Práctica 1

Formato y fecha de entrega

Hay que entregar la Práctica antes del día **13 de Mayo a las 23:59**. Para la entrega hará falta que entreguéis un fichero en formato **ZIP** de nombre ***logincampus*_pr1** en minúsculas (donde *logincampus* es el nombre de usuario con el qué hacéis login en el Campus). El ZIP tiene que contener:

- Workspace CodeLite entero, con todos los ficheros que se piden.

Para reducir la medida de los ficheros y evitar problemas de envío que se pueden dar al incluir ejecutables, hay que eliminar lo que genera el compilador. Podéis utilizar la opción “Clean” del workspace o eliminarlos directamente (las subcarpetas Menu y Test son las que contienen todos los binarios que genera el compilador).

Hay que hacer la entrega en el apartado de entregas de EC del aula de teoría.

Presentación

En esta práctica se presenta el caso en que trabajaremos durante este semestre. Habrá que utilizar lo que habéis trabajado en las primeras PECs, también la composición iterativa y la utilización de *strings*. También tendréis que empezar a poner en juego una competencia básica para un programador, la capacidad de entender un código ya dado y saberlo adaptar a las necesidades de nuestro problema. Con este objetivo, se os facilita gran parte del código, en el cual hay métodos muy similares a los que se os piden. Se trata de aprender mediante ejemplos, una habilidad muy necesaria cuando se está programando.

Competencias

Transversales

- Capacidad de comunicación en lengua extranjera.

Específicas

- Capacidad de diseñar y construir aplicaciones informáticas mediante técnicas de desarrollo, integración y reutilización.
- Conocimientos básicos sobre el uso y la programación de los ordenadores, sistemas operativos, y programas informáticos con aplicación a la ingeniería.

Objetivos

- Practicar los conceptos estudiados en la asignatura
- Profundizar en el uso de un IDE, en este caso CodeLite
- Analizar un enunciado y extraer los requerimientos tanto de tipos de datos como funcionales (algoritmos)
- Aplicar correctamente la composición alternativa cuando haga falta
- Aplicar correctamente la composición iterativa cuando haga falta
- Utilizar correctamente tipos de datos estructurados
- Utilizar correctamente el tipo de datos Tabla
- Aplicar correctamente el concepto de Modularidad
- Aplicar correctamente los esquemas de búsqueda y recorrido
- Analizar, entender y modificar adecuadamente código existente
- Hacer pruebas de los algoritmos implementados

Recursos

Para realizar esta actividad tenéis a vuestra disposición los siguientes recursos:

- Materiales en formato web de la asignatura
- Laboratorio de C

Criterios de valoración

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

- Los ejercicios en lenguaje C, tienen que compilar para ser evaluados. En tal caso, se valorará:
  - Que funcionen correctamente
  - Que se respeten los criterios de estilo (véase la Guía de estilo de programación en C que tenéis en la Wiki)
  - Que el código esté comentado (preferiblemente en inglés)
  - Que las estructuras utilizadas sean las adecuadas

Descripción del proyecto

Este semestre nos han pedido crear una aplicación para gestionar un compañía de transporte aéreo: los aviones, los pasajeros y los vuelos. En las PECs habéis ido definiendo ciertas variables y programando pequeños algoritmos relacionados con esta aplicación. En el código que os proporcionamos como base para la realización de esta práctica podréis encontrar estas variables ya conocidas de las PECs y algunos de los algoritmos. Recordaréis, por ejemplo, como definisteis variables para almacenar datos de un avión.

Ahora en esta práctica os pediremos que implementáis la gestión de los pasajeros.

Y las acciones que querremos tener programadas serán las siguientes:

- Leer, mediante un menú, los datos correspondientes a un pasajero
- Cargar los datos desde ficheros y también almacenarlos
- Hacer búsquedas, aplicar filtros y obtener datos estadísticos.

Además, a pesar de que esta primera versión será una aplicación en línea de comandos, queremos que todas estas funcionalidades queden recogidas en una **API** (Application Programming Interface), lo que nos permitirá en un futuro poder utilizar esta aplicación en diferentes dispositivos (interfaces gráficas, teléfonos móviles, tablets, web, ...).

Estructuración del código

Junto con el enunciado se os facilita un proyecto Codelite que será el esqueleto de la solución. A continuación se dan algunas indicaciones del código que os hemos dado:

main.c: Contiene el inicio del programa. Está preparado para funcionar con dos modos diferentes:

- **Menu**: Muestra un menú que permite al usuario gestionar los datos.
- **Test**: se ejecutan un conjunto de pruebas sobre el código para asegurar que funciona. Inicialmente muchas de estas pruebas fallarán pero, una vez realizados todos los ejercicios, todas tendrían que pasar correctamente.

Si se ejecuta normalmente, la aplicación muestra el menú. Para que funcione en modo test, hay que pasar el parámetro "-t" a la aplicación. Tal como se ha configurado el proyecto de Codelite, si lo ejecutáis en modo Test (seleccionando "Test" en el desplegable "workspace build configuration") se ejecutarán los tests, y si lo hacéis en modo Menu, veréis el menú.

data.h: Se definen los tipos de datos que se utilizan en la aplicación. A pesar de que se podrían haber separado en los diferentes ficheros de cabecera, se han agrupado todos para facilitar la lectura del código.

passenger.h/passenger.c: contienen todo el código que gestiona los pasajeros.

plane.h/plane.c: contienen el código que gestiona los aviones.

menu.h/menu.c: contienen todo el código para gestionar el menú de opciones que aparece cuando se ejecuta en modo Menu.

api.h/api.c: contienen los métodos (acciones y funciones) públicos de la aplicación, el que sería el API de nuestra aplicación. Estos métodos son los que se llaman desde el menú de opciones y los que utilizaría cualquier otra aplicación que quisiera utilizar el código.

test.h/test.c: contienen todas las pruebas que se pasan al código cuando se ejecuta en modo Test.

La mayoría de los ejercicios referencian a acciones y funciones ya existentes al código proporcionado que son muy similares a las que se os piden. Analizadlas y utilizadlas como base.

Enunciados

[10%] Ejercicio 1: Definición de datos

Tal como hemos comentado en la presentación de la práctica, una de las estructuras de información que usaremos es la del pasajero. Utilizaremos un tipo **tPassenger** que está parcialmente definido en el fichero **data.h**:

- id: Identificador del pasajero (valor entero)
- nombre: Es el nombre de pila del pasajero (cadena de como máximo 15 caracteres alfanuméricos y sin espacios)

Y en este ejercicio lo tenemos que completar con los siguientes campos:

- surname: es una cadena de, como máximo, 25 caracteres (sin espacios) que indica el primer apellido del pasajero
- docType: es el tipo de documento identificativo (tipo enumerado tDocumentType) que podrá ser uno de los siguientes valores: NIF, NIE, PASSPORT, DRIVING_LICENSE y un valor genérico que servirá para designar otros documentos (OTHER_DOCUMENTS).
- docNumber: es el número de documento, que será una cadena de caracteres alfanuméricos de, como mucho, 20 posiciones.
- birthDate: fecha de nacimiento del pasajero. Tendréis que apoyaros en una estructura auxiliar tDate que incluya los valores de año, mes y día de nacimiento (los tres, enteros)
- countryISOCode: es el país de nacimiento del pasajero, una cadena de dos caracteres que representa un país según codificación ISO
- customerCardHolder: es un booleano que indica si este pasajero posee la tarjeta de fidelización de la compañía aérea.
- cardNumber: es el número de tarjeta de fidelización del pasajero (entero positivo). Este campo sólo tendrá un valor diferente de cero en caso de que customerCardHolder tenga valor CIERTO.
- fidelityPoints: es el número de puntos de la tarjeta de fidelización (entero positivo). Este campo sólo tendrá un valor diferente de cero en caso de que customerCardHolder tenga valor CIERTO.

[20%] Ejercicio 2: Entrada interactiva de datos

Cuando trabajamos con aplicaciones que utilizan la línea de pedidos para comunicarse con el usuario, a menudo se hace necesario utilizar menús de opciones. El programa muestra la lista de opciones identificadas por un número al usuario, de forma que el usuario pueda elegir la opción que le interesa introduciendo por teclado este identificador.

Completad la implementación de la acción **readPassenger**, que encontraréis al archivo **menu.c**, que permite dar de alta de manera interactiva un nuevo pasajero. Esta acción es llamada desde la opción interactiva de añadir un nuevo pasajero "2) ADD PASSENGER". Para acceder hay que elegir primero "3) MANAGE PASSENGERS" del menú principal de la aplicación. Podéis usar estas opciones para comprobar que la entrada interactiva de datos funciona correctamente.

Los datos que hay que pedir por cada pasajero son: id, name, surname, docType, docNumber, birthDate (tres enteros), countryISOCode y customerCardHolder.

Además, en caso de que customerCardHolder sea CIERTO, habrá que leer el número de tarjeta (cardNumber) y el número de puntos obtenidos hasta ahora (fidelityPoints). En caso contrario, ambos valores mencionados se tendrán que inicializar a cero.

Habrá que comprobar si los valores introducidos están dentro de los límites especificados en el ejercicio 1 y, si es así, hará falta que el parámetro de salida de la acción retVal contenga el valor OK y, si no, el valor ERROR. Tomad como modelo la lectura de datos de aviones que se hace a la acción **readPlane** del mismo archivo menu.c.

[20%] Ejercicio 3: Serialización y deserialización de la estructura pasajero

Los tipos estructurados de datos son una buena herramienta para representar objetos complejos como por ejemplo pasajeros y aviones. No obstante, a menudo interesa obtener una representación más textual de estos objetos, por ejemplo, para poder mostrarla al usuario cuando convenga, o leerla o escribirla en un fichero. La acción de convertir en texto unos datos de una estructura lo denominamos serialitzar. La acción contraria, convertir una cadena de texto en una estructura de datos, lo denominamos deserialitzar.

Los datos de un pasajero nos llegarán (o los escribiremos) separados por un único espacio y en el mismo orden que aparecen en el enunciado del ejercicio 1. Todos los reales tendrán 2 decimales.

a) Completad en el fichero **passenger.c** la acción **getPassengerStr** que permite pasar una estructura de tipo tPassenger a una cadena de caracteres. Observad como se ha codificado la acción getPlaneStr en el fichero plane.c

b) Completad en el fichero **passenger.c** la acción **getPassengerObject** que permite pasar de una cadena de caracteres a una estructura tPassenger. Observad como se ha codificado la acción getPlaneObject en el fichero plane.c

[20%] Ejercicio 4: Operadores de la estructura pasajero

Un problema que nos encontramos con los tipos estructurados es que muchos de los operadores que tenemos definidos con los tipos básicos de datos, como por ejemplo los de comparación (==, !=, <, >, ...) o el de asignación (=), no funcionan para los nuevos tipos que creamos.

Por este motivo a menudo se hace necesario definir métodos que nos den estas funcionalidades. Por ejemplo, ya hemos visto que para asignar una cadena de caracteres no lo hacemos con el operador de asignación normal (=), sino que tenemos que recurrir a la función **strcpy**. El mismo pasa en las comparaciones, donde en vez de utilizar los operadores normales (==, !=, <, >, ...) utilizamos **strcmp**.

Se pide:

a) Completad en el fichero **passenger.c** la acción **passenger_cpy** que permite copiar todos los datos de una estructura tPassenger a otra. Observad como se ha codificado la acción plane_cpy en el fichero plane.c

b) Completad en el fichero **passenger.c** la función **passenger_cmp** que, dados dos pasajeros d1 y d2, nos devuelve:

-1 si d1<d2 0 si d1==d2 1 si d1>d2

El orden de los pasajeros vendrá dado por el valor de sus campos en el orden de prioridad siguiente:

1. Nombre (ascendente)
2. Apellido (ascendente)
3. Tipo de documento (ascendente)
4. Número de documento (ascendente)
5. Fecha de nacimiento (ascendente)
6. País (ascendente)
7. Si es poseedor de la tarjeta de fidelización (ascendente)
8. Número de tarjeta (ascendente)
9. Puntos de la tarjeta (ascendente)

Esto quiere decir, que si el nombre de d1 es “Albert” y el de d2 “Paz", consideraremos que d1<d2. En caso de que los nombres sean iguales, habrá que comprobar el apellido para desempatar. Si los apellidos también son iguales, habrá que comprobar el tipo de documento, y así sucesivamente... Si todos los datos son iguales, significará que d1==d2.

Nota: Para hacer este ejercicio podéis utilizar el método **strcmp** de la librería **string.h.** Observad como se ha codificado la acción plane_cmp en el fichero **plane.c**.

[10%] Ejercicio 5: Persistencia de datos

Para que los datos de pasajeros y aviones con las que hemos sido trabajando no se pierdan cuando finaliza la ejecución del programa, es decir, para que sean persistentes, hay varios mecanismos. Una estrategia bastante utilizada consiste en obtener una representación textual (la serialización que ya hemos visto en el ejercicio 3) y escribirlas en un fichero de texto, una estructura a cada línea. El paso inverso, la carga de los datos desde un fichero de texto en una ejecución posterior del programa, requiere de la deserialización (volver a obtener la estructura de datos original a partir de cada línea del fichero).

a) Implementar en passenger.c la acción ***passengerTable_save***

que dada una tabla de pasajeros y un nombre de fichero, guarde todos los pasajeros de la tabla en formato textual en el fichero. Hay que utilizar en su código la acción **getPassengerStr** por serializar cada estructura pasajero de la tabla y escribir cada cadena de caracteres en una línea nueva del fichero. El parámetro de salida retVal contendrá un valor **OK** si ha podido guardar los datos o un valor **ERR_CANNOT_WRITE** en caso de que, por algún motivo, no se pueda crear el fichero de salida. Podéis utilizar como ejemplo la implementación de la acción **planesTable_save** del fichero **plane.c**.

b) Implementar en passenger.c la acción ***passengerTable_load***

que dada una tabla de pasajeros y un nombre de fichero, lea todos los datos de pasajeros en formado texto del fichero y las añada a la tabla. Hace falta que uséis la función **getPassengerObject** para deserializar cada una de las líneas del fichero y convertirlas en estructuras de tipos pasajeros que se puedan añadir a la tabla**.** El parámetro de salida retVal contendrá un valor **OK** si ha podido guardar los datos o un valor **ERR_CANNOT_READ** en caso de que, por algún motivo, no se pueda abrir el fichero de entrada. Podéis utilizar como ejemplo la implementación de la acción **planesTable_load** del fichero **plane.c**.

[10%] Ejercicio 6: Filtro de pasajeros

Tenemos todos los pasajeros guardados en una tabla de tipo tPassengerTable y la aplicación ya nos permite gestionarlos (añadir, eliminar, etc). En este ejercicio nos interesará encontrar los pasajeros que cumplan unas ciertas condiciones. Dado que no conocemos el número de pasajeros que cumplirán las condiciones, utilizaremos una tabla de pasajeros tPassengerTable para devolver los resultados de las búsquedas.

Se pide implementar en el fichero **passenger.c**, utilizando las acciones que necesitéis de este mismo fichero:

a) La acción ***passengerTable_filterByCountry***

   Que, dada una tabla de pasajeros y un código de país ISO, nos devuelva todos los pasajeros de este país.

b) La acción ***passengerTable_filterByBirthDateInterval***

   Que, dada una fecha inicial y una fecha final, devuelva todos los pasajeros con fecha de nacimiento dentro de estos dos límites.

[10%] Ejercicio 7: Cálculos estadísticos de pasajeros

En este ejercicio nos interesará obtener información estadística de la tabla de pasajeros.

Se pide que implementáis, en el fichero **passenger.c**, las funciones:

a) La función ***passengerTable_getNumberOfCardHolders***

   Que, dada una tabla de pasajeros, nos devuelva el número de pasajeros que tengan tarjeta de fidelización de la compañía.

b) La función ***passengerTable_getMaxFidelityPoints***

   Que, dada una tabla de pasajeros, nos devuelva el número máximo de puntos que tiene un pasajero de la compañía. Tened presente que sólo hay que contemplar el número de puntos de aquellos pasajeros que sean poseedores de la tarjeta de fidelización.