Programación I

Práctica #9 **Curso 2024-2025** **1º curso**

Elaboración de proyectos de programación

Introducción

En esta práctica, aprenderemos a automatizar la compilación de proyectos modulares de programación, mediante el uso del comando `make`.

Para ello, necesitaremos escribir un fichero de configuración, llamado `Makefile`, y, después, entender cómo funciona y cómo se usa dicho comando `make`.

Previamente, aprenderemos a fortificar nuestros programas, mediante el uso de operaciones de entrada de datos robustas. Nos basaremos, para ello, en las funciones de la librería estándar de entrada/salida `fgets()` y `sscanf()`.

El resultado final de esta sesión debe ser una versión robusta de nuestro proyecto de programación.

Objetivos

Esta práctica tiene como objetivos fundamentales:

- Conocer el comando `make`.
- Saber crear un `Makefile`.
- Dominar las operaciones robustas de entrada de datos.
- Avanzar el proyecto.

Planificación

Sesión 10 de laboratorio (3 horas):

1. Realización del ejercicio 1.
2. Realización del ejercicio 2 y 3.
3. Realización del ejercicio 4.

Trabajo fuera del aula:

- Trabajo en el diseño y codificación del proyecto, a partir del ejercicio 4 de este enunciado

Entrada robusta de datos

Protección frente a entrada de datos errónea

En tu implementación actual, cuando esperas que el usuario teclee un único carácter como respuesta al menú, y quieres que tu programa lea ese carácter, tu función `take_char()` llama a la función `scanf()` con la especificación de formato `" %c"`. Esto, si el usuario, efectivamente introduce un solo carácter, funciona correctamente. Pero, si no, produce resultados erróneos. Así:

- si el usuario se limita a pulsar solo la tecla `Intro`, sin introducir ningún carácter, el programa se queda esperando una entrada válida; debería decir que esa entrada es incorrecta,
- si el usuario introduce una cadena que empiece por uno de los caracteres válidos, el programa aceptará ese primer carácter, y el resto de la entrada será leída por la siguiente operación de entrada de datos; por ejemplo, si el usuario teclea: `"Natalia"`, el programa aceptará la `'N'` como una entrada válida, pasará a ejecutar la función `p_new()`, y, allí, intentará leer el nombre de un lector, y "pensará" que el nombre introducido es `"atalia"`.

Comprueba con tu programa estas circunstancias.

Para paliar estos problemas, vamos a seguir un esquema en el que:

- primero, leeremos la línea completa introducida por el usuario, mediante la función `fgets()`, proporcionando como parámetro una cadena de caracteres muy larga,
- después, extraeremos de esa cadena el contenido relevante, mediante `sscanf()`, y,
- en paralelo, comprobaremos que el contenido es el que esperamos, es decir:
  - que hay el tipo de datos que esperamos,
  - que no falta ninguna entrada, y
  - que no sobra ninguna información.

La clave para estas comprobaciones está en que `sscanf()` retorna el número de las variables que le hemos proporcionado como parámetro a las que ha sido capaz de asignarle un valor válido.

Por ejemplo, si quisiéramos leer un entero, un real y una cadena, haríamos:

```
int ent;
float real;
char cad[26];
char linea[256];
char extra[256];

do {
    printf ("Dame un entero, un real y una cadena: ");
    fgets (linea, sizeof(linea), stdin);
    if (sscanf (linea, "%d%f%s%s", &ent, &real, cad, extra) == 3) {
        // Los datos son aceptables, luego los procesamos
        …
    }
    else {
        // Entrada errónea: tomamos las medidas que correspondan
        …
    }
```

Entrada de datos de tipo carácter

Ejercicio 1:

Ve al directorio de la práctica 9 (pr09) y copia allí el código del ejercicio 8 de la práctica 8 (`library.c, access.c, access.h, database.c` y `database.h`).

En la función `take_char()`, del módulo `access`, sustituye el `scanf()` por:
- un `fgets()`, en el que pases como parámetro un cadena muy larga (por ejemplo, de 256 caracteres), seguido de:
- un `sscanf()`, aplicado sobre dicha cadena, y al que le vas a pasar 2 parámetros:
  - la dirección del **carácter** que quieres leer, y
  - una cadena de caracteres extra.

¿Cuál es la especificación de formato que debes usar para estos 2 parámetros?

Ahora, si `sscanf()` retorna un valor distinto de 1, será porque el usuario ha introducido:
- datos "de menos" (una línea vacía), o
- datos "de más" (hay caracteres no válidos detrás del primer carácter).

En cualquier caso, si ese valor es distinto de 1, rechazaremos la entrada del usuario, volviendo a presentar el mensaje de invitación, y esperaremos una nueva entrada del usuario.

Cuando usamos `take_char()` desde la función `main()` del módulo `library.c`:

```
N) New reader
A) Add book
V) Submit vote
C) Clean reader
S) Show info
X) Exit

Enter operation (NAVCSX): Nav
Enter operation (NAVCSX):
Enter operation (NAVCSX): N A V
Enter operation (NAVCSX): S

Readers:
…
```

Y, cuando usamos `take_char()` desde la función `validate()` del propio módulo `database.c`:

```
Confirm exit? (YN): YN
Confirm exit? (YN):
Confirm exit? (YN): y

usuario@maquina:~/pr09$
```

Entrada de datos de tipo entero

Ejercicio 2:

Modifica ahora la función `take_int()`, del módulo `access`, siguiendo exactamente el mismo esquema del ejercicio anterior; o sea, sustituye el `scanf()` por:
- un `fgets()`, en el que pases como parámetro un cadena muy larga, seguido de
- un `sscanf()`, aplicado sobre dicha cadena, y con 2 parámetros: la dirección del **entero** que quieres leer, y una cadena de caracteres extra.

Si el valor de retorno de `sscanf()` es distinto de 1, rechazaremos la entrada del usuario, volviendo a presentar el mensaje de invitación, y esperaremos una nueva entrada del usuario:

```
Code [0-200]: 24A
Code [0-200]:
Code [0-200]: 1200
Code [0-200]: 45

Name (1-25 char): …
```

Entrada de datos de tipo cadena

Ejercicio 3:

Modifica ahora la función `take_string()`, del módulo `access`, siguiendo exactamente el mismo esquema de los ejercicios anteriores; o sea, sustituye el `scanf()` por:
- un `fgets()`, en el que pases como parámetro un cadena muy larga, seguido de
- un `sscanf()`, aplicado sobre dicha cadena, y con 2 parámetros: la **cadena** que quieres leer, y una cadena de caracteres extra.

Si el valor de retorno de `sscanf()` es distinto de 1, rechazaremos la entrada del usuario, volviendo a presentar el mensaje de invitación, y esperaremos una nueva entrada del usuario:

```
Name (1-25 char): juan gomez
Name (1-25 char):
Name (1-25 char): juan_gomez
Code [0-200]: …
```

Automatización de la compilación. El comando `make`

Hemos creado un proyecto modular, en el que podemos compilar sus diversas partes por separado:

```
gcc -Wall -c access.c
gcc -Wall -c database.c
gcc -Wall library.c access.o database.o -o library
```

¿Tenemos que compilar todos nuestros módulos cada vez que hacemos un cambio en nuestro proyecto? En realidad, no. Solo aquello que haya cambiado. En un proyecto pequeño como el que estamos desarrollando en nuestro curso puede ser sencillo darse cuenta de cuáles son los módulos que necesitan se compilados y cuáles no. Y, en cualquier caso, si no estamos seguros, compilarlo todo no supone un gran malgasto de tiempo.

Sin embargo, en proyectos más grandes, esto no es así. Por ello, y para conseguir una mayor eficiencia, se usan herramientas que permitan automatizar este proceso.

La más sencilla de estas herramientas es el comando `make`, cuya página de manual (`man make`) nos dice lo siguiente:

"El propósito de la utilidad make es determinar automáticamente qué partes de un programa complejo necesitan volver a ser compiladas, y lanzar los comandos para volver a compilarlas".

¿Y cómo sabe el comando `make` qué partes necesitan volver a ser compiladas?

Lo sabe mediante la información que aportamos en un fichero de configuración llamado `Makefile` que almacenamos en el mismo directorio que nuestros ficheros fuente.

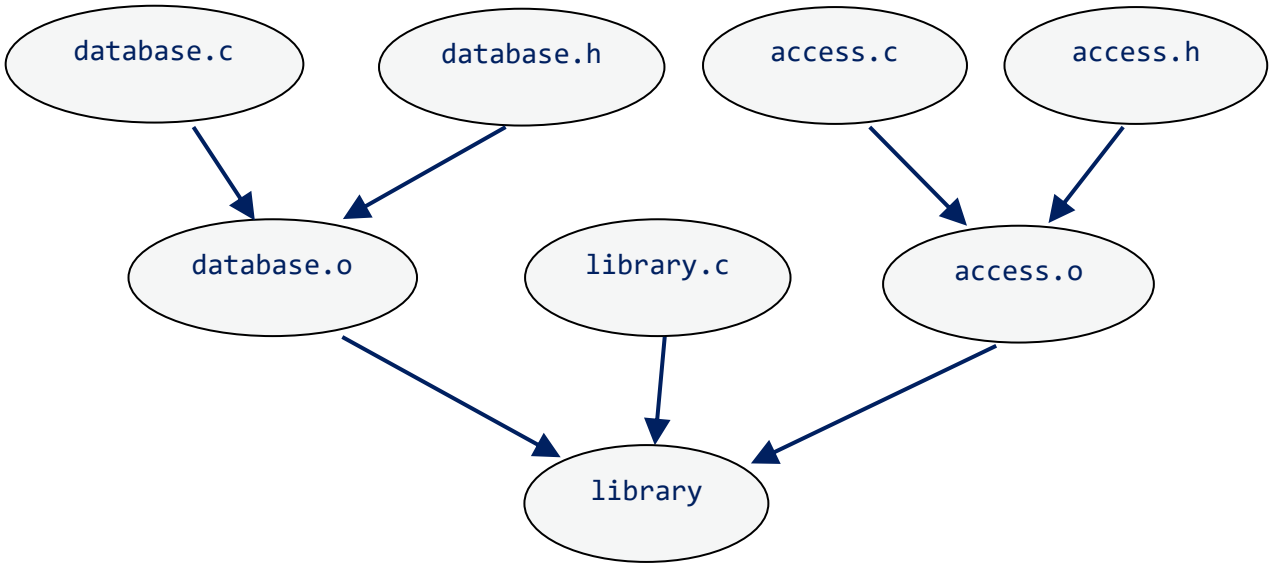
Mediante dicho fichero `Makefile` le indicamos al comando `make`:

- el/los objetivo/s: el/los programa/s que tiene que actualizar, es decir, los módulos que tiene que mantener actualizados
- las dependencias entre las partes del programa, es decir, de qué archivos depende cada módulo que hay que compilar:
    - si un módulo objetivo no existe, `make` intentará generarlo
    - si existe, pero es más antiguo que alguna de sus dependencias, también intentará generarlo
- los comandos necesarios para generar los objetivos; en nuestro caso cómo se debe compilar cada módulo.

En nuestro proyecto `library`:

- el objetivo es el módulo `library`
- el módulo `library` depende del archivo `library.c` y de los módulos `database.o` y `access.o`
- el módulo `database.o` depende de los archivos `database.c` y `database.h`
- el módulo `access.o` depende de los archivos `access.c` y `access.h`

Gráficamente:



Con esto, un sencillo `Makefile` para nuestro proyecto sería:

```
# ********************************************************************
# Programación I.  Curso 2024-2024
# Compilación del proyecto Library
# ********************************************************************

CC     = gcc  -Wall

all: library

access.o: access.c  access.h
	$(CC)  -c  access.c

database.o: database.c database.h
	$(CC)  -c database.c

library: library.c  access.o  database.o
	$(CC)  access.o  database.o  library.c  -o  library

clean:
	rm  -f  library  *.o  *.bak  *~
```

Antes de cada comando, no hay caracteres en blanco, sino un tabulador

Ejercicio 4:

Crea un `Makefile`, tal como se ha indicado en esta sección. Prueba lo siguiente:

1) Modifica `access.c` (añadiendo, por ejemplo una línea en blanco)

Si ejecutamos: `make`

- El resultado será que, primero, se compilará `access.c`:

```
gcc  -c  access.c
```

- Y, luego, se compilará `library.c`:

```
gcc  -Wall  access.o  database.o  library.c  -o  library
```

2) Modifica `library.c`

Si ejecutamos: `make`

- El resultado será que sólo se compilará `library.c`:

```
gcc  -Wall  access.o  database.o  library.c  -o  library
```

3) Si ejecutamos: `make clean`

- El resultado será que se borrará `library`, `access.o`, `database.o` y los eventuales archivos de respaldo (`*.bak` y `*~`) que pueda haber.

Resumen

Los principales resultados esperados de esta práctica son:

- Hacer que nuestro proyecto tenga una entrada de usuario robusta.
- Creación de un `Makefile` y manejo del comando `make` para automatizar la compilación del proyecto.