Principios de Programación

Funciones

Subprogramas en C para separar tareas, reutilizar lógica y organizar mejor un programa: definición, llamada, parámetros, retorno, alcance y prototipos.

Recordando la semana anterior

Lo ya trabajado

- Arreglos y matrices para organizar varios datos del mismo tipo
- Cadenas como arreglos de `char` con terminador `'\0'`
- Bucles para recorrer y procesar esas estructuras

Límite que aparece ahora

- Un mismo procesamiento puede aparecer en varios lugares del programa
- Copiar y pegar esa lógica dispersa reglas y dificulta corregirlas
- Hace falta encapsular tareas para reutilizarlas con más claridad

Pregunta guía

Si un programa debe calcular promedios, validar claves o mostrar resultados en varios puntos, ¿conviene reescribir cada bloque o separar esa tarea en una función?

Motivación: evitar código repetido

SIN FUNCIÓN

```
float prom_lia = (lia1 + lia2) / 2.0;
float prom_luis = (luis1 + luis2) / 2.0;
float prom_eva = (eva1 + eva2) / 2.0;
```

USANDO `promedio(...)`

```
float prom_lia = promedio(lia1, lia2);
float prom_luis = promedio(luis1, luis2);
float prom_eva = promedio(eva1, eva2);
```

Qué problema trae

- La fórmula del promedio queda repetida
- Si cambia el criterio, hay que corregir varias líneas
- La intención del programa se mezcla con el detalle del cálculo

Qué aporta una función

- `promedio(n1, n2)` concentra el cálculo en un solo lugar
- El nombre describe con claridad la tarea
- Después puede llamarse tantas veces como haga falta

Definición

Una función es un bloque de código con una tarea específica. Puede recibir datos, procesarlos y devolver un resultado.

Analogía

En matemáticas, `f(x)` toma una entrada y produce una salida. En programación ocurre lo mismo, pero la función además puede ejecutar varias instrucciones internas.

Qué aporta

- Reutilización
- Modularidad
- Lectura más clara del programa

Modelo de caja negra

Entrada

Parámetros con los datos que la función necesita.

Procesamiento

Instrucciones internas que resuelven la tarea.

Salida

Valor de retorno, si la función produce uno.

Ejemplo

`calcularArea(base, altura)` recibe dos valores, multiplica esos datos y devuelve el área. Para usarla importa qué recibe y qué entrega; la implementación se consulta cuando hace falta modificarla.

Estructura de una función en C

SINTAXIS BÁSICA

```c
tipo nombre_funcion(parametros) {
    // instrucciones
    return valor;
}
```

Lectura

- El tipo inicial indica qué devuelve la función
- El nombre describe la tarea que realiza
- Entre paréntesis van los parámetros, si la función necesita datos de entrada
- El cuerpo se escribe entre llaves

Regla

Si la función promete devolver un dato, el `return` debe entregar un valor compatible con ese tipo.

Definir y llamar una función

DEFINICIÓN

```c
int sumar(int a, int b) {
    int resultado = a + b;
    return resultado;
}
```

LLAMADA / INVOCACIÓN DESDE `main()`

```c
int total;
total = sumar(x, y);
printf("%d\n", total);
```

Diferencia

Definir una función es escribir su comportamiento una sola vez. Llamarla es usar esa definición con datos concretos desde otra parte del programa.

```c
int sumar(int a, int b) {
    int resultado = a + b;
    return resultado;
}

int main() {
    int x = 5;
    int y = 3;
    int total = sumar(x, y);
    printf("%d\n", total);
    return 0;
}
```

TRAZA

```
Paso 11
    salida mostrada
--> return 0
```

Lectura

En cada avance, el control queda detenido en una línea distinta. Cuando aparece una llamada, `main()` espera, la función trabaja y luego el programa regresa al punto siguiente.

CÓDIGO

```c
int calcular_doble(int numero) {
    int resultado = numero * 2;
    return resultado;
}

int main() {
    int valor = 5;
    int doble = calcular_doble(valor);
    printf("%d\n", doble);
    return 0;
}
```

SECUENCIA

```
main: valor = 5
main: llamada a calcular_doble(5)
  calcular_doble: numero = 5
  calcular_doble: resultado = 10
  calcular_doble: return 10
main: doble = 10
main: imprime 10
```

Lectura

La ejecución no ocurre toda a la vez. `main()` se detiene en la llamada, la función trabaja con sus variables locales y luego devuelve el control con el valor calculado.

EJEMPLO

```c
int sumar(int a, int b) {
    return a + b;
}

int main() {
    int x = 5;
    int y = 3;
    int total = sumar(x, y);
    return 0;
}
```

En la definición

`a` y `b` son **parámetros formales**: nombres que representan la información que llegará a la función.

En la llamada

`x` e `y` son **parámetros actuales** o **argumentos**: valores concretos que se copian a `a` y `b`.

LO QUE OCURRE

```c
void modificar(int numero) {
    numero = 100;
}

int main() {
    int x = 5;
    modificar(x);
    printf("%d\n", x);
    return 0;
}
```

Lectura correcta

- `numero` recibe una copia de `x`
- La modificación ocurre dentro de la función
- `x` sigue valiendo `5` al volver a `main()`

SALIDA

```
5
```

Consecuencia

Para cambiar el valor original no alcanza con modificar el parámetro recibido. En este curso, por ahora, se trabaja pensando las funciones como cajas que reciben copias y devuelven resultados.

CON RETORNO

```c
int mayor(int x, int y) {
    if (x > y) {
        return x;
    }
    return y;
}
```

SIN RETORNO

```c
void imprimir_mensaje() {
    printf("Hola\n");
}

void cortar_si_hace_falta() {
    if (condicion) {
        return;
    }
    printf("Sigue\n");
}
```

Si el tipo es `void` puede no llevar `return` o usar `return;`

Ejemplo completo: devolver el mayor

PROGRAMA COMPLETO

```c
#include <stdio.h>

int mayor(int x, int y) {
    if (x > y) {
        return x;
    }
    return y;
}

int main() {
    int numero1, numero2;
    int maximo;

    printf("Ingrese dos numeros: ");
    scanf("%d %d", &numero1, &numero2);

    maximo = mayor(numero1, numero2);
    printf("El mayor es: %d\n", maximo);
    return 0;
}
```

Lectura

Si ambos números son iguales, la condición `x > y` es falsa y la función devuelve `y`. El resultado sigue siendo correcto porque ambos tienen el mismo valor.

Alcance de variables

Variables locales

- Se declaran dentro de una función
- Solo pueden usarse dentro de esa función
- Existen mientras esa función se ejecuta

Variables globales

- Se declaran fuera de todas las funciones
- Son visibles desde varias partes del programa
- Exigen más cuidado porque aumentan el acoplamiento

Regla base

Una variable solo puede usarse donde su nombre es visible. Con funciones, eso obliga a distinguir con claridad qué información entra como parámetro y cuál queda encerrada dentro del cuerpo.

EJEMPLO

```c
int sumar(int a, int b) {
    int resultado = a + b;
    return resultado;
}

int main() {
    int x = 5;
    int y = 3;
    int total = sumar(x, y);
    return 0;
}
```

Qué variables hay

- En `sumar()`: `a`, `b`, `resultado`
- En `main()`: `x`, `y`, `total`

Consecuencia

`resultado` no existe en `main()` y `x` no existe en `sumar()`. Tener el mismo programa abierto no significa compartir todas las variables entre funciones.

EJEMPLO CON GLOBAL

```c
int contador_global = 0;

void incrementar() {
    contador_global = contador_global + 1;
}

int main() {
    incrementar();
    printf("%d\n", contador_global);
    return 0;
}
```

Por qué se ve desde `main()`

`contador_global` fue declarada fuera de cualquier función, así que ambas funciones pueden leerla y modificarla.

Criterio del curso

En esta etapa conviene preferir variables locales y parámetros. Las globales simplifican algunos ejemplos, pero vuelven más difícil rastrear cambios y dependencias.

Error de alcance

CÓDIGO CON ERROR

```c
int sumar(int a, int b) {
    int resultado = a + b;
    return resultado;
}

int main() {
    int total = sumar(5, 3);
    printf("%d\n", resultado);
}
```

MENSAJE

```
error: 'resultado' undeclared
```

Corrección

`resultado` solo existe dentro de `sumar()`.

```c
printf("%d\n", total);
```

Mismo nombre, distinto alcance

DOS VARIABLES DISTINTAS

```c
int sumar(int a, int b) {
    int resultado = a + b;
    return resultado;
}

int main() {
    int resultado = 0;
    resultado = sumar(5, 3);
    printf("%d\n", resultado);
    return 0;
}
```

Lectura correcta

Hay dos variables llamadas `resultado`, pero no compiten entre sí. Una vive dentro de `sumar()` y la otra dentro de `main()`.

Idea

El nombre puede repetirse si el alcance cambia. Aun así, conviene elegir nombres que hagan fácil seguir la lógica del programa.

Por qué hacen falta prototipos

LLAMADA ANTES DE DECLARAR

```c
int main() {
    int x = sumar(5, 3);
    return 0;
}

int sumar(int a, int b) {
    return a + b;
}
```

MENSAJE

```
error: call to undeclared function 'sumar'
```

Soluciones

- Definir `sumar()` antes de `main()`
- O declarar un **prototipo** antes de `main()`

Prototipo de función

DECLARACIÓN Y DEFINICIÓN

```c
#include <stdio.h>

int sumar(int a, int b);

int main() {
    int x = sumar(5, 3);
    printf("%d\n", x);
    return 0;
}

int sumar(int a, int b) {
    return a + b;
}
```

Qué informa el prototipo

Los **prototipos** los podemos pensar como promesas que le hacemos al compilador, diciéndole que en un «futuro» va a existir una función con cierta forma. Dicha promsa incluye:

- Nombre de la función
- Tipo de retorno
- Tipo y cantidad de parámetros

Dónde suele ir

Después de los `#include` y antes de `main()`. En programas grandes, muchas veces aparece en un archivo `.h`.

`stdio.h`: incluye *promesas* para las funciones `printf` y `scanf`.

El prototipo no incluye lógica. Solo adelanta la **firma** para que el compilador sepa cómo se llama y forma que tiene la función.

DECLARACIÓN Y DEFINICIÓN

```c
#include <stdio.h>

int sumar(int a, int b);

int main() {
    int x = sumar(5, 3);
    printf("%d\n", x);
    return 0;
}

int sumar(int a, int b) {
    return a + b;
}
```

Qué muestra la implementación

Acá ya no aparece solo la **promesa** de la función, sino su desarrollo concreto. La **implementación** incluye:

- La **misma firma** que el prototipo
- El cuerpo entre llaves
- Las instrucciones que resuelven la tarea

Relación con el prototipo

El prototipo adelanta cómo se llama la función y qué recibe. La implementación completa esa promesa mostrando qué hace realmente.

Promedio

Diseñar `float promedio(int a, int b)`:

- recibe dos enteros
- calcula `(a + b) / 2.0`
- devuelve un `float`

Paridad

Diseñar `int es_par(int n)`:

- devuelve `1` si `n` es par
- devuelve `0` si `n` es impar
- puede usarse directo en `if`

ESQUELETO

```c
float promedio(int a, int b) {
    // completar
}
```

IDEA CLAVE

```c
int es_par(int n) {
  // completar
}
```

Diseñar

`int potencia(int base, int exponente):`

- iniciar **resultado** en `1`
- repetir **exponente** veces
- multiplicar por `b*se` en cada vuelta

Organizar un archivo con:

- prototipos al inicio
- `main()` como punto de coordinación
- `sumar`, `restar` y `multiplicar` con una única responsabilidad cada una

ESQUELETO

```c
int potencia(int base, int exponente) {
  ...
}
```

PROTOTIPOS

```c
int sumar(int a, int b);
int restar(int a, int b);
int multiplicar(int a, int b);
```

Error frecuente: falta return

CÓDIGO CON ERROR

```
1 int sumar(int a, int b) {
2     int resultado = a + b;
3 }
```

MENSAJE

```
warning: control reaches end
of non-void function
```

Corrección

Una función que promete devolver un `int` debe terminar con un valor de ese tipo.

```
int sumar(int a, int b) {
    int resultado = a + b;
    return resultado;
}
```

Cantidad incorrecta de argumentos

CÓDIGO CON ERROR

```
1 int sumar(int a, int b) {
2     return a + b;
3 }
4
5 int main() {
6     int r = sumar(5);
7     return 0;
8 }
```

CORRECCIÓN

Pasar todos los parámetros requeridos:

```
1 int r = sumar(5, 3);
```

Tipo de dato inadecuado

CÓDIGO CON ERROR

```
1 int sumar(int a, int b) {
2     return a + b;
3 }
4
5 int main() {
6     float x = 5.5;
7     int r = sumar(x, 3);
8     return 0;
9 }
```

CORRECCIÓN

Revisar si la función realmente debería trabajar con enteros o con otro tipo. La firma debe coincidir con el uso esperado.

Buenas prácticas: nombres y responsabilidad

NOMBRES DÉBILES

```c
// que hace f?
int f(int x, int y);

// que tipo de calculo?
int calc(int a, int b);

int hacer_algo(int n);
```

NOMBRES DESCRIPTIVOS

```c
// que hace sumar? es obvio
int sumar(int a, int b);
float promedio(int a, int b);
int es_par(int numero);
```

Resumen

Función

Bloque reutilizable con un nombre, parámetros y, en muchos casos, un valor de retorno.

Parámetros

En C se pasan por valor: la función recibe copias de los argumentos.

Alcance

Las variables locales existen solo en la función donde fueron declaradas. Al finalizar la función, las variables locales desaparecen.

Prototipos

Permiten usar una función antes de mostrar su definición completa.