Video[6] (1)

Introducción a C – Structs

Conceptos Principales

1. a
2. b
3. c

Notas

1. **Qué es un struct**:
    - Un `struct` es un **bloque de memoria** que agrupa distintos tipos de datos bajo un mismo nombre.
    - Se define **primero como plantilla** (tipo) y luego se pide memoria para instancias concretas.
2. **Definición de un struct**:

```
struct Isla {
    float radio;
    int poblacion;
    int cantidadRanas;
};
```

    - Esto **no asigna memoria**, solo define la plantilla.
3. **Creación de una instancia dinámica en heap**:

```
struct Isla *martinica;
martinica = (struct Isla*) calloc(1, sizeof(struct Isla));
```

    - `calloc` reserva memoria limpia en **heap**.
    - `martinica` es un puntero que apunta a la dirección de memoria asignada.
    - La dirección del bloque se puede guardar en la **pila** usando el nombre de la variable (`martinica`).
4. **Acceso y modificación de campos**:
    - Para acceder a los campos de un struct dinámico (puntero), se usa el **operador flecha (`->`)**:

```
martinica->radio = 3.183;
martinica->poblacion = 15;
```

- Si el struct no es puntero (variable automática en pila), se usa el **punto (`.`)**:

```
struct Isla isla_local;
isla_local.radio = 2.5;
```

5. **Impresión de campos**:

```
printf("%d\n", martinica->cantidadRanas);
```

- Como `calloc` inicializa todo en cero, campos no asignados mostrarán `0`.

6. **Importancia de heap vs stack**:

- Crear structs en heap evita que los datos se pierdan al salir de la función.
- Si se crean en stack, se perderán al terminar la función, como pasa con arrays automáticos.

Resumen práctico:

- `struct` → define la **plantilla** de un bloque de datos.
- `calloc` → reserva memoria limpia en heap.
- `>` → acceder a campos de un struct **puntero**.
- `.` → acceder a campos de un struct **directo en stack**.
- Siempre preferir heap si se quiere que los datos persistan más allá del ámbito de la función.