

Sesión 04: *Módulos* en C

Programación para Sistemas

Ángel Herranz

Otoño 2018

Universidad Politécnica de Madrid

En capítulos anteriores...

Moodle UPM

<https://github.com/aherranz/pps>

- Unix y Bash
- Editor
- gcc y ejecución de programas C
- gdb
- ejercicios

En el capítulo de hoy...

- Funciones, variables y ámbito
- Recursión
- Estructura de un programa C

Chapter 4. Functions and Program Structure

(The C Programming Language, K&R 2nd. edition)

- make

GNU Make Manual

The Free Software Foundation (FSF)

<http://makefiletutorial.com/>

Chase Lambert

Una recomendación

- Crear un directorio para el material de la asignatura:

```
$ mkdir pps
```

- Crear un directorio para el material de las clases:

```
$ cd pps
```

```
$ mkdir clases
```

- Crear un directorio por sesión, hoy:

```
$ cd clases
```

```
$ mkdir 04
```

- Trabaja en ese directorio:

```
$ pwd
```

```
/home/angel/pps/clases/04
```

Otra recomendación

Los programas o los ficheros tipo **Makefile** tienen que seguir una gramática formal. Si no prestas mucha atención lo más probable es que cometas errores de sintaxis. El compilador o **make** quizás te digan qué error has cometido pero es posible que aún no lo entiendas bien. **Transcribe todo con mucho cuidado**, presta especial atención a los espacios y los cambios de línea.

LCG: generando números *aleatorios*

- LCG = *Linear Congruential Generator*
- Algoritmo que genera números *pseudoaleatorios* utilizando una ecuación lineal.

$$X_{n+1} = (a \times X_n + c) \mod m$$


💬 Juguemos con $a = 7$, $c = 1$, $m = 11$ y $X_0 = 0$

LCG: generando números *aleatorios*

- LCG = *Linear Congruential Generator*
- Algoritmo que genera números *pseudoaleatorios* utilizando una ecuación lineal.

$$X_{n+1} = (a \times X_n + c) \mod m$$

 Juguemos con $a = 7$, $c = 1$, $m = 11$ y $X_0 = 0$

 Implementar una función `generar_aleatorio` que cada vez que se le llame genere un número aleatorio usando el LCG anterior. Elaborar un programa `lcg1.c` para mostrar su funcionamiento.

lcg1.c

```
#include <stdio.h>
```

```
#define A 7
```

```
#define C 1
```

```
#define M 11
```

```
int x = 0;
```

```
int generar_aleatorio() {  
    int anterior = x;  
    x = (A * x + C) % M;  
    return anterior;  
}
```

```
int main()
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < M; i++) {
```

```
        printf(  
            "%i -> %i\n",  
            i,  
            generar_aleatorio());  
    }
```

```
    return 0;
```

```
}
```


make i

- ¿Cansados de gcc -Wall -Werror ...?
- Automaticemos las tareas con la herramienta make

 Probemos a crear un fichero Makefile con este contenido:

```
CFLAGS=-Wall -g -pedantic

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

 Ejecutar make lcg1 para construir el ejecutable:

```
$ make lcg1
cc -Wall -g -pedantic -c -o lcg1.o lcg1.c
cc -Wall -g -pedantic -o lcg1 lcg1.o
```

make ii

- Y ya se puede ejecutar `lcg1`:

```
$ ./lcg1
```

```
0 -> 0
```

```
1 -> 1
```

```
2 -> 8
```

```
3 -> 2
```

```
4 -> 4
```

```
5 -> 7
```

```
6 -> 6
```

```
7 -> 10
```

```
8 -> 5
```

```
9 -> 3
```

```
10 -> 0
```

```
$ |
```

Makefile explicado i

- La primera línea define una variable con los *flags* de gcc:

```
CFLAGS=-Wall -g -pedantic
```

- Las otras dos líneas¹:

```
lcg1: lcg1.o
```

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

“para construir el fichero `lcg1` necesitas el fichero `lcg1.o`
y entonces tienes que ejecutar la orden
`$(CC) $(CFLAGS) -o lcg1 lcg1.o`”

- La herramienta `make` tiene algunas reglas por defecto².

¹Cuidado con el `tabulador!`

²Ejecuta `make -p` para ver dichas reglas

Makefile explicado ii

- make utiliza los **tiempos de modificación de los ficheros** para saber si tiene que volver a **realizar las tareas** o no
- En un mismo fichero Makefile se pueden escribir varias reglas

💬 ¿Qué pasa al ejecutar `make lcg1` por segunda vez?

💬 ¿Por qué crees que pasa eso?

💬 ¿Qué pasa si modificas `lcg1.c` y ejecutas `make lcg1`?

💬 ¿Por qué crees que pasa eso?

Dissección de `lcg1.c` i

```
#include <stdio.h>

#define A 7
#define C 1
#define M 11

int x = 0;

int generar_aleatorio() {
    int anterior = x;
    x = (A * x + C) % M;
    return anterior;
}
```

- *Includes*
- *Copy and paste* de `/usr/include/stdio.h` realizado por el *preprocesador* antes de compilar
- Los ficheros *header* (`.h`) *no contienen código*, ni variables, ni funciones, sólo *declaraciones*.


Disección de lcg1.c ii

```
1 #include <stdio.h>
2
3 #define A 7
4 #define C 1
5 #define M 11
6
7 int x = 0;
8
9 int generar_aleatorio() {
10     int anterior = x;
11     x = (A * x + C) % M;
12     return anterior;
13 }
```

- *Macros*
- *Find and replace*: antes de ejecutar serán substituidas por el texto indicado:

A por 7, C por 1, y M por
11

- Es el **preprocesador** el encargado de realizar el trabajo

 gcc -E lcg1.c para ver el efecto de **#define**

Dissección de `lcg1.c` iii

```
1 #include <stdio.h>
2
3 #define A 7
4 #define C 1
5 #define M 11
6
7 int x = 0;
8
9 int generar_aleatorio() {
10     int anterior = x;
11     x = (A * x + C) % M;
12     return anterior;
13 }
```

- **Definición** de una variable **global**: `x`
- (**definir** vs. **declarar**)
- **Tiempo**: dicha variable existe durante la ejecución completa del programa
- **Ámbito** (*scope*): dicha variable es accesible desde cualquier parte del programa (ver línea 10)

Dissección de lcg1.c iv

```
1 #include <stdio.h>
2
3 #define A 7
4 #define C 1
5 #define M 11
6
7 int x = 0;
8
9 int generar_aleatorio() {
10     int anterior = x;
11     x = (A * x + C) % M;
12     return anterior;
13 }
```

- Definición de una función: `generar_aleatorio()`
- No tiene argumentos
- Devuelve `int`
- Las funciones **no se pueden anidar** (casi nada en C se puede anidar)
- Las funciones **son globales** y no se pueden esconder

Dissección de `lcg1.c` v

```
1 #include <stdio.h>
2
3 #define A 7
4 #define C 1
5 #define M 11
6
7 int x = 0;
8
9 int generar_aleatorio() {
10     int anterior = x;
11     x = (A * x + C) % M;
12     return anterior;
13 }
```

- Variable **automática** o **local**.
- **Tiempo**: se crea una variable en la pila de ejecución con cada llamada y se destruye al terminar la llamada.
- **Ámbito**: sólo es accesible desde la función (ver línea 12).

Sumar números del 0 a n : sum1.c³

```
#include <stdio.h>

unsigned sum(unsigned i) {
    if (i < 1) {
        return 0;
    }
    else {
        return i + sum(i-1);
    }
}
```

```
int main() {
    unsigned n = 10;
    printf(
        "0+1+...+%u = %u\n ",
        n,
        sum(n)
    );

    return 0;
}
```

³*Fuerza bruta recursiva ;)*

Actualizamos el Makefile

- 📄 Añadimos una **nueva regla** a nuestro Makefile, como la anterior substituyendo `lcm1` por `sum1`:

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

- Recordemos, el significado de esa regla es:
“para construir el fichero **sum1** necesitas el fichero **sum1.o** y entonces tienes que ejecutar la orden **`$(CC) $(CFLAGS) -o sum1 sum1.o`**”

¿Cómo se comporta el programa sum1?

- Probemos con diferentes valores de n
- ¿1 000? ¿100 000? ¿500 000?
- Editar y recompilar y ejecutar con cada cambio:
make sum1 y ./sum1



¿Qué ocurre?

Segmentation fault (core dumped) (prueba `ls -l`)

⁴Si no ves el fichero core, prueba ejecutando `ulimit -c unlimited` antes de ejecutar el programa

¿Cómo se comporta el programa sum1?

- Probemos con diferentes valores de n
- ¿1 000? ¿100 000? ¿500 000?
- Editar y recompilar y ejecutar con cada cambio:
make sum1 y ./sum1



¿Qué ocurre?

Segmentation fault (core dumped) (prueba `ls -l`)

- El programa se rompe con un *stackoverflow* y genera un volcado de toda su huella en la memoria: core⁴

⁴Si no ves el fichero core, prueba ejecutando `ulimit -c unlimited` antes de ejecutar el programa

- Vamos a estructurar nuestro código del LCG en módulos.
- Un *módulo* con la función `main`.
- Un *módulo* con la variable global y con la función `generar_aleatorio`.

LCG en *módulos*: primer intento i

generador_lcg.c

```
#define A 7
#define C 1
#define M 11

int x = 0;

int generar_aleatorio() {
    int anterior = x;
    x = (A * x + C) % M;
    return anterior;
}
```

lcg2.c

```
#include <stdio.h>
int main() {
    int i;
    for (i = 0; i < M; i++) {
        printf(
            "%i -> %i\n",
            i,
            generar_aleatorio());
    }
    return 0;
}
```

LCG en *módulos*: primer intento ii

Añadimos dos nuevas reglas al Makefile:

```
...  
lcg2: lcg2.o generador_lcg.o  
      $(CC) $(CFLAGS) -o $@ $^
```


LCG en *módulos*: primer intento ii

Añadimos dos nuevas reglas al Makefile:

```
...  
lcg2: lcg2.o generador_lcg.o  
      $(CC) $(CFLAGS) -o $@ $^
```

```
$ make lcg2
```

```
cc -Wall -g -pedantic -c -o lcg2.o lcg2.c
```

```
lcg2.c: In function 'main':
```

```
lcg2.c:4:19: error: 'M' undeclared (first use in this function)
```

```
    for (i = 0; i < M; i++) {  
        ^
```

```
lcg2.c:8:7: warning: implicit declaration of function
```

```
    'generar_aleatorio' [-Wimplicit-function-declaration]
```

```
    generar_aleatorio();
```

```
    ^~~~~~
```

LCG en *módulos*: primer intento iii

- El compilador **no encuentra ni M ni generar_aleatorio**, no sabe lo que son ni de qué tipo.
- El compilador tiene que ser capaz de compilar `lcg2.c` sin ver lo que hay en `generador_lcg.c`.
- **Convención:** todo lo que es público se lleva a un *header*
generador_lcg.h
- Y se hace un **#include** "`generador_lcg.h`" desde `lcg2.c` y desde `generador_lcg.c`

LCG en *módulos*: segundo intento i

generador_lcg.h

```
#define A 7
#define C 1
#define M 13

extern int generar_aleatorio();
```

generador_lcg.c

```
#include "generador_lcg.h"

int x = 0;

int generar_aleatorio() {
    int anterior = x;
    x = (A * x + C) % M;
    return anterior;
}
```

lcg2.c

```
#include <stdio.h>
#include "generador_lcg.h"


int main() {
    int i;
    for (i = 0; i < M; i++) {
        printf(
            "%i -> %i\n",
            i,
            generar_aleatorio());
    }
    return 0;
}
```

LCG en *módulos*: segundo intento ii

```
$ make lcg2
cc -Wall -g -pedantic -c -o lcg2.o lcg2.c
cc -Wall -g -pedantic -c -o generador_lcg.o generador_lcg.c
cc -Wall -g -pedantic -o lcg2 lcg2.o generador_lcg.o
$ ./lcg2
0 -> 0
1 -> 1
2 -> 8
3 -> 2
4 -> 4
...
10 -> 0
$ |
```



LCG en *módulos*: segundo intento iii

- Modifiquemos sólo el fichero en `generador_lcg.h`, por ejemplo **#define** M 13
 - Ejecutamos `make lcg2` y luego nuestro programa `./lcg2`
-  ¿Qué ocurre? ¿Qué debería ocurrir?

LCG en *módulos*: segundo intento iii

- Modifiquemos **sólo** el fichero en `generador_lcg.h`, por ejemplo **#define** M 13
- Ejecutamos `make lcg2` y luego nuestro programa `./lcg2`



¿Qué ocurre? ¿Qué debería ocurrir?



Hay que decirle a make que tanto `generador_lcg.o` como `lcg2.o` **dependen además de** `generador_lcg.h` para que sepa que tiene que **recompilar**.

- Añadimos estas dos reglas a nuestro Makefile

```
...  
generador_lcg.o: generador_lcg.c generador_lcg.h  
  
lcg2.o: lcg2.c generador_lcg.h
```

LCG en *módulos*: segundo intento iv

- El resultado final es:

```
$ make lcg2
cc -Wall -g -pedantic -c -o lcg2.o lcg2.c
cc -Wall -g -pedantic -c -o generador_lcg.o generador_lcg.c
cc -Wall -g -pedantic -o lcg2 lcg2.o generador_lcg.o
$ ./lcg2
0 -> 0
1 -> 1
2 -> 8
3 -> 5
4 -> 10
5 -> 6
...
12 -> 0
$ |
```