

Prácticas de Programación

PR1 - 20242

Fecha límite de entrega: **05 / 04 / 2025**

Formato y fecha de entrega

Se debe entregar la práctica antes del día **5 de abril de 2025** a las 23:59.

Se deben entregar los archivos requeridos en la herramienta DSLab dentro del ejercicio PR1. Solo se valorará la última entrega dentro del plazo establecido.

Concretamente, en esta actividad se requieren los siguientes archivos:

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

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

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

- Los archivos **film.c**, **film.h**, **api.c** y **api.h**.

Finalmente, también **se debe entregar un documento en formato PDF en el apartado de entregas de AC del aula de teoría que contenga un breve informe (máximo 2 páginas)** sobre el código desarrollado.

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

Objetivos

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

Criterios de corrección

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

- No seguir el **formato de entrega**, tanto en el **tipo y nombre de los archivos** como en el contenido requerido, implicará una **penalización importante** o la calificación con una **D en la actividad**.
- El código entregado **debe compilar para ser evaluado**. Si compila, se valorará:
 - Que **funcione** tal como se describe en el enunciado.
 - Que obtenga el **resultado esperado** dadas unas condiciones y datos de entrada específicos (pruebas proporcionadas). No es necesario pasar todas las pruebas, pero al menos debe mostrar el resultado en pantalla.
 - Que se respeten los **criterios de estilo** y que el código esté **comentado**. Se valorará especialmente el uso de comentarios en inglés.
 - Que las **estructuras** utilizadas sean las correctas.
 - Que se **separe adecuadamente la declaración e implementación** de las acciones y funciones, utilizando los archivos correspondientes.
 - El **grado de optimización** en tiempo y recursos utilizados en la solución entregada.
 - Que se realice una **gestión adecuada de la memoria**, liberándola 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 ésta debe ser individual y diferenciada del resto. Las entregas serán analizadas con **herramientas de detección de plagio**.

El uso de herramientas de inteligencia artificial (IA) no está permitido en esta asignatura. Su utilización no es recomendable porque afecta el proceso de aprendizaje, ya que es necesario hacer los ejercicios para adquirir los conocimientos.

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. Es

importante destacar que no se hará distinción entre coincidencias debidas a plagio entre estudiantes o al uso de la IA.

Guía citación: <https://biblioteca.uoc.edu/es/contenidos/Como-citar/index.html>

Monográfico sobre plagio:

<https://biblioteca.uoc.edu/es/biblioguias/biblioguia/Plagio-academico/>

Observaciones

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 se utiliza memoria dinámica, lo 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, existen herramientas que ejecutan un análisis dinámico del programa. Una herramienta de código abierto muy utilizada es Valgrind (<https://valgrind.org/>). El uso de esta herramienta queda fuera del alcance del curso.

Para entender el significado de los **códigos de error**, podéis utilizar el siguiente enlace, donde encontraréis también algunos ejemplos de código que os ayudarán a entender cuándo se generan estos errores:

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

DSLAb

En esta práctica, se introduce el uso de la herramienta DSLAb (<https://sd.uoc.edu/dslab/>). Esta herramienta también se utiliza en otras asignaturas y tiene como objetivo:

- Proporcionar un entorno común donde evaluar los ejercicios de codificación.

Os aconsejamos hacer envíos periódicos a la herramienta de los diferentes ejercicios de código, ya que os permitirá detectar posibles errores antes de la entrega final. **Tened en cuenta que es la herramienta utilizada para corregir vuestros códigos, y que no se corregirá ningún código en otro entorno o máquina.** Por lo tanto, si vuestro código no funciona en la herramienta DSLAb, se considerará que no funciona, aunque lo haga en vuestro ordenador.

Recordad que **la entrega que se evaluará será la última que hagáis antes de que termine el plazo de entrega** en la herramienta DSLAb. En cualquier caso, también **debéis entregar el informe del código desarrollado en el apartado correspondiente del aula, tal como indica el enunciado.**

La información básica que os será útil al utilizar DSLAb:

- DSLAb considera la entrega correcta únicamente si pasa todos los tests.
- Se muestra un resumen rápido del número de tests pasados. Por norma general, no será necesario pasarlos todos para aprobar la entrega.
- En los detalles se muestra:
 - El detalle de los tests pasados y de los que han fallado.
 - Es posible descargar un archivo con el texto que el programa muestra en pantalla (salida estándar). Se ha incluido el uso de **valgrind** en la salida estándar, de modo que en este apartado podréis ver el informe sobre la **gestión de la memoria**. Es recomendable revisarlo para asegurarse de que se hace un uso correcto de los punteros y de la memoria.
- También hay un log de ejecución que guarda la evolución de la ejecución del programa. Si, debido a una codificación incorrecta, el programa falla y no es capaz de mostrar el resultado de los tests, se debe revisar este log para determinar en qué punto se interrumpió la ejecución.

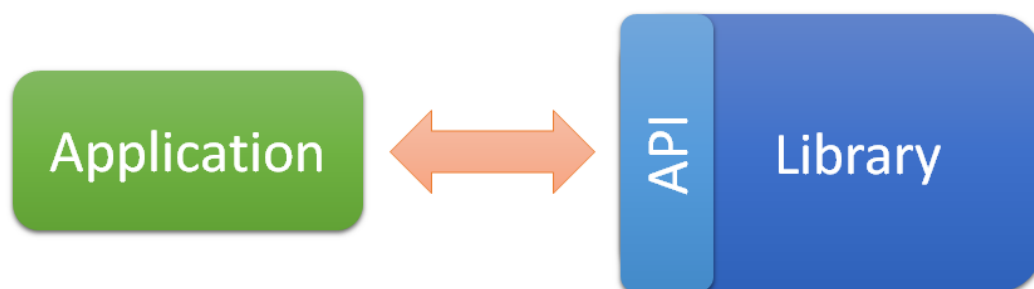
Nota: si encontráis algún problema con la herramienta DSLAb, informad a los profesores para que puedan corregir la incidencia lo antes posible.

Enunciado

En las PECs hemos estado trabajando de forma aislada partes del problema introducido en la PEC1. En las prácticas veremos cómo construir una aplicación más compleja que vaya incorporando de manera incremental lo trabajado 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 que definimos los métodos y los datos, permitiendo que otras aplicaciones puedan interactuar con la nuestra sin necesidad de saber cómo se han implementado dichos métodos.

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



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.
- **Biblioteca:** Será un proyecto de tipo "Static library". En este caso, el resultado de la compilación y enlace no produce un ejecutable, sino un archivo .a o .lib, dependiendo del sistema operativo. Este archivo puede ser utilizado desde otra biblioteca o desde una aplicación. A diferencia de las aplicaciones, las bibliotecas no implementan el método *main*.

Ejercicio 1: Preparación del entorno [0%]

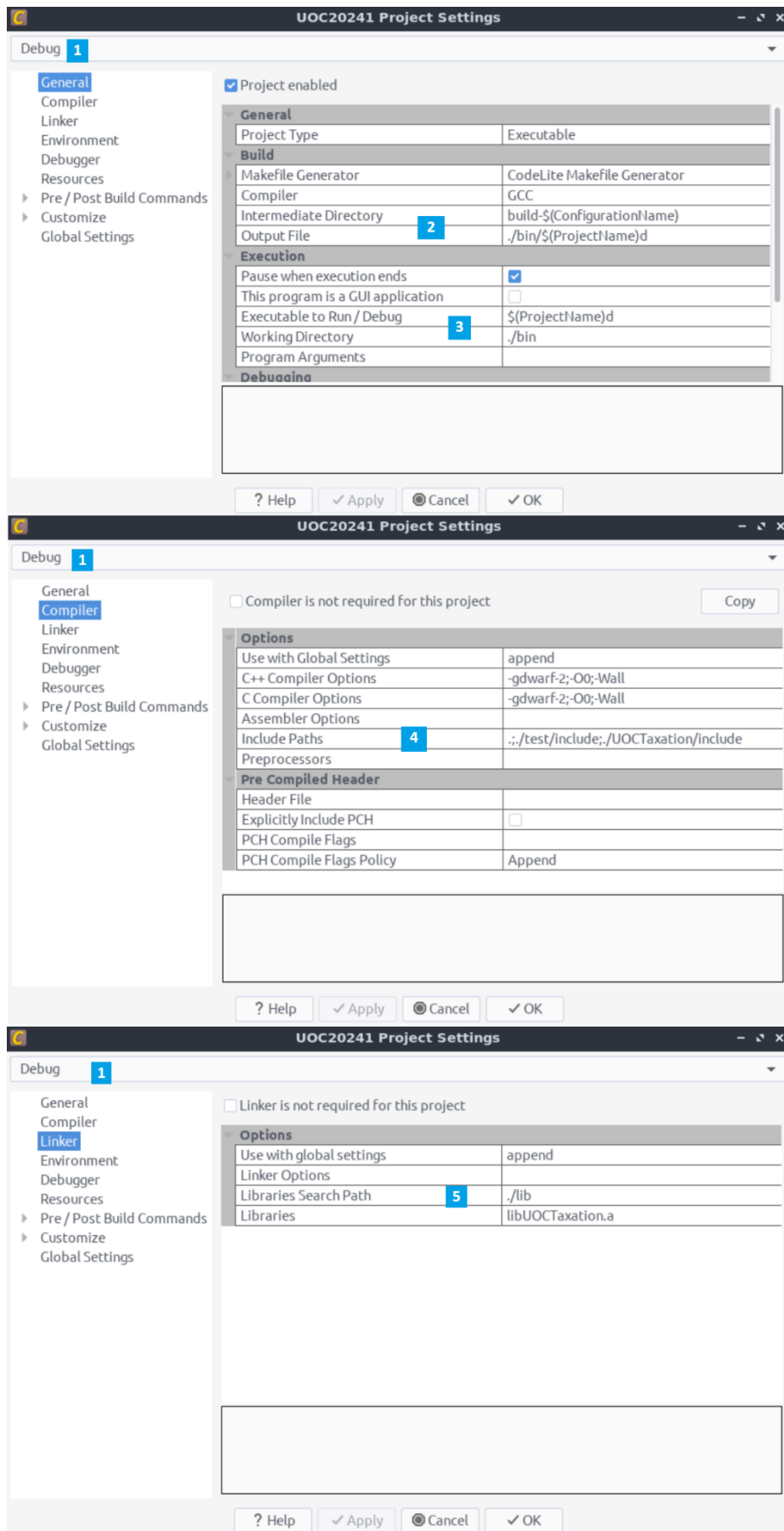
Junto con el enunciado se facilita un archivo de código con un espacio de trabajo (Workspace) que contiene dos proyectos. A continuación, se detallan las principales características de cada uno de ellos:

- **UOCPlay:** Este proyecto es la biblioteca en la que iremos añadiendo toda la funcionalidad de la práctica.
 - El código está dividido en declaraciones (include) e implementaciones (src).
 - Los archivos **api.h** y **api.c** contienen la declaración e implementación de la API, y por lo tanto, los métodos a los que se accederá desde la aplicación.
 - Al compilar, debe buscar los archivos de cabecera en la carpeta include.
 - La biblioteca debe generarse en la carpeta “lib” del Workspace.
 - El nombre de la biblioteca añadirá una “d” cuando se compile en modo Debug.
- **UOC20242:** Este proyecto es nuestra aplicación. Se encarga de ejecutar las distintas pruebas para verificar el correcto funcionamiento de la biblioteca.
 - El código principal está en la carpeta src, y el código de las pruebas en la carpeta test, separado en declaraciones (include) e implementaciones (src).
 - Al compilar, debe buscar los archivos de cabecera (*.h) tanto en la carpeta de las pruebas (test/include) como en la carpeta correspondiente de la biblioteca (UOCPlay/include).
 - Al vincular (link), es necesario indicar que busque las bibliotecas en el directorio lib del Workspace, incluyendo la biblioteca generada por el proyecto anterior.
 - El ejecutable debe generarse en la carpeta “bin” del Workspace, que también será el directorio de trabajo cuando ejecutemos (working directory).
 - El nombre del ejecutable añadirá una “d” cuando se compile en modo Debug.

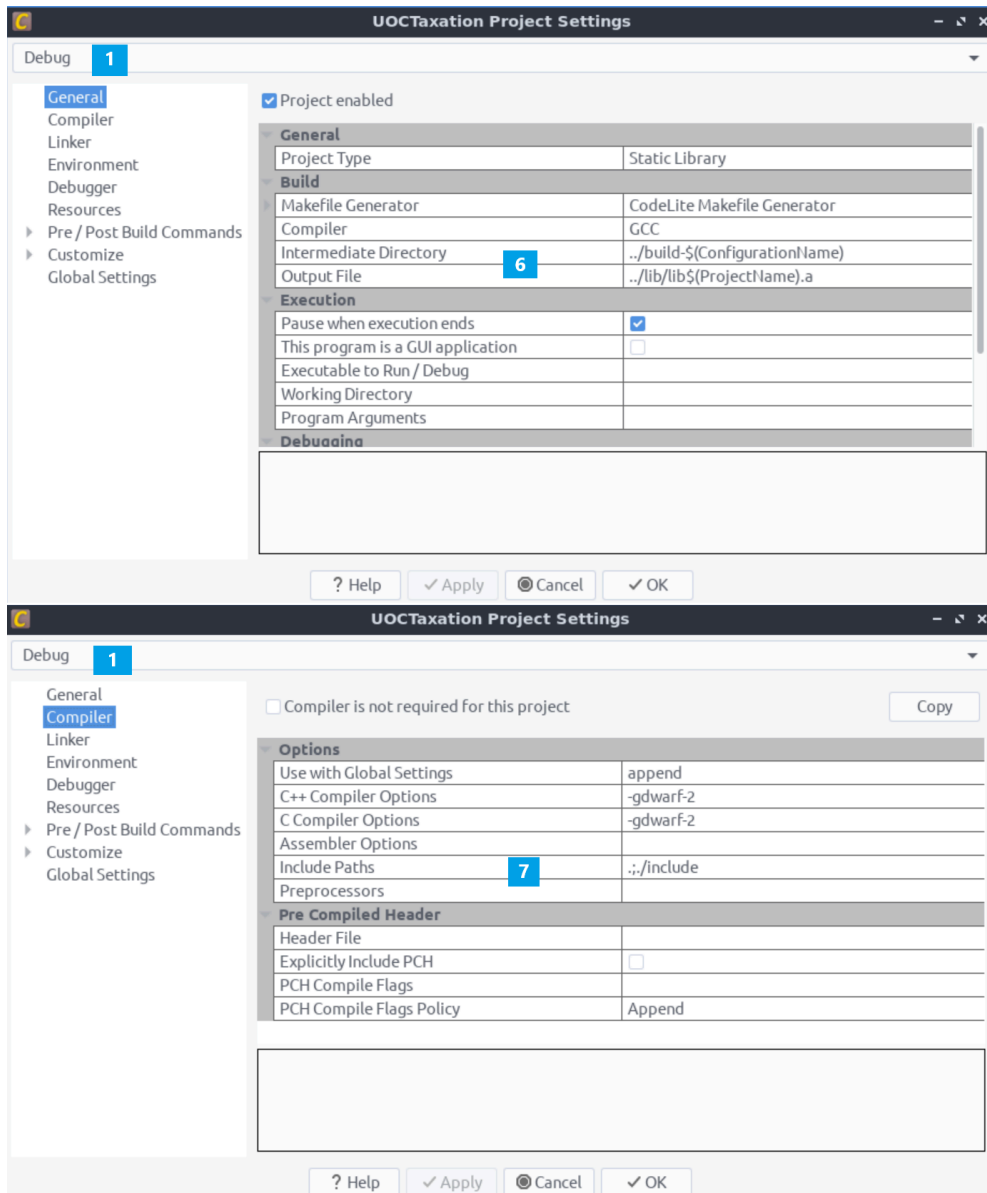
El objetivo de este ejercicio es tener el entorno proporcionado en funcionamiento. El Workspace está preparado para funcionar en la máquina virtual de la asignatura. Si no la utilizáis, puede ser necesario modificar las opciones de los proyectos para que funcione.

Si se utiliza un entorno de programación diferente de CodeLite, se debe realizar el mismo ejercicio adaptándolo al entorno utilizado. Es muy importante organizar correctamente el código, y este es uno de los objetivos que se deben alcanzar al finalizar esta práctica.


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



Localización de las opciones de configuración del proyecto UOC20242.



Localización de las opciones de configuración del proyecto UOCPlay.

1. Permite cambiar entre la configuración de Debug y Release.
2. Define dónde se generan los archivos resultantes.
3. Define qué se ejecuta al usar el botón de play de CodeLite (). Es necesario indicar el directorio de trabajo y el nombre de la aplicación, que será diferente en Debug y en Release.
4. Define en qué directorios se buscarán los archivos de cabecera. **En la imagen se configura para un proyecto llamado “UOCTaxation”, debéis sustituirlo por “UOCPlay”.**
5. Define en qué directorios se buscarán las bibliotecas y qué bibliotecas se deben añadir para generar la aplicación.
6. Define dónde se generan los archivos resultantes.
7. Define en qué directorios se buscarán los archivos de cabecera.

Una vez seleccionadas todas las opciones correctas, al ejecutar deberíais ver el resultado de las pruebas. Inicialmente, todas, excepto la del ejercicio 1, aparecerán como fallidas. También os aparecerá que el nombre y correo electrónico no se han proporcionado (izquierda). Colocad los datos dentro del archivo README.txt del Workspace, tal como se indica en el apartado de entrega, y deberíais ver que se incorporan a la salida (derecha).



```

=====
Name: <not provided>
Email: <not provided>
=====

TEST RESULTS
=====

Tests for PR1 exercises
=====
[OK]: [PR1_EX1_1] Read version information.
[FAIL]: [PR1_EX2_1] Initialize the API data s
[FAIL]: [PR1_EX2_2] Load data from file
[FAIL]: [PR1_EX2_3] Add an entry with invalid
[FAIL]: [PR1_EX2_4] Add a person entry with

```

```

=====
Name: Name Surname
Email: learner@uoc.edu
=====

TEST RESULTS
=====

Tests for PR1 exercises
=====
[OK]: [PR1_EX1_1] Read version informa
[FAIL]: [PR1_EX2_1] Initialize the API d
[FAIL]: [PR1_EX2_2] Load data from file
[FAIL]: [PR1_EX2_3] Add an entry with i

```

Ejemplo de posible salida del programa.

Ejercicio 2: Manipulación de TAD de TAD [20%]

Material de consulta

- *Apartados de la xWiki: TAD en memoria dinámica y TAD de TAD*

En este ejercicio se deben codificar las funciones que permiten la manipulación del catálogo de películas de la aplicación. Como podéis observar en el archivo **film.h**, la aplicación almacena el conjunto de películas dentro de una lista enlazada. Además, también almacena una lista enlazada con las referencias de las películas que están disponibles en el plan gratuito. Es decir, la primera de la listas (**tFilmList**) contiene todas las películas y, la segunda (**tFreeFilmList**), contiene una referencia a las películas gratuitas.

Notar que en todo momento **se habla de referencia en la segunda lista**, por lo que **se debe almacenar un puntero a una película ya existente** en la primera lista.

Asimismo, veréis que muchas de las funciones de este archivo devuelven el tipo de dato **tApiError**, una enumeración que podéis encontrar en el archivo **error.h**. A continuación, os mostramos una tabla de los tipos de errores que se espera que devuelvan estas funciones para este ejercicio:

E_SUCCESS	Operación ejecutada correctamente.
E_NOT_IMPLEMENTED	La funcionalidad aún no está implementada.
E_FILM_DUPLICATED	Ya existe una película con ese nombre.
E_FILM_NOT_FOUND	No existe ninguna película con ese nombre.

Por lo tanto, en este ejercicio se solicita:

- a) **PR1_2a:** Codifica la función **catalog_init** para que se pueda iniciar la estructura **tCatalog** recibida por parámetro. En caso de que se inicialice correctamente, esta función debe devolver **E_SUCCESS**.
- b) **PR1_2b:** Codifica la función **catalog_add** para que añada una película dentro del catálogo de películas. Por lo tanto, se solicita que la película sea añadida en la lista de películas de la aplicación de tipo (**tFilmList**) y, si esta es gratuita, también se añada una referencia a la lista de películas gratuitas de tipo (**tFreeFilmList**). En caso de que la película ya esté añadida, la función debe devolver **E_FILM_DUPLICATED**; si se añade correctamente, debe devolver **E_SUCCESS**. Controlad los errores debidamente en todo momento así como sus implicaciones (i.e. si la película no se ha podido añadir en la lista de películas, tampoco debe hacerlo en la lista de películas gratuitas a pesar de que lo sea).
- c) **PR1_2c:** Codifica la función **catalog_del** para que se elimine una película del catálogo de películas. Esto implica que la película sea eliminada de la lista de películas de tipo (**tFilmList**) y, si es gratuita, que también sea eliminada de la lista de películas gratuitas (**tFreeFilmList**). En caso de que la película no exista en la lista de películas, la función debe devolver **E_FILM_NOT_FOUND**; en caso contrario, debe devolver **E_SUCCESS**.
- d) **PR1_2d:** Codifica las funciones **catalog_len** y **catalog_freeLen** para que devuelvan la cantidad de películas totales y la cantidad de películas gratuitas respectivamente.
- e) **PR1_2e:** Codifica la función **catalog_free** para que elimine toda la información del catálogo de películas. En caso de que lo elimine correctamente, esta función debe devolver **E_SUCCESS**.

Ejercicio 3: Entrada de datos [40%]

Material de consulta

- *Apartados de la xWiki: Diseño descendente y TAD de TAD*

Hasta ahora hemos estado trabajando solo con los datos de personas (**tPeople**), sus suscripciones (**tSubscriptions**) y, más recientemente, con el catálogo de películas de la aplicación (**tCatalog**). 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 almacenar los siguientes datos:

- **people:** Conjunto de personas que son usuarias de la aplicación. Cada persona está identificada de manera única a partir de su documento de identidad. Los datos de una persona se almacenan en un tipo **tPerson**. El conjunto de personas se guarda en una tabla (**tPeople**).

- **subscriptions:** Conjunto de suscripciones dadas de alta en el sistema. Cada suscripción está identificada de manera única a partir de un identificador entero y se puede relacionar con una persona previamente dada de alta a partir de su documento de identidad. Los datos de cada suscripción se almacenan en una estructura de tipo **tSubscription**. El conjunto de suscripciones se guarda en una tabla (**tSubscriptions**).

A partir de esta práctica, se almacena un TAD que contiene dos listas enlazadas con la información de las películas dadas de alta en la aplicación tal como se ha puesto en práctica en el ejercicio anterior: una que contiene todas las películas (**tFilmList**) y otra que contiene las referencias a las películas gratuitas (**tFreeFilmList**). Por lo tanto, la estructura de datos (**tApiData**) también debe almacenar los siguientes datos:

- **catalog:** Esta estructura almacena las películas de la aplicación. Cada película se identifica de manera única a partir de su nombre. Los datos de las películas se almacenan en un tipo **tFilm**. El conjunto de películas se guarda en una lista enlazada (**tFilmList**). Asimismo, las referencias a las películas gratuitas se almacenan en otra lista enlazada (**tFreeFilmList**). El conjunto de listas enlazadas mencionadas anteriormente se almacenan en un TAD de tipo **tCatalog**.

Como parte del enunciado de la práctica se proporcionan los siguientes 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 retornará la API (ya mencionado en el ejercicio anterior).
- **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.
- **subscription.h/subscription.c** con la declaración e implementación de los tipos de datos y métodos relacionados con las suscripciones.
- **film.h/film.c** con la declaración e implementación de los tipos de datos y métodos relacionados con las películas y el catálogo.

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

A continuación se detallan algunos de los posibles errores:

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_FILM_DUPLICATED	Ya existe una película con ese nombre.
E_FILM_NOT_FOUND	No existe ninguna película con ese nombre.
E_PERSON_DUPLICATED	Ya existe una persona con ese documento.
E_PERSON_NOT_FOUND	No existe ninguna persona con ese documento.
E_SUBSCRIPTION_DUPLICATED	Ya existe una suscripción con ese identificador.
E_SUBSCRIPTION_NOT_FOUND	No existe ninguna suscripción con ese identificador.

En general, las funciones devuelven el valor inicial **E_NOT_IMPLEMENTED**. Una vez implementadas, devuelven el valor **E_SUCCESS** si todo ha funcionado correctamente. En caso contrario, devuelven un código de error asociado al tipo de error producido.

Se solicita:

- PR1_3a:** Completa la definición del tipo de datos **tApiData** en el archivo **api.h** para que almacene todos los datos requeridos.
- PR1_3b:** Implementa la función **api_initData** en el archivo **api.c**, que inicializa una estructura de tipo **tApiData** dada. La función debe devolver **E_SUCCESS**.
- PR1_3c:** Implementa la función **api_addPerson** en el archivo **api.c** para que, dada una estructura de tipo **tApiData** y una persona en formato CSV **tCSVEntry**, añada esta persona a los datos de la aplicación. Si la persona ya existe, la función no añadirá nada y deberá devolver un error de tipo **E_PERSON_DUPLICATE**.

Para cada dato, será necesario comprobar que el formato es correcto (asumimos que es correcto si el número de campos es el esperado, error asociado **E_INVALID_ENTRY_FORMAT**) y que el tipo es "PERSON" (puedes

acceder al tipo mediante el método `csv_getType`, error asociado **E_INVALID_ENTRY_TYPE**).

En caso de finalizar correctamente, la función debe devolver **E_SUCCESS**.

Nota: En los archivos **person.h** y **person.c** encontrarás una nueva definición de **person_parse** que devuelve la información de una persona en una estructura de tipo **tPerson**. También encontrarás los métodos para gestionar la tabla de personas **tPeople**. Asimismo, tienes el método **person_free** para eliminar la memoria dinámica reservada temporalmente para guardar los datos de las personas.

- d) **PR1_3d:** Implementa la función **api_addSubscription** en el archivo **api.c** para que, dada una estructura de tipo **tApiData** y una suscripción en formato CSV **tCSVEntry**, añada esta suscripción a los datos de la aplicación. Si la suscripción ya existe, la función debe devolver un error de tipo **E_SUBSCRIPTION_DUPLICATED**. Además, si no existe ninguna persona con el documento de identidad recibido dentro de la suscripción, la función debe devolver un error de tipo **E_PERSON_NOT_FOUND**.

De manera análoga a la función anterior, se debe comprobar que el formato sea correcto y que el tipo también lo sea, siendo en este caso "SUBSCRIPTION". En caso de que no lo sea por alguno de los dos motivos anteriores, la función debe devolver **E_INVALID_ENTRY_FORMAT** o **E_INVALID_ENTRY_TYPE**.

En caso de finalizar correctamente, la función debe devolver **E_SUCCESS**.

Nota: Es recomendable revisar las funciones que encontrarás en los archivos **subscription.c** y **subscription.h**.

- e) **PR1_3e:** Implementa la función **api_addFilm** en el archivo **api.c** para que, dada una estructura de tipo **tApiData** y una película en formato CSV **tCSVEntry**, agregue esta película en los datos de la aplicación. Tened en cuenta que, si la película es gratuita, también se debe añadir una referencia a la lista enlazada de películas gratuitas. Por otro lado, si la película ya existe dentro del catálogo de la aplicación, esta función debe devolver un error de tipo **E_FILM_DUPLICATED**.

De nuevo, para cada dato, será necesario comprobar que el formato sea correcto y que el tipo sea "FILM". En caso de que no cumpla alguna de las dos condiciones, deberá retornar los errores **E_INVALID_ENTRY_FORMAT** o **E_INVALID_ENTRY_TYPE**.

En caso de finalizar correctamente, la función debe devolver **E_SUCCESS**.

Nota: Es recomendable revisar las funciones que encontrarás en los archivos **film.c** y **film.h**.

- f) **PR1_3f:** Implementa los métodos **api_peopleCount**, **api_subscriptionsCount**, **api_filmsCount** y **api_freeFilmsCount** que, dada una estructura de datos de tipo **tApiData**, retornen la cantidad de personas, suscripciones, total de películas y la cantidad de películas gratuitas, respectivamente.
- g) **PR1_3g:** Implementa la función **api_freeData** en el archivo **api.c** que elimina toda la información almacenada en una estructura de tipo **tApiData** dada.
- h) **PR1_3h:** Implementa la función **api_addDataEntry** en el archivo **api.c** que, dada una estructura de tipo **tApiData** y un nuevo dato en formato CSV **tCSVEntry**, guarde este nuevo dato dentro de la estructura **tApiData**. Una entrada de datos puede ser de tipo "PERSON", "SUBSCRIPTION" o "FILM". Para cada dato, será necesario comprobar que el formato sea correcto (asumimos que es correcto si el número de campos es el esperado). Los posibles valores de retorno de esta función son los mencionados en los apartados anteriores. En cualquier caso, si la función finaliza sin ningún error, debe devolver **E_SUCCESS**.

Importante: para realizar esta práctica, es necesario aplicar el diseño descendente. De esta manera, es posible simplificar la solución de los ejercicios y evitar repetir código, es decir, no será necesario implementar dos o más veces la misma funcionalidad.

Ejercicio 4: Acceso a los datos [40%]

Material de consulta

- *Apartados de la xWiki: Diseño descendente y TAD de TAD*

Con el fin 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 en un archivo CSV corresponde a un dato (fila, objeto, ...), y que, por lo tanto, el archivo es un conjunto de datos. Utilizaremos el tipo **tCSVData** para intercambiar múltiples datos (por ejemplo, listados) y el tipo **tCSVEntry** para objetos únicos.

Se solicita:

- a) **PR1_4a:** Implementa la función **api_getSubscription** en el archivo **api.c** que, dada una estructura de tipo **tApiData** y el identificador de una suscripción, guarde los datos de la suscripción en una estructura de tipo **tCSVEntry**. El formato de la suscripción que debe devolver es el siguiente:

“id;document;start_date;end_date;plan;price;num_devices”

Por ejemplo:

“1;98765432J;01/01/2025;31/12/2025;Free;0;1”
“2;33365111X;01/05/2025;30/04/2026;Standard;29.95;3”
“3;12345672C;15/06/2025;14/06/2026;Standard;29.95;3”
“4;55565432Z;21/03/2025;20/03/2026;Free;0;1”
“5;47051307Z;01/01/2023;31/12/2028;Premium;29.95;3”

El tipo de registro contendrá el valor “SUBSCRIPTION”. Si la suscripción no existe, la función debe devolver un error de tipo **E_SUBSCRIPTION_NOT_FOUND**; en caso contrario, **E_SUCCESS**.

Nota: Es importante que el número de decimales para el campo **price** sea el mínimo posible tal como sucede en los ejemplos y que las fechas tengan el formato dd/mm/yyyy.

- b) **PR1_4b:** Implementa la función **api_getFilm** en el archivo **api.c** que, dada una estructura de tipo **tApiData** y el nombre de una película, almacene la información de la película con dicho nombre en una estructura de tipo **tCSVEntry**.

El formato de la película que debe devolver es el siguiente:

“name;duration;genre;release;rating;is_free”

Por ejemplo:

“Interstellar;02:49;4;07/11/2014;4.8;0”
“Blade Runner 2049;02:44;4;06/10/2017;4.6;1”
“The Matrix;02:16;4;31/03/1999;4.9;1”
“Inception;02:28;0;16/07/2010;4.7;1”
“Mad Max: Fury Road;02:00;0;15/05/2015;4.5;0”

El tipo de registro contendrá el valor “FILM”. Si la película no existe, la función debe devolver un error de tipo **E_FILM_NOT_FOUND**; en caso contrario, **E_SUCCESS**.

Nota: Es importante que el número de decimales para el campo **rating** sea solo 1, que la duración tenga el formato hh:mm y la fecha dd/mm/yyyy.

- c) **PR1_4c:** Implementa la función **api_getFreeFilms** en el archivo **api.c** que, dada una estructura de tipo **tApiData**, guarde los datos de todas las películas gratuitas registradas en una estructura de tipo **tCSVData**. Cada película se almacenará en una estructura de tipo **tCSVEntry** con el mismo formato que el apartado anterior:

“name;duration;genre;release;rating;is_free”

El tipo de registro contendrá el valor “FILM”. La función debe devolver **E_SUCCESS**.

- d) **PR1_4d:** Implementa la función **api_getFilmsByGenre** en el archivo **api.c** que, dada una estructura de tipo **tApiData** y un género, guarde los datos de todas películas del género recibido en una estructura de tipo **tCSVData**. Cada película se almacenará en una estructura de tipo **tCSVEntry** con el mismo formato que los dos apartados anteriores:

“name;duration;genre;release;rating;is_free”

El tipo de registro contendrá el valor “FILM”. La función debe devolver **E_SUCCESS**.

Nota: Podéis asumir que una cadena en formato CSV nunca superará los 2048 caracteres (2KB) de longitud. Los siguientes métodos pueden ser útiles para realizar este ejercicio:

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