Prácticas de Programación
PEC2 - 20212

Fecha límite de entrega: **20 / 03 / 2022**

Formato y fecha de entrega

La PEC debe entregarse antes del día **20 de marzo de 2022** 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 **vaccine.h** y **vaccine.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 es respeten los **criterios de estilo y** que el código esté **debidamente comentado**.
    - Que les **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 esta debe que 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 presenta el proyecto que se desarrollará durante las distintas actividades del semestre, que se ha simplificado y adaptado a las necesidades académicas.

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 estarán dadas de alta en el sistema de salud. Por este motivo, se ha modificado la lista de personas para utilizar memoria dinámica, evitando la limitación de 100 personas inicial:

```
type
      tPopulation = 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 **tVaccineLotData** para que pueda almacenar un número indeterminado de posiciones, eliminando la restricción de máximo 100 posiciones de la PEC1.

SOLUCIÓN

```
type
      tVaccineLotData = record
            elems : pointer to tVaccineLot;
            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 **tVaccineLotData** (del ejercicio 1). Utiliza tus ficheros **vaccine.h** y **vaccine.c** de la PEC1 como base o los de la solución publicada:

a) **vaccineData_init:** Dada una estructura de tipo **tVaccineLotData** la inicializa correctamente obteniendo una estructura vacía.
b) **vaccineData_add**: Dada una estructura de tipo **tVaccineLotData**, y un lote de vacunas (**tVaccineLot)**, añade este lote a la lista de lotes de vacunas (**tVaccineLotData**). Si ya existe un lote para el mismo código postal, fecha, hora y tipo de vacuna (nombre), se sumará el número de dosis al lote ya existente.
c) **vaccineData_del:** Dada una estructura de tipo **tVaccineLotData**, un código postal, una estructura de tipo **tDateTime**, el nombre de la vacuna, y un número de dosis, reduce el número de dosis en el lote correspondiente. Si el número de dosis queda en cero o negativo, se eliminará la información del lote. En el caso de que no existiera ningún lote para el código postal, momento de tiempo y tipo de vacuna (nombre), no se hace nada.
d) **[Nuevo método] vaccineData_free:** Dada una estructura de tipo **tVaccineLotData**, elimina todos 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 fichero **main.c** sin ninguna modificación. Esta rutina ejecuta las siguientes tareas:

1. Añade 3 lotes de vacunas.
2. Añade otros 3 lotes de vacunas.
3. Elimina parcialmente un lote de vacunas.
4. Elimina las dosis restantes del lote anterior.
5. Elimina un lote de vacunas que no existe.
6. Añade N>100 nuevos lotes de vacunas.
7. Elimina todos los datos.

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

```
0;01/01/2022;13:45;08000;PFIZER;2;21;300
1;01/02/2022;11:00;08018;MODERNA;1;0;150
2;01/03/2022;17:15;25001;PFIZER;2;21;100

0;01/01/2022;13:45;08000;PFIZER;2;21;350
1;01/02/2022;11:00;08018;MODERNA;1;0;350
2;01/03/2022;17:15;25001;PFIZER;2;21;350

0;01/01/2022;13:45;08000;PFIZER;2;21;350
1;01/02/2022;11:00;08018;MODERNA;1;0;300
2;01/03/2022;17:15;25001;PFIZER;2;21;350

0;01/01/2022;13:45;08000;PFIZER;2;21;350
1;01/03/2022;17:15;25001;PFIZER;2;21;350

0;01/01/2022;13:45;08000;PFIZER;2;21;350
1;01/03/2022;17:15;25001;PFIZER;2;21;350

Added new 150 vaccine lots

No vaccine lots
```

Note: 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 la vacunación, necesitamos saber cuántas vacunas tendremos en cada centro de salud (**tHealthCenter**). Cada centro tiene una previsión de las vacunas de las que dispondrá cada día (**tVaccineStockData**), que se calcula a partir de los lotes que se prevén recibir (**tVaccineLotData**) y las vacunas que se prevé administrar. Para saber cuántas vacunas se administrarán cada día, el centro gestiona una agenda con las citas de vacunación (**tAppointmentData**) que tiene programadas. A continuación se muestra la definición de los nuevos tipos de datos:

```
const
      MAX_STOCKS: integer = 20;
      MAX_APPOINTMENTS: integer = 20;
end const

type
      tVaccineStock = record
            day : tDate;
            vaccine : tVaccine;
            doses : integer;
      end record

      tVaccineStockData = record
            elems : vector [MAX_STOCKS] of tVaccineStock;
            count : integer;
      end record

      tAppointment = record
            timestamp : tDateTime;
            person : tPerson;
            vaccine : tVaccine;
      end record

      tAppointmentData = record
            elems : vector [MAX_APPOINTMENTS] of tAppointment;
            count : integer;
      end record

      tHealthCenter = record
            cp: string;
            appointments: tAppointmentData;
            stock: tVaccineStockData;
      end record
```

```
end type

{Compare two dates and return -1 if date1 < date2, +1 if date1 > date2 and
0 if both dates are equals}
function date_cmp(in date1: tDate, in date2: tDate): integer;

{Adds a day to a date}
action date_inc(inout date: tDate);
```

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

```
action compute_stock(in vaccineLots: tVaccineLotData, inout center:
tHealthCenter, in date: tDate)
```

que dado un centro de salud (**tHealthCenter**), una lista de lotes de vacunas (**tVaccineLotData**) y una fecha (**tDate**), calcule el número de vacunas diarias hasta la fecha dada, empezando por la fecha de recepción del primer lote de vacunas (**tVaccineLot**). Hay que tener en cuenta:

- El número de vacunas disponibles en un día dado se calcula a partir de:
  - El número de vacunas del día anterior (inicialmente asumimos que ningún centro de salud tiene vacunas disponibles)
  - Los lotes de vacunas que llegan al centro de vacunación (identificado con su código postal) en ese día.
- Al número de vacunas disponibles, hay que descontar las que se administrarán ese día, según las citas de vacunación.
- Podéis asumir que inicialmente el stock de vacunas del centro está vacío.
- Podéis asumir que los lotes de vacunas están ordenados por fecha y hora dentro de **tVaccineLotData**.
- Puede darse el caso de que el número de dosis de una vacuna sea negativo (hay más citas de vacunación que vacunas disponibles).
- Los elementos (**tVaccineStock**) dentro del stock del centro (**tVaccineStockData**) deben quedar **ordenados ascendentemente** por fecha.

Para realizar este ejercicio podéis asumir que tenéis ya implementados los métodos:

- **date_cmp**: Compara dos fechas y devuelve:
  - -1 si la primera fecha es menor a la segunda.
  - 0 si ambas fechas son iguales.
  - +1 si la primera es mayor a la segunda.
- **date_inc:** Dada una fecha, la incrementa en un día.

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

Solución

Nivel 1:

```
action compute_stock(in vaccineLots: tVaccineLotData, inout center:
tHealthCenter, in date: tDate)

var
   currentDate: tDate;
end var

{ Initialize the stock }
initStock(center.stock);

{ If the list of vaccine lots have elements, start updating the stock }
if vaccineLots.count > 0 then
    { Get the initial date from the list of lots }
    currentDate := vaccineLots.elems[1].timestamp.date;

    { Iterate day by day }
    while date_cmp(currentDate, date) <= 0 do
        { Update the stock with vaccine lots }
        updateStock(center, vaccineLots, currentDate);

        { Discount doses from appointments }
        processAppointments(center.stock, center.appointments,
                            currentDate);

        { Move to next day }
        date_inc(currentDate);
    end while
endif

end action
```

Nivel 2:

```
action initStock(inout stock: tVaccineStockData)
    stock.count := 0;
end action
```

```
action updateStock(inout center: tHealthCenter, in lots: tVaccineLotData,
in date: tDate)
var
   i: integer;
   stop: boolean;
   cmp: integer;
end var

{ Search over lots }
i := 1;
stop := false;
while i <= lots.count and stop = false do
    { Compare dates }
    cmp := date_cmp(lots.elems[i].timestamp.date, date);
    if cmp > 0 then
       { Stop if date is after current date }
       stop := true;
    else if cmp = 0 and lots.elems[i].cp = center.cp  then
       { Update if it is for today }
       addDoses(center.stock, date, lots.elems[i].vaccine,
                lots.elems[i].doses);
    end if
    i := i + 1;
end while
end action



action processAppointments(inout stock: tVaccineStockData, in
appointments: tAppointmentData, in date: tDate)
var
   i: integer;
   stop: boolean;
end var

{ Traverse all appointments }
for i := 1 to appointments.count do
    if date_cmp(date, appointments.elems[i].timestamp.date) = 0 then
       { Update if it is for today }
       addDoses(stock, date, appointments.elems[i].vaccine, -1);
    end if
end for
end action
```

Nivel 3:

```
action addDoses(inout stock: tVaccineStockData, in date: tDate, in
vaccine: tVaccine, in doses: integer)
var
   i: integer;
   stockDoses: integer;
   found: boolean;
end var

{ Initialize the number of available doses for this vaccine }
stockDoses := 0;

{ Traverse all the stocks }
found := false;
for i := 1 to stock.count do
   if stock.elems[i].vaccine.name = vaccine.name then
      if data_cmp(stock.elems[i].day, date) < 0 then
         { While we are not on the given date, just store doses }
         stockDoses := stock.elems[i].doses;
      else
         { Update the number of doses}
         found := true;
         stock.elems[i].doses := stock.elems[i].doses + doses;
      end if
   end if
end for

{ Insert a new register at the end if this vaccine was not found }
if found = false then
   stock.count := stock.count + 1;
   stock.elems[stock.count].day := date;
   stock.elems[stock.count].vaccine := vaccine;
   stock.elems[stock.count].doses := stockDoses + doses;
end if

end action
```

Nota: Aquí os proponemos una posible solución, pero hay otras. Por ejemplo, en este caso no se añade ningún elemento para fechas anteriores a la aparición de una vacuna, pero se podría añadir un elemento con 0 dosis.