Programación I

Práctica #10 **Curso 2024-2025** **1º curso**

Listas enlazadas

Introducción

En esta práctica, completaremos nuestro proyecto `Library`.

Además, aprenderemos a manejar estructuras dinámicas de datos, concretamente las listas simplemente enlazadas. Aprenderemos a crear listas, a usarlas (insertar y extraer nodos) y a pasarlas como argumentos de funciones.

En esta sesión, dada la íntima relación entre el concepto de lista y:

- por un lado, el de `struct`, y,
- por otro, el de puntero,

también continuaremos mejorando nuestro dominio de estos dos últimos elementos.

Además, seguiremos profundizando en nuestro conocimiento de las **librerías** de funciones.

El resultado final de esta sesión debe ser la versión definitiva de nuestro proyecto.

Objetivos

Esta práctica tiene como objetivos fundamentales:

- Entender las limitaciones de los tipos de datos definidos de forma estática.
- Comprender el concepto de lista enlazada.
- Dominar las principales operaciones sobre listas.
- Continuar avanzando en el dominio de las librerías de funciones y la estructura modular de los proyectos.
- Completar el proyecto.

Planificación

Sesión 10 de laboratorio (3 horas):

1. Realización de los ejercicios 1 a 3.
2. Realización de los ejercicios 4 a 6.
3. Realización del ejercicio 7

Trabajo fuera del aula:

- Trabajo en la implementación y pruebas del proyecto, previo a su entrega final.

Limitaciones de los tipos de datos estáticos

Proyecto basado en arrays de estructuras

En prácticas anteriores, hemos aprendido a manejar:

- el tipo `array` para representar una colección de variables del mismo tipo, y
- el tipo `struct` para representar una colección de variables de tipos diversos.

Con ello, en nuestra actual implementación de nuestro proyecto, la variable que representa a la información que queremos mantener (la relativa a un grupo de lectores) es un `array` de `structs`, cada una de las cuales tiene los datos de un lector. Además, necesitaremos una variable de tipo entero que indicará cuántos elementos hay actualmente con información útil en el `array`.

Además de lectores, en nuestra aplicación, también vamos a tener libros, así que necesitaremos otro `array` de `structs`, cada una de las cuales tiene los datos de un libro, y otra variable de tipo entero que indicará cuántos elementos hay actualmente con información útil en este otro `array`.

```
// Definimos las estructuras
struct Reader {
  char name[26];
  int code;
};

struct Book {
  char author[26];
  char title[26];
  int votes;
  int points;
  int signature;
};


…

int main () {
    // Declaramos las variables que almacenan las tablas
    struct Reader readers[100];
    struct Book books[100];
    int numReaders, numBooks;
…
```

Con esta organización, las funciones de nuestra base de datos recibirán como parámetros dichos arrays (por referencia, como es normativo en los arrays) y el número de elementos que hay actualmente en los arrays (por referencia o por valor, según nuestra función deba modificar dicho número o no):

```
int p_init (struct Reader *tab_r, int *n_r, struct Book *tab_b, int *n_b);
int p_show (struct Reader *tab_r, int n_r, struct Book *tab_b, int n_b);
…

int main () {
    struct Reader readers[100];
    struct Book books[100];
    int numReaders, numBooks;
    …
    p_init (readers, &numReaders, books, &numBooks);
    …
    p_show (readers, numReaders, books, numBooks);
    …
}
```

Más adelante, vamos a ver cómo esta configuración puede suponer innecesarios (y costosos en términos de tiempo) movimientos de datos en memoria. Con ello, vamos a motivar la necesidad de usar estructuras dinámicas de datos cuando no sabemos de antemano cuántos elementos vamos a tener que manejar.

Ejercicio 1:

Ve al directorio de la práctica 10 (pr10). Crea allí el subdirectorio `arrays`:

```
usuario@maquina:~/pr10$ mkdir arrays
```

Ve al subdirectorio `arrays` y copia allí el código del ejercicio 4 de la práctica 9 (`library.c`, `access.c`, `access.h`, `database.c`, `database.h` y `Makefile`).

Modifica la función `p_init()`, para que, en vez de solicitar los datos por teclado, lea los lectores de un fichero llamado `readers.txt`, consistente en líneas con el siguiente formato:

```
Nombre Código
```

Y lea los libros de un fichero llamado `books.txt`, consistente en líneas con el siguiente formato:

```
Signatura Autor Título Votos Puntos
```

Los ficheros puede no existir, o existir y estar vacíos. En cualquiera de estos casos, no hay error: simplemente, nuestra cuenta de lectores y/o de libros seguirá a cero.

Pero, si existen y no están vacíos, los vamos leyendo y copiando la información (que asumimos correcta) a nuestros `arrays` de lectores y de libros, una `struct` por cada línea.
Usa para esta función `p_init()` el prototipo indicado en la página 2.

Ejercicio 2:

Modifica ahora la función `p_show()`, para que, en vez mostrar sólo la información de los lectores, muestre también la información de los libros, de la siguiente forma:

```
…
Enter operation (NAVCSX): S
Show

Readers:
Juan_Perez;2
Luisa_Costa;7
Angel_Garcia;25

Books:
1;Osvaldo_Battistuti;Fundamentos_Programacion;5;40
2;Gurevich_&_Huggins;Semantics_C_programming;0;0
3;Kernighan_&_Ritchie;Lenguaje_Programacion_C;0;0
4;X_Leroy;Verification_C_Compiler;3;23
5;Antonakos_&_Mansfield;C_Structured_Programming;0;0

N) New reader
…
```

Usa para esta función `p_show()` el prototipo indicado en la página 2.

Ejercicio 3:

Modifica la función `take_int()`, para que el mínimo valor aceptado sea **1**.

Modifica ahora la función `p_new()`, para que, en vez de limitarse a pedir un nombre y mostrarlo por la pantalla, se use para añadir un nuevo lector a la base de datos:

- pida el nombre del lector, mediante `take_string()` y usando `"Name"` como cadena de invitación,
- le asigne un código igual al mayor código actual en la tabla más 1,
- guarde esa información en la siguiente posición libre del `array` de lectores,
- y muestre el mensaje: `"Reader NOMBRE;CODIGO"`

```
…
Enter operation (NAVCSX): n
New
Name (1-25 char): Ricardo
Reader Ricardo;4

N) New reader
…
```

Ojo, ahora la función `p_new()` debe recibir parámetros. ¿Cuáles?

Si, después de llamar a `p_new()`, llamamos a `p_show()`, nuestro nuevo lector aparecerá en la tabla, pero, si salimos del programa y observamos el fichero de lectores, ese nuevo lector no aparece. Eso es lo siguiente que debemos corregir.

Ejercicio 4:

Modifica ahora el programa principal `main()`, para que, una vez que ha confirmado que el usuario quiere terminar el programa, tras abandonar el bucle del menú y antes de salir:
- guarde, en el fichero `readers.txt`, la tabla de lectores, de manera que cada lector ocupe una línea, con el formato:

```
Nombre Código
```

- y guarde, en el fichero `books.txt`, la tabla de libros, de manera que cada libro ocupe una línea, con el formato:

```
Signatura Autor Título Votos Puntos
```

Esta operación debe hacerla mediante una nueva función `p_fin()`, perteneciente al gestor. Es decir, declarada en `database.h` y definida en `database.c`.

Ojo, la función `p_fin()` debe recibir parámetros. ¿Cuáles?

Ahora, cuando salgamos del programa, observaremos que el fichero de lectores contiene a los que había originalmente, más los que hayamos añadido mediante `p_new()`.

Ejercicio 5:

Modifica ahora la función `p_add()`, para que, en vez de limitarse a mostrar por la pantalla `"Add"`:

- pida una 1ª cadena, usando `take_string()`, y usando `"Author"` como cadena de invitación,
- pida una 2ª cadena, usando `take_string()`, y usando `"Title"` como cadena de invitación,
- le asigne una signatura igual al número actual de libros más 1,
- le asigne un número de votos y de puntos igual a cero,
- guarde esa información en la siguiente posición libre del `array` de libros,
- y muestre el mensaje: `"Book AUTOR;TITULO;SIGNATURA"`

```
…
Enter operation (NAVCSX): a
Add
Author (1-25 char): Juan_Gomez
Title (1-25 char): C_Maravilla
Book Juan_Gomez;C_Maravilla;6

N) New reader
…
```

Ojo, ahora la función `p_add()` debe recibir parámetros. ¿Cuáles?

Ejercicio 6:

Modifica ahora la función `p_vote()`, para que:
- Si no hay ningún libro muestre el mensaje `"No books yet"` y retorne al programa principal.
- Cuando pida el 1º entero, usando `take_int()`, y usando `"Book"` como cadena de invitación, use el número actual de libros como el máximo aceptable, y, después,
- pida un 2º entero, usando `take_int()`, y usando `"Points"` como cadena de invitación, y `10` como el máximo aceptable,
- incremente en 1 la cuenta de votos de ese libro.
- sume el valor de los puntos recibido al total de puntos del libro,
- y muestre el mensaje: `"Registered vote"`

```
…
Enter operation (NAVCSX): v
Vote
Book [1-6]: 4
Points [1-10]: 9
Registered vote

N) New reader
…
```

Ojo, ahora la función `p_vote()` debe recibir parámetros. ¿Cuáles?

Ejercicio 7:

Modifica ahora la función `p_clean()`, para que, en vez de limitarse a mostrar por la pantalla `"Clean"`:
- Si no hay ningún lector muestre el mensaje `"No readers yet"` y retorne al programa principal.
- Si hay alguno, pida un nombre, mediante `take_string()` y usando `"Name"` como cadena de invitación,
- busque en la tabla de lectores uno con ese nombre, y
- si existe, lo elimine de la tabla (si hay más de uno, eliminará el primero), mostrando el mensaje `"Cleaned reader"`, y
- si no, muestre el mensaje de error: `"Unknown reader"`.

```
…
Enter operation (NAVCSX): c
Clean
Name (1-25 char): Manolo
Unknown reader

…
Enter operation (NAVCSX): c
Clean
Name (1-24 char): Juan
Cleaned reader
…
```

Ojo, ahora la función `p_clean()` debe recibir parámetros. ¿Cuáles?

Proyecto basado en listas enlazadas

Habrás observado que, en esta implementación de `p_clean()` basada en un `array` de estructuras, tenemos que mover gran cantidad de información al objeto de mantener nuestra información compacta. Lo mismo nos pasaría en `p_new()` si deseásemos insertar a los nuevos lectores con un determinado orden. O en `p_add()` si deseásemos insertar a los nuevos libros con un determinado orden. O si tuviésemos una función para borrar libros.

Nuestra base de datos es pequeña, pero si necesitásemos manejar una gran cantidad de estructuras de gran tamaño, esto resultaría muy costoso en términos de tiempo.

Además, hemos definido nuestros `arrays` de estructuras con un tamaño de 100 elementos. Si tenemos muchos menos lectores y/o libros, estamos malgastando la memoria; pero, si tenemos más, "no nos caben" en nuestra base de datos.

Por estas razones, una base de datos construida alrededor de listas enlazadas de nodos, donde cada nodo contiene la información de un lector o un libro, es una opción mucho más eficiente.

Vamos a cambiar la implementación de nuestro proyecto, de forma que la información que vamos a mantener son 2 listas de nodos, cada uno de las cuales contiene:

- los datos de un lector o de un libro, y
- la información de enlace al siguiente nodo:

Con ello, las variables que representa a estas listas son simples punteros al primero de esos nodos. Y ya no necesitamos el entero que indica cuántos elementos tenemos con información útil.

```
// Definimos las estructuras que representan a los nodos
struct Reader {
  char name[26];
  int code;
  struct Reader *sig;
};

struct Book {
  char author[26];
  char title[26];
  int votes;
  int points;
  int signature;
  struct Book *sig;
};

…

int main () {
    // Declaramos variables que apuntan a la cabeza de las listas
    struct Reader *lectores;
    struct Book *books;
…
```

Ahora tenemos que modificar todas nuestras funciones del módulo gestor de la base de datos para que usen estas nuevas variables.

Esto afecta por un lado a sus prototipos: los parámetros que reciben deben ser diferentes a los que recibían en la implementación basada en `arrays` de estructuras.

Por otro lado, también afecta a su forma de trabajar:

- en lugar de usar un `for` controlado por una variable de tipo entero, que nos permitía ir indexando en los sucesivos elementos del `array`:

```
int p_show (struct Reader *tab_r, int n_r, struct Book *tab_b, int n_b) {
int i;

for (i=0; i<n_r; i++) {
    // Acceder a tab_r[i].name y tab_r[i].code
}
```

- ahora usaremos un `while` controlado por una variable de tipo puntero a estructura, que nos permitirá ir accediendo a los sucesivos nodos, por medio de sus punteros de enlace (campo `sig` de la `struct`):

```
int p_show (struct Reader *lista_r, struct Book *lista_b) {
struct Reader *pos;

pos = lista_r;
while (pos != NULL) {
    // Acceder a pos->name y pos->code
    pos = pos->sig;
}
```

Por último, también afecta a la forma de invocarlas:

- en lugar de pasar el `array` (siempre por referencia) y el número de lectores (por referencia o por valor en función de si puede ser modificado por la función, o no):

```
int main () {
    struct Reader readers[100];
    struct Book books[100];
    int numReaders, numBooks;
…
p_init (readers, &numReaders, books, &numBooks);
…
p_show (readers, numReaders, books, numBooks);
```

- ahora pasaremos solo los punteros a la cabeza de las listas (por referencia o por valor en función de si dichos punteros pueden ser modificados por la función, o no):

```
int main () {
    struct Reader *readers;
    struct Book *books;
…
p_init (&readers, &books);
…
p_show (readers, books);
…
```

Ejercicio 8:

Vuelve al directorio `pr10` y haz una copia del subdirectorio `arrays` con el nombre `lists`:

```
usuario@maquina:~/pr10$ cp -a arrays lists
```

(Ese comando creará una copia de todos los archivos que haya en el subdirectorio `arrays`).
Ve al subdirectorio `lists` y realiza los siguientes cambios en tu programa:
- modifica tanto la definición de `Reader` como la declaración de `readers` para que, ahora, estén basados en listas enlazadas,
- modifica tanto la definición de `Book` como la declaración de `books` para que, ahora, estén basados en listas enlazadas, y
- modifica las funciones `p_init()`, `p_fin()`, `p_new()`, `p_add()`, `p_vote()`, `p_clean()` y `p_show()` para que, ahora, usen la nueva declaración de `readers` y de `books`.

Ten en cuenta que:
- muchas de estas nuevas versiones de las funciones no necesitan el parámetro del número de lectores y/o libros,
- las funciones que **nunca** van a **modificar** la cabeza de la(s) lista(s) (`p_fin()`, `p_show()` y `p_vote()`), recibirán la variable que representa a dicha cabeza por **valor**, pero
- las funciones que **pueden modificar** la cabeza de la(s) lista(s) (`p_init()`, `p_new()`, `p_add()` y `p_clean()`), recibirán la variable que representa a dicha cabeza por **referencia**,
- las llamadas a todas estas funciones desde `main()` también deben ser modificadas,
- cuando hayas modificado algunas de tus funciones, pero no todas, si compilas, recibirás gran cantidad de errores; céntrate en los que tengan que ver con lo ya modificado, ignorando los demás.

Resumen

Los principales resultados esperados de esta práctica son:

- Terminar de desarrollar el proyecto.
- Aprender a manejar listas enlazadas.

Tareas para el/la estudiante:

- Finalizar todos los ejercicios no completados en clase.

Evaluación del proyecto

El Examen Práctico Final (EFL):

Tal y como se especifica en la Guía Docente de la asignatura, las prácticas serán evaluadas mediante el Examen Final de Laboratorio (EFL).

En dicho EFL, se solicitará que realices una o varias sencillas modificaciones de tu código. Esto significa que se definirán unos pequeños cambios en la especificación de las prácticas de laboratorio y se pedirá que adaptes tu código a la nueva especificación.

Plazo de entrega:

Deberás subir el código de las prácticas #8, #9 y #10 a Moovi no más tarde del **miércoles 15** de enero de 2025.

Condiciones de la entrega

Deberás subir a Moovi:

- Todos los ficheros con código C (`"*.c"`) y con cabeceras (`"*.h"`) en los que hayan organizado tus prácticas.

- El correspondiente `Makefile` para generar el ejecutable del proyecto de la práctica #10.

- En forma de un solo archivo comprimido, cuyo nombre debe coincidir con tu número de DNI. Así, si la compresión usada es zip y el número de DNI es 12345678Z, el archivo subido debe ser.

"12345678Z.zip".

Es tu responsabilidad asegurarte de que el código subido:
1. se compila sin errores, y
2. tiene un comportamiento conforme a lo especificado en las prácticas de laboratorio