

Sesión 08: Más sobre tipos y módulos

Programación para Sistemas

Ángel Herranz

2022-2023

Universidad Politécnica de Madrid

Recordatorio *structs*

- Dos **variables** representando puntos en Cartesianas:

```
struct {  
    float x;  
    float y;  
} a, b;
```

Recordatorio *structs*

- Dos **variables** representando puntos en Cartesianas:

```
struct {  
    float x;  
    float y;  
} a, b;
```

- Otra **variable** más:

```
struct {  
    float x;  
    float y;  
} c;
```

Recordatorio *structs*

- Dos **variables** representando puntos en Cartesianas:

```
struct {  
    float x;  
    float y;  
} a, b;
```

- Otra **variable** más:

```
struct {  
    float x;  
    float y;  
} c;
```

- Para no repetir:

Recordatorio *structs*

- Dos **variables** representando puntos en Cartesianas:

```
struct {  
    float x;  
    float y;  
} a, b;
```

- Otra **variable** más:

```
struct {  
    float x;  
    float y;  
} c;
```

- Para no repetir:

```
struct punto_s {  
    float x;  
    float y;  
};
```

```
struct punto_s a, b;  
struct punto_s c;
```

- **punto_s** es una **etiqueta**
- **struct punto_s** es un **tipo**

Recordatorio punteros a *structs*

```
rectp = (struct rectangulo_s *) malloc(sizeof(struct rectangulo_s));
```

Recordatorio punteros a *structs*

```
rectp = (struct rectangulo_s *) malloc(sizeof(struct rectangulo_s));
```

Masivamente utilizados en C

FOPEN(3) Linux Programmer's Manual FOPEN(3)

NAME

 fopen, fdopen, freopen - stream open functions

SYNOPSIS

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

```
FILE *fopen(const char *pathname, const char *mode);
```

Recordatorio: typedef

- Usando **typedef** podríamos hacer esto

```
struct punto_s {float x; float y};  
typedef struct punto_s punto_t;
```

- Podríamos declarar variables de una forma más **legible**:

```
punto_t a, b;  
rectangulo_t r, s;
```

- Podríamos usarlo al invocar malloc de esta forma:

```
rectp = (rectangulo_t *) malloc(sizeof(rectangulo_t ));  
  
rectp = (struct rectangulo_s *) malloc(sizeof(struct rectangulo_s));
```

En el capítulo de hoy...

- Más sobre tipos: *Enum*, *Union*
- Repaso de la sintaxis (y semántica): algunas convenciones
- Módulos

Enum

- Una forma asociar constantes a nombres es **#define**
- Muchas veces lo que queremos es simplemente hacer una enumeración: ej. días de la semana, tipos de figuras geométricas, etc.
- Para ello C introduce **enum**

```
enum forma {CIRCULO, CUADRADO};
```

- forma es una **etiqueta**
- **Nuevo tipo:** **enum** forma
- **Dos constantes:** CIRCULO y CUADRADO
- El siguiente código declara la variable f:

```
enum forma f;
```

- Semántica

$$\llbracket \text{CIRCULO} \rrbracket = 0$$
$$\llbracket \text{CUADRADO} \rrbracket = 1$$
$$\llbracket \text{enum forma} \rrbracket = \{0, 1\}$$

- ¡Todo son enteros en C!

¿Qué significa?

```
enum mes {ENERO, FEBRERO, MARZO, ..., DICIEMBRE};
```



¿Qué significa?

```
enum mes {ENERO, FEBRERO, MARZO, ..., DICIEMBRE};
```

$$\llbracket \text{ENERO} \rrbracket = 0$$
$$\llbracket \text{FEBRERO} \rrbracket = 1$$
$$\llbracket \text{MARZO} \rrbracket = 2$$
$$\vdots$$
$$\llbracket \text{DICIEMBRE} \rrbracket = 11$$
$$\llbracket \text{enum mes} \rrbracket = \{0, 1, 2, \dots, 11\}$$

Meses i

- Función que recibe un mes (del tipo **enum** mes) y que devuelve los días que tiene dicho mes

```
int dias(enum mes m) {  
    int d;  
    switch (m) {  
        case FEBRERO:  
            d = 28;  
            break;  
        case ABRIL:  
        case JUNIO:  
        case SEPTIEMBRE:
```

```
        case NOVIEMBRE:  
            d = 30;  
            break;  
        default:  
            d = 31;  
    }  
    return d;  
}
```

- 🏠 Escribe una función que reciba un mes (del tipo **enum** mes) y que devuelva el nombre del mes en español
- 💻 Escribe una función que reciba un string con el nombre en español de un mes y que devuelva el valor correcto del tipo **enum** mes 🕒 5'

enum iii

```
enum dia {LUNES = 1, MARTES, MIERCOLES, ..., DOMINGO};
```

enum iii

```
enum dia {LUNES = 1, MARTES, MIERCOLES, ..., DOMINGO};
```

$$\llbracket \text{LUNES} \rrbracket = 1$$
$$\llbracket \text{MARTES} \rrbracket = 2$$
$$\llbracket \text{MIERCOLES} \rrbracket = 3$$
$$\vdots$$
$$\llbracket \text{DOMINGO} \rrbracket = 7$$
$$\llbracket \text{enum dia} \rrbracket = \{1, 2, \dots, 7\}$$

Union

A union is a variable that may hold at different times objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program.

Capítulo 6, K&R

union ii

- Empezamos creando una variable para información de contacto: un teléfono o (exclusivo) un email

```
union {  
    char telefono[16];  
    char email[31];  
} c;
```

- El código anterior declara la variable `c`,
- capaz de almacenar dos strings de 15 y 30 caracteres aunque no a la vez,
- strings accesibles con la sintaxis `c.telefono` y `c.email` como en *struct*

Sintaxis similar a *struct*

- Semántica completamente diferente:

struct	union
×	∪

-  Escribe un programa con una variable *union* como la anterior y explora sintaxis y semántica. ⌚ 5'

// Ejemplo para explorar:

```
printf("sizeof(c) == %u\n", sizeof(c));  
strcpy(c.telefono, "34123456789");  
strcpy(c.email, "johndoe@example.org");  
printf("telefono == %s\n", c.telefono);  
printf("email == %s\n", c.email);  
printf("sizeof(c) == %u\n", sizeof(c));
```

union iii

- Igual que ocurre con **struct**, la frase

```
union {char telefono[16]; char email[31];}
```

se puede considerar como **un nuevo tipo** que se puede declarar con una *etiqueta (tag)* de esta forma

```
union contacto {  
    char telefono[16];  
    char email[31];  
};
```

- Ahora **la etiqueta contacto** nos permite declarar variables así:

```
union contacto c1, c2;
```

- Es posible **combinar** declaraciones *union*, *structs* y *arrays*

🏠 Profundizar en la siguiente representación de figuras geométricas:

```
enum forma_de_figura {TRIANGULO, RECTANGULO, CIRCULO};  
struct figura {  
    enum forma_de_figura forma;  
    union {  
        struct {struct punto c, float r} circulo;  
        struct {struct punto so, ne} rectangulo;  
        struct {struct punto a, b, c} triangulo;  
    } contenido;  
}
```

- Observa que **si f es una figura**,
f.contenido.triangulo sólo tiene sentido si **f.forma == TRIANGULO**

Code conventions (aka coding style)

Nombres de tipos	sufijo <code>_t</code>
Etiquetas (<i>tag</i>) de enum	sufijo <code>_e</code>
Etiquetas de struct	sufijo <code>_s</code>
Etiquetas de union	sufijo <code>_u</code>

💬 ¿Reglas? ¿Por qué?

¹Solo son dos ejemplos, busca y siéntete bien con `unas`.

Code conventions (aka coding style)

Nombres de tipos	sufijo <code>_t</code>
Etiquetas (<i>tag</i>) de enum	sufijo <code>_e</code>
Etiquetas de struct	sufijo <code>_s</code>
Etiquetas de union	sufijo <code>_u</code>

💬 ¿Reglas? ¿Por qué?

- Lo más importante no son **qué reglas** si no **usar unas**

NASA C Style Guide, GNU Coding Standards¹

🏠 Adapta lo que hayas hecho hoy en clase a estas reglas. Sigue estas reglas el resto de la sesión y de la asignatura.

¹Solo son dos ejemplos, busca y siéntete bien con **unas**.

Ejemplo: tipo pila

```
/* Declaración de un struct, sólo el nombre */  
struct nodo_pila_s;
```

Ejemplo: tipo pila

```
/* Declaración de un struct, sólo el nombre */
```

```
struct nodo_pila_s;
```

```
/* Definición del tipo pila_t */
```

```
typedef struct nodo_pila_s *pila_t;
```

Ejemplo: tipo pila

```
/* Declaración de un struct, sólo el nombre */
```

```
struct nodo_pila_s;
```

```
/* Definición del tipo pila_t */
```

```
typedef struct nodo_pila_s *pila_t;
```

```
/* Definición del struct */
```

```
struct nodo_pila_s {
```

```
    int cima;
```

```
    pila_t resto;
```

```
};
```

Ejemplo: tipo árbol binario de enteros

```
/* Declaración de un struct, sólo el nombre */  
struct arbol_bin_int_s;
```

Ejemplo: tipo árbol binario de enteros

```
/* Declaración de un struct, sólo el nombre */
```

```
struct arbol_bin_int_s;
```

```
/* Definición del tipo arbol_bin_int_t */
```

```
typedef struct arbol_bin_int_s *arbol_bin_int_t;
```

Ejemplo: tipo árbol binario de enteros

```
/* Declaración de un struct, sólo el nombre */  
struct arbol_bin_int_s;  
  
/* Definición del tipo arbol_bin_int_t */  
typedef struct arbol_bin_int_s *arbol_bin_int_t;  
  
/* Definición del struct */  
struct arbol_bin_int_s {  
    int raiz;  
    arbol_binario_t hi;  
    arbol_binario_t hd;  
};
```

Operadores

Precedencia y asociatividad



Tabla 2-1 de KR:

*Operators in the same line have the same precedence;
rows are in order of decreasing precedence*

OPERATORS	ASSOCIATIVITY
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
?:	right to left
= += -= *= /= %= &= ^= = <=> >=	right to left
,	left to right

Unary +, -, and * have higher precedence than the binary forms.



¡Siempre a mano! (la de cualquier lenguaje)

Pon los paréntesis donde los pondría C

`c > a > b`

`c == a > b`

`c > a = b`

`1 > 2 + 3 && 4`

`1 == 2 != 3`

`e = (a + b) * c / d`

`a * a - 3 * b + a / b`

`a & b || c`

`a = b || c`

`q && r || s--`

`p == 0 ? p += 1: p += 2`

¿De qué tipo es x?

```
int *x();  
int (*x)();  
char **x;  
int (*x)[13];  
int *x[13];
```

¿De qué tipo es qsort? ¿Y compar?

QSORT(3)

Linux Programmer's Manual

QSORT(3)

NAME

qsort, qsort_r - sort an array

SYNOPSIS

```
#include <stdlib.h>
```

```
void qsort(void *base, size_t nmemb, size_t size,  
           int (*compar)(const void *, const void *));
```

...

Punteros a funciones

```
void qsort(void *base, size_t nmemb, size_t size,  
          int (*compar)(const void *, const void *));
```

```
int comparc(char *c1, char *c2) {  
    if (*c1 < *c2) return -1;  
    if (*c1 > *c2) return 1;  
    return 0;  
}  
  
...  
  
/* Ordena todos los caracteres de palabra */  
qsort(palabra, strlen(palabra), sizeof(char), &comparc);
```

Módulos en C

Módulos en C: la biblioteca estándar i

- El **aspecto superficial** de un módulo en C es un fichero **header** que incluimos (**#include**) cuando queremos usar dicho módulo²
- C tiene un conjunto de módulos que forman su

biblioteca estándar

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

²Salvando 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 (generador de números pseudoaleatorios) 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;
}
```

lcg3.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:

```
...  
lcg3: generador_lcg.o lcg3.o  
      $(CC) $(CFLAGS) -o $@ $^
```

LCG en *módulos*: primer intento ii

Añadimos dos nuevas reglas al Makefile:

```
...  
lcg3: generador_lcg.o lcg3.o  
      $(CC) $(CFLAGS) -o $@ $^
```

```
$ make lcg3  
cc -Wall -g -c -o generador_lcg.o generador_lcg.c  
cc -Wall -g -c -o lcg3.o lcg3.c  
lcg3.c: In function 'main':  
lcg3.c:4:19: error: 'M' undeclared (first use in this function)  
    for (i = 0; i < M; i++) {  
                      ^  
lcg3.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 `lcg3.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 `lcg3.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;
}
```

lcg3.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 lcg3
cc -Wall -g -c -o lcg3.o lcg3.c
cc -Wall -g -c -o generador_lcg.o generador_lcg.c
cc -Wall -g -o lcg3 lcg3.o generador_lcg.o
$ ./lcg3
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 lcg3` y luego nuestro programa `./lcg3`

💬 ¿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 lcg3` y luego nuestro programa `./lcg3`

 ¿Qué ocurre? ¿Qué debería ocurrir?

 Hay que decirle a make que tanto `generador_lcg.o` como `lcg3.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  
  
lcg3.o: lcg3.c generador_lcg.h
```

LCG en *módulos*: segundo intento iv

- El resultado final es:

```
$ make lcg3
cc -Wall -g -c -o lcg3.o lcg3.c
cc -Wall -g -c -o generador_lcg.o generador_lcg.c
cc -Wall -g -o lcg3 lcg3.o generador_lcg.o
$ ./lcg3
0 -> 0
1 -> 1
2 -> 8
3 -> 5
4 -> 10
5 -> 6
...
12 -> 0
$ |
```

Si acabas haciendo
#include "mi_modulo.c"
es por que algo estás entendiendo mal

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. */
```

Módulo para árboles binarios de enteros

arbol_bin_int.h

```
/* Devuelve un árbol vacío */
extern arbol_bin_int_t
    crear_vacio();

/* Devuelve un árbol no vacío */
extern arbol_bin_int_t
    crear_nodo(int r,
               arbol_bin_int_t i,
               arbol_bin_int_t d);

/* Inserta un dato en "orden" */
extern arbol_bin_int_t
    insertar(arbol_bin_int_t a,
             int dato);
```

```
/* Devuelve el hijo izquierdo */
extern arbol_bin_int_t
    hi(arbol_bin_int_t a);

/* Devuelve el hijo derecho */
extern arbol_bin_int_t
    hd(arbol_bin_int_t a);

/* Devuelve la raiz del arbol */
extern int
    raiz(arbol_bin_int_t a);

/* Decide si es vacío */
extern int
    es_vacio(arbol_bin_int_t a);
```



- Aunque ya había algo en los ejercicios de la sesión anterior...
- Si a es de tipo `arbol_bin_int_t`

```
typedef struct arbol_bin_int_s *arbol_bin_int_t;  
arbol_bin_int_t a;
```



¿Cómo se accede a la raíz?



- Aunque ya había algo en los ejercicios de la sesión anterior...
- Si a es de tipo `arbol_bin_int_t`

```
typedef struct arbol_bin_int_s *arbol_bin_int_t;  
arbol_bin_int_t a;
```



¿Cómo se accede a la raíz?

`*a.raiz`

¿Error?



- Aunque ya había algo en los ejercicios de la sesión anterior...
- Si a es de tipo `arbol_bin_int_t`

```
typedef struct arbol_bin_int_s *arbol_bin_int_t;  
arbol_bin_int_t a;
```



¿Cómo se accede a la raíz?

`*(a.raiz)`

C pone ahí los paréntesis



- Aunque ya había algo en los ejercicios de la sesión anterior...
- Si a es de tipo arbol_bin_int_t

```
typedef struct arbol_bin_int_s *arbol_bin_int_t;  
arbol_bin_int_t a;
```



¿Cómo se accede a la raíz?

`(*a).raiz`

¡Qué feo!



- Aunque ya había algo en los ejercicios de la sesión anterior...
- Si a es de tipo arbol_bin_int_t

```
typedef struct arbol_bin_int_s *arbol_bin_int_t;  
arbol_bin_int_t a;
```



¿Cómo se accede a la raíz?

```
a->raiz          ;)
```



Ordenar enteros

- Escribe un programa que ordene enteros de menor a mayor
- La entrada estándar tiene
 - Un entero positivo n en la primera línea
 - n enteros en las n siguientes líneas
- La salida de tu programa tiene los n enteros después de la primera línea ordenados de menor a mayor



Ordenar enteros

- Escribe un programa que ordene enteros de menor a mayor
- La entrada estándar tiene
 - Un entero positivo n en la primera línea
 - n enteros en las n siguientes líneas
- La salida de tu programa tiene los n enteros después de la primera línea ordenados de menor a mayor

Usamos el módulo de árboles binarios

while (n) Ordenar

Evita consumir más memoria de la necesaria



- Por convención en los *headers*, para evitar dobles inclusiones:

```
#ifndef _ARBOL_BIN_INT_H
#define _ARBOL_BIN_INT_H
...
#endif
```

- **#include** "arbol_bin_int.h" tanto en arbol_bin_int.c como en ordenar.c
- gcc -o arbol_bin_int.o -c arbol_bin_int.c
- gcc -o ordenar.o -c ordenar.c
- gcc -o ordenar ordenar.o arbol_bin_int.o

Transparencias recordatorio

Recordatorio *módulos*

generador_lcg.h

```
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;
}
```

lcg3.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;
}
```

Convención (evitará *dobles inclusiones*)

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;
}
```

lcg3.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;
}
```

Convención (evitará *dobles inclusiones*)

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;
}
```

lcg3.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;
}
```

#ifndef GENERADOR_LCG_H: Si no está definida la macro `GENERADOR_LCG_H` entonces se procesa todo hasta **#endif** (en otro caso no se procesa)

🔍 Busca *headers* de la biblioteca estándar como `stdio.h` o `limits.h` y mira cómo siguen la convención.