Prácticas de Programación
PR2 - 20212

Fecha límite de entrega: **27 / 04 / 2022**

Formato y fecha de entrega

La práctica debe entregarse antes del día **27 de abril de 2022** a las 23:59.

Es necesario entregar un fichero en formato **ZIP**, que contenga una carpeta **UOC20212** con el directorio principal de vuestro proyecto, siguiendo la estructura de carpetas y nombres de ficheros especificados en el enunciado de la práctica. No debe contener ningún fichero ZIP en su interior. Esta carpeta debe contener:

- Un fichero **README.txt** con el siguiente formato (ver ejemplo):

```
Formato:
Correo electrónico UOC
Apellidos, Nombre
Sistema operativo utilizado

Ejemplo:
estudiante1@uoc.edu
Apellido1 Apellido2, Nombre
Windows 10
```

- Los ficheros de prueba sin modificaciones.
- Los ficheros *.c y *.h resultantes de los ejercicios realizados.
- Los ficheros .workspace y .project que definen el espacio de trabajo y los proyectos de Codelite.
- Todos los ficheros deben estar dentro de las carpetas correctas (src, test, …).

La entrega debe realizarse en el apartado de entregas de EC del aula de teoría antes de la fecha límite de la entrega. **Únicamente el último envío** dentro del periodo establecido será evaluado.

El incumplimiento del formato de entrega especificado anteriormente puede suponer un suspenso de la práctica.

Objetivos

- Saber interpretar y seguir el código de terceras personas.
- Saber compilar proyectos de código organizados en carpetas y librerías.
- Saber implementar un proyecto de código a partir de su especificación.

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**.
- El código entregado **debe compilar para ser evaluado**. Si compila, se valorará:
    - Que **funcionen** tal como se describe en el enunciado.
    - Que obtenga el **resultado esperado** dadas unas condiciones y datos de entrada diseñadas (pruebas proporcionadas).
    - Que se respeten los **criterios de estilo y** que el código esté **debidamente comentado**. Se valorará especialmente el uso de comentarios en inglés.
    - Que les **estructuras** utilizadas sean las correctas.
    - Que se **separe correctamente la declaración e implementación** de las acciones y funciones, utilizando los ficheros correctos.
    - El **grado de optimización** en tiempo y recursos utilizados en la solución entregada.
    - Que se realice una **gestión de memoria** adecuada, liberando la memoria cuando sea necesario.

Aviso

Aprovechamos para recordar que **está totalmente prohibido copiar en las PECs y prácticas** 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

El punto de partida para esta práctica es el resultado de la PR1. El código facilitado con el enunciado incorpora todos los archivos necesarios de las actividades anteriores, con lo que veréis que todas las pruebas de la PR1 pasan correctamente.

En esta práctica implementaremos la estructura de datos planteada en la PEC3 (con pequeñas modificaciones) y los métodos relacionados para poder gestionar el stock de vacunas:

```
type
      tVaccine = record
            name : string;
            required : integer;
            days : integer;
      end record

      tVaccineStock = record
            vaccine : pointer to tVaccine;
            doses : integer;
      end record

      tVaccineStockNode = record
            elem : tVaccineStock;
            next : pointer to tVaccineStockNode;
      end record

      tVaccineDailyStock = record
            day : tDate;
            first : pointer to tVaccineStockNode;
            next : pointer to tVaccineDailyStock;
            count : integer;
      end record

      tVaccineStockData = record
            first : pointer to tVaccineDailyStock;
            last : pointer to tVaccineDailyStock;
            count : integer;
      end record
end type
```

Recordad que se guardan los estocs de las distintas vacunas disponibles por cada día. En esta práctica, implementaremos la gestión de los estocs para los distintos centros de salud.

Ejercicio 1: Gestión de los estocs [50%]

En el código proporcionado encontraréis los ficheros ***stock.h*** y ***stock.c*** con la definición de los tipos de datos anteriores y los principales métodos relacionados. En el fichero ***stock.c*** implementad:

a) El método ***stockList_init*** que dada una estructura de tipo **tVaccineStockData**, la inicialice correctamente. Inicialmente, no habrá dosis para ningún día.

b) El método ***stockList_update*** que dada una estructura de tipo **tVaccineStockData**, una fecha **tDate**, el puntero a la definición de una vacuna **tVaccine** y un número de dosis, modifica el número de dosis de esta vacuna (aumentandolo o disminuyéndolo). Se debe tener en cuenta:

i) Si el estoc está vacío, o no hay dosis para esta vacuna, el número de dosis para esta vacuna se añade directamente en la fecha correspondiente.
ii) Si existen dosis para la vacuna, se deberá modificar las entradas posteriores a la fecha afectada para reflejar el cambio. En el caso de que las entradas fueran anteriores, se deberá añadir los elementos necesarios.
iii) Todos los elementos están ordenados por fecha, y entre dos fechas solo hay un día de diferencia.
iv) Si el número de dosis para una fecha es 0, se debe eliminar el registro de esa vacuna de la fecha correspondiente.
v) La primera/última fecha de la lista de estocs debe tener datos, en caso contrario se debe eliminar del estoc.
vi) Dentro de una fecha, las vacunas deben estar ordenadas por nombre.

Nota: Al final del ejercicio tenéis tres ejemplos de la evolución del estoc a partir de un estado inicial y varias llamadas al método ***stockList_update*** con una fecha, vacuna y número de dosis. En los ficheros **date.h** y **date.c** encontraréis el método ***date_addDay***, que os permite añadir o restar días a una fecha dada. También encontraréis el método ***date_cmp*** que permite comparar dos fechas.

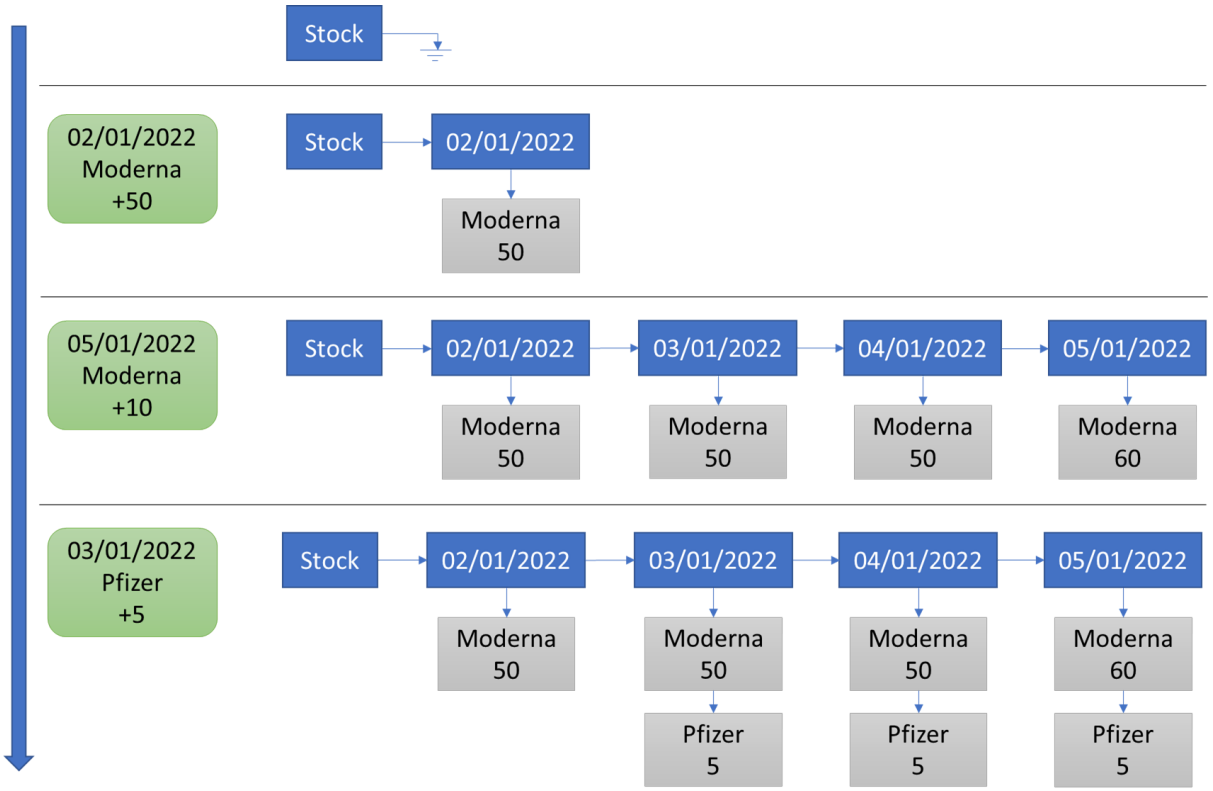
Nota: Para este ejercicio es muy recomendable aplicar la metodología del diseño descendente, definiendo todos los métodos adicionales que consideréis.

c) El método ***stockList_getDoses*** que dada una estructura de tipo **tVaccineStockData**, una fecha **tDate**, y el puntero a la definición de una vacuna **tVaccine** nos retorna el número de dosis de esta vacuna en estoc. Se debe tener en cuenta:

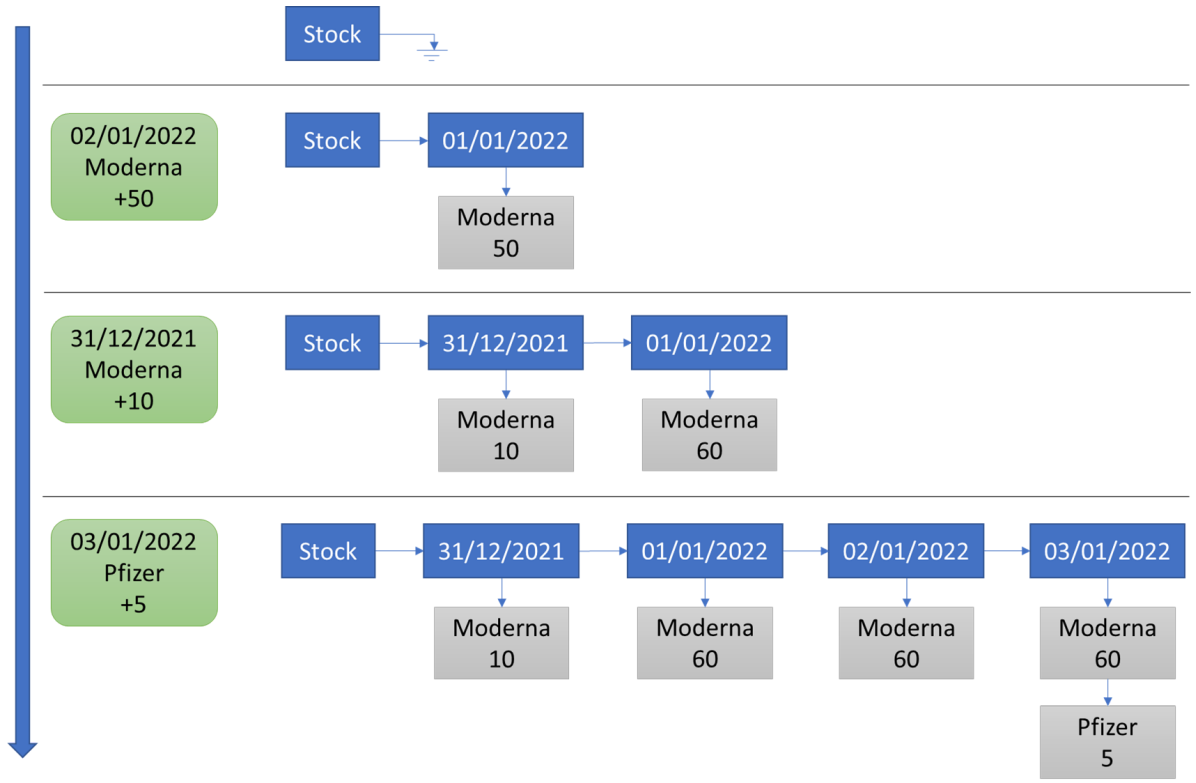
i) Si existe la fecha, pero no hay datos para esta vacuna, consideramos que el número de dosis es 0.

ii) Si la fecha es anterior a la primera fecha de que tenemos estoc, consideramos que el número de dosis es 0.

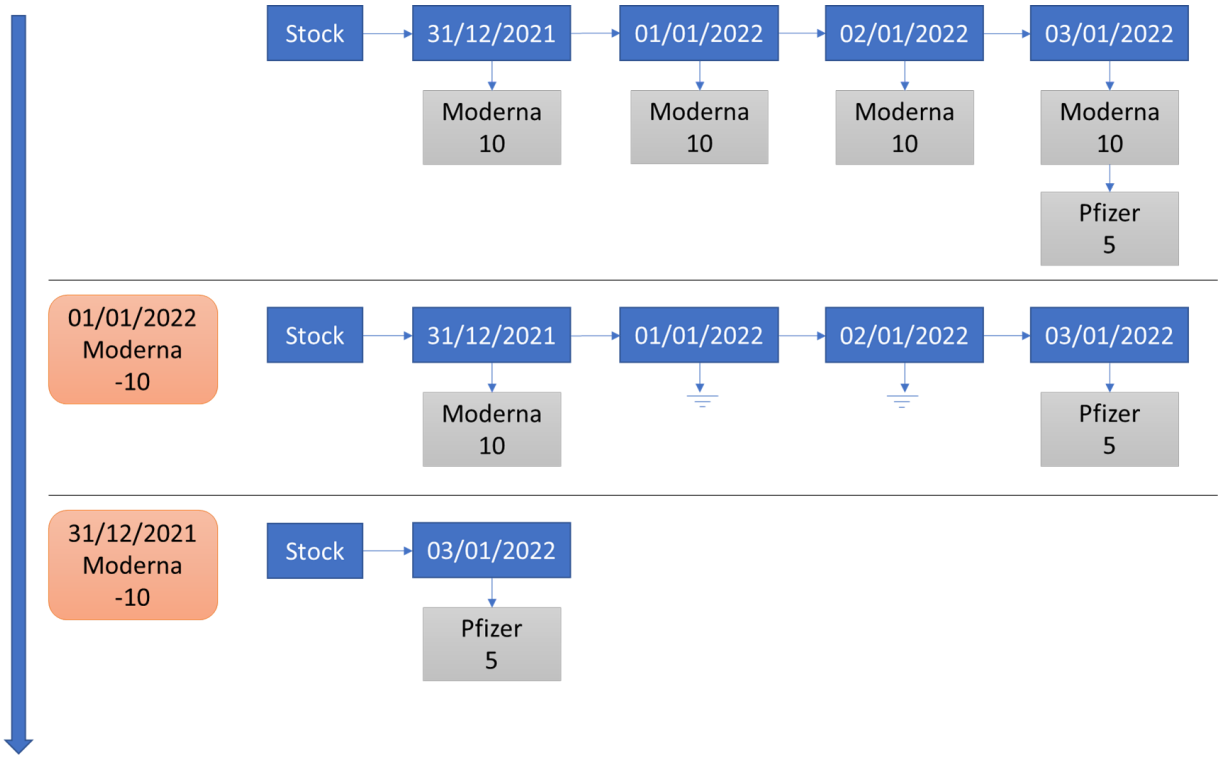
iii) Si la fecha es posterior a la última fecha de que tenemos estoc, el número de dosis corresponderá al número de dosis de la última fecha.



Ejemplo 1: *A partir de una lista vacía, añadimos tres lotes de vacunas.*



Ejemplo 2: *A partir de una lista vacía, añadimos tres lotes de vacunas.*



Ejemplo 3: *A partir de una lista iniciada, eliminamos vacunas dos veces.*

Ejercicio 2: Gestión de los centros de salud [30%]

En el código proporcionado encontraréis los archivos **center.h** y **center.c** con la definición de los tipos de datos para guardar los centros de salud y la declaración de los métodos relacionados. En el archivo ***center.c*** implementad:

a) El método ***center_init***, que dado un objeto de tipo **tHealthCenter**, lo inicie con los datos de un centro. Inicialmente, el centro tendrá el estoc de vacunas vacío.
b) El método ***center_free***, que dado un objeto de tipo **tHealthCenter**, elimine los datos que contiene. Como parte de la implementación, será necesario implementar también el método ***stockList_free*** del archivo **stock.c**, que elimine todos los datos del stock del centro.
c) El método ***centerList_init*** que dada una lista de centros **tHealthCenterList** la inicialice.
d) El método ***centerList_free*** que dada una lista de centros **tHealthCenterList** elimine todos los datos que contiene.
e) El método ***centerList_insert*** que dada una lista de centros **tHealthCenterList** y el código postal de un nuevo centro, añade este nuevo centro a la lista de centros. Los centros deben quedar ordenados por código postal de menor a mayor. En caso de que el centro ya exista, no se hace nada.
f) El método ***centerList_find*** que dada una lista de centros **tHealthCenterList** y el código postal de un centro, devuelve el puntero al centro **tHealthCenter** con ese código postal. En caso de que no exista ningún centro con este código postal, el método devolverá un valor NULL.

Ejercicio 3: Integración en la API [20%]

Finalmente, queremos incorporar la gestión de los centros y sus respectivos estocs de vacunas en la estrucutra del API, definida e implementada en los ficheros ***api.h*** y ***api.c***.

Se pide:

a) Completa la definición del tipo de datos ***tApiData*** del archivo **api.h** para que guarde la lista de los centros de salud ***tHealthCenterList*** definida en el ejercicio anterior.

b) Modifica la función **api_initData** del archivo **api.c** para que inicialice la lista de centros de salud.

c) Modifica el método **api_addVaccineLot** del archivo **api.c** para que cuando se añada un nuevo lote de vacunas:

- **i)** Si el centro de salud al que va el lote no existe, añada este centro a la lista de centros y actualice su stock de vacunas.
- **ii)** Si ya existe el centro de salud, actualice el stock de vacunas de ese centro.

d) Modifica el método **api_freeData** del archivo **api.c**, para que elimine todos los datos referentes a centros de salud.

e) Implementa el método **api_centersCount** del archivo **api.c**, que dada una estructura de tipo tApiData, nos devuelva el número de centros de salud que contiene.