



## Práctica: Estructuras

### Contenido:

Esta práctica está diseñada para que el estudiante comience a programar usando las estructuras de datos utilizadas más comúnmente en aplicaciones más cercanas a las que se enfrentará como ingeniero electrónico.

1) Dada la siguiente estructura que permite representar tiempos:

```
typedef struct tiempo{  
    int anio, mes, dia, hora, minuto, segundo;  
} Tiempo;
```

Implemente las siguientes funciones:

1. `int compara_tiempos( Tiempo *t1, Tiempo * t2);`  
que retorna:

1	si *t1 es anterior a *t2
0	si *t1 es igual a *t2
-1	si *t1 es posterior a *t2

2. `void imprime_tiempo( Tiempo t );`  
que imprime el contenido de la estructura `t` con el siguiente formato: "**dia/mes/año**  
**hora:minuto:segundo**" (por ejemplo: "3/8/1974 18:23:59").

Para representar un archivo se escogió la siguiente estructura que utiliza la anterior:

```
typedef struct {  
    char * nombre;  
    Tiempo ultima_mod;  
} Archivo;
```

Donde, **nombre** es un cadena de caracteres con el nombre del archivo y **ultima\_mod** es una estructura **Tiempo** que almacena la fecha y hora en que se modificó por última vez el archivo. Dado un arreglo **lista** de **n** archivos, implemente funciones para ordenarlo alfanuméricamente y temporalmente de acuerdo a los siguientes prototipos

- a. `void ordena_alfa( Archivo * lista, int n );`  
(utilice la función **strcmp** de la biblioteca estándar para comparar los nombres de los archivos al ordenarlos alfabéticamente).
- b. `void ordena_temporal(Archivo * lista, int n );`  
(utilice la función **compara\_tiempos** para comparar los tiempos de última modificación de los archivos).

2) Dada la siguiente estructura

```
typedef struct
{
    int n;
    double *coeficiente;
} Polinomio;
```

Que representa a un polinomio de orden  $n$ , cuyos  $n+1$  coeficientes  $c_0, c_1, \dots, c_n$ , almacenados en el vector apuntado por `coeficiente` corresponden a las potencias  $x^0, x^1, \dots, x^n$ , cree las siguientes funciones:

```
a. /* crea y retorna un nuevo Polinomio */
Polinomio * creaPolinomio(int orden);
b. /* asigna el n_ésimo coeficiente del Polinomio */
void setCoef(int n, double c, Polinomio * P);
c. /* retorna el n_ésimo coeficiente del Polinomio */
double getCoef(int n, Polinomio * P);
d. /* calcula el polinomio en x usando:
((...((c[n]*x+c[n-1])*x+c[n-2])*x+ ...+c[1]*x)+c[0]) */
double especializa( double x, Polinomio * P );
e. /* suma dos Polinomios retorna un nuevo Polinomio con el resultado */
Polinomio * sum( Polinomio *p1, Polinomio *p2);
f. /* multiplica dos Polinomios y retorna un nuevo Polinomio con el
resultado */
Polinomio * mult( Polinomio *p1, Polinomio *p2);
g. /* deriva un Polinomio retornando un nuevo Polinomio con el
resultado */
Polinomio * deriv( Polinomio *p );
h. /* libera la memoria asociada con el polinomio */
void destruyePolinomio( Polinomio *p );
i. /* busque por el método de bisección un cero de un polinomio dentro
de un intervalo [a,b], con una precisión dada por épsilon y lo
retorne*/
double ceropol( Polinomio *p, double a, double b, double epsilon);
```

3) La siguiente estructura de datos permite representar números enteros con precisión arbitraria:

```
typedef struct {
    char sign;
    unsigned char num_bytes;
    unsigned char * bytes;
} APint;
```

Donde `sign` es el signo (-1 0 o 1), `num_bytes` es el número de bytes que se utilizan para representar el número (por ejemplo 4 equivaldría aproximadamente a un `int` en una arquitectura de 32 bits), `bytes` es un arreglo que contiene los valores con los que se representa el número.

Deberá implementar las siguientes funciones para sumar y multiplicar dichos enteros:

```
a. APint * suma_APint( APint * i1, APint * i2);
```

b. `APint * producto_APint(APint * i1, APint * i2);`

- 4) Se define TipoCiudad como un struct para almacenar la posición de una ciudad en una representación de dos dimensiones, es decir, en un plano.

```
struct TipoPunto{
    double abscisa;
    double ordenada;};
struct TipoCiudad{
    TipoPunto situacion;
    char nombre[50];
};
```

Para almacenar varias ciudades, se construirá un vector de TipoCiudad. Se pide construir una función que, a partir de un vector de TipoCiudad y dado el nombre de una ciudad, reordene ascendentemente el vector atendiendo a la distancia euclídea del resto de las ciudades con respecto a la elegida. Por ejemplo, si elegimos "Granada", la ciudad con dicho nombre deberá ponerse como la primera componente del vector; la segunda será la ciudad más cercana a "Granada" y así sucesivamente. El prototipo de la función será:

```
void Reordenar (TipoCiudad ciudades[], int num_ciudades, const char
                nombre_ciudad_referencia[]);
```

Recordemos que la distancia euclídea entre dos puntos se define como la raíz cuadrada de la suma de los cuadrados de las diferencias de las abscisas y las ordenadas. No pueden usarse vectores auxiliares.

- 5) Escribir un programa para sumar dos matrices:

- ↵ Cree una estructura con dos enteros con las dimensiones y un puntero a la matriz.
- ↵ Llene con valores enteros en forma aleatoria.
- ↵ Desarrolle una función que realice la suma pasando como argumento las estructuras de matrices y devuelva el puntero al resultado.
- ↵ Mostrar el resultado de la suma en forma matricial.

```
struct matrix { short filas, col; int **matriz; };
```

- 6) Dadas las siguientes estructuras de datos

```
typedef struct{
    unsigned char R, G, B; //componentes primarios de un color
} RGB;

typedef struct{
    int ancho, alto;
    RGB **pixel;
} ImagenRGB;

typedef unsigned char Gris;
```

```
typedef struct{
    int ancho, alto;
    Gris **pixel;
} ImagenGris;
```

Implemente las siguientes funciones:

- a. /\* crea y retorna una nueva ImagenRGB \*/  
**ImagenRGB \* creaImagenRGB( int ancho, int alto );**
- b. /\* crea y retorna una nueva ImagenGris \*/  
**ImagenGris \* creaImagenGris( int ancho, int alto );**
- c. /\* asigna el pixel de la fila y columna dadas \*/  
**void setPixelRGB( ImagenRGB \*im, int fila, int columna, RGB \* pix );**
- d. /\* asigna el pixel de la fila y columna dadas \*/  
**void setPixelGris( ImagenGris \*im, int fila, int columna, Gris \* pix );**
- e. /\* retorna el pixel de la fila y columna dadas \*/  
**RGB \* getPixelRGB( ImagenRGB \*im, int fila, int columna );**
- f. /\* retorna el pixel de la fila y columna dadas \*/  
**Gris getPixelGris( ImagenGris \*im, int fila, int columna );**
- g. /\* libera la memoria asociada con la imagen im \*/  
**void destruyeImagenRGB( ImagenRGB \* im);**
- h. /\* libera la memoria asociada con la imagen im \*/  
**void destruyeImagenGris( ImagenGris \* im);**
- i. /\* convierte un pixel RGB en uno Gris usando la fórmula:  
 $Gris = 0.299*R + 0.587*G + 0.114*B$  \*/  
**Gris RGBtoGris( RGB \* pix );**
- j. /\* transforma la imagenRGB en una nueva ImagenGris \*/  
**ImagenGris \* transforma( ImagenRGB \* im );**