# PEC3

## Formato y fecha de entrega

Hay que entregar la PEC antes del día **8 de mayo de 2017 a las 23:59**. Para la entrega deberéis entregar un fichero en formato **ZIP**, que contenga:

- Este documento con las respuestas a los diferentes ejercicios que se plantean.
- Un **workspace Codelite** que contenga un proyecto y los archivos .h y .c pedidos por cada ejercicio. El **main** de cada proyecto debe contener las llamadas a las pruebas que se solicitan, de forma que se muestre por pantalla el resultado de cada prueba.

Hay que hacer la entrega en el apartado de entregas de EC del aula de teoría.

## Presentación

En esta PEC trabajaremos tanto en lenguaje algorítmico como con pequeños programas en C. Se utilizarán tablas y tipos abstractos de datos. También se continuará trabajando con los tipos de datos estructurados y la modularidad, tanto a nivel de acciones y funciones, como en la utilización de diversos archivos de código en Codelite.

## Competencias

***Transversales***

- Capacidad de comunicación en lengua extranjera.

***Específicas***

- Capacidad de diseñar y construir aplicaciones informáticas mediante técnicas de desarrollo, integración reutilización.
- Conocimientos básicos sobre el uso y programación de los ordenadores, sistemas operativos, bases de datos y programas informáticos con aplicación en la ingeniería

## Objetivos

- Saber modularizar el código utilizando acciones y funciones
- Saber crear y utilizar archivos de cabecera
- Saber definir y utilizar tablas
- Saber definir y utilizar un TAD
- Saber definir pruebas para comprobar el correcto funcionamiento de un código
- Crear pequeños programas en C

## Recursos

Para realizar esta actividad tenéis a vuestra disposición los siguientes recursos:

### Básicos

- Materiales en formato web de la asignatura de las 11 primeras semanas
- Laboratorio de C

### Complementarios

- Internet: La forma más efectiva de encontrar información sobre cualquier duda sobre C es la búsqueda a través de un buscador.

## Criterios de valoración

Cada ejercicio lleva asociada la puntuación sobre el total de la actividad. Se valorará tanto la corrección de las respuestas como su completitud.

- Los ejercicios en lenguaje algorítmico deben respetar el formato. (Véase documento *Nomenclátor* de los materiales).
- Los ejercicios en lenguaje C deben compilar para ser evaluados. En tal caso, se valorará que:
  - funcionen correctamente
  - respeten los criterios de estilo. (Véase la *Guía de estilo de programación en C* que tenéis en los materiales)
  - el código está comentado (preferiblemente en inglés)

- las estructuras utilizadas sean las adecuadas

## [40%] Ejercicio 1: Operaciones con tablas

Una tienda se dispone a premiar la fidelidad de sus clientes ofreciendo cupones de descuento que podrán canjear en futuras compras. En este ejercicio os pedimos que, a partir de las siguientes definiciones de tipo **tDiscount** y **tDiscountShop** en lenguaje algorítmico,

```
const
     MAXDISCOUNTS: integer= 50;
endconst

type
      tDiscount  = record
          dni: integer;
          discount: real;
          changed: boolean;
      end record

     tDiscountsShop = record
          discounts: vector [MAXDISCOUNTS] of tDiscount;
          numDiscounts: integer;
     end record
endtype
```

y en lenguaje C,

```
#define MAXDISCOUNTS 50

typedef enum {FALSE, TRUE} bool;

typedef struct {
	int	dni;
	float discount;
	bool changed;
} tDiscount;

typedef struct {
	tDiscount discounts[MAXDISCOUNTS];
	int	numDiscounts;
} tDiscountsShop;
```

implementa en lenguaje algorítmico y en C:

1. Una acción **sortDiscountsShop** que dada una tabla de tipo **tDiscountsShop**, que contiene los cupones de descuento generados, devuelva la tabla ordenada por DNI.

```
action sortDiscountsShop(inout discountsShop: tDiscountsShop)
       { variables declaration }
       var
          i, j: integer;
       fvar

       { iterate for each discount on the table }
       for i:=1 to discountsShop.numDiscounts do
         for j:=1 to discountsShop.numDiscounts-i do
            { compare dni of each discount }
           if discountsShop->discounts[j].dni >
discountsShop->discounts[j+1].dni then
              { switch positions of discounts on the table }
              swap(discountsShop.discounts[j],
discountsShop.discounts[j+1]);
           end if
         end for
       end for
end action
```

2. Una acción **discountChanged** que...
    a. Dada una tabla de tipo **tDiscountsShop** que contiene cupones de descuento y un **DNI** que identifica un determinado cupón de descuento
    b. Busca el **cupón** en la tabla y pone el valor del campo **changed** a TRUE.
    c. Si dado un **DNI** que identifica un cupón, éste no existe en la tabla, se mostrará un aviso que lo indique.
    d. De forma similar, si dado un **DNI** que identifica un cupón, éste tiene el campo **changed** con valor TRUE, se mostrará un aviso indicando que este cupón de descuento ya se ha canjeado previamente.

```
action discountChanged(inout discountsShop:tDiscountsShop,
dni:integer)
        { variables declaration }
        var
           i, state: integer;
        fvar

        { variables init }
        i := 1;
        state := -1;

        { iterate for each discount on the table }
        while i < discountsShop.numDiscounts AND state <= 0 do
                { check if both dni are the same }
                if discountsShop->discounts[i].dni = dni then
                        { check if the discount is already been used }
                        if discountsShop->discounts[i].changed = TRUE
        then
                                { show 'discount used' message }
                                writeString("Error. Sorry, but this
                        discount is already been used.");
                        else
                                { update changed value }
```

```
                        discountsShop->discounts[i].changed := TRUE
                    end if
                    { cancel search }
                    state := 0;
                end if
                { update index search }
                i := i+1;
            end while

            { show 'discount not found' message }
            if (state = -1) then
                writeString("Error. Sorry, we could not find the
            discount.");
            end if
        end action
```

3. (**Sólo en lenguaje C**) Una acción **testDiscounts** que implemente las pruebas necesarias para comprobar que el código funciona en los siguientes casos (hay que mostrar el resultado por pantalla).
    a. Para hacer los test hay que declarar e inicializar una variable de tipo **tDiscountsShop** que contenga información de cinco cupones de descuento.
    b. Los cupones de descuento deben estar desordenados por DNI.
    c. Los cupones tendrán el campo **changed** con valor FALSE excepto uno de los cupones que lo tendrá con valor TRUE.
    d. Tests:
        1. Ordenamos por **DNI** la variable de tipo **tDiscountsShop**.
        2. Buscamos un cupón cuyo **DNI** existe y tiene el campo **changed** a **FALSE**.
        3. Buscamos un cupón cuyo **DNI** existe y tiene el campo **changed** a **TRUE**.
        4. Buscamos un cupón cuyo **DNI** no existe.

**Nota:**

- El código en lenguaje C de los apartados a y b de este ejercicio debe ir en los ficheros **discount.c** y **discount.h**.

- En el caso de los test del apartado c, deberán ir en los ficheros **tests.c** y **tests.h**.
- Añade una llamada en el método **main** en la acción **testDiscounts**.

## [60%] Exercici 2: Operaciones con listas

A partir de la siguiente definición del tipo **tClient** y del tipo **tShop** en lenguaje algorítmico:

```
const
     MAXPURCHASES: integer= 50;
     MAXCLIENTS: integer= 50;
end const

type
     tClient=record
          dni: integer; {sin letra}
          name: string;
          email: string;
          accumulated: real;
          thereIsCupon: boolean;
     end record

     tShop=record
          clients: vector [MAXCLIENTS] of tClient;
          numClients: integer;
     end record
endtype
```

y en lenguaje C,

```
#define MAXPURCHASES 50
#define MAXCLIENTS 50
#define MAXSTRINGLEN 100

typedef enum {FALSE, TRUE} bool;

typedef struct{
     int dni;
     char name [MAXSTRINGLEN];
     char email [MAXSTRINGLEN];
     float accumulated;
     bool thereIsCupon;
} tClient;

typedef struct{
     tClient clients [MAXCLIENTS];
     int numClients;
} tShop;
```

y teniendo como ayuda adicional la implementación en lenguaje algorítmico de las acciones **shopInit** para inicializar la lista de clientes de una variable de tipo **tShop** y **clientCopy** para copiar dos variables de tipo **tClient**,

```
action shopInit (inout shop: tShop)
     shop.numClients:= 0;
end action

action clientCopy (in client1: tClient, out client2: tClient)
     {to copy strings in C you need to use 'strcpy'}
     client2.dni:= client1.dni;
     client2.name:= client1.name
     client2.email:= client1.email;
     client2.accumulated:= client1.accumulated;
     client2.thereIsCupon:= client1.thereIsCupon;
end action
```

diseña en lenguaje algorítmico y en lenguaje C:

**a)** Una **función posClient** que:

1. Recibe dos variables como parámetro:
   i. Una de tipo **tShop**
   ii. Una de tipo entero (que representa el **DNI** de un cliente),
2. Devuelve:
   i. Un entero que indique la **posición** del cliente en la lista.
   ii. En el caso que el cliente no esté, debe devolver un **-1**.

```
function posClient(inout shop:tShop, in dni:integer) : integer
        { variables declaration }
        var
           i, position: integer;
           founded: bool;
        fvar

        { variables init }
        i := 1;
        position := -1;
        founded = FALSE;

        { iterate over each client of shop }
        while i < shop.numClients AND founded = FALSE do
                { check if any dni match }
                if shop.clients[i].dni = dni then
                        { return position }
                        position := i;
                        founded := TRUE;
                endif
                { update index search }
                i := i+1;
        end while
```

```
        return position;
end action
```

**b)** Una **acción insertClient** que,

1. Dada una
   i. variable de tipo **tShop** (con clientes ordenados por DNI de forma creciente)
   ii. y una variable de tipo **tClient**,
2. Compruebe, mediante el **DNI**, que el cliente **no está dentro** de la variable de tipo **tShop**
   i. Y haga la inserción **ordenándolo por DNI**.
   ii. En caso contrario, no se inserta y se muestra un mensaje por pantalla avisando que el cliente ya está dentro de la variable de tipo **tShop**.
   iii. Si no se puede hacer la inserción porque la tabla está llena, se mostrará el aviso que lo indique.
3. Para hacer esta acción se pueden utilizar las funciones/acciones anteriores.

```
void insertClient(inout shop:tShop, client:tClient)
        { variables declaration }
        var
           i, j, positionToInsert: integer;
           founded: bool;
        fvar

        { setting default values to variables }
        i := 0;
        founded := FALSE;

        { check if this client exists into our shop }
        position = posClient(shop, client.dni);

        { if client is not register yet }
        if position = -1 then
```

```
                { check if the shop table have enough space for another
client}
                if shop.numClients < MAXCLIENTS then
                        { Search the right position to insert the new client }
                        while i <= shop.numClients AND founded = FALSE do
                                if shop.clients[i].dni > client.dni then
                                        positionToInsert := i;
                                        founded := TRUE;
                                end if

                                { Update index search }
                                i := 1;

                                { If there is not a match, the position to
insert is the last one }
                                if i = shop.numClients then
                                        positionToInsert = i;
                                end if
                        end while

                        { Move the clients one position to the right.
                          Starting from position of new client }
                        for j == shop.numClients-1;
                            j >== positionToInsert;
                            j-- then
                               shop.clients[j+1] = shop.clients[j];
                        end for

                        { Update number of clients }
                        shop.numClients := shop.numClients + 1;

                        { Insert new client }
                        shop.clients[positionToInsert] = client;
                else
                        { show 'not enough space' message }
                        writeString("Error. Sorry, currently there is not
```

```
            enough space for another client");
            end if
        else
            { show 'client registered' message }
            writeString("Error. Sorry, this client is already
        registered.");
        end if
end action
```

**c)** Una **acción addAmountClient** que,

1. dada
   - i. una variable de tipo **tShop**,
   - ii. un **DNI** que identifica a un cliente
   - iii. y un **real** que representa el **importe de una nueva compra** de este cliente,
2. mire si el cliente está en la lista y localice su posición...
   - i. Si el cliente existe,
     1. añada el importe de la nueva compra al importe acumulado en el campo **accumulated**.
   - ii. Si el cliente no está en la lista debe mostrar un aviso.

```
action addAmountClient(inout shop: tShop, dni:integer, amount:real)
        { variables declaration }
        var
           position: integer;
        fvar

        { check if this client exists into our shop }
        position = posClient(shop, dni);

        if position != -1 then
            { if client exists, update accumulated value }
            shop->clients[position].accumulated :=
        shop->clients[position].accumulated + amount;
        else
```

```
                { show 'client not registered' message }
                writeString("Error. Sorry, this client is not
        registered.");
        end if
end action
```

**d)** Una **acción generateDiscount** que,

1. dada una variable de tipo **tShop**,
2. se ejecuta a petición
3. y mira si puede generar cupones de descuento que los clientes podrán canjear en la tienda en la próxima compra.
4. El valor del **cupón de descuento** es del **3%** del importe de las compras **acumuladas** hasta el momento.
5. Este **cupón** se generará siempre y cuando el importe **acumulado** de las compras sea igual o superior a **100** € (se genera independientemente del valor del campo **thereIsCupon**).
6. Para generar el **cupón** se escribe un mensaje por pantalla con el texto ("Client *C*: new discount of *X* euros"),
   i. donde *C* representa el **DNI** del cliente
   ii. y *X* representa el **valor del cupón de descuento**.
7. En el caso que se genere el **cupón**, se inicializará el importe **acumulado** de las compras (se pone a cero) y se pone a **TRUE** el valor de campo **thereIsCupon**.
8. En el caso que no llegue al mínimo,
   i. no se generará ningún cupón de descuento
   ii. y el importe acumulado de las compras quedará intacto para la siguiente vez que se ejecute la acción.

```
action generateDiscount(inout shop: tShop)
        { variables declaration }
        var
                i: integer;
                discountCoupon: real;
                minimumForDiscount:real;
                discountRate:real;
        fvar

        { set default values to variables }
        minimumAccumulatedForDiscount := 100.0;
        discountRate := 0.03;
```

```
        { iterate over each client on our shop }
        for i:=1 to shop.numClients do
                { check if accumulated is enough for get a discount coupon  }
                if shop.clients[i].accumulated >= minimumForDiscount
then
                        { generate discount coupon }
                        discountCoupon := shop.clients[i].accumulated *
discountRate;

                        { show discount message }
                        writeString("Client %d: new  discount  of %d
                euros", shop.clients[i].dni, discountCoupon);

                        { reset accumulated value }
                        shop.clients[i].accumulated := 0;

                        { update thereIsCupon }
                        shop.clients[i].thereIsCupon = TRUE;
                end if
        end for
end action
```

**e)** (**Sólo en lenguaje C**) Una acción **testShop** que haga las pruebas necesarias para comprobar que el código implementado funciona en los siguientes casos (hay que mostrar el resultado por pantalla):

1. Añadimos un cliente a una lista vacía.
2. Añadimos un cliente a una lista no vacía.
    i. Este nuevo cliente va al principio de la lista.
    ii. Este nuevo cliente va entre otros clientes de la lista.
    iii. Este nuevo cliente va al final de la lista.
3. Intentamos insertar un cliente que ya existe.
4. Intentamos añadimos un nuevo cliente a una lista llena (es decir, ya contiene el número máximo de clientes que caben).
5. Verificamos la generación de descuentos a los clientes de la lista.
    i. Hay un cliente para el que el importe acumulado de las

compras es **igual o superior** a 100 €:

1. se genera cupón de descuento.

**ii.** Hay un cliente para el que el importe acumulado de las compras es **inferior** a 100 €:

1. no se genera cupón de descuento.

**Nota:**

- El código en lenguaje C de los apartados **a-d** debe ir en los ficheros **shop.c** y **shop.h**.
- Los test 1-5 de la acción **testShop** del apartado e, deben ir en los ficheros **tests.c** y **tests.h**.
- Añade una llamada en el método **main** a la acción **testShop**.