

09-transpas-tipos.pdf



Anónimo



Programación Para Sistemas



2º Grado en Ingeniería Informática



**Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid**

Sesión 09: Más sobre tipos y sintaxis

Programación para Sistemas

Ángel Herranz

2021-2022

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 más:

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

Recordatorio *structs*

- Dos variables representando puntos en Cartesianas:

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

- Otra 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 más:

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

- Para no repetir:

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

```
struct punto a, b;  
struct punto c;
```

- punto es una *etiqueta*

Recordatorio punteros a *structs*

```
rectp = (struct rectangulo *)  
        malloc(sizeof(struct rectangulo));  
  
        (*rectp).ne
```

Recordatorio punteros a *structs*

```
rectp = (struct rectangulo *)  
        malloc(sizeof(struct rectangulo));  
  
(*rectp).ne, mucho mejor: rectp->ne
```


Recordatorio punteros a *structs*

```
rectp = (struct rectangulo *)  
        malloc(sizeof(struct rectangulo));  
  
        (*rectp).ne, mucho mejor: rectp->ne
```

- 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);
```

```
...
```

En el capítulo de hoy...

- *Enum*
- *Union*
- *Typedef*
- Repaso de la sintaxis (y semántica)

Enum

enum i

- 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};
```

- **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};
```

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

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 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,
- los strings son accesibles con la sintaxis `c.telefono` y `c.email`

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;
```


union iv

- Es posible **combinar** declaraciones *union*, *structs* y *arrays*
- 🏠 Profundizar en el tipo **struct** figura como representación de figuras geométricas:

```
enum tipo_de_figura {TRIANGULO, RECTANGULO, CIRCULO};  
struct figura {  
    enum tipo_de_figura tipo;  
    union {  
        struct {struct punto a, b, c} triangulo;  
        struct {struct punto so, ne} rectangulo;  
        struct {struct punto c, int r} circulo;  
    } contenido;  
}
```

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

Typedef

typedef: definiendo nuevos tipos

- Podemos definir nuevos tipos con **typedef**
- Ejemplo

```
typedef long long unsigned int natural;
```

typedef: definiendo nuevos tipos

- Podemos definir nuevos tipos con **typedef**
- Ejemplo

```
typedef long long unsigned int natural;
```

- Funciona igual que la definición de una variable,
- pero define un nuevo tipo, **natural**, que es igual a **long long unsigned int**

 Por convención, voy a usar el sufijo **_t** para los tipos

```
typedef long long unsigned int natural_t;
```

```
// Ejemplo de decl de variable de tipo natural_t:  
natural_t n, m;
```

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

💬 ¿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;
```

```
};
```

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

Operadores

Operadores

- ¿Qué entendemos por operador?
- Mejor que una definición...

`+, -, *, /, %, >, >=, <, <=, ==, !=, &&, ||, !, (-), ++, --, &, |, ^, <<, >>, ~, =, binop=, *, &, ->, _?_:_`

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];  
char ((*x())[5])();  
char ((*x[3])())[5];  
(*x[5])() int (*x)(int a, int b)
```