

Sesión 7: Programas y paso de parámetros

Programación para Sistemas

Ángel Herranz

Curso 2024-2025

En el capítulo de hoy...

- Funciones, variables, ámbito y pila
- Paso de parámetros por valor y por referencia
- Más `make`

GNU Make Manual

The Free Software Foundation (FSF)

<http://makefiletutorial.com/>

Chase Lambert

Una 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

Funciones, variables y ámbito

LCG: generando números aleatorios

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

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

💬 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.

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

- 💬 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

```
#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** (cada vez un poco más)

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

```
CFLAGS=-Wall -Wextra -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
```

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 líneas¹:

```
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*².

¹!Cuidado 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?

Disección 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**

Disección de 1cg1.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**: anterior
- **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 con recursión: `sum1.c`

```
#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;
}
```

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

Actualizamos el Makefile

- 📄 Añadimos una **nueva regla** a nuestro Makefile, como la anterior substituyendo `lsg1` 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?

³Si 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: `core3`

³Si no ves el fichero `core`, prueba ejecutando `ulimit -c unlimited` antes de ejecutar el programa

This is C

*p

&x

Direcciones de memoria

- C permite un **control absoluto** de la memoria
- Nueva **sintaxis** para expresiones:

&x

Dirección de memoria de x
(x puede ser cualquier *lvalue*)

- Usaremos el *conversion specifier* **%p** de `printf` para mostrar **direcciones de memoria**

¿Donde está la variable?

```
int x = 42;  
printf("El contenido de x es %i\n", x);  
printf("La dirección de memoria de x es %p\n", &x);
```

¿Donde está la variable?

```
int x = 42;  
printf("El contenido de x es %i\n", x);  
printf("La dirección de memoria de x es %p\n", &x);
```

El contenido de x es 42

La dirección de memoria de x es 0x7ffc4e20391c



```
#include <stdio.h>
int global1;
int global2;

void f (int arg) {
    int local;
    printf("f(%i): &arg: %p\n",
           arg, &arg);
    printf("f(%i): &local: %p\n",
           arg, &local);
    if (arg) f(!arg);
}
```

```
int main() {
    int local;
    printf("main: &local: %p\n", &local);
    printf("main: &global1: %p\n",
           &global1);
    printf("main: &global2: %p\n",
           &global2);
    printf("main: &f: %p\n", &f);
    printf("main: &main: %p\n", &main);
    f(1);
    return 0;
}
```



- 💬 ¿Puedes ver donde están las variables globales?
- 💬 ¿Puedes ver lo que ocupan?
- 💬 ¿Puedes ver cómo se distribuyen las variables y argumentos en el *stack*?
- 💬 ¿Has observado que las funciones son variables globales?
- 🏠 Añade más variables locales y argumentos

Punteros: variables con direcciones de memoria

- Sintaxis para declarar punteros:

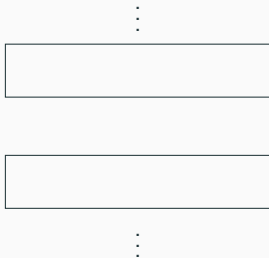
$$T \ *p;$$

- p es una variable que contiene una dirección de memoria,
- en la que hay un elemento de tipo T (donde T es cualquier tipo, ej. **char**, **int**, **long unsigned int**, etc.)
- accesible usando la expresión

$$*p$$

*Contenido de la dirección de
memoria almacenada en p*

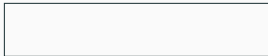
Encajando las piezas i



Encajando las piezas i

```
int x;
```

⋮
x: -243291612 0x7fff15fb17d0



⋮

Encajando las piezas i

```
int x;  
int *p;
```

⋮

x: -243291612 0x7fff15fb17d0

p: 0x560a2995479d 0x7fff15fb17c8

⋮

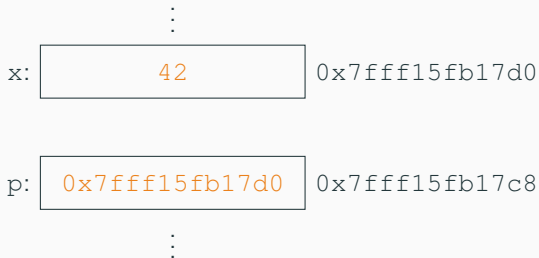
Encajando las piezas i

⋮
x: 42 0x7fff15fb17d0

p: 0x560a2995479d 0x7fff15fb17c8
⋮

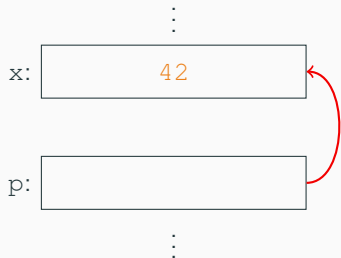
```
int x;  
int *p;  
x = 42;
```

Encajando las piezas i



```
int x;  
int *p;  
x = 42;  
p = &x;
```

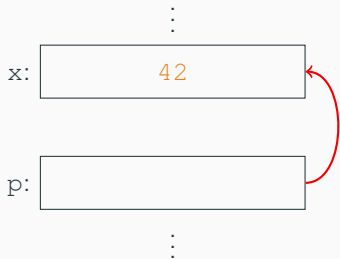
Encajando las piezas i



```
int x;  
int *p;  
x = 42;  
p = &x;
```

Representación habitual

Encajando las piezas i

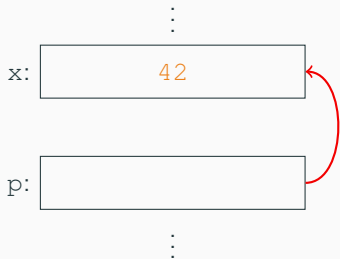


```
int x;  
int *p;  
x = 42;  
p = &x;
```

Representación habitual

```
printf("%i\n", *p)
```

Encajando las piezas i



```
int x;  
int *p;  
x = 42;  
p = &x;
```

Representación habitual

```
printf("%i\n", *p)  
42
```

Encajando las piezas ii

 ¿Qué hacen estas dos líneas después del código anterior?

```
*p = 27;
```

```
printf("%i\n", x);
```

Encajando las piezas ii

 ¿Qué hacen estas dos líneas después del código anterior?

```
*p = 27;  
printf("%i\n", x);
```

 Entender estas últimas transparencias es **muy importante**



```
int x = 42, y = 27;  
printf("Antes de intercambiar: (%i, %i)\n", x, y);  
intercambiar(x,y);  
printf("Despues de intercambiar: (%i, %i)\n", x, y);
```

Lo esperado:

Antes de intercambiar: (42, 27)

Despues de intercambiar: (27, 42)



```
void intercambiar(int x, int y) {  
    int aux = x;  
    x = y;  
    y = aux;  
}
```



```
void intercambiar(int x, int y) {  
    int aux = x;  
    x = y;  
    y = aux;  
}
```

💬 ¿Qué ocurre? (¡dibujémoslo en cajas!)



```
void intercambiar(int x, int y) {  
    int aux = x;  
    x = y;  
    y = aux;  
}
```

💬 ¿Qué ocurre? (¡dibujémoslo en cajas!)

⚠ Paso por valor: el contenido de las variables
se **copia** en los argumentos

📁 ¿Y si pasamos los **punteros como argumento**?



- El **paso de parámetros por referencia** se usa para tener **parámetros de salida**
- Por ejemplo, la siguiente cabecera es la de una función `sum` que **devuelve en el segundo parámetro** la suma de los números de 0 a n

```
void sum(unsigned n, unsigned *s)
```

- 🏠 Prepara un programa `sum2.c` que tome como argumento n e imprima la suma de números de 0 a n en la salida estándar:

```
$ ./sum2 6
```

```
21
```

```
$ |
```



Código: **UFEBD**