Principios de Programación

Estructuras (`struct`)

Tipos compuestos en C para reunir datos relacionados de una misma entidad: definición, declaración, acceso a campos, arreglos de estructuras y uso básico con funciones.

Recordando la semana anterior

Lo ya trabajado

- Variables simples para guardar un dato
- Arreglos para reunir muchos datos del mismo tipo
- Cadenas como arreglos de `char`
- Funciones para separar tareas

Límite que aparece ahora

- Un mismo alumno puede necesitar nombre, edad y promedio
- Esos datos pertenecen a la misma entidad, pero no son del mismo tipo
- Hace falta agruparlos sin repartirlos en variables separadas

Definición

Una estructura es un tipo de dato compuesto que agrupa varios campos bajo un mismo nombre. Esos campos pueden ser de tipos distintos.

Ventaja

- Reúne datos que pertenecen a la misma entidad
- Cada campo puede tener un tipo distinto
- Permite trabajar con la entidad completa bajo un solo nombre

Definición de `struct` en C

FORMA GENERAL

```c
struct Nombre {
    tipo campo1;
    tipo campo2;
    tipo campo3;
};
```

Lectura

- `struct Alumno` nombra el nuevo tipo
- Cada línea interna declara un campo
- La llave final se cierra con `;`

EJEMPLO

```c
struct Alumno {
    char nombre[20];
    int edad;
    float promedio;
};
```

Observación

Definir una estructura no crea todavía una variable concreta. Solo describe cómo será ese tipo de dato.

Declarar variables de estructura

DEFINICIÓN DEL TIPO

```
struct Alumno {
    char nombre[20];
    int edad;
    float promedio;
};
```

VARIABLES CONCRETAS

```
struct Alumno a1;
struct Alumno a2;
struct Alumno grupo[30];
```

Diferencia

La definición describe qué campos tiene el tipo. La declaración crea variables de ese tipo, igual que antes se declaraban `int x;` o `float nota;`.

Acceso a campos con `.`

LECTURA Y ESCRITURA

```c
struct Alumno a1;

a1.edad = 19;
a1.promedio = 8.7;

printf("%d\n", a1.edad);
printf("%.1f\n", a1.promedio);
```

Regla

Para acceder a un campo se usa:

`variable.campo`

Lectura correcta

`a1.edad` significa: tomar la variable `a1` y luego su campo `edad`.

DEFINICIÓN

```
struct Alumno {
    char nombre[20];
    int edad;
    float promedio;
};
```

Lectura

Un campo también puede ser un arreglo. En este caso, `nombre` es una cadena y necesita espacio suficiente para las letras y para `'\0'`.

Relación con la clase anterior

Dentro de una estructura siguen valiendo las mismas reglas ya vistas para arreglos y cadenas.

ASIGNACIONES CAMPO A CAMPO

```
struct Alumno a1;

strcpy(a1.nombre, "Ana");
a1.edad = 19;
a1.promedio = 8.7;
```

INICIALIZACIÓN CON LLAVES

```
struct Alumno a1 = {
    "Ana", 19, 8.7
};
```

Regla

En la inicialización con llaves, el orden de los valores debe coincidir con el orden de los campos dentro de la estructura.

Tipo

`struct Alumno`

Variable

`a1`

Campo

`a1.promedio`

Distinción

El tipo describe la forma del dato. La variable guarda un valor de ese tipo. El campo es una parte interna de esa variable.

Asignación entre estructuras

ASIGNACIÓN

```c
struct Alumno a1 = {"Ana", 19, 8.7};
struct Alumno a2;

a2 = a1;

printf("%s\n", a2.nombre);
printf("%d\n", a2.edad);
printf("%.1f\n", a2.promedio);
```

Lectura correcta

La estructura completa se copia de una variable a otra porque ambas tienen el mismo tipo.

SALIDA

```
Ana
19
8.7
```

DECLARACIÓN

```
struct Alumno grupo[3];
```

ACCESO

```
grupo[0].edad = 19;
grupo[1].edad = 20;
grupo[2].edad = 18;
```

Lectura correcta

Primero se elige qué alumno del arreglo se quiere usar y luego qué campo de ese alumno se necesita.

Patrón

`grupo[i].promedio` combina dos ideas ya conocidas: índice de arreglo + acceso a campo.

MOSTRAR LOS DATOS DEL GRUPO

```c
for (int i = 0; i < 3; i++) {
    printf("%s - %d - %.1f\n",
           grupo[i].nombre,
           grupo[i].edad,
           grupo[i].promedio);
}
```

1. Índice

`i` selecciona al alumno actual

2. Campo

Se accede a `nombre`, `edad` y `promedio`

3. Repetición

El mismo patrón se aplica a todo el arreglo

UN ALUMNO

```c
struct Alumno a1;

printf("Nombre: ");
scanf("%19s", a1.nombre);

printf("Edad: ");
scanf("%d", &a1.edad);

printf("Promedio: ");
scanf("%f", &a1.promedio);
```

Nombre

Para una cadena dentro de la estructura se usa el campo `a1.nombre`, igual que con cualquier arreglo de `char`.

Edad y promedio

Para campos simples se usa `&a1.edad` y `&a1.promedio` porque `scanf` necesita la dirección de esas variables.

Función que recibe una estructura

PARÁMETRO DE TIPO `struct Alumno`

```c
void mostrar_alumno(struct Alumno a) {
    printf("%s\n", a.nombre);
    printf("%d\n", a.edad);
    printf("%.1f\n", a.promedio);
}
```

Lectura correcta

La función recibe un alumno completo como parámetro. Dentro de la función, ese parámetro se usa con el mismo operador `.`.

Relación con la semana anterior

Igual que con otros parámetros en C, el argumento se pasa por valor. En esta etapa conviene pensarlo como una copia del alumno recibido.

Función que devuelve una estructura

RETORNO

```
struct Alumno crear_alumno() {
    struct Alumno a = {"Ana", 19, 8.7};
    return a;
}
```

Lectura

El valor devuelto por la función es un alumno completo, no un único campo.

USO EN `main()`

```
struct Alumno a1;
a1 = crear_alumno();
```

Paso por valor con estructuras

EJEMPLO

```c
void cambiar_edad(struct Alumno a) {
    a.edad = 30;
}

int main() {
    struct Alumno a1 = {"Ana", 19, 8.7};
    cambiar_edad(a1);
    printf("%d\n", a1.edad);
    return 0;
}
```

Lectura

La modificación se hace sobre el parámetro local `a`, no sobre la variable `a1` de `main()`. En esta clase, si una función actualiza una estructura recibida por valor, debe devolver la copia modificada.

SALIDA

```
19
```

Error frecuente: usar un campo sin su variable

Campo fuera de contexto

CÓDIGO CON ERROR

```
1 struct Alumno a1;
2 edad = 19;
```

CORRECCIÓN

El campo debe usarse a través de una variable de estructura.

```
1 a1.edad = 19;
```

Confusión entre arreglo y campo

CÓDIGO CON ERROR

```
1 struct Alumno grupo[3];
2 printf("%d\n", grupo.edad);
```

CORRECCIÓN

Primero hay que elegir una posición del arreglo.

```
1 printf("%d\n", grupo[0].edad);
```

Opcional si aparece: `typedef`

SIN `typedef`

```c
struct Alumno a1;
struct Alumno grupo[3];
```

CON `typedef`

```c
typedef struct {
    char nombre[20];
    int edad;
    float promedio;
} Alumno;

Alumno a1;
Alumno grupo[3];
```

Criterio del curso

`typedef` puede hacer más cómoda la escritura, pero no cambia la idea central de la clase. Primero importa entender qué es una estructura y cómo se usan sus campos.

Estructuras anidadas

Una estructura puede tener campos que también sean estructuras. Esto permite modelar entidades con partes internas.

Memoria y padding

El tamaño real de una estructura puede ser mayor que la suma directa de sus campos, porque el compilador puede insertar espacios internos.

Criterio

Estos temas ayudan a entender C con más profundidad, pero no son necesarios para resolver el práctico principal de estructuras.

EJEMPLO

```c
struct Fecha {
    int dia;
    int mes;
    int anio;
};

struct Alumno {
    char nombre[20];
    struct Fecha nacimiento;
    float promedio;
};
```

Acceso

Para llegar a un campo interno se encadenan accesos:

`a1.nacimiento.anio`

Lectura

Primero se elige el alumno, luego su fecha de nacimiento, y finalmente el campo `anio`.

Idea

Aunque `char` ocupa 1 byte y `int` suele ocupar 4 bytes, la estructura puede ocupar más de 5 bytes.

TAMAÑO ESPERADO VS. REAL

```c
struct Ejemplo {
    char letra;
    int numero;
};

printf("%zu\n", sizeof(struct Ejemplo));
```

Por qué pasa

El compilador puede agregar bytes de relleno para que algunos campos queden ubicados en direcciones más convenientes para la máquina.

Regla práctica

No asumir el tamaño exacto de una estructura: medirlo con `sizeof`.

1. **Máquina**

La arquitectura trabaja mejor, o a veces solo puede trabajar, con ciertos tipos ubicados en direcciones alineadas.

2. **ABI**

La plataforma define reglas concretas de tamaño y alineación para que programas, librerías y compiladores sean compatibles.

3. **Compilador**

El compilador arma el layout de la estructura siguiendo esas reglas y agrega padding cuando hace falta.

Qué garantiza C

Los campos aparecen en el orden declarado y no hay padding antes del primer campo. Pero puede haber bytes de relleno entre campos y al final de la estructura.

Consecuencia

El padding no lo escribe el programador y no es un campo accesible. Sale de las reglas de representación elegidas por la implementación para esa máquina.

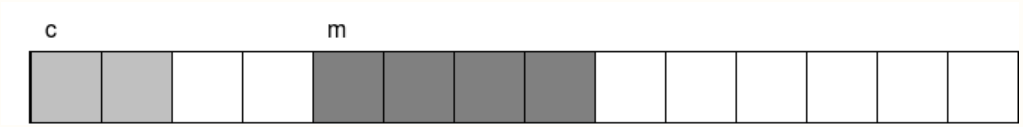

Imagen: Thedsadude, Wikimedia Commons, CC BY 3.0.

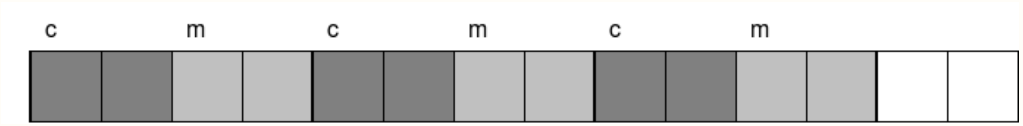

Imagen: Thedsadude, Wikimedia Commons, CC BY 3.0.

Lectura

Los campos se guardan en orden, pero pueden aparecer huecos entre ellos. Esos huecos no son campos del programa: son bytes de relleno.

Arreglos de estructuras

El tamaño total también se ajusta para que cada elemento de un arreglo empiece bien alineado.

Idea clave

`sizeof(struct X)` cuenta campos visibles y padding invisible.

REGLA DE OFFSET

```
offset actual: dónde empieza el próximo campo
align: alineación requerida por ese campo

padding =
  (align - (offset % align)) % align

offset alineado = offset + padding
```

Traducción

Si el próximo campo necesita alineación de `4`, debe empezar en un offset múltiplo de `4`: `0`, `4`, `8`, `12`, etc.

Caso sin padding

Si `offset % align == 0`, el campo ya está alineado y se agregan `0` bytes.

Al final

El tamaño total de la estructura se redondea a un múltiplo de la mayor alineación de sus campos.

ESTRUCTURA

```c
struct Sensor {
    char id;       // 1 byte
    int lectura;   // 4 bytes
    char estado;   // 1 byte
};
```

Suposición

En una máquina típica: `char` alinea a `1`, `int` alinea a `4`.

OFFSETS

```
id:
  offset 0, ocupa 1 byte
  próximo offset = 1

lectura:
  offset 1 no es múltiplo de 4
  padding = 3
  lectura empieza en offset 4
  próximo offset = 8

estado:
  offset 8, ocupa 1 byte
  próximo offset = 9

padding final:
  9 no es múltiplo de 4
  padding = 3

sizeof(struct Sensor) == 12
```

MÁS PADDING

```
struct SensorA {
    char id;
    int lectura;
    char estado;
};

// 1 + 3 + 4 + 1 + 3 = 12
```

MENOS PADDING

```
struct SensorB {
    int lectura;
    char id;
    char estado;
};

// 4 + 1 + 1 + 2 = 8
```

Criterio

C mantiene el orden de declaración de los campos. Si se necesita ahorrar memoria, conviene evaluar el orden de campos grandes a chicos, midiendo siempre con `sizeof`.

Ejercicio 1

Definir `struct Alumno`, declarar variables, inicializarlas con llaves, modificar campos con `.` y mostrar sus datos.

Ejercicio 2

Escribir una función que muestre un alumno y otra función que cree y devuelva un alumno cargado por teclado.

Ejercicio 3

Cargar un grupo de alumnos, recorrerlo con `for`, calcular el promedio general y encontrar el alumno con mayor promedio.