \* v2, int dim2)

{

int i, encontrado;

for(i = 0, encontrado = 1; i < dim1 && encontrado; i++){

encontrado = pertenece(v1[i], v2, dim2);

}

return encontrado;

}

int

**pertenece**(int elem, const int \* v, int dim) {

int i;

for(i = 0; i < dim; i++){

if(elem == v[i]) {

return 1;

}

}

return 0;

}

***Ejercicio 16***

***Errores más comunes cometidos:***

* Eficiencia: Seguir recorriendo la matriz cuando ya se sabe que la matriz no es ascendente ni descendente.
* Eficiencia: En lugar de consultar los flags en cada comparación, usar dos contadores con cantidad de comparaciones que dieron menor y cantidad de comparaciones que dieron mayor. Luego si la cantidad de comparaciones que dieron menor es igual a FILS \* (COLS-1) entonces la matriz es descendente. De esta forma siempre se recorre la matriz de principio a fin, resultando menos eficiente que la solución propuesta.
* Asociar la situación de igualdad con sólo ascendente o descendente cuando sirve para ambas. Y para los que utilizaron contadores, en la igualdad no aumentar ambos.
* Acceder a posiciones inválidas. Si en el código se accede a m[i][j+1] entonces j < COLS - 1.
* Realizar dos funciones por separado esAscendente y esDescendente. Repite código ya que sólo difiere en una línea de comparación de elementos. De no usar correctamente los flags también puede afectar eficiencia recorriendo la matriz dos veces.
* No comparar el primer elemento de una fila con el último elemento de la fila anterior.

***Ejercicio 23***

#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 24***

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 25***

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;

}

***Ejercicio 26***

a) A continuación se presenta una primera versión de la función **insertaDesde**. Notar que el parámetro s2 tiene el calificador *const* ya que nuestra función nunca debería modificar el segundo string.

void

**insertaDesde**(char \* s1, const char \* s2, char c)

{

while(\*s1 && \*s1 != c) {

s1++;

}

if(\*s1 == c) {

while(\*s2) {

\*s1++ = \*s2++;

}

}

}

Una mejor solución consiste en separar la tarea a realizar en dos funciones:

La función **char \* encuentraCaracter(char \* s, char c)** localiza la primera aparición de c en la cadena de caracteres s.

La función **char \* copiaCadena(char \* d, const char \* f)** copia la cadena apuntada por f a la cadena apuntada por d.

void

**insertaDesde**(char \* s1, const char \* s2, char c)

{

char \* pos = encuentraCaracter(s1, c);

if(pos != NULL)

copiaCadena(pos, s2);

}

char \*

**encuentraCaracter**(char \* s, char c) {

while(\*s) {

if(\*s == c) {

return s;

}

s++;

}

return NULL;

}

char \*

**copiaCadena**(char \* d, const char \* f) {

while(\*f) {

\*d++ = \*f++;

}

return d;

}

Mientras la consigna no prohíba el uso de las funciones de la biblioteca estándar, se recomienda su uso cuando la funcionalidad pedida por la consigna se ajusta a la de una o más funciones ya implementadas en la biblioteca.

La función de biblioteca estándar **char \* strchr(const char \* s, char c)** localiza la primera aparición de c en la cadena de caracteres s.

La función de biblioteca estándar **char \* strcpy(char \* d, const char \* f)** copia la cadena apuntada por f (incluyendo el carácter nulo) a la cadena apuntada por d.

De esta forma, utilizando las funciones anteriores, el cuerpo de la función **insertaDesde** es de sólo tres líneas y no se necesitan implementar funciones auxiliares.

void

**insertaDesde**(char \* s1, const char \* s2, char c)

{

char \* pos = strchr(s1, c);

if(pos != NULL)

strcpy(pos, s2);

}

Se recomienda tener presente las distintas funciones de la String Library que fueron presentadas en la clase teórica **12. Cadenas de Caracteres**.

b)

El siguiente ejemplo falla en ejecución porque la cadena de caracteres **str1 es inmutable**.

int

main(void)

{

char \* str1 = "manuel";

char str2[] = "javi";

insertaDesde(str1, str2 ,'n');

printf("%s", str1);

return 0;

}

***Ejercicio 30***

1. Falso, no es necesario indicar un nombre
2. No, TDay no es una variable, es un alias para un entero, que nos permitirá definir variables enteras que deberían tomar solo los valores enumerados
3. No, TVec no es un vector, es un alias para el tipo de dato vector de 10 enteros
4. No, m1 tiene 5 elementos, donde cada elemento es un vector de 10 enteros ( o "matriz" de 5 x 10), mientras que m2 es un vector de 10 elementos, donde cada elemento es un vector de 5 enteros (o "matriz" de 10 x 5 )
5. Verdadero
6. No, por lo explicado en el punto d)
7. No, ya que se le puede asignar cualquier valor entero (lo cual seguramente ocasionará errores de ejecución por asignar valores incorrectos).
8. Verdadero
9. Verdadero. TVec es un tipo de datos, por lo tanto no reservará memoria

***Ejercicio 32***

Presentamos a continuación una primera solución, en la cual antes de proceder a examinar el número validamos si es correcto y si es un valor distinto de cero.

El cálculo de la potencia se hace en base a la distancia entre el punto decimal del número original y el punto decimal de la respuesta, considerando si se eliminaron ceros.

int

numeroCorrecto(const char \* numero)

{

int huboPunto = 0, respuesta = 1;

while ( respuesta && \*numero )

{

if ( \*numero == '.' )

if ( huboPunto )

respuesta = 0;

else

huboPunto = 1;

else

respuesta = isdigit( \*numero );

numero ++;

}

return respuesta;

}

char \*

CopiaOriginal(const char \* original)

{

char \* aux;

int i;

*/\* Se reserva espacio para la copia \*/*

if ((aux=malloc(strlen(original)+1)) == NULL)

return NULL;

*/\* Se ignoran los ceros a izquierda antes del punto \*/*

while (\*original == '0')

original++;

*/\* Se hace la copia \*/*

strcpy(aux, original);

*/\* Se ignoran los ceros a derecha después del punto (al final)\*/*

i = strlen(aux) - 1;

while (i >= 0 && aux[i] == '0')

aux[i--]= 0;

*/\* Se elimina el punto si quedó al final \*/*

if (i >= 0 && aux[i] == '.')

aux[i]= 0;

return aux;

}

int

normalizar(const char \* original, char \* salida)

{

int mayorUno; */\* Indica si el número es mayor(o igual) o menor que uno \*/*

int potencia=0, j=0;

char \* num, \* prim, \*punto;

*/\* Inicializa la salida con el string vacio \*/*

salida[j] = 0;

*/\* Si el número no es correcto, no se puede hacer una copia o la copia es string vacio \*/*

if ( !numeroCorrecto(original) ||

((num=CopiaOriginal(original)) == NULL) || !\*num )

return 0;

*/\* Se busca la posición del punto, si no está se lo supone al final de la cadena \*/*

punto = strchr(num, '.');

if (punto==NULL)

punto = num + strlen(num);

*/\* Se busca el primer dígito significativo \*/*

prim = strpbrk(num,"123456789");

*/\* La potencia depende de la posición del punto y el primer dígito significativo \*/*

if (mayorUno = (\*num != '.'))

potencia = punto - prim - 1;

else

potencia = prim-num;

*/\* Se copian los dígitos \*/*

salida[j++] = \*prim++;

if (\*prim)

{

for(salida[j++] = '.'; \*prim; prim++)

{

if (\*prim != '.')

salida[j++]=\*prim;

}

}

*/\* Se arma la expresión final, con la potencia normalizada \*/*

sprintf(salida+j," E%c%02d", (mayorUno)? '+' : '-', potencia);

return 1;

}

En esta versión no verificamos que el número sea correcto antes de comenzar la transformación sino a medida que procesamos el número. Si bien el algoritmo recorre una sola vez el string, el código es menos claro y modular que en el ejemplo anterior (y supera la cantidad de líneas solicitadas).

int

normalizar(const char \* original, char \* salida) {

int mayorUno=1, potencia=0, j=0, sumarPotencia=1;

salida[j] = 0;

while (\*original == '0')

original++;

if ( \*original != '.' && !isdigit(\*original))

return 0;

if ( \*original == '.' ) {

original ++;

for ( mayorUno = 0, potencia = 1; \*original == '0'; original++)

potencia ++;

sumarPotencia = 0;

}

salida[j] = \*original++;

if ( \*original != 0 )

salida[++j] = '.';

while ( \*original != 0 )

{

if ( \*original == '.' )

{

if ( ! sumarPotencia )

{

salida[0] = 0;

return 0;

}

sumarPotencia = 0;

}

else

{

if ( !isdigit(\*original))

{

salida[0] = 0;

return 0;

}

salida[++j] = \*original;

}

if ( sumarPotencia )

potencia ++;

original++;

}

while ( j >=0 && salida[j] == '0' )

j--;

if ( j >= 0 && salida[j] == '.' )

j--;

sprintf(salida+j+1,"E%c%02d", ( mayorUno )? '+' : '-', potencia);

return 1;

}

En esta tercera versión se realiza el procesamiento en un solo ciclo, haciendo el código más compacto y claro. Si bien la conversión es correcta, presenta en su funcionamiento un defecto y se deja como ejercicio para el alumno detectarlo y corregirlo.

int

normalizar(const char \* original, char \* salida)

{

int mayorUno=1, potencia=0, digitoSig = 0, huboPunto = 0;

if ( ! numeroCorrecto(original) )

return 0;

while ( \*original )

{

if (isdigit(\*original))

{

if ( digitoSig )

{

\*salida++ = \*original++;

if (! huboPunto )

potencia ++;

}

else if ( \*original != '0' )

{

digitoSig = 1;

\*salida++ = \*original ++;

\*salida++ = '.';

}

else

{

if ( !mayorUno )

potencia++;

original++;

}

}

else

{

huboPunto = 1;

original ++;

if ( !digitoSig )

{

mayorUno = 0;

potencia = 1;

}

}

}

sprintf(salida,"E%c%02d", (mayorUno)? '+' : '-', potencia);

return 1;

}

***Ejercicio 33***

Ver el código en la solución. Para decodificar debemos recorrer el vector hasta encontrar el carácter codificado, y en base a la posición que ocupa calcular la letra correspondiente. ¿Qué habría que cambiar en el diseño si quisiéramos que también sea de O(1) la decodificación de un carácter?