Prácticas de Programación
PR1 - 20212

Fecha límite de entrega: **03 / 04 / 2022**

Formato y fecha de entrega

La práctica debe entregarse antes del día **3 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:

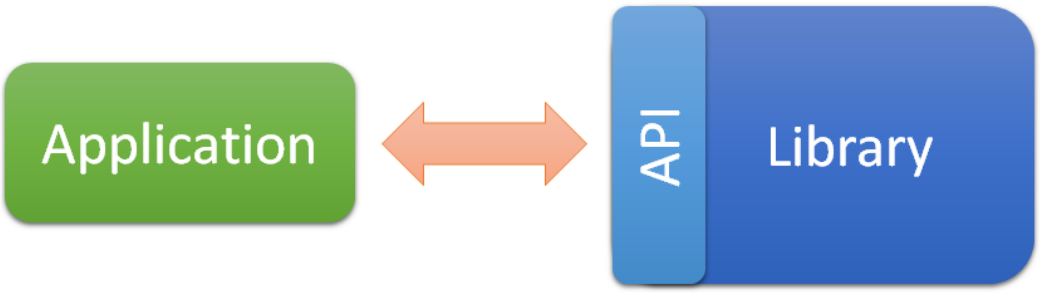
https://bytes.usc.edu/cs104/wiki/valgrind/

Enunciado

En las PECs hemos trabajado de forma aislada partes del problema introducido en la PEC1. En las prácticas veremos como construir una aplicación más compleja que vaya incorporando de forma incremental lo que se va trabajando en las PECs.

Es habitual que las aplicaciones definan una API (Application Programming Interface) o interfaz de programación de aplicaciones. Básicamente se trata de una abstracción de nuestra aplicación, en la cual definimos los métodos y los datos, y permitimos que otras aplicaciones puedan interactuar con nuestra aplicación sin necesidad de saber como se han implementado estos métodos.

Además, encapsularemos todas las funcionalidades en una librería, que podrá ser utilizada por cualquier programa. En la siguiente figura se muestra la estructura de la práctica, en que tendremos una aplicación (ejecutable) que utilizará los métodos (acciones y funciones) de la API, implementada en una librería.



A nivel de código, lo que tendremos será un espacio de trabajo (Workspace) con dos proyectos:

- **Aplicación**: Será un proyecto igual al que utilizamos en las PECs, creado como un ejecutable simple.
- **Librería:** Será un proyecto de tipo “Static library”. En este caso, el resultado de la complicación y entrelazado no produce un ejecutable, sino un fichero .a o .lib dependiendo del sistema operativo. Este fichero se puede utilizar desde otra librería o desde una aplicación. A diferencia de las aplicaciones, las librerías no implementan el método *main*.

Ejercicio 1: Preparación del entorno [20%]

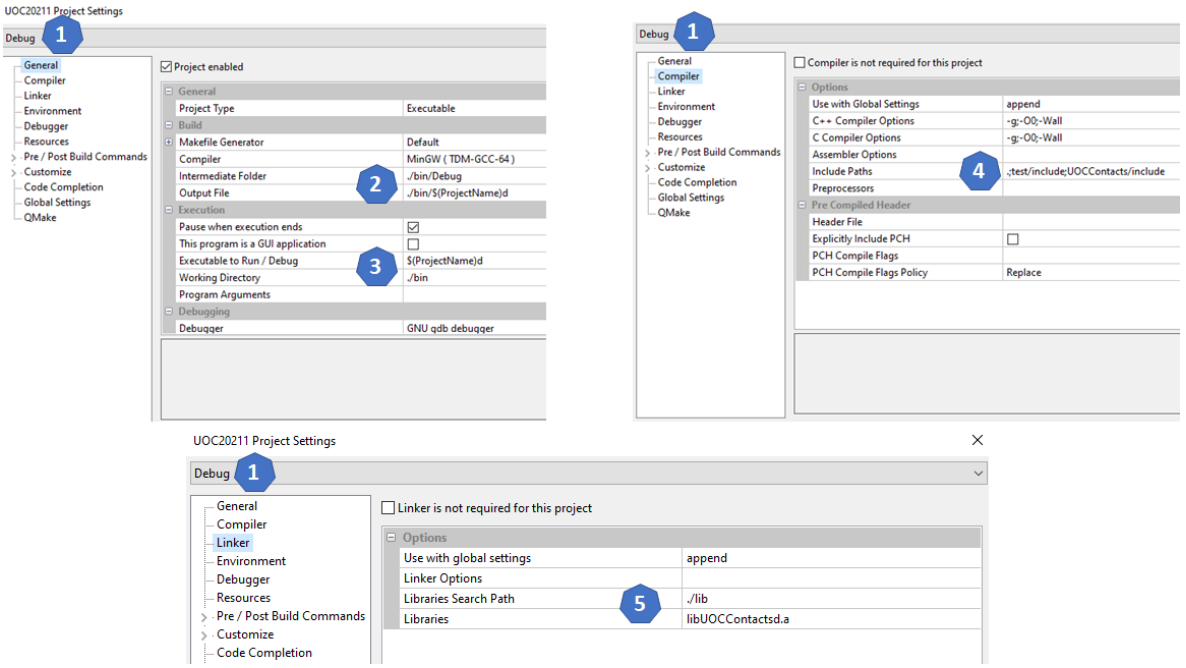
Junto con el enunciado se facilita un fichero de código con un espacio de trabajo (Workspace) que contiene dos proyectos. A continuación se detallan las principales características de cada uno de ellos:

- **UOCVaccine**: Este proyecto corresponde a la librería donde iremos añadiendo toda la funcionalidad de la práctica.
    - El código se divide en declaraciones (include) e implementaciones (src). Los ficheros **api.h** y **api.c** contienen la declaración de la API, y por lo tanto, los métodos que se accederán desde la aplicación.
    - Cuando se compila, debe ir a buscar los ficheros de cabecera en la carpeta include.
    - La librería se debe generar en la carpeta “lib” del Workspace.
    - El nombre de la librería añadirá una “d” cuando se compile en modo Debug.

- **UOC20212:** Este proyecto es nuestra aplicación. Se encarga de ejecutar las diferentes pruebas para verificar el correcto funcionamiento de la librería.
    - El código principal está en la carpeta src, y el código de las pruebas en la carpeta test, separando las declaraciones (include) y la implementación (src).
    - Cuando se compila, debe ir a buscar los ficheros de cabecera (*.h) tanto en la carpeta de las pruebas (test/include) como en la carpeta correspondiente de la librería (UOCVaccine/include).
    - Cuando se hace el entrelazado (link), se debe indicar que vaya a buscar las librerías en el directorio lib del Workspace, y que incluya la librería generada por el proyecto anterior.

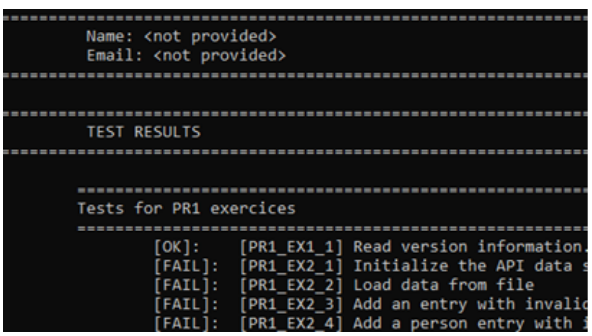
El objetivo de este ejercicio es tener el entorno proporcionado funcionando. Por tanto, será necesario que modifiquéis las opciones de los proyectos para que os funcione.

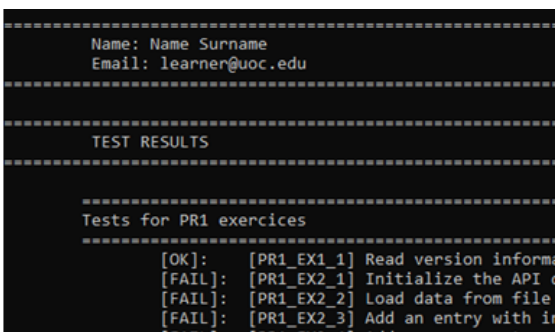
A continuación se muestra una guía de las opciones (settings) de los proyectos en donde se definen las características anteriores. Recordad que para acceder a las opciones del proyecto lo podéis hacer a través del menú contextual (botón derecho) y la opción Settings.



1. Permite cambiar entre la configuración de Debug y Release.
2. Define dónde se generan los ficheros resultantes
3. Define qué se ejecuta cuando se utiliza el botón de play de Codelite ( ). Se debe indicar el directorio de trabajo y el nombre de la aplicación, que será distinta en Debug y en Release.
4. Define en qué directorios se van a buscar los ficheros de cabecera.
5. Define en qué directorios se va a buscar las librerías y qué librerías se deben añadir para generar la aplicación.

Cuando se hayan seleccionado las opciones correctas, al ejecutar debéis ver el resultado de las pruebas. Inicialmente, todas, excepto la del ejercicio 1 aparecen como fallidas. También os saldrá que el nombre y el correo electrónico no se ha proporcionado (izquierda). Introducid los datos en el fichero README.txt del Workspace tal como se indica en el apartado de entrega, y se deberían incorporar.



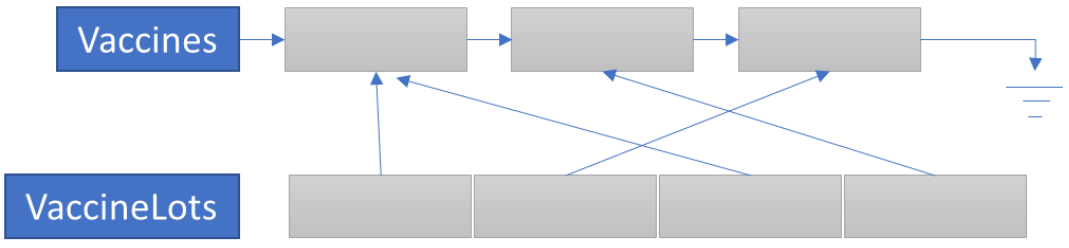


Ejercicio 2: Entrada de datos [40%]

Hasta ahora hemos estado trabajando solamente con los datos de los lotes de vacunas ***tVaccineLotData***. Para facilitar la interacción con la API, queremos agrupar todos los datos con los que trabajamos en una única estructura de datos (***tApiData***). Esta estructura debe guardar los siguientes datos:

- **Population**: Conjunto de personas dadas de alta en el sistema de salud. Cada persona está identificada de forma única a partir de su documento de identidad. Los datos de una persona se guardan en un tipo ***tPerson***. El conjunto de personas se guardan en una tabla (***tPopulation***).
- **Vaccines**: Lista de vacunas aceptadas por el sistema de salud. Cada vacuna se identifica de forma única a partir de su nombre. Los datos de una vacuna se guardan en una estructura de tipo ***tVaccine***. La lista de vacunas se guarda ordenada por nombre en una lista encadenada (**tVaccineList**).
- **VaccineLots**: Lotes de vacunas que se recibirán en cada centro de salud. Los datos de un lote de vacunas se guardan en una estructura de tipo ***tVaccineLot***. Los lotes de vacunas disponibles son guardados en una tabla (***tVaccineLotData***).

Con el fin de evitar guardar la información de las vacunas varias veces, cuando se almacenan los lotes de vacunas no se guarda la información de la vacuna, sino que se ha redefinido el tipo ***tVaccineLot*** para que apunte a la vacuna correspondiente dentro de la lista de vacunas (ver archivo **vaccine.h**). A continuación se muestra gráficamente una representación de la lista de vacunas y cómo los elementos de la tabla de lotes apuntan a la vacuna correspondiente:



Como parte del enunciado de la práctica se proporcionan los archivos:

- **api.h/api.c** con la declaración e implementación de los tipos de datos y métodos de la API.
- **error.h** con la declaración de los tipos de error que devolverá la API.

- **csv.h/csv.c** con la declaración e implementación de los tipos de datos y métodos relacionados con la gestión de datos en formato CSV.
- **date.h/date.c** con la declaración e implementación de los tipos de datos y métodos relacionados con la manipulación de fechas.
- **person.h/person.c** con la declaración e implementación de los tipos de datos y métodos relacionados con la gestión de personas.
- **vaccine.h/vaccine.c** con la declaración e implementación de los tipos de datos y métodos relacionados con la gestión de vacunas. Tened en cuenta que la definición de los datos de un lote de vacunas difiere de la utilizada en la PEC2.

Estos datos se consultarán y manipularán a través de los métodos de la API (definidos en el fichero **api.h**), los cuales retornarán generalmente un valor de tipo ***tApiError*** que indicará si se ha producido algún error o si la acción se ha ejecutado correctamente. Encontraréis los códigos de error definidos en el fichero **error.h** de la librería.

Se pide:

a) Completa la definición del tipo de datos **tApiData** del fichero **api.h**, para que se guarden todos los datos especificados.

b) Implementa la función **api_initData** del fichero **api.c**, que inicializa una estructura de tipo ***tApiData*** dada. Los valores de retorno de esta función se detallan en la siguiente tabla:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.

c) Implementa la función **api_addVaccineLot** del fichero **api.c**, para que dada una estructura de tipo ***tApiData*** y un lote de vacunas en formato csv ***tCSVEntry***, añada este lote a los datos de la aplicación. Se debe tener en cuenta:

i) Si la vacuna no existe en la lista de vacunas, será necesario añadirla.
ii) 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.

Para cada dato, se deberá comprobar que el formato sea correcto (asumimos que es correcto si el número de campos es el esperado), y que su tipo es "VACCINE_LOT" (podéis acceder al tipo mediante el método ***csv_getType***). Los valores de retorno de esta función se detallan en la siguiente tabla:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.
E_INVALID_ENTRY_TYPE	El tipo de dato es incorrecto.
E_INVALID_ENTRY_FORMAT	El formato del dato no es correcto.

Nota: En los archivos **vaccine.h y vaccine.c** encontraréis una nueva definición de ***vaccineLot_parse***, que devuelve la información de la vacuna en una estructura de tipo ***tVaccine*** y el lote en una estructura ***tVaccineLot*** sin la información de la vacuna. También encontraréis los métodos para gestionar la lista de vacunas ***tVaccineList***. También tenéis los métodos **vaccine_free y vaccineLot_free** para eliminar la memoria dinámica reservada para guardar los datos de vacunas y lotes.

d) Implementa los métodos **api_populationCount**, **api_vaccineCount** y **api_vaccineLotsCount**, que dada una estructura de datos de tipo ***tApiData*** devuelven el número de personas, vacunas y lotes de vacuna respectivamente. En el caso de los lotes de vacuna, los lotes que se hayan fusionado por ser de la misma vacuna, centro, fecha y hora, contarán como un único lote.

e) Implementa la función **api_freeData** del archivo **api.c**, que elimina toda la información guardada en una estructura de tipo ***tApiData*** dada. Los valores de retorno de esta función se detallan en la siguiente tabla:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.

f) Implementa la función **api_addDataEntry** del fichero **api.c**, que dada una estructura de tipo ***tApiData*** y un nuevo dato en formato csv ***tCSVEntry***, guarda este nuevo dato dentro de la estructura ***tApiData***. Una entrada de

datos puede ser de tipo “PERSON” o “VACCINE_LOT” (podéis acceder al tipo mediante el método ***csv_getType***). Para cada dato, será necesario comprobar que el formato sea correcto (asumimos que es correcto si el número de campos es el esperado). Los valores de retorno de esta función se detallan en la siguiente tabla:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.
E_INVALID_ENTRY_TYPE	El tipo de dato es incorrecto.
E_INVALID_ENTRY_FORMAT	El formato del dato no es correcto.
E_DUPLICATED_PERSON	Se está intentando añadir una persona que ya está registrada.

Ejercicio 3: Acceso a los datos [40%]

Con la finalidad de no exponer los tipos de datos internos de la API, se ha decidido que todos los intercambios de datos a través de la API se realizarán utilizando CSV. Recordad que cada entrada de un fichero CSV corresponde a un dato (fila, objeto, …), y que por lo tanto el fichero es un conjunto de datos. Utilizaremos el tipo ***tCSVData*** para intercambiar múltiples datos (por ejemplo listados), y el tipo ***tCSVEntry*** para objetos únicos.

Se pide:

a) Implementa la función **api_getVaccine,** del fichero **api.c**, que dada una estructura de tipo ***tApiData*** y el nombre de una vacuna, nos retorne los datos de la vacuna en una estructura de tipo ***tCSVEntry***. El formato de la vacuna será:

"vaccine;required;days"

El tipo de registro contendrá el valor "VACCINE". Los valores de retorno de esta función se detallan en la siguiente tabla:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.
E_VACCINE_NOT_FOUND	No se ha encontrado ninguna vacuna registrada con el nombre proporcionado.

b) Implementa la función **api_getVaccineLot,** del fichero **api.c**, que dada una estructura de tipo ***tApiData*** y los datos de un lote de vacunas, nos retorne la información de este lote en una estructura de tipo ***tCSVEntry***. El formato será el mismo que para los datos de entrada:

"date;time;CP;vaccine;required;days;doses"

El tipo de registro contendrá el valor "VACCINE_LOT". Los valores de retorno de esta función se detallan en la siguiente tabla:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.
E_LOT_NOT_FOUND	No se ha encontrado ningún lote con los datos proporcionados.

c) Implementa la función **api_getVaccines,** del fichero **api.c**, que dada una estructura de tipo ***tApiData*** nos retorne los datos de todas las vacunas registradas en una estructura de tipo ***tCSVData***. Cada vacuna estará guardada en una estructura de tipo ***tCSVEntry*** en el formato:

"vaccine;required;days"

El tipo de registro contendrá el valor "VACCINE". Los valores de retorno de esta función se detallan en la siguiente tabla:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.

d) Implementa la función **api_getVaccineLots,** del fichero **api.c**, que dada una estructura de tipo ***tApiData*** nos retorne los datos de todos los lotes de vacunas en una estructura de tipo ***tCSVData***. Cada lote de vacunas estará guardado en una estructura de tipo ***tCSVEntry*** en el formato:

"date;time;CP;vaccine;required;days;doses"

El tipo de registro contendrá el valor "VACCINE_LOT". Los valores de retorno de esta función se detallan en la siguiente tabla:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.

Nota: Podéis asumir que un string en formato csv nunca superará los 2048 caracteres (2Kb) de longitud. Os pueden resultar de ayuda para realizar este ejercicio los siguientes métodos:

csv_init / csv_free	Inicializa / libera una estructura de tipo tCSVData
csv_parseEntry	Rellena una estructura de tipo tCSVEntry con la información contenida en un string en formato csv.
csv_addStrEntry	Añade a una estructura de tipo tCSVData una nueva entrada (tCSVEntry) a partir de un string en formato csv.
sprintf	Método similar a printf, pero que en lugar de mostrar una información formateada por pantalla, la guarda en un string.