

Introducción a la Programación

Tipos

`typedef`, `enum`, uniones etiquetadas y comportamiento indefinido

Gerson Garrido

Semana 7

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Un tipo dice qué existe y qué se puede hacer
2. `typedef` y `enum`
3. `union` y unión etiquetada
4. Una variante que posee memoria
5. Conversiones: `-1 < 1u`
6. Comportamiento indefinido: por qué “me funcionó” no prueba nada

> **Lectura**
>
> Apuntes de la semana 7. Código en `code/week-07`.

SECCIÓN 01

Un tipo dice qué existe

01

El mismo `int`, tres significados

Un `int` que vale 2 puede ser una nota, 2 pasajeros o el estado “terminado”.

- Un tipo responde: **qué valores existen** y **qué operaciones tienen sentido**.
- Con un `int` se puede escribir el estado 7, o sumar dos estados.
- Son **estados inválidos**: el tipo los permite, y el problema no.

> **Idea clave**
>
> Pregúntense qué valores **no** deberían existir. Mientras menos se puedan representar, menos hay que revisar y probar.

SECCIÓN 02

typedef y enum

02



`typedef`: otro nombre, no otro tipo

typedef_units.c

```c
typedef int Meters;
typedef int Seconds;

double speed(Meters distance, Seconds time) {
    return (double) distance / time;
}
```

terminal

```
$ build/typedef_units
bien: 20.00 m/s
al revés: 0.05 m/s
```

Sin ninguna advertencia. Para `gcc` son el mismo tipo, con tres nombres.

Ojo

`typedef` documenta, pero no protege.

Dos `struct` sí son tipos distintos

errors/struct_units.c

```c
typedef struct { int value; } Meters;
typedef struct { int value; } Seconds;
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/struct_units.c -o build/struct_units
errors/struct_units.c:15:28: error: incompatible type for argument 1 of 'speed'
errors/struct_units.c:15:34: error: incompatible type for argument 2 of 'speed'
```

Ahora el error es de compilación. El precio: escribir `distance.value` en todas partes. Pocas veces vale la pena para unos metros, pero la puerta existe.

Primer intento: números con nombre

trip_defines.c

```c
#define TRIP_WAITING 0
#define TRIP_ON_ROUTE 1
#define TRIP_FINISHED 2

const char *state_name(int state) {
    if (state == TRIP_WAITING) {
        return "esperando";
    }
    if (state == TRIP_ON_ROUTE) {
        return "en ruta";
    }
    return "terminado";
}
```

terminal

```
$ build/trip_defines
1: en ruta
7: terminado
```

- El estado 7 no existe, y dijo “terminado”.
- `#define` reemplaza texto: el compilador solo ve un 1.

`enum`: un tipo con valores con nombre

trip_enum.c

```c
enum trip_state { TRIP_WAITING, TRIP_ON_ROUTE,
      TRIP_FINISHED };

const char *state_name(enum trip_state state) {
    switch (state) {
    case TRIP_WAITING:
        return "esperando";
    case TRIP_ON_ROUTE:
        return "en ruta";
    case TRIP_FINISHED:
        return "terminado";
    }
    return "desconocido";
}
```

enum_values.c

```c
enum fare { FARE_STUDENT = 100,
      FARE_ADULT = 250,
      FARE_NIGHT };
```

terminal

```
$ build/enum_values
0 1 2
100 250 251
```

Se numeran solos desde 0, o desde el valor que se les dé.

Primera ganancia: `gdb` sabe los nombres

con `#define`

```
Breakpoint 1, state_name (state=1) at trip_defines.c:9
(gdb) print state
$1 = 1
(gdb) whatis state
type = int
```

con `enum`

```
Breakpoint 1, state_name (state=TRIP_WAITING) at trip_enum.c:7
(gdb) print state
$1 = TRIP_WAITING
(gdb) print/d state
$2 = 0
(gdb) whatis state
type = enum trip_state
```

Compilado con `-g -O0`.

Segunda ganancia: el `switch` revisado

Agregamos `TRIP_CANCELLED` y nos olvidamos de atenderlo:

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/missing_case.c -o build/missing_case
errors/missing_case.c:7:5: warning: enumeration value 'TRIP_CANCELLED' not handled in switch [-Wswitch]
```

`gcc` les hace la lista de lugares por cambiar. Pero con `default`, se calla: `errors/default_hides.c` compila sin avisos.

Error común

Poner `default` por costumbre sobre un `enum`. Si atienden todas las etiquetas, no lo pongan.

Tercera ganancia, y el límite

errors/enum_mix.c

```c
void show(enum trip_state state, int passengers) {
show(SEAT_AISLE, 20);
show(20, TRIP_ON_ROUTE);
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/enum_mix.c -o build/enum_mix
errors/enum_mix.c:12:10: warning: implicit conversion from 'enum seat' to 'enum trip_state' [-Wenum-conversion]
$ build/enum_mix
estado 1, 20 pasajeros
estado 20, 1 pasajeros
```

- Dos errores, **un** aviso. Mezclar dos `enum` avisa. Un `int` entra en silencio.
- Por dentro es un entero: `enum trip_state state = 7;` compila.
- Por eso `state_name` necesita un `return` después del `switch`: si no, `-Wreturn-type`.

¿Cuánto protege cada uno?

Situación	`#define`	`typedef int`	`enum`	`struct`
Nombre en `gdb`	No	No	**Sí**	Muestra el campo
Avisa si falta un caso en `switch`	No	No	**Sí**	No aplica
Avisa si se mezclan dos tipos así	No	No	**Sí**	**Error**
Avisa si se usa un `int` en su lugar	No	No	No	**Error**

Regla práctica

`enum` si los valores forman un conjunto. `#define` o `const` si es un número suelto, como un tamaño máximo.

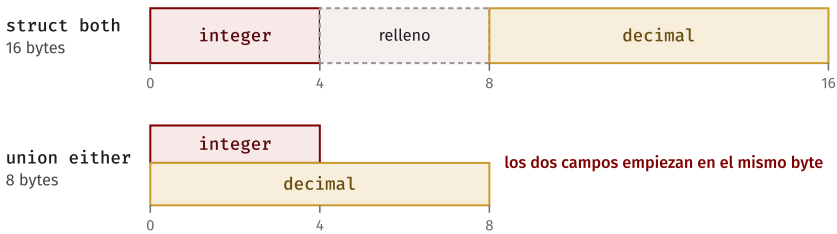
SECCIÓN 03

union

03



`struct` y `union`, en la memoria



- `struct`: los campos van uno después del otro, y existen todos a la vez.
- `union`: los campos van uno encima del otro. Hay espacio para uno, el más grande.
- Los tamaños dependen de la máquina. Aquí, el `double` debe empezar en un múltiplo de 8.

Escribir un campo pisa a los demás

union_bytes.c

```c
union either u;
u.integer = 7;
printf("guardé un int: %d\n", u.integer);

u.decimal = 3.14;
printf("guardé un double: %.2f\n", u.decimal)
    ;
printf("leído como int: %d\n", u.integer);
```

terminal

```
$ build/union_bytes
guardé un int: 7
guardé un double: 3.14
leído como int: 1374389535
```

- Son 4 de los 8 bytes de 3.14, leídos como `int`. Depende de la máquina, y no significa nada.
- La `union` no recuerda cuál campo se escribió.

SECCIÓN 04

Unión etiquetada

04

Un pago con todos sus campos

Un pago de pasaje es en efectivo, con tarjeta **o** con billetera del celular.

payment_sizes.c

```c
struct loose_payment {
    int kind;
    double cash;
    int card_end;
    char *phone;
};
struct loose_payment bad = {0, 2.5, 4821,
      NULL};
```

- ¿En efectivo, y con tarjeta? ¿Cuál vale?
- Es un estado que el problema no tiene.
- Cada función, y cada prueba, tendría que considerarlo.

Unión etiquetada: las tres herramientas juntas

payment.h

```c
typedef enum { PAY_CASH, PAY_CARD, PAY_WALLET }
      PaymentKind;

typedef struct {
    PaymentKind kind;
    union {
        double cash;    // PAY_CASH: soles
      recibidos
          int card_end;  // PAY_CARD: últimos
      cuatro dígitos
          char *phone;    // PAY_WALLET: número del
       celular; lo posee el Payment
    } as;
} Payment;
```

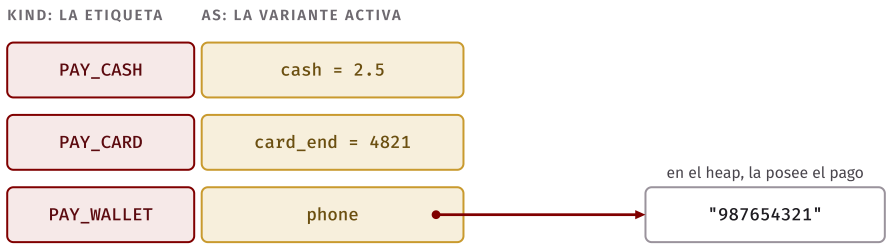
Unión etiquetada *(tagged union)*

Un `struct` con una etiqueta y una `union`. La etiqueta dice qué variante está activa.

terminal

```
$ build/payment_sizes
sizeof(struct loose_payment) = 32
sizeof(Payment) = 16
```

La etiqueta dice qué mirar



Un solo arreglo, de un solo tipo, con pagos de formas distintas. La mezcla de antes ya no se puede escribir.

Dos reglas de uso

1. Se crea con un constructor

payment.c

```c
Payment payment_cash(double soles) {
    Payment p;
    p.kind = PAY_CASH;
    p.as.cash = soles;
    return p;
}
```

La etiqueta y la variante concuerdan en un solo lugar. Y el `switch` va sin `default`, para que `-Wall` revise los casos.

2. Se lee después de mirar la etiqueta

payment.c

```c
switch (p->kind) {
case PAY_CASH:
    printf("efectivo: S/ %.2f\n", p->as.cash);
    break;
case PAY_CARD:
    printf("tarjeta terminada en %04d\n", p->as.card_end);
    break;
case PAY_WALLET:
    printf("billetera del %s\n", p->as.phone);
    break;
}
```

`gdb` no sabe de etiquetas

gdb, en payment_print

```
(gdb) print *p
$1 = {kind = PAY_CASH, as = {cash = 2.5, card_end = 0,
    phone = 0x4004000000000000 <error: Cannot access memory at address 0x4004000000000000>}}
(gdb) print *p
$2 = {kind = PAY_CARD, as = {cash = 2.5000000000002141, card_end = 4821,
    phone = 0x40040000000012d5 <error: Cannot access memory at address 0x40040000000012d5>}}
(gdb) print *p
$3 = {kind = PAY_WALLET, as = {cash = 9.2711410816661211e-310,
    card_end = -1431563616, phone = 0xaaaaaaac12a0 "987654321"}}
```

- Muestra **las tres variantes siempre**. Solo una tiene sentido, y la etiqueta dice cuál.
- Los números de las variantes inactivas dependen del entorno.

¿Y si leen `v.as.number` y hay una cadena?

Se lee el `double`

errors/wrong_variant.c

```c
v.kind = VALUE_TEXT;
v.as.text = "Callao";
printf("v.as.number = %g\n", v.as.number)
    ;
```

depende de la corrida

```
v.as.number = 9.27115e-310
```

Se usa como dirección

errors/wrong_text.c

```c
v.kind = VALUE_NUMBER;
v.as.number = 2.5;
printf("v.as.text = %s\n", v.as.text);
```

ASan; se recortan las direcciones

```
ERROR: AddressSanitizer: SEGV on unknown address
    #4 ... in main errors/wrong_text.c:19
```

Ni `gcc`, ni ASan, ni UBSan dicen nada.

Con precisión

- Leer otra variante **reinterpreta los bytes**. No es comportamiento indefinido por sí solo.
- Pero es un bug invisible: ninguna herramienta lo ve.
- Usar esos bytes como lo que no son, por ejemplo como dirección, sí es comportamiento indefinido.

> **Idea clave**
>
> La etiqueta es una promesa que ustedes deben cumplir: C no la revisa.

¿Por qué `Payment` y `Value` no son opacos?

Nada impide escribir `p.kind = PAY_WALLET;` sobre un pago en efectivo. Un tipo opaco cerraría esa puerta.

	Opaco	**Visible**
¿Quién garantiza que está bien formado?	El módulo	Quien lo usa
¿Se cambia la representación sin tocar a quien lo usa?	Sí	No
¿Se agrega una operación sin tocar el módulo?	No	Sí, otro `switch`

SECCIÓN 05

Una variante que posee memoria

05



El pago es owner de su cadena

payment.c

```c
Payment payment_wallet(const char *phone) {
    char *copy = malloc(strlen(phone) + 1);
    if (copy == NULL) {
        return payment_cash(0.0);
    }
    strcpy(copy, phone);
```

Guarda una **copia**, en memoria nueva.

payment.c

```c
void payment_free(Payment *p) {
    if (p->kind == PAY_WALLET) {
        free(p->as.phone);
    }
    *p = payment_cash(0.0);
}
```

Mira la etiqueta, y deja un pago válido sin memoria.

Error común

Liberar sin mirar la etiqueta: `free` recibiría los bytes de un `double`.

Asignar copia la dirección, no la cadena

```
Payment first = payment_wallet("987654321");
Payment second = first;
payment_free(&first);
payment_free(&second);
```



second sigue siendo PAY_WALLET, y su dirección es la misma: ASan informa un double-free.

El informe de ASan cuenta la historia

ASan; se recortan las direcciones

```
ERROR: AddressSanitizer: attempting double-free on 0x502000000010 in thread T0:
    #2 ... in main errors/shared_owner.c:11
freed by thread T0 here:
    #2 ... in main errors/shared_owner.c:10
```

Se reservó en la línea 5, se liberó en la 10, y se intentó liberar otra vez en la 11. Mientras nadie libere, compartir no se nota.

Hay que elegir, y escribirlo

Mover

Después de `second = first`, el owner es `second`. `first` ya no se usa ni se libera.

Es lo que hace el laboratorio: una celda recibe lo que devolvió `value_parse`.

Copiar de verdad

Una función aparte reserva memoria nueva para la cadena. Nadie comparte nada.

Cuesta memoria y un `malloc` más.

Idea clave

Un `struct` con un puntero que posee memoria no se asigna sin pensar. La regla va escrita en el encabezado.

SECCIÓN 06

Conversiones

06



¿Qué escribe?

errors/minus_one.c

```c
if (-1 < 1u) {
    puts("menor");
} else {
    puts("mayor");
}
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/minus_one.c -o build/minus_one
errors/minus_one.c:5:12: warning: comparison of integer expressions of different signedness: 'int' and 'unsigned int' [-Wsign-compare]
$ build/minus_one
mayor
```

- Los dos operandos deben tener el mismo tipo, y la regla elige `unsigned int`.
- Convertir suma 2^{32}: la comparación real es `4294967295 < 1`, y es falsa.

Las reglas

Si los operandos son	Se convierten a	Ejemplo
Algún `double`	`double`	`7 / 2.0` es 3.5
Más chicos que `int`	`int`, siempre	`a + b` con `unsigned char`
`int` y `unsigned int`	`unsigned int`	`-1 < 1u`
`int` y `size_t`	`size_t`	`position < strlen(word)`

En una asignación, un parámetro o un `return`: se convierte al tipo de destino, aunque no quepa.

Ya lo vieron

Introducción a la Computación: `size_t`, `strlen(s) - 1` y `-Wsign-compare`. Hoy les ponemos reglas.

Con signo y sin signo, en la vida real

errors/size_compare.c

```c
const char *word = "hola";
int position = -1;

if (position < strlen(word)) {
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/size_compare.c -o build/size_compare
$ build/size_compare
la posición está después del final
```

`gcc` avisa con `-Wsign-compare`. Se comparó con 18446744073709551615 (`size_t` de 8 bytes en este entorno). Error común: callar la advertencia sin entenderla. Un cast dice que ustedes revisaron que el valor cabe.

Promoción de enteros

promotion.c

```c
unsigned char a = 200;
unsigned char b = 100;

int sum = a + b;
unsigned char small_sum = a + b;
```

promotion.c

```c
unsigned char top = 255;
printf("top + 1 == 0: %d\n", top + 1 == 0);

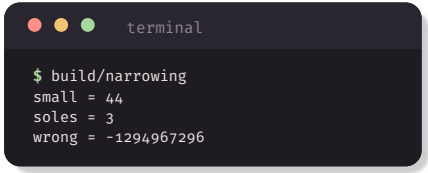
unsigned int big_top = 4294967295u;
printf("big_top + 1 == 0: %d\n", big_top +
      1 == 0);
```

terminal

```
$ build/promotion
sum = 300, small_sum = 44
sizeof(a) = 1, sizeof(a + b) = 4
top + 1 == 0: 0
big_top + 1 == 0: 1
```

- `a + b` se hace en `int`: 300.
- Guardarlo en 8 bits reduce módulo 256: 44.
- `top + 1` es 256: no dio la vuelta.
- `big_top` no se promueve, y sí da la vuelta.

Conversiones que pierden información



`-Wall -Wextra` no dice nada.
`-Wconversion` avisa de las tres.

Conversión	Qué dice C
A sin signo, si no cabe	Definida: módulo 2^n
`double` a `int`, si cabe	Definida: descarta decimales
A con signo, si no cabe	La decide el compilador
`double` a `int`, si no cabe	Comportamiento indefinido

Regla práctica

`int` para contar e indexar. No mezclen signos. `unsigned` para bits y bytes, no para decir “no puede ser negativo”.

SECCIÓN 07

Comportamiento indefinido

07

Tres grupos

Grupo	Significa	Ejemplo
Definido	El estándar dice qué pasa	`unsigned char` que da la vuelta
Por la implementación	Cada compilador elige y documenta	`int wrong = large;`
Indefinido	El estándar **no exige nada**	Overflow de un `int`, índice fuera de rango, local sin inicializar

Comportamiento indefinido ***(undefined behavior)***

No es “da un número raro”. El programa deja de tener un significado definido, y **el compilador puede suponer que eso nunca ocurre**.

El mismo programa, dos resultados

errors/overflow_compare.c

```c
int next = x + 1;
printf("next = %d, next > x: %d\n", next
      , next > x);

int steps = 0;
for (int i = x; i > 0 && steps < 5; i++)
     {
    steps++;
}
printf("steps = %d\n", steps);
```

`-O2` pide optimizar: un ejecutable más rápido, sin cambiar lo que un programa válido hace.
`-O0` no optimiza. Ninguna da advertencias.

Corramos los dos, con el mayor `int`

terminal

```
$ gcc -Wall -Wextra -std=c17 -O0 errors/overflow_compare.c -o build/overflow_o0
$ gcc -Wall -Wextra -std=c17 -O2 errors/overflow_compare.c -o build/overflow_o2
$ echo 2147483647 | build/overflow_o0
next = -2147483648, next > x: 0
steps = 1
$ echo 2147483647 | build/overflow_o2
next = -2147483648, next > x: 1
steps = 5
```

Lean la línea de -O2

`next` vale -2147483648, y es mayor que 2147483647.

Por qué “me funcionó” no prueba nada

El razonamiento del compilador: el overflow no ocurre en un programa válido. Entonces `x + 1 > x` siempre es cierto, y no hace falta comparar.

Puede fallar mañana porque depende de:

- **El compilador**, su versión y sus opciones. Estos resultados son de este `gcc`.
- **Lo que hay en la memoria**. Cambiar una línea lejana cambia esos bytes.
- **Los datos**. Con 5 como entrada, los dos ejecutables hacen lo mismo.

> **Idea clave**
> Una prueba que pasa solo dice que, en esa corrida y con esos datos, no se notó.

UBSan y ASan los encuentran... si esa línea corre

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/index_out.c -o build/index_out
$ echo 4 | build/index_out
errors/index_out.c:10:42: runtime error: index 4 out of bounds for type 'int [4]'
```

UBSan, en la salida de errores

```
errors/overflow_compare.c:10:9: runtime error: signed integer overflow: 2147483647 + 1 cannot be represented in type 'int'
```

Con una entrada válida, los dos callan: la prueba tiene que pasar por ese camino.

La local sin inicializar se les escapa

errors/uninitialized.c

```c
int discount(int age) {
    int percent;
    if (age < 18) {
        percent = 50;
    } else if (age >= 65) {
        percent = 30;
    }
    return percent;
}
```

el valor depende del entorno

```
$ echo 30 | build/uninitialized
descuento: 0%
$ echo 30 | build/uninitialized_gdb
descuento: 65535%
```

La primera tiene ASan y UBSan. La segunda, `-g -O0`. `0%` parece correcto, y las pruebas pasarían. Los sanitizers no dicen nada.

valgrind; se recorta el prefijo

```
Use of uninitialised value of size 8
   by 0x10886B: main (uninitialized.c:19)
 Uninitialised value was created by a stack allocation
   at 0x1087E8: discount (uninitialized.c:4)
```

Ninguna herramienta lo ve todo

Error	`-Wall -Wextra`	**ASan + UBSan**	**valgrind**
Overflow de un `int`	No	**Sí**, si corre	No
Índice fuera de rango, local	No, si se lee al correr	**Sí**, si corre	No ve el índice
Local sin inicializar	Solo con `-O2`	No	**Sí**, si se usa
Doble `free`	No	**Sí**	**Sí**
`double` que no cabe en `int`	No	No, hace falta otra opción	No
Leer otra variante, sin usarla como dirección	No	No	No

`test_examples.py` comprueba cada fila.

La defensa: revisar antes de operar

lab/value.c

```c
if (value > (INT_MAX - digit) / 10) {
    return 0;
}
value = value * 10 + digit;
```

- Después de multiplicar ya es tarde: el overflow ocurrió.
- `sscanf` con `%d` no sirve aquí: si el número no cabe, también es comportamiento indefinido.
- Lo van a leer en `value_parse`, en el laboratorio.

SECCIÓN 08

Laboratorio

08

Un valor con varios tipos

lab/value.h

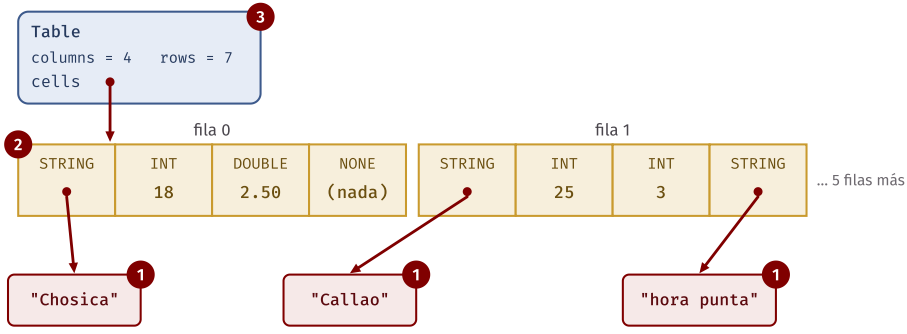
```c
typedef enum { VALUE_NONE, VALUE_INT,
     VALUE_DOUBLE, VALUE_STRING } ValueKind
     ;

typedef struct {
    ValueKind kind;
    union {
        int integer;   // VALUE_INT
        double number; // VALUE_DOUBLE
        char *text;    // VALUE_STRING:
      cadena en el heap, que posee este
      Value
    } as;
} Value;
```

1. Leer `value.h`, `value.c` y `table.h`
2. Escribir el resto de `Value`, con pruebas
3. Escribir `add_row`, `table_free` y `table_summary`

Una sola sesión: la otra es la Práctica 2.

La tabla: quién posee qué



table_free libera en este orden: 1, las cadenas; 2, el bloque de celdas; 3, la tabla.

Resumen

- `typedef` documenta. `enum` agrega nombres, `-Wswitch` y el aviso al mezclar, pero por dentro es un entero.
- Una `union` tiene espacio para un campo, y no recuerda cuál.
- Una **unión etiquetada** se crea con constructores y se lee con `switch`, sin `default`.
- Asignar un `struct` con un puntero deja dos owners: se mueve o se copia, y se escribe en el encabezado.
- `-1 < 1u` es falso. No mezclen signos.
- Con **comportamiento indefinido**, el compilador supone que no ocurre: “me funcionó” no prueba nada.
- ASan y UBSan solo ven lo que se ejecuta, y no ven una local sin inicializar. Para eso, valgrind.

Para la próxima clase

- Terminen el laboratorio: `make test-lab`, sin informes de ASan.
- Resuelvan los ejercicios de los apuntes, sobre todo el 3, el 4 y el 9.
- Respondan las preguntas de repaso de la guía de la semana 7.
- La semana 8 es el examen parcial, sobre las semanas 1 a 7.

Antes de escribir un tipo

¿Qué valores no deberían existir? ¿Quién es owner de qué?