CI2126 Computación II, Ene/Mar 2014

Resolución del Examen I (30%)

1.- (7 puntos) Indique con **V**(erdadero) o **F**(also) a las expresiones siguientes:

a.- `typedef float mireal` declara la variable "mireal" _F_
b.- Los punteros ocupan espacio en memoria sólo cuando no son nulos. _F_
c.- La expresión `(*puntero).atributo` y `puntero->atributo` dan resultados diferentes _F_
d.- Si *p* es puntero a **int**, `*p = malloc(sizeof(32))` crea una cadena de caracteres _F_
e.- La cola es una estructura del tipo FIFO (*First In First Out*). _V_
f.- La pila es una estructura del tipo LILO (*Last In Last Out*). _F_
g.- Dos pilas encadenadas, una construida a partir de vaciar la otra, equivalen a otra pila _V_
h.- Una cola se puede simular con una pila, si desencolar invierte la pila antes.. _V_
i.- Si *p* es puntero a `int` no nulo, `p[0] = 3;` es una instrucción legal _V_
j.- Si declaramos `FILE* f; f = fopen("a.txt", "w");` Si ∃ "a.txt", se produce error. _F_
k.- Un puntero a función no se puede pasar como parámetro a otras funciones. _F_
l.- Declarando **int*** p, entonces `p = calloc(25,sizeof(int))` crea un arreglo de **int** _V_
m.- `double call (double (*f) (double x), x) { return f(x); }` es correcto _V_
n.- `double (*f) (double x); f = sqrt; double x = f(4.0);` es correcto _V_

4.- (3 pts) Dado el siguiente programa escrito en lenguaje C, indique qué hace el mismo y cuáles valores produciría cuando al ejecutarlo el usuario introduce como entrada "**4**":

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

double cube (double x) { return x*x*x; }

int main() {
      /* fabs  calcula el valor absoluto,   sqrt  calcula la raíz cuadrada */
      static double (*funcion[])(double) = {fabs, sqrt, cube};
      double x;
      int i;

      fprintf (stdout, "\n Introduzca un numero real: ");
      fscanf  (stdin,  "%lf", &x);

      for (i=0; i<3 ;++i) {
            double r = ( funcion[i] ) ( ( funcion[(i+1)%3] ) (x) );
            fprintf(stdout, "\n %lf provee los resultados r: %lf \n", x, r);
      }
      exit (0);
}
```

```
fabs ( sqrt ( 4.0 ) ) -->       4.000000 provee los resultados r: 2.000000

sqrt ( cube ( 4.0 ) ) -->       4.000000 provee los resultados r: 8.000000

cube ( fabs ( 4.0 ) ) -->       4.000000 provee los resultados r: 64.000000
```

2.- **(6 puntos)** Codifique una función *sonIguales* que reciba una ***Pila*** *p* y una ***Cola*** *q* y determine si al *extraer sus elementos* ambas tienen los mismos valores en el mismo orden, usando ***solamente*** las operaciones de los TADs ***Pila*** y ***Cola.*** ***OJO***: sin destruir las originales *p* y *q*.

```
int sonIguales (Pila p, Cola q):
```

Debe retornar : **1** si *p* y *q* tienen los mismos elementos; **0** si no.

```
        /* Los elementos son enteros */
typedef int Elem_t;

    /* Definición apuntador del TDA Elem */
typedef Elem_t* Elem;
```

```
         /* El TDA recursivo Nodo */
typedef struct NodoCDT_tag {
    Elem info;
    struct NodoCDT_tag* next;
} NodoCDT_t;

    /* Definición apuntador del TDA Nodo */
typedef NodoCDT_t* Nodo;
```

```
 /* Selector para obtener un elem de un nodo */
Elem getElem (Nodo n)   /* _pre: ∃ n */
```

```
       /* TDA Pila_CDT_t como un TDC  */
typedef struct PilaCDT_tag {
       Nodo primero;
} PilaCDT_t;

    /* Definición apuntador del TDA Pila */
typedef PilaCDT_t* Pila;
```

```
       /* TDA Cola_CDT_t como un TDC  */
typedef struct ColaCDT_tag {
    Nodo primero;
    Nodo ultimo;
} ColaCDT_t;

    /* Definición apuntador del TDA Cola */
typedef ColaCDT_t* Cola;
```

```
   /* Devuelve 1 si los elems son iguales */
bool igualElems ( Elem a, Elem b);
/* _pre: ∃ a, b */
```

```
         /* Devuelve una pila vacía */
Pila consPila ();

    /* Libera memoria ocupada por la pila */
void destPila (Pila* p);   /* _pre: ∃ p */

       /* Devuelve el tope de la pila */
Elem topePila (Pila p);   /* _pre: ∃ p */


       /* Empila e y devuelve la Pila */
Pila empilar (Pila p, Elem e);
/* _pre: ∃ p */

   /* Desempila el tope actual de la Pila */
Pila desempilar(Pila p, Elem* e);
/* _pre: ∃ p */

       /* Dice si la Pila está vacía */
bool esPilaVacia (Pila p);  /* _pre: ∃ p */

       /* Hace una copia de la Pila */
Pila clonarPila(Pila p);   /* _pre: ∃ p */
```

```
        /* Devuelve una cola vacía */
Cola consCola ();

   /* Libera memoria ocupada por la cola */
void destCola (Cola* q);   /* _pre: ∃ q */


     /* Devuelve el primero de la cola */
Elem primeroCola (Cola q);   /* _pre: ∃ q */

     /* Devuelve el ultimo de la cola */
Elem ultimoCola (Cola q);   /* _pre: ∃ q */


     /* Encola e como último de la cola */
Cola encolar (Cola q, Elem e);
/* _pre: ∃ q */

     /* Desencola el primero de la cola */
Cola desencolar(Cola q, Elem* e);
/* _pre: ∃ q */

       /* Dice si la Cola está vacía */
bool esColaVacia (Cola q);   /* _pre: ∃ q */

       /* Hace una copia de la cola */
Cola clonarCola(Cola q);    /* _pre: ∃ q */
```

```
#include "hoare.h"
```

Primera Solución

```
bool sonIguales1 (Pila p, Cola q) {
    _pre ( (p != NULL) and (q != NULL) );

    Pila s = clonarPila(p);          // Se crea una copia de p como la pila auxiliar s
    Cola c = clonarCola(q);          // Se crea una copia de q como la cola auxiliar c
    bool iguales = true;             // De principio suponemos que son iguales
    Elem a, b;                       // Declaramos apuntadores vacíos a elementos

    while ((not esVaciaPila(s)) and (not esVaciaCola(c)) and (iguales)) {
        desempilar(s, &a);           // Sacamos a de la pila s
        desencolar(c, &b);           // Sacamos b de la cola c
        iguales = igualElems(a,b);   // Vemos si a y b son iguales
    }
                                     // Retornamos que son iguales sólo si ambas quedaron vacías
    iguales = iguales and esVaciaPila(s) and esVaciaCola(c);

    destCola(&c);                    // Se destruye la cola auxiliar c
    destPila(&s);                    // Se destruye la pila auxiliar s

    return iguales;
}
```

Segunda Solución

```
bool sonIguales2 (Pila p, Cola q) {
    _pre ( (p != NULL) and (q != NULL) );

    Pila s = consPila();             // Se crea pila auxiliar vacía s
    Cola c = consCola();             // Se crea cola auxiliar vacía c
    bool iguales = true;             // De principio suponemos que son iguales
    Elem a, b;                       // Declaramos apuntadores vacíos a elementos

    while ((not esVaciaPila(p)) and (not esVaciaCola(q)) and (iguales)) {
        desempilar(p, &a);           // Sacamos a de la pila p
        desencolar(q, &b);           // Sacamos b de la cola q
        iguales = igualElems(a,b);   // Vemos si a y b son iguales
        empilar(s, a);               // Metemos a en la pila s
        encolar(c, b);               // Metemos b en la cola c
    }
                                     // Retornamos que son iguales sólo si ambas quedaron vacías
    iguales = iguales and esVaciaPila(p) and esVaciaCola(q);

    while (not esVaciaPila(s)) {     // Restauramos la pila p
        desempilar(s, &a);           // Sacamos a de la pila s
        empilar(p, a);               // Metemos a en la pila p
    }

    while (not esVaciaCola(q)) {     // Vaciar el resto de la cola q
        desencolar(q, &b);           // Sacamos b de la cola q
        encolar(c, b);               // Metemos b en la cola c
    }

    q->primero = c->primero;         // Restauramos la cola q
    q->ultimo  = c->ultimo;          // Solo copiamos los apuntadores

    destCola(&c);                    // Se destruye la cola auxiliar
    destPila(&s);                    // Se destruye la pila auxiliar

    return iguales;                  // Se devuelve si eran iguales la pila y cola
}
```

Tercera Solución Usando recursion de cola);

```
bool sonIgualesAux (Pila p, Cola q, bool iguales, Pila s, Cola c)
{
    Elem a, b;                      // Declaramos apuntadores vacíos a elementos
    bool ningunaVacia = ( not ( esVaciaPila(p) ) ) and ( not esVaciaCola(q) );
                                    // Si cualquiera se acaba, fin
    if ( iguales and ningunaVacia) {
        desempilar(p, &a);          // Sacamos a de la pila p
        desencolar(q, &b);          // Sacamos b de la cola q
        empilar(s, a);              // Metemos a en la pila s
        encolar(c, b);              // Metemos b en la cola c

        return sonIgualesAux(p, q, igualElems(a,b), s, c);      // Vemos si a y b son iguales
    }

    return iguales;
 }


bool sonIguales3 (Pila p, Cola q)
{
    _pre ( (p != NULL) and (q != NULL) );

    Pila s = consPila();
    Cola c = consCola();
    Elem e;

    // De principio suponemos que son iguales
    // Se devuelve si eran iguales la pila y cola
    bool iguales = sonIgualesAux(p, q, true, s, c) and esVaciaPila(p) and esVaciaCola(q);

        //if ((iguales and ambasVacias) or (not iguales) )
    while (not esVaciaPila(s)) {    // Restauramos la pila p
        desempilar(s, &e);          // Sacamos e de la pila s
        empilar(p, e);              // Metemos e en la pila p
    }

    while (not esVaciaCola(q)) {    // Vaciar el resto de la cola q
        desencolar(q, &e);          // Sacamos e de la cola q
        encolar(c, e);              // Metemos e en la cola c
    }

    q->primero = c->primero;        // Restauramos la cola q
    q->ultimo  = c->ultimo;         // Solo copiamos los apuntadores

    destCola(&c);                   // Se destruye la cola auxiliar
    destPila(&s);                   // Se destruye la pila auxiliar

    return iguales;
}
```

3.- **(6 pts)** Se define el TAD ***Estudiante_t***, para almacenar la información de un estudiantes USB.

Estudiante_t: {carnet, nombres, apellidos, fecha_ingreso, edad, carrera, créditos_aprobados, índice}
donde:

carnet, apellidos, nombres $\in$ ***string***	(cadenas con 8, 20 y 20 caracteres respectivamente)
fecha_ingreso $\in$ ***Fecha_t,***	{año (4 dígitos), mes (2 dígitos), día (2 dígitos)}
edad, créditos_aprobados, carrera $\in \mathbb{Z}$	(2 dígitos), (3 dígitos), (2 dígitos)
indice $\in \mathbb{R}$	(5 dígitos más el punto)

El tipo ***Estudiante*** es un apuntador a ***Estudiante_t***.

Se define el TAD **Nomina_t** : {maxEst : $\mathbb{Z}$; Est: apuntador (arreglo) de *maxEst* estudiantes}

El tipo ***Nomina*** es un apuntador a ***Nomina_t***.

(a) Declarar correctamente los TADs:

(b) Suponiendo que conoce estas Operaciones primitivas de los TADs:

Estudiante *consEstudiante* (carnet, nombres, apellidos, fecha_i, edad, credaprob, carrera);
void *destEstudiante* (***Estudiante**** e);
Nomina *consNomina ( );*
void *destNomina* (***Nomina**** n);

Codifique la función

Nomina *leerEstudiantes* : (***char**** nombreArchivo)

Recibe un string de 256 caracteres que es el nombre de un archivo de texto cuya primera línea indica cuantas líneas siguen, cada línea conteniendo los datos de un estudiante, y devuelve una Nomina de Estudiantes. Se adjunta un ejemplo con 3 estudiantes.

```
0.........1.........2.........3.........4.........5.........6.........7....
01234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234

3
07—38887Fulanito            Menganez            1992 07 10 22 104 15 3.4513
08—38887Zutanita            Perencejo           1993 03 08 21  92 12 3.0010
09—38887Gomen               Nasai               1994 01 09 20  45 11 4.5234
```

(a) Declarar correctamente los TADs:

```
typedef struct Fecha_tag {            /* La estructura de Fecha          */
    short anyo;
    char  mes;
    char  dia;
} Fecha_t;

typedef struct Estudiante_tag {       /* La estructura del Estudiante    */
    char carnet[8];                   /* carnet                          */
    char apellidos[20];               /* apellidos                       */
    char nombres[20];                 /* nombres                         */
    Fecha_t fe_in;                    /* Fecha de ingreso                */
    char edad;                        /* Edad                            */
    short cred_aprobados;             /* Créditos aprobados              */
    char carrera;                     /* Carrera                         */
    double indice;                    /* Índice                          */
} Estudiante_t;

typedef Estudiante_t*  Estudiante;    /* El apuntador a Estudiante       */

typedef struct Nomina_tag {           /* La estructura de la Nómina      */
    int maxEst;                       /* Cantidad total de estudiantes   */
    Estudiante est;                   /* El apuntador a Estudiante       */
} Nomina_t;

typedef Nomina_t* Nomina;             /* El apuntador a la nómina        */
```

(b) Codifique la función **Nomina** leerEstudiantes : (**char*** nombreArchivo)

```
Nomina  leerEstudiantes (char* nombreArchivo)
{
    FILE* f;
    int k, cant;
    if (f = fopen(nombreArchivo, "r"))   /* abrir archivo exitoso   */
    {
        fscanf(f, "%i", &cant);
        Nomina n = consNomina();
        n->maxEst = max;
        n->est = (Estudiante) malloc(max*sizeof(Estudiante_t));
         /* o tambien: n->est = (Estudiante) calloc(max,sizeof(Estudiante_t));  */

        for (k = 0; k < max; k++)
        {
           fscanf(f, "%s", &n->est[k].carnet);
           fscanf(f, "%s", &n->est[k].apellidos);
           fscanf(f, "%s", &n->est[k].nombres);
           fscanf(f, "%i %i %i", &n->est[k].fe_in.anyo,
                                 &n->est[k].fe_in.mes,
                                 &n->est[k].fe_in.dia);
           fscanf(f, "%i", &n->est[k].cred_aprobados);
           fscanf(f, "%i", &n->est[k].carrera);
           fscanf(f, "%f", &n->est[k].indice);
        }

        if (!fclose(f))      /* El archivo se cerró correctamente */
        {
            destEstudiante ( Estudiante &e );
            return n;        /* Se devuelve la nómina             */
        }

        printf("Hubo un error cerrando %s \n",  nombreArchivo);
    }

    printf("Hubo un error abriendo %s \n",  nombreArchivo);
    return NULL;
}
```

5.- **(8 puntos)** Dado la siguiente especificación funcional sintáctica del **TDA Matriz NxM** (N columnas y M filas), diseñe la especificación formalmente en C (lo que sería el "Matriz.h")

(a) Las declaraciones necesarias para crear la estructura de la matriz

```
typedef  ...  MatrizTDA_t;  /* La estructura del TDA Matriz NxM */

typedef ...   Matriz;       /* El apuntador a la Matriz */
```

(b) Los prototipos necesarias de las operaciones de la matriz, con sus precondiciones

`consMatriz :  ncolumnas x nfilas ---> Matriz`	(Genera una matriz llena de ceros)
`consIdentidad : nfilas ---> Matriz`	(Devuelve una matriz identidad cuadrada)
`esCuadrada : Matriz ---> Boolean`	(Indica si una matriz es cuadrada)
`esIdentidad : Matriz ---> Boolean`	(Indica si la matriz es la Identidad)
`esFila : Matriz ---> Boolean`	(Indica si la matriz contiene una única fila)
`esColumna : Matriz ---> Boolean`	(Indica si la matriz contiene una única columna)
`getNumFilas : Matriz ---> Int`	(Indica la cantidad de filas de una matriz)
`getNumColumnas : Matriz ---> Int`	(Indica la cantidad de columnas de una matriz)
`setValor : Matriz x fila x columna x Real ---> Matriz`	(Coloca un elemento en la posición [fila,columna] indicada dentro de la matriz)
`getValor :  Matriz x fila x columna --->  Real`	(Devuelve el elemento en esa posición)
`sumaMatriz : Matriz x Matriz ---> Matriz`	(Suma matrices de iguales dimensiones)
`multMatriz : Matriz x Matriz ---> Matriz`	(Multiplica matrices. El *NumColumnas* de la primera debe ser igual al *NumFilas* de la segunda)
`multEscalar : Matriz x Real ---> Matriz`	(El producto un escalar a toda la matriz)
`destMatriz : Matriz --->`	(Elimina matriz. Retorna recursos al sistema)

(c) Haga la implementación de *consIdentidad , sumaMatriz*

BONO*:* Haga la implementación de *multMatriz*

(a) Las declaraciones necesarias para crear la estructura de la matriz

```
typedef struct MatrizTDA_tag {    /* La estructura del TDA Matriz NxM */
    int nc;                       /* num columnas                     */
    int nf;                       /* num filas                        */
    double* cells;                /* apuntador al arreglo de celdas   */
} MatrizTDA_t;

typedef MatrizTDA_t*  Matriz;     /* El apuntador a la Matriz         */
```

(b) Los prototipos necesarios de las operaciones de la matriz, con sus precondiciones

```
“Matriz.h”

Matriz consMatriz (int nc, int nf);
    /* _pre (( nc > 0) and (nf > 0)) */  /* m = (Matriz) calloc(nc*nf, sizeof(double)) */

Matriz consIdentidad (int n);
    /* _pre ( n > 0) */

bool esCuadrada (Matriz m);
    /* _pre ( m != NULL) */

bool esIdentidad (Matriz m);
    /* _pre ( m != NULL) */

bool esFila (Matriz m);
    /* _pre ( m != NULL) */

bool esColumna (Matriz m);
    /* _pre ( m != NULL) */

int getNumFilas (Matriz m);
    /* _pre ( m != NULL) */

int getNumColumnas (Matriz m);
    /* _pre ( m != NULL) */

Matriz setValor (Matriz m, int fila, int columna, double valor);
    /* _pre ( m != NULL) and ( 0 <= fila < m->nf ) and ( 0 <= columna < m->nc ) */

double getValor (Matriz m, int fila, int columna);
    /* _pre ( m != NULL) and ( 0 <= fila < m->nf ) and ( 0 <= columna < m->nc ) */

Matriz sumaMatriz (Matriz a, Matriz b);
    /* _pre ( ( a != NULL) and ( b != NULL) and ( a->nf == b->nf ) and ( a->nc == b-> nc) ) */

Matriz multMatriz (Matriz a, Matriz b);
    /* _pre ( ( a != NULL) and ( b != NULL) and ( a->nc == b->nf ) ) */

Matriz multEscalar (Matriz m, double e);
    /* _pre ( m != NULL) */

void destMatriz (Matriz* m);
    /* _pre ( m != NULL) */
```

(c) Haga la implementación de *consIdentidad* , *sumaMatriz*

```
Matriz consIdentidad (int n) {
    _pre( n > 0 );

    int k;
    Matriz m = consMatriz(n, n);                    /* Llamamos al constructor     */

    for (k = 0; k < n; k++)
        setValor (m, k, k, 1.0);                    /* Llenamos de 1's la diagonal */

    return m;
}
```

```
Matriz = sumaMatriz (Matriz a, Matriz b) {
    _pre ( ( a != NULL ) and ( b != NULL ) and ( a->nf == b->nf ) and ( a->nc == b->nc) );

    int i, j;
    int nc = getNumColumnas(a);
    int nf = getNumFilas(a);

    Matriz suma = consMatriz(nc, nf);

    for (j = 0; j < nc; j++)
        for (i = 0; i < nf; i++)                    /* suma[i][j] +=  a[i][j] + b[i][j] */
            setValor( suma, i, j, getValor(a, i, j) + getValor(b, i, j) );

    return suma;
}
```

BONO

```
Matriz = multMatriz (Matriz a, Matriz b) {
    _pre ( ( a != NULL) and ( b != NULL) and ( a->nc == b->nf ) );

    int i, j, k;
    double total;

    int nc = getNumFilas(a);
    int nf = getNumColumnas(b);
    int nk = getNumFilas(b);                        /* Tambien es getNumColumnas(a)  */

    Matriz mult = consMatriz(nc, nf);

    for (j = 0; j < nc; j++)
        for (i = 0; i < nf; i++)
        {
            total = 0.0;                            /* suma inicial es 0 por celda  */
            for (k = 0; k < nk; k++)                /* total +=  a[i][k] * b[k][j]  */
                total += getValor(a, i, k) * getValor(b, k, j);
            setValor( mult, i, j, total);           /* suma[i][j] = total           */
        }

    return mult;
}
```