Programación I

Práctica #8 **Curso 2024-2025** **1º curso**

Estructuras. Programación modular

Introducción

En esta práctica, introduciremos el tipo de datos `struct`, que nos permite representar una colección de datos de tipos diversos. Aprenderemos a definir variables de dicho tipo, a usarlas y a pasarlas como argumentos de funciones.

En esta sesión, también aprenderemos a utilizar las funciones como mecanismo para reutilizar código entre diferentes programas. Para ello tendremos que profundizar un poco en el proceso de generación de código ejecutable a partir de código fuente.

Además, aprenderemos a crear y utilizar las llamadas **librerías** de funciones (en realidad, deberíamos llamarlas **bibliotecas** de funciones, ya que su name deriva de "*library*").

Mediante las librerías de funciones conseguiremos no tener que volver a compilar nuestras funciones cada vez que creemos un programa que las utiliza.

El resultado final de esta sesión debe ser una estructura modular de nuestro proyecto.

Objetivos

Esta práctica tiene como objetivos fundamentales:

- Aprender a utilizar el tipo `struct` para representar una colección de datos de tipos diversos.
- Mejorar la comprensión de las funciones como mecanismo básico de programación modular en C.
- Comprender el uso de funciones como mecanismo de reutilización de código entre programas.
- Introducir el concepto de *librería de funciones*.
- Crear la estructura general

Planificación

Sesión 8 de laboratorio (3 horas):

1. Realización de los ejercicios 1 y 2.
2. Realización de los ejercicios 3 a 6.
3. Realización de los ejercicios 7 y 8

Trabajo fuera del aula:

- Trabajo en el diseño y codificación del proyecto, a partir del ejercicio 8 de este enunciado.

Estructuras

El tipo `struct`

En prácticas anteriores, hemos aprendido a manejar el tipo `array` para representar una colección de variables. A la hora de representar datos más complejos, el tipo `array` tiene la limitación de que sus componentes deben ser todos del mismo tipo. Cuando queremos representar una colección de variables de diferentes tipos debemos usar un nuevo tipo de datos: el tipo **`struct`**. Veamos un ejemplo:

```
// Definimos la estructura
struct Reader {
  char name[26];
  int code;
};

// Declaramos una variable del tipo struct
struct Reader persona;
```

A cada componente de la estructura (name, code, …), le llamamos "*campo*". Para acceder a cada campo usamos el formato `variable.campo` , donde cada `variable.campo` es, en realidad, una variable del tipo del que hayamos definido el campo, y con la que podemos hacer cualquier operación que podamos hacer con cualquier variable individual de ese tipo. Por ejemplo, asignarle un valor:

```
strcpy (persona.name, "Juan");
persona.code = 18;
```

Una vez definida una estructura, podemos querer representar una colección de dichas estructuras, es decir, podemos querer declarar un `array` cuyo tipo base sea la estructura. Esto es tan sencillo como declarar un `array` de cualquier tipo simple:

```
// Declaramos un array de 100 readers
struct Reader readers[100];

…
// Le asignamos un valor al campo "code" del 4º elemento del array
readers[3].code = 20;
```

También podemos usar punteros a estructuras:

```
// Declaramos un puntero a un tipo struct
struct Reader *pReader;
…
// Le asignamos la dirección de una variable de tipo struct
pReader = &persona;
```

Y podemos pasar estructuras (y arrays de estructuras) como parámetros a funciones. Analiza el siguiente código:

```
int p_init (struct Reader *tab_r, int *numero) {
    int code;

    *numero = 0;
    code = take_int ("Code", 200);
    while (code > 0) {
        tab_r[*numero].code = code;
        take_string ("Name", tab_r[*numero].name);
        (*numero)++ ;
        code = take_int ("Code", 200);
    }
    return 0;
}

…

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

Ejercicio 1:

Copia el ejercicio 8 de la Práctica #6.

```
usuario@maquina:~/pr08$ cp ../pr06/ej8.c ej1.c
```

Modifica tu programa:
- añade, tras las líneas con los `include`, la definición proporcionada de `Reader`
- añade, como variables locales de la función `main()`, las declaraciones proporcionadas para `readers` y `numReaders`
- haz que `main()`, antes de mostrar la carátula, llame a la función `p_init()`
- modifica el menú y el `switch` para que, ahora, cuando el usuario seleccione la nueva opción `'S'`, una nueva función `p_show()` muestre los valores introducidos, con el siguiente formato:

```
Show
Readers:
Pepe_Perez;18
Juan_Gomez;24
Luisa_Costa;20
…
```

Nota: el prototipo de `p_show()` debe ser el siguiente:

```
int p_show (struct Reader *tab_r, int numero);
```

Programación modular

Introducción

Tal como estamos programando hasta ahora, no podemos reutilizar en diferentes programas el código que estamos escribiendo. Sin embargo, esto es algo que querremos hacer a menudo.

Ejercicio 2:

En este ejercicio, no tienes que escribir ningún código. Solo reflexionar: para crear otro programa como el del ejercicio 1, pero que imprima:

```
++++++++++++++++++++++++++++++++++++++++
MENSAJES
++++++++++++++++++++++++++++++++++++++++
```

¿Qué **habría** que hacer?

La respuesta es que habría que repetir la mayor parte del código en ambos programas. Ésta no es una situación deseable, no sólo porque supone una repetición de trabajo, sino porque hace difícil mantener nuestro código.

Supongamos que ahora queremos modificar de nuevo nuestras funciones `validate()`, `boundary()` y `mark()` porque hemos detectado un error, o porque queremos mejorarlas en algún aspecto. ¡Tendríamos que editar los dos programas!

Y, si hubiéramos utilizado las funciones en más programas, tendríamos que modificarlas en todos esos programas.

Por suerte, C nos proporciona mecanismos para solucionar esta situación.

Dividiendo el código en varios ficheros

La solución consiste en escribir el código fuente de la función en un fichero adicional y conseguir que cualquiera de nuestros programas pueda utilizar dicho código.

Ejercicio 3:

Copia el programa que desarrollaste en el ejercicio 1, y llámale `"library.c"`

```
usuario@maquina:~/pr08$ cp ej1.c library.c
```

Usando el editor, **corta** de tu programa el código con **la definición** de tus funciones `boundary()`, `mark()` `validate()`, `take_string()`, `take_int()` y `take_char()`.

Usando el editor, toma el código que has cortado y pégalo en un nuevo fichero, llamado `"access.c"`

Ahora prueba a compilar tu programa como has hecho hasta ahora:

```
usuario@maquina:~/pr08$ gcc -Wall library.c -o library
```

¿Qué ocurre?

Obviamente, ahora tienes un problema, y es que estás llamando desde tu código a una función que ya no está definida dentro de ese código, puesto que la has borrado.

Ejercicio 4:

Compila de nuevo tu programa, pero indicando ahora al compilador que utilice dos ficheros fuente:
- el `"library.c"`, que contiene tu función `main()`,
- y el `"access.c"`

O sea, ejecuta:

```
usuario@maquina:~/pr08$ gcc  -Wall  library.c  access.c  -o library
```

Has conseguido eliminar el problema, ya que ahora el código de las funciones `boundary()`, `mark()`, `validate()`, `take_string()`, `take_int()` y `take_char()` está escrito una sola vez, y puedes utilizarlo en tantos programas como quieras (por ejemplo, con el programa insinuado en el ejercicio 2).

Es más, aun cuando no pienses reutilizar funciones, cuando desarrolles programas grandes, es muy conveniente dividir el código en varios ficheros, tal como has hecho en el último ejercicio, ya que el código será mucho más manejable. Una buena división, por ejemplo, agrupando en ficheros diferentes funciones que realicen tareas relacionadas, es una buena práctica dentro de lo que denominamos **programación modular**.

Librerías (o bibliotecas) de funciones

Creando una librería de funciones

Tal como hemos compilado en el último ejercicio, cada vez que compilemos un programa que utilice nuestro fichero de funciones "access.c", estamos compilando también este último fichero. Puesto que nuestra librería de funciones es pequeña, esto no supone una gran sobrecarga de trabajo. Pero si nuestra librería contuviera cientos de funciones, el proceso de compilación se podría ralentizar bastante, y además sin motivo, ya que muchísimas veces no habremos cambiado nada en nuestra librería de funciones entre una compilación y otra.

En este ejercicio, vamos a aprender como compilar un fichero una sola vez y utilizarlo tantas veces como queramos. Esta forma de trabajar tiene una ventaja adicional, y es que nos va a permitir proporcionarle (o venderle) a otros programadores nuestra librería para que la utilicen sin tener que proporcionarles el código fuente de la misma.

La solución consiste en generar el código objeto de nuestra librería. Esto se hace utilizando las siguientes opciones del compilador:

```
$ gcc -Wall -c access.c
```

Con esto, el compilador genera el fichero *objeto* "access.o" a partir del fichero fuente en código *C* "access.c". Ahora podríamos utilizar ese fichero tantas veces como queramos de la siguiente manera:

```
$ gcc -Wall access.o  ejemplo.c -o ejemplo
```

Este comando compilaría un código fuente contenido en cierto "ejemplo.c", lo enlazaría con el código objeto que contiene "access.o" (sin tener que volver a compilar "access.c") y generaría el correspondiente programa ejecutable "ejemplo". Para ello, no se ha necesitado disponer del fichero "access.c".

Ejercicio 5:

Crea el código objeto de tu librería[1] a partir del fichero "access.c" utilizando el mecanismo que hemos visto en esta sección.
Comprueba que, después de haber generado una vez el fichero objeto de tu librería ("access.o"), puedes compilar tu programa para generar el correspondiente ejecutable a partir de esa librería, sin necesidad de su código fuente ("access.c")

```
usuario@maquina:~/pr08$  gcc  -Wall  -c  access.c
usuario@maquina:~/pr08$  gcc  -Wall  library.c  access.o  -o  library
```

[1] En realidad una librería de C es un concepto un poco más complejo, ya que suele contener el agregado de varios ficheros de código objeto y se suele utilizar de otra forma desde la línea de comandos (con la opción –l), pero esto no lo aprenderemos en esta asignatura. Para nuestros propósitos lo que hemos visto es una librería a todos los efectos.

Ficheros de cabecera

Con lo que hemos hecho hasta ahora, no hemos resuelto todos nuestros problemas, ya que seguimos teniendo código duplicado en nuestro programa relacionado con las funciones de nuestra librería. Veamos el problema.

Ejercicio 6:

Modifica la función `boundary()` de `"access.c"`, usada para dibujar la carátula, para que tome un parámetro adicional: el número de caracteres que hay que imprimir en cada línea (hasta ahora era un número fijo, 50).
Modifica el programa `"library.c"` para que llame adecuadamente a esta función, con el citado parámetro adicional.
Vuelve a generar el código objeto de la librería de entrada/salida (`"access.o"`) y recompila tu programa.

Ahora, además de modificar la definición de la función `boundary()`, dentro de la librería (`"access.c"`), también hemos tenido que modificar su declaración en el programa (y, si estuviésemos manteniendo el otro programa insinuado en el ejercicio 2, también la tendríamos que modificar allí). Es decir, seguimos teniendo código duplicado.

Para arreglar este problema, la solución es la misma de antes, introducir el código necesario en un nuevo fichero en vez de en todos los programas. Ahora bien, este nuevo fichero sólo va a contener declaraciones, no código fuente que dé lugar a código ejecutable. Por tanto, será un fichero que **no es necesario compilar explícitamente**, ya que lo utilizaremos de otra forma.

La solución consiste en cortar las declaraciones de las funciones de nuestra librería y pegarlas en un nuevo fichero denominado **fichero de cabecera** y que, por convención, tiene la extensión `".h"`.

A continuación, podemos eliminar dichas declaraciones de funciones de nuestro programa y sustituirlas por la siguiente directiva[2] (si hemos llamado a nuestro fichero de cabecera `"access.h"`):

```
#include "access.h"
```

Durante el proceso de compilación, el compilador sustituirá dicha línea por el contenido del fichero indicado.

Ejercicio 7:

Modifica tu programa, para que utilice el fichero de cabecera `"access.h"`, es decir:
- **corta** de `"library.c"` la **declaración** de las funciones `boundary()`, `mark()`, `validate()`, `take_string()`, `take_int()` y `take_char()` y pégala en un nuevo fichero, llamado `"access.h"`
- añade tanto a `"library.c"` como a `"access.c"` la directiva `#include`:

```
#include "access.h"
```

- recompila tu programa.

[2] Ojo a las **comillas**, en lugar de los caracteres de ángulo: < y >

Estructura modular general del Proyecto

Vamos a desarrollar la estructura en módulos de nuestro proyecto. Tras el próximo ejercicio, "solo" necesitaremos darle contenido a nuestras funciones.

Ejercicio 8:

Realiza las siguientes modificaciones sobre el código de tu programa:

1. Corta del módulo principal (`"library.c"`) las **definiciones** de las 6 funciones: `p_init()`, `p_new()`, `p_add()`, `p_vote()`, `p_clean()` y `p_show()`, y pégalas en un nuevo fichero, para la librería con el gestor de la base de datos, que se denomine `"database.c"`.

2. Corta del módulo principal (`"library.c"`) las **declaraciones** de las 6 funciones: `p_init()`, `p_new()`, `p_add()`, `p_vote()`, `p_clean()` y `p_show()`, y pégalas en un nuevo fichero de cabecera del gestor de la base de datos, llamado `"database.h"`.

3. Corta del módulo principal (`"library.c"`) la **definición** de `struct Reader`, y pégala en `"database.h"`.

4. Añade tanto a `library.c` como a `database.c` las directivas `#include`:

```
#include "access.h"
#include "database.h"
```

5. Genera el ejecutable, a partir del programa principal y las 2 librerías:

```
usuario@maquina:~/pr08$ gcc  -Wall -c access.c
usuario@maquina:~/pr08$ gcc  -Wall -c database.c
usuario@maquina:~/pr08$ gcc  -Wall  library.c  access.o  database.o  -o  library
```

Continuando con el proyecto

Pensando el proyecto Library:

Reflexiona:

- ¿Queremos tener que compilar manualmente todos los módulos del proyecto cada vez que realicemos un cambio simple en nuestro código?

En una práctica posterior, aprenderemos a crear un `Makefile` por mediación del cual se pueda compilar automáticamente este programa. Dicho `Makefile` es un requisito obligatorio para el examen final de laboratorio (EFL).

- ¿Es robusta la interfaz de usuario de tu programa?

- ¿Qué pasa si el usuario escribe una opción vacía, o un entero vacío, o una cadena vacía (solo pulsa la tecla `"Intro"`)?

- ¿Qué pasa si escribe una opción formada por más de un carácter (por ejemplo: `"NA"`)?

- ¿Qué pasa si escribe letras cuando el programa espera un entero?

Pista: siguiendo lo que has aprendido en la Práctica #7, en las funciones `take_string()`, `take_int()` y `take_char()` vamos a querer utilizar `fgets()`, seguido de `sscanf()`, en lugar de usar solo `scanf()`.

Resumen

Los principales resultados esperados de esta práctica son:

- Saber manejar el tipo struct.
- Crear la estructura general del proyecto.

Como posibles líneas de trabajo adicional del alumno, se proponen las siguientes:

- Terminar los ejercicios no completados en clase.