Sesión 4: Funciones en C

Hoja de problemas

Programación para Sistemas

Ángel Herranz
aherranz@fi.upm.es
Universidad Politécnica de Madrid

2022-2023

Ejercicio 1. Repasa las transparencias de clase. Presta especial atención a los detalles sintácticos del fichero `Makefile`. Intenta entender el *guión* que lleva a solucionar cada problema que va surgiendo.

Ejercicio 2. En las transparencias puedes encontrar varias referencias, una al K&R, capítulo 4 y otras dos para la herramienta `make`. No dejes de tenerlas a mano y explorarlas:

Chapter 4. Functions and Program Structure
(*The C Programming Language*, K&R 2nd. edition)

GNU Make Manual
The Free Software Foundation (FSF)

`http://makefiletutorial.com/`
Chase Lambert

Ejercicio 3. Asegúrate de realizar todos los ejercicios de las transparencias antes de continuar. Deberías tener en un directorio los siguientes ficheros: `lcg1.c`, `sum1.c`, y un maravilloso `Makefile` con el que puedes *fabricar* los ejecutables `lcg1` y `sum1`.

Ejercicio 4. Recuerda que la herramienta `make` tiene algunas reglas por defecto que no es necesario que tú mismo escribas. Dependiendo de la instalación las reglas pueden variar (aunque las relacionadas con el lenguaje C son bastante estables en todas las distribuciones de Unix). Para ver las reglas predefinidas ejecuta:

```
$ make -p -f /dev/null
```

Ejercicio 5. Antes de continuar, vamos a enriquecer el `Makefile` para que con la simple ejecución de `make` se creen todos los ejecutables. Para ello basta con que añadas esta regla justo después de que establezcas la variable `CFLAGS`:

```
CFLAGS=-Wall -g -pedantic

todos: lcg1 sum1

...
```

Prueba ahora a ejecutar

```
$ make todos
```

o simplemente

```
$ make todos
```

Ejercicio 6. A veces, después de ejecutar `make`, el directorio en el que estás trabajando se llena de ficheros *feos* o inservibles. Vamos a añadir un par de reglas al final a nuestro el `Makefile` para realizar una limpieza:

```
...

limpio:
        rm -f *.o

muylimpio: limpio
        rm -f lcg1 sum1 core
```

Ahora, la ejecución de

```
$ make limpio
```

borrará todos los ficheros `.o` y

```
$ make muylimpio
```

hará lo mismo que `make limpio` y además borrará todos los ejecutables.

Nota: por convención, la mayor parte de los programadores usan `clean` y `veryclean` como objetivos así que es habitual ver:

```
$ make clean
```

y

```
$ make veryclean
```

Ejercicio 7. Deberías haber sido capaz de hacer el `Makefile` por ti mismo. Puedes compararlo con el siguiente:

```
CFLAGS=-Wall -g -pedantic

todos: lcg1 sum1

lcg1: lcg1.o
	$(CC) $(CFLAGS) -o lcg1 lcg1.o

sum1: sum1.o
	$(CC) $(CFLAGS) -o sum1 sum1.o

clean:
	rm -f *.o

veryclean: clean
	rm -f lcg1 sum1 lcg2 test_lcg core
```

Ejercicio 8. Ejecución paso a paso: `gdb lcg1`

- Poner en marchar el depurador `gdb`.
- Colocar un *breakpoint* en `main`.
- Ejecutar el programa paso a paso y explorar las variables[1]
- ¿Puedes ver el valor de `anterior` cuando está ejecutando `main`? ¿Y el valor de x?
- ¿Puedes ver el valor de `i`, variable de `main` cuando está ejecutando `generar_aleatorio`?

Ejercicio 9. Ejecución paso a paso: `gdb sum1` (usa un número bajito `;)`)

- Poner en marchar el depurador `gdb`.
- Colocar un *breakpoint* en `main`.
- Ejecutar el programa paso a paso y explorar las variables[2]
- ¿Puedes ver el valor de `n` cuando está ejecutando `main`? ¿Y el valor de i?
- ¿Puedes ver el valor de `n`, variable de `main` cuando está ejecutando `sum`?

[1] Ver transparencias de la sesión 2
[2] Ver transparencias de la sesión 2

Ejercicio 10. Poco a poco vamos viendo nueva sintaxis para las expresiones del lenguaje C. En esta sesión hemos descubierto dos sintaxis nuevas y muy importantes:

$$*e \qquad \&e$$

Antes de continuar vuelve a asegurarte de que entiendes la semántica de ambas sintaxis:

- $*e$: se evalúa e, su resultado es entonces interpretado como una dirección de memoria, y $*e$ hace referencia al contenido de dicha dirección de memoria.
- $\&e$: en este caso e no puede ser cualquier expresión, sólo puede ser un *lvalue*[3], normalmente un identificador. El significado es «la dirección de memoria de e».

Ejercicio 11. No olvides realizar una implementación correcta de la función `intercambiar`.

Ejercicio 12. Ejecuta la función `intercambiar` bajo el control de `gdb` y vete explorando las variables.

Ejercicio 13. No olvides realizar una implementación correcta de la función `sum` no recursiva (fichero `sum2.c`).

Ejercicio 14. Hasta ahora has usado "masivamente" la función `printf` de la biblioteca estándar de C para poder escribir en la salida estándar. Dicha función forma parte de una familia de funciones entre las que están `printf, fprintf, dprintf, sprintf` y `snprintf`. Puedes verlas todas invocando la *sección 3* del manual desde Bash:

```
$ man 3 printf
```

Existe una versión dual de `printf` para poder leer datos de la entrada estándar: **`scanf`**. El siguiente ejemplo lee un entero de la entrada estándar:

```
int x;
scanf("%i", &x);
printf("El dato leído es: %i\n", x);
```

Como podrás observar es fundamental entender el uso del paso de parámetros por referencia para poder usar la función `scanf`. El resto del ejercicio, además de probar esas líneas, consiste en leer diferentes tipos de datos y entender cómo funcionan las funciones de la familia: `scanf, fscanf` y `sscanf`. Para ello tienes a tu disposición el manual:

```
$ man 3 scanf
```

[3] Todo aquello que puede aparecer en la izquierda de una asignación.