Prácticas de Programación
PEC2 - 20222

Fecha límite de entrega: **02 / 04 / 2023**

Formato y fecha de entrega

La PEC debe entregarse antes del día **2 de abril de 2023** a las 23:59.

Es necesario entregar un fichero en formato ZIP, que contenga:

- Un documento en formato **pdf** con las respuestas de los **ejercicios 1 y 3**. El documento debe indicar en su primera página el **nombre y apellidos** del estudiante que hace la entrega.
- Los ficheros **donation.h** y **donation.c** solicitados en el **ejercicio 2**.

Es necesario hacer la entrega en el apartado de entregas de EC del aula de teoría antes de la fecha de entrega.

Objetivos

- Saber interpretar y seguir el código de terceras personas.
- Saber realizar pequeños programas para solucionar problemas a partir de una descripción general del problema.
- Saber utilizar el tipo de datos puntero.
- Saber definir tipos de datos usando memoria dinámica.
- Saber reducir un problema dado en problemas más pequeños mediante el diseño descendente.

Criterios de corrección:

Cada ejercicio tiene asociada su puntuación sobre el total de la actividad. Se valorará tanto que las respuestas sean correctas como que también sean completas.

- No seguir el **formato de entrega**, tanto por lo que se refiere al **tipo y nombre de los ficheros** como al contenido solicitado, comportará una **penalización importante** o la cualificación con una **D de la actividad**.
- En los ejercicios en que se pide lenguaje algorítmico, se debe respetar el **formato.**
- En el caso de ejercicios en lenguaje C, estos **deben compilar para ser evaluados**. Si compilan, se valorará:
    - Que **funcionen** tal como se describe en el enunciado.
    - Que se respeten los **criterios de estilo y** que el código esté **debidamente comentado**.
    - Que las **estructuras** utilizadas sean las correctas.

Aviso

Aprovechamos para recordar que **está totalmente prohibido copiar en las PECs** de la asignatura. Se entiende que puede haber un trabajo o comunicación entre los estudiantes durante la realización de la actividad, pero la entrega de ésta debe ser individual y diferenciada del resto. Las entregas serán analizadas con **herramientas de detección de plagio.**

Así pues, las entregas que contengan alguna parte idéntica respecto a entregas de otros estudiantes serán consideradas copias y todos los implicados (sin que sea relevante el vínculo existente entre ellos) suspenderán la actividad entregada.

Guía citación: https://biblioteca.uoc.edu/es/contenidos/Como-citar/index.html
Monográfico sobre plagio:
http://biblioteca.uoc.edu/es/biblioguias/biblioguia/Plagio-academico/

Observaciones

Esta PEC continúa el proyecto que se desarrolla durante las distintas actividades del curso.

En este documento se utilizan los siguientes símbolos para hacer referencia a los bloques de diseño y programación:


Indica que el código mostrado es en **lenguaje** algorítmico.


Indica que el código mostrado es en **lenguaje** C.


Muestra la ejecución de un programa en **lenguaje** C.

Análisis dinámico

En esta actividad empezamos a utilizar memoria dinámica, que requiere que el programador reserve, inicialice y libere la memoria. Para ayudar a detectar memoria que no se ha liberado correctamente, o errores en las operaciones con punteros relacionadas, hay herramientas que ejecutan un análisis dinámico del programa. Una herramienta de código abierto muy empleada es Valgrind (https://valgrind.org/). La utilización de esta herramienta queda fuera del ámbito del curso, pero os facilitamos sus resultados como parte del análisis de la herramienta PeLP. Podéis acceder a los errores detectados por Valgrind en la pestaña de errores:

Para entender el significado de los **códigos de error**, podéis consultar el siguiente enlace, donde encontraréis ejemplos de código que os ayudarán a entender cuando se dan estos errores:

https://bytes.usc.edu/cs104/wiki/valgrind/

Enunciado

Siguiendo con el problema introducido en la PEC1, se ha decidido que dado el volumen de datos que habrá que gestionar, no podemos predecir con exactitud el número de personas que serán dadas de alta en el sistema. Por este motivo, se ha modificado la lista de personas para utilizar memoria dinámica, evitando la limitación de 100 personas inicial:

```
type
      tPeople = record
            elems : pointer to tPerson;
            count : integer;
      end record
end type
```

Además se nos ha facilitado los ficheros **person.h** y **person.c** con la declaración e implementación en lenguaje C de los tipos de datos y métodos básicos para la gestión de personas, ya incorporando el uso de memoria dinámica.

Ejercicio 1: Definición de tipos de datos [10%]

A partir de la descripción de los datos de entrada del enunciado, redefine **en lenguaje algorítmico** el tipo **tDonationData** para que pueda almacenar un número indeterminado de donaciones, eliminando la restricción de máximo 100 posiciones de la PEC1.

SOLUCIÓN

```
type
      tDonationData = record
            elems : pointer to tDonation;
            count : integer;
      end record

end type
```

Ejercicio 2: Manipulación de tablas con memoria dinámica [40%]

A partir de las estructuras de datos y los métodos definidos e implementados en la PEC1, **define o redefine** los siguientes métodos (acciones o funciones) en lenguaje C, para que se adapten a la nueva definición de **tDonationData** (del ejercicio 1). Utiliza tus ficheros **donation.h** y **donation.c** de la PEC1 como base o los de la solución publicada:

a) **donationData_init:** Dada una estructura de tipo **tDonationData,** la inicializa correctamente obteniendo una estructura vacía.
b) **donationData_add**: Dada una estructura de tipo **tDonationData** y una donación (**tDonation**), añade esta donación a la lista de donaciones (**tDonationData**) manteniendo la ordenación por el código del proyecto. Si ya existe una donación de la misma persona para la misma fecha y proyecto, la donación se ignorará y no se añadirá. Se recuerda que la persona se identifica con su documento de identidad y que los proyectos tienen un código único que los identifica.
c) **donationData_del:** Dada una estructura de tipo **tDonationData**, una fecha, el documento de una persona y un código de proyecto, se eliminará esa donación de la lista. En el caso de que no exista ninguna donación con esas características, entonces no se hará nada.
d) **[Nuevo método] donationData_free:** Dada una estructura de tipo **tDonationData**, elimina todo su contenido, obteniendo una estructura vacía.

Nota: Para este ejercicio disponéis de los ficheros **csv.h** y **csv.c** de la PEC1, y los ficheros **person.h** y **person.c** como referencia.

Debéis utilizar la rutina principal que se os facilita en el archivo **main.c** sin ninguna modificación. Esta rutina ejecuta las siguientes tareas (y muestra los datos almacenados después de cada paso):

1. Añade 3 donaciones.
2. Añade 2 donaciones nuevas.
3. Añade una donación ya existente.
4. Elimina 2 donaciones existentes.
5. Elimina una donación que no existe.

Para considerar que el programa funciona correctamente, el resultado final por pantalla deberá ser el siguiente:

```
0;15/04/2023;12345678N;ACN;ACN8455;15.50
1;30/12/2022;55755010C;CRJ;CRJ0333;3500.60
2;11/12/2022;76203765G;MSF;MSF0012;600.20

0;15/04/2023;12345678N;ACN;ACN8455;15.50
1;30/12/2022;55755010C;CRJ;CRJ0333;3500.60
2;12/05/2022;65239334J;GRP;GRP0020;23.00
3;25/08/2022;87654321B;MSF;MSF0000;100.00
4;11/12/2022;76203765G;MSF;MSF0012;600.20

0;15/04/2023;12345678N;ACN;ACN8455;15.50
1;30/12/2022;55755010C;CRJ;CRJ0333;3500.60
2;12/05/2022;65239334J;GRP;GRP0020;23.00
3;25/08/2022;87654321B;MSF;MSF0000;100.00
4;11/12/2022;76203765G;MSF;MSF0012;600.20

0;15/04/2023;12345678N;ACN;ACN8455;15.50
1;30/12/2022;55755010C;CRJ;CRJ0333;3500.60
2;25/08/2022;87654321B;MSF;MSF0000;100.00

0;15/04/2023;12345678N;ACN;ACN8455;15.50
1;30/12/2022;55755010C;CRJ;CRJ0333;3500.60
2;25/08/2022;87654321B;MSF;MSF0000;100.00
```

Nota: Los ficheros **person.h y person.c** se facilitan con el enunciado a modo de ejemplo, pero no se precisan para la solución del ejercicio.

Ejercicio 3: Diseño descendiente [50%]

Como parte de la gestión de las donaciones necesitamos conocer la lista de proyectos (**tProjectData**) existentes. Además, en base a las donaciones que ha recibido cada proyecto (**tProject**), podremos obtener qué presupuesto tiene. De manera que el presupuesto no es más que la suma de todas las donaciones que ha recibido el proyecto.

También es importante conocer qué personas hacen donaciones de manera regular a cada ONG. Y no menos importante es cuánto dinero aportan estos benefactores (**tBenefactor**) con las distintas donaciones que han hecho a los proyectos de la ONG. Nada impide que una misma persona haga donaciones a varias ONGs y, por tanto ser benefactor en más de una. Eso sí, cada ONG tendrá su propia contabilidad y una puede haber recibido más dinero que otra.

Finalmente, con todos los datos obtenidos podemos determinar para cada ONG (**tNGO**) qué proyectos está realizando, así como un listado de sus benefactores (**tBenefactorData**). A continuación se muestra la definición de los nuevos tipos de datos:

```
const
      MAX_PROJECTS: integer = 100;
      MAX_PEOPLE: integer = 100;
end const

type
     tProject = record
            code: string;
            ngoCode : string;
            budget: real;
      end record

      tProjectData = record
            elems: vector [MAX_PROJECTS] of tProject;
            count: integer;
      end record

      tBenefactor = record
            personDocument: string;
            amount: real;
      end record

      tBenefactorData = record
            elems: vector [MAX_PEOPLE] of tBenefactor;
            count: integer;
      end record

      tNGO = record
            code: string;
            name: string;
```

```
            projects: tProjectData;
            benefactors: tBenefactorData;
      end record
end type
```

Se pide implementar en **lenguaje algorítmico** la función:

```
function calculateBenefactorsNGO(donations: tDonationData, ngo: tNGO)
: tBenefactorData
```

que dada una lista de donaciones (**tDonationData**) y una ONG (**tNGO**), nos devuelve una lista de los benefactores de la ONG. Hay que tener en cuenta:

- La lista de donaciones puede contener donaciones a distintas ONGs.
- Para cada benefactor es necesario calcular cuánto dinero ha donado a la ONG.
- El monto aportado en cada donación será siempre positivo y superior a cero. Es una precondición del sistema que nos asegura algo obvio, que esperamos recibir dinero y no regalarlo.
- En la lista de beneficiarios devuelta la misma persona no puede aparecer dos veces. Además la lista estará ordenada alfabéticamente por el documento de la persona.
- Para resolver el problema utiliza la definición de **tDonation** y de **tDonationData** de la solución de la PEC1, que te recordamos a continuación.

```
type
      tDonation = record
            date: tDate;
            personDocument: string;
            ngo: string;
            projectCode: string;
            amount: real;
      end record

      tDonationData = record
            elems: vector [MAX_DONATIONS] of tDonation;
            count: integer;
      end record
end type
```

Descompón esta función en acciones o funciones más simples utilizando diseño descendente e implementa **en lenguaje algorítmico** las acciones y funciones resultantes.

Solución

Nivel 1:

```
function calculateBenefactorsNGO (donations: tDonationData, ngo: tNGO) :
tBenefactorData

var
   i: integer;
   benefactors: tBenefactorData;
end var

benefactors.count := 0;

{ Loop through donations list }
for i:=1 to donations.count do
      if donations.elems[i].ngo = ngo.name then
      addBenefactor(donations.elems[i].personDocument,
                    donations.elems[i].amount, benefactors);
      end if
end for

return benefactors;
end function
```

Nivel 2:

```
action addBenefactor(in document: string, in amount: real, inout
benefactors: tBenefactorData)

var
   i: integer;
   stop: boolean;
   found: boolean;
end var

{ Loop through benefactors list }
i := 1;
stop := false;
found := false;
while i <= benefactors.count and stop = false and found = false do
      if benefactors.elems[i].personDocument = document then
          found := true;
      else if benefactors.elems[i].personDocument > document then
          stop := true;
      else
          i := i + 1;
      end if
end while
```

```
if found = true then
      { Increase amount }
      benefactors.elems[i].amount := benefactors.elems[i].amount + amount;
else
      { Insert new benefactor }
      benefactors.count := benefactors.count + 1;
      if stop = true then
          { Shift elements to insert in order }
          shiftBenefactors(i, benefactors);
      end if
      insertBenefactor(i, document, amount, benefactors);
end if

end action
```

Nivel 3:

```
action shiftBenefactors(in position: integer, inout benefactors:
tBenefactorData)

var
   i: integer;
end var

i := benefactors.count;
while i > position do
      { Move elements to the right until reaching the position }
      benefactors.elems[i]:= benefactors.elems[i-1];
      i := i-1;
end while

end action


action insertBenefactor(in position: integer, in document: string, in
amount: real, inout benefactors: tBenefactorData)
{ Insert new benefactor }
benefactors.elems[position].personDocument := document;
benefactors.elems[position].amount := amount;

end action
```

Nota: Aquí os proponemos una posible solución, pero hay otras válidas.