

# Sesión 04: *Módulos y Vectores* (?)

Programación para Sistemas

---

Ángel Herranz

Otoño 2018

Universidad Politécnica de Madrid

# En capítulos anteriores...

Moodle

<https://github.com/aherranz>

- 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
- Arrays?

# 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
```

# 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 (transcribir)

```
#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;
}
```

## Makefile

- ¿Cansados de gcc -Wall -Werror ...?
- Automaticemos la tarea con 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 y observar qué obtenemos:

```
$ make lcg1
```

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

```
cc -Wall -g -pedantic -o lcg1 lcg1.o
```



# Makefile explicado i

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

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

- Las otras dos líneas<sup>1</sup>:

```
lcg1: lcg1.o
```

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

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

- La herramienta `make` tiene algunas *reglas por defecto*<sup>2</sup>.

---

<sup>1</sup>Cuidado con el tabulador `marcado`

<sup>2</sup>Ejecuta `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.

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

💬 ¿Por qué?

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

# 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;
}
```

- *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 }
```

- Define tres **macros**: **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)

# Disección 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).

## Ejecución paso a paso: `gdb lcg`

- Poner en marcha el depurador `gdb`.
- Colocar un *breakpoint* en `main`.
- Ejecutar el programa paso a paso y explorar las variables<sup>3</sup>

💬 ¿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`?

---

<sup>3</sup>Ver transparencias de la sesión 2



## Sumar números del 0 a $n$ : sum1.c<sup>4</sup>

```
#include <stdio.h>

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

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

    return 0;
}
```

 Añadimos una nueva regla a nuestro Makefile.

---

<sup>4</sup>*Fuerza bruta recursiva ;)*

# ¿Qué hace el programa?

- Probemos otros valores de `n` (editar y recompilar con `make sum1`)
- ¿20?
- ¿1 000?
- ¿100 000?
- ¿500 000?

 ¿*Segmentation fault (core dumped)*?

- Antes de volver a ejecutar: **`ulimit -c unlimited`**
- `ls -l`
- ¿Qué ocurre?

- 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

### Makefile

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

# LCG en *módulos*: primer intento ii

## 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 11

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

## Makefile

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

```
$ 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 lc2 lcg2.o generador_lcg.o
$ lc2
0 -> 0
1 -> 1
...
```



## LCG en *módulos*: segundo intento iii

- Modifiquemos el fichero `generador_lcg.h` (cambiamos alguno de los parámetros).
- Ejecutamos `make lcg2`.



¿Qué ocurre? ¿Qué debería ocurrir?

## LCG en *módulos*: segundo intento iii

- Modifiquemos el fichero `generador_lcg.h` (cambiamos alguno de los parámetros).
- Ejecutamos `make 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**.

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

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

# Vectores (*Arrays*)

---

# Definición de variables de tipo *array*

- Sintaxis:


*tipo variable[número];*

- Esa definición crea un espacio de **memoria contigua**,
- **tan grande como lo que indican** *número* y **`sizeof(tipo)`**
- **accesible a través de** *variable* usando la sintaxis

*variable[i]*

- donde *i* deberá estar entre *0* y *número - 1*

# Modificar el programa lcg2.c

 Almacenar M datos en un array y luego imprimirlos.

```
#include <stdio.h>
#include "generador_lcg.h"
int main() {
    int i;
    int aleatorio[M];
    for (i = 0; i < M; i++) {
        aleatorio[i] =
            generar_aleatorio();
    }
```

```
    for (i = 0; i < M; i++) {
        printf(
            "%i -> %i\n",
            i,
            aleatorio[i]);
    }
    return 0;
}
```

## ⚠ ¿Qué pasa si me salgo del array?

```
for (i = -1; i <= M; i++) {  
    printf(  
        "%i -> %i\n",  
        i,  
        aleatorio[i]);  
}
```