Sesión 04: *Módulos* en C

Programación para Sistemas

Ángel Herranz

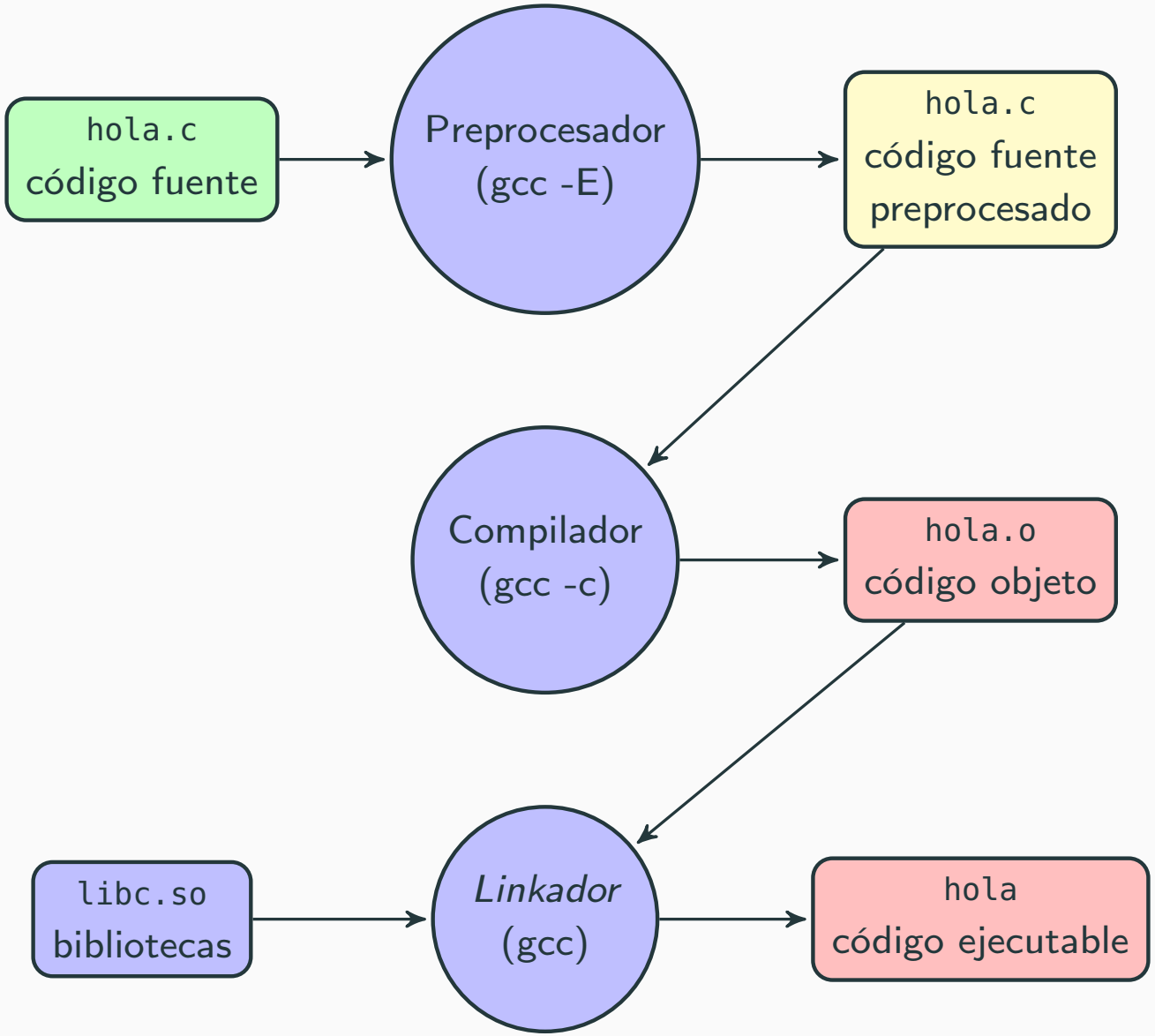
2021-2022

Universidad Politécnica de Madrid

Accesos al servidor

¿Cómo van esos accesos a `triqui`? (`ssh`)

Minirepaso Sesión 1



Minirepaso Sesión 2

- Función `main`
- Argumentos de invocación (`argc` y `argv`)
- Funciones (ej. `factorial`)
- Depuración (método 1): `trazas`, es decir `printf` para ver por donde va el programa
- Depuración (método 2): `gdb`

Minirepaso Sesión 3

- No vemos el detalle de mucha sintaxis de C porque se parece mucho a la de Java: libro de C a mano
- Intuimos el tipo *string*: `char *` (en la siguiente sesión)
- Tipos básicos (enteros): **`char`** e **`int`** y sus *modificadores* **`unsigned`** y **`long`**
- Tipos básicos (coma flotante): **`float`** y **`double`**
- Función de biblioteca `printf` y *conversores*

  `\n`, `%d`, `%i`, `%c`, `%f`, `%.6f`, `%s`, etc.

  para más detalles, desde el terminal: `man 3 printf`
- Makefile: `make basicos` vs. `gcc -ansi -Wall ...`

En el capítulo de hoy...

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

  Chapter 4. Functions and Program Structure

  (*The C Programming Language*, K&R 2nd. edition)
- Sobre la biblioteca estándar
- Más `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 espacios, tabuladores y 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. Ver siguiente transparencia.

lcg1.c: copia, compila y ejecuta

```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;
}
```

```c
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`[1]

Probemos a crear un fichero `Makefile` con este contenido:

```
CFLAGS=-Wall -g


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

Ejecutar `make lcg1` para construir el ejecutable:

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

[1] Un uso un poco más potente cada vez.

make ii

- Y ya se puede ejecutar `lcg1` (observa `./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

- Primera línea: *variable* con nuestros *flags* de gcc:

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

- Las otras dos líneas[2]:

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

  dicen "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 defecto[3].

[2] ¡Cuidado con el tabulador !

[3] 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
- 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?

Disección de `lcg1.c` i

```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;
}
```

- *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`**

Disección 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

Disección 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).
- Cuidado: límite de pila.

Sumar números del 0 a n: `sum1.c`[4]

```c
#include <stdio.h>

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

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

  return 0;
}
```

Repetir los pasos para `lcg1.c` con `sum1.c`

[4] *Fuerza bruta recursiva* ;)

Actualizamos el `Makefile`

- Añadimos una nueva regla a nuestro `Makefile`, como la anterior substituyendo `lcg1` 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?

[5] Si 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: `core`[5]

[5] Si no ves el fichero `core`, prueba ejecutando **`ulimit`** `-c unlimited` antes de ejecutar el programa

Módulos en C

Módulos en C: la biblioteca estándar i

- El aspecto superficial de un módulo en C es un fichero *header* que incluímos (**`#include`**) cuando queremos usar dicho módulo[6]
- C tiene un conjunto de módulos que forman su

biblioteca estándar

- Cada módulo define una serie de tipos y sus funciones
- En esta asignatura hay que aprenderse alguno de esos módulos (ver siguiente transparencia)

[6] Salvando las distancias con Java se parece a **`import`**.

Biblioteca Estándar (apéndice B del libro de C)

- Input and Output: `<stdio.h>` (`man 3 stdio`)
- Character Class Tests: `<ctype.h>`
- String Functions: `<string.h>` (`man 3 string`)
- Mathematical Functions: `<math.h>`
- Utility Functions: `<stdlib.h>` (busca el fichero `.h`)
- Diagnostics: `<assert.h>`
- Variable Argument Lists: `<stdarg.h>`
- Non-local Jumps: `<setjmp.h>`
- Signals: `<signal.h>`
- Date and Time Functions: `<time.h>`
- Implementation-defined Limits: `<limits.h>` y `<float.h>`
- Y otros módulos (`<errno.h>`, `<sysexits.h>`, etc.)

Módulos en C: nuestro primer módulo

- 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

```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

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

LCG en *módulos*: primer intento ii

Añadimos dos nuevas reglas al `Makefile`:

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

```
$ make lcg2
cc -Wall -g   -c -o generador_lcg.o generador_lcg.c
cc -Wall -g   -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**: 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

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

extern int generar_aleatorio();
```

generador_lcg.c

```c
#include "generador_lcg.h"
int x = 0;
int generar_aleatorio() {
  int anterior = x;
  x = (A * x + C) % M;
  return anterior;
}
```

lcg2.c

```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;
}
```

- **extern**: sólo declaración

LCG en *módulos*: segundo intento ii

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

Aviso

Si acabas haciendo

```
#include "mi_modulo.c"
```

es por que algo estás entendiendo mal

Aviso

Si acabas haciendo

```
#include "mi_modulo.c"
```

es por que algo estás entendiendo mal

include sólo de *headers* (`.h`)

Convención para evitar dobles inclusiones

generador_lcg.h

```
#ifndef _GENERADOR_LCG_H
#define _GENERADOR_LCG_H

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

extern int generar_aleatorio();

#endif /* generador_lcg.h included.  */
```

Cada semana una hoja

Entre otros ejercicios:

- Usar GDB con los ejemplos de hoy
- Exponer variables globales de un módulo (ej. `generador_lcg.c`)