Programación I

Pizarras y Ejemplos de clase, noviembre 2025

Este documento contiene las pizarras y ejemplos de código que se utilizaron en las clases de Programación I del grupo G1M3, durante la primera mitad el mes de noviembre de 2025.

1 . Clase 5 noviembre 2025

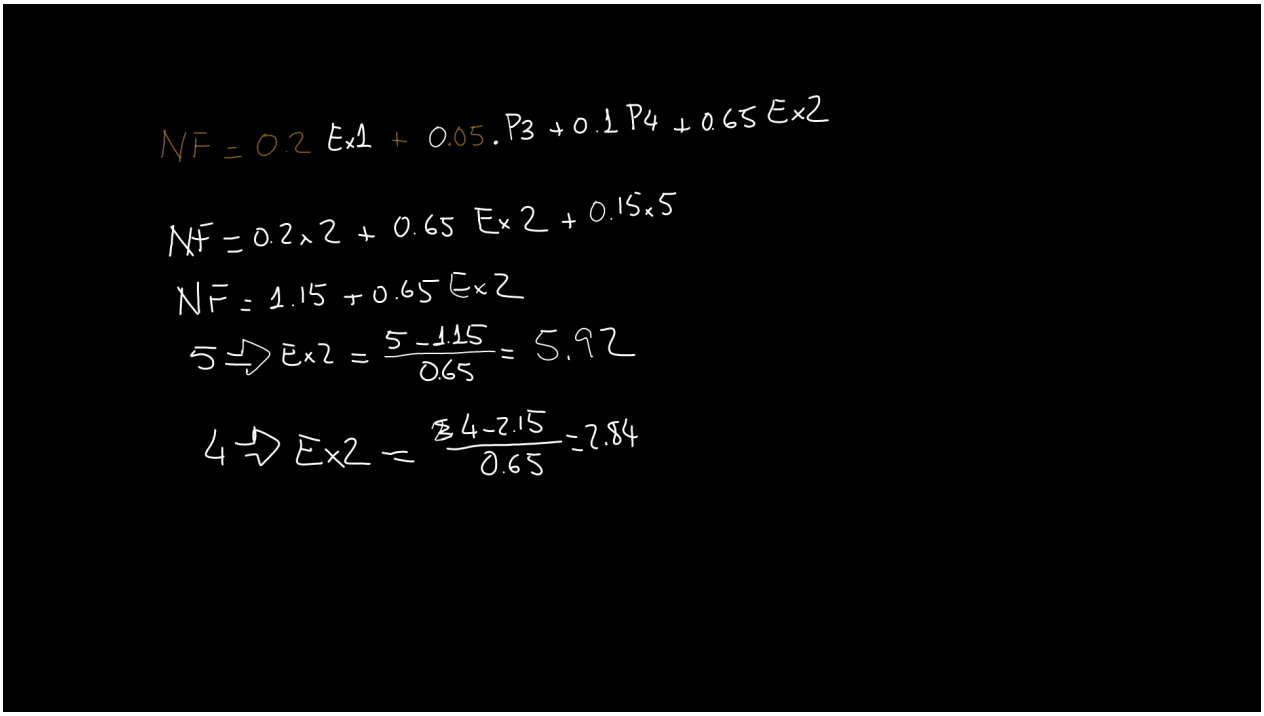

Figura 1: Fórmula para el cálculo de la nota final ponderada en base a las notas obtenidas en los diferentes exámenes de la evaluación continua.

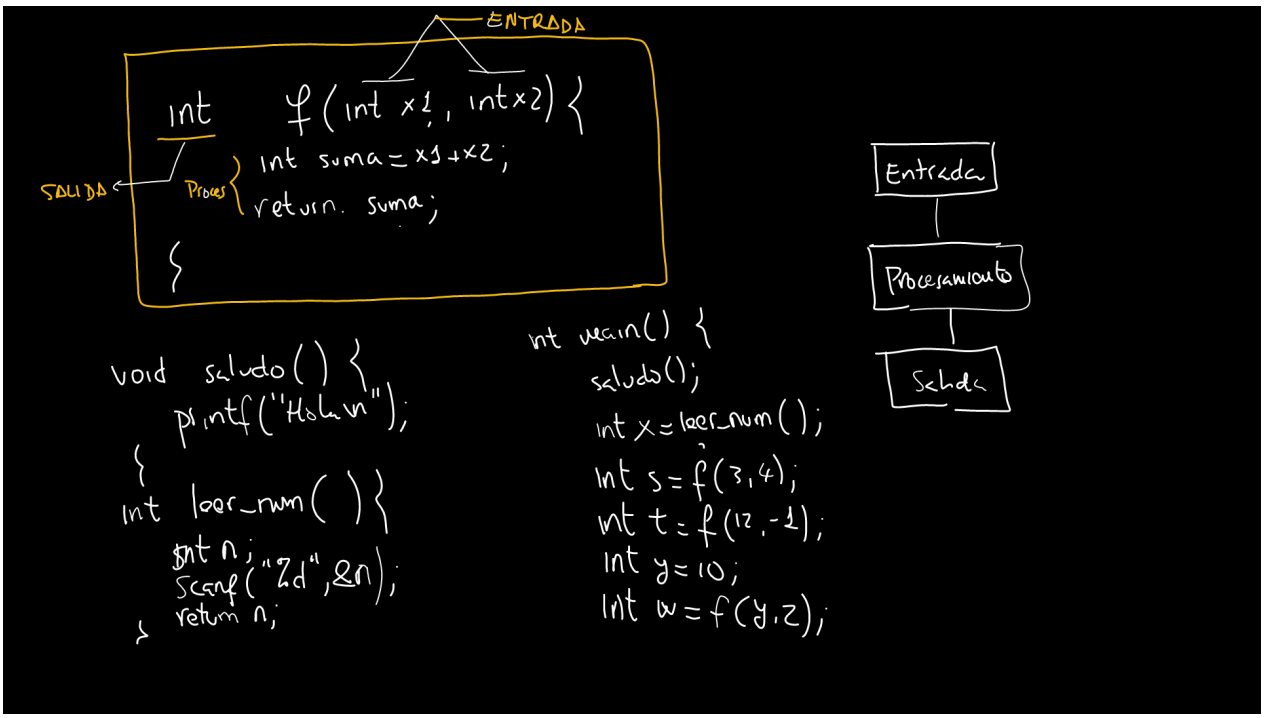

Figura 2: Las funciones siguen el esquema de cualquier programa: **entrada**, a través de los parámetros, **procesamiento**, las instrucciones contenidas en el bloque de código y **salida**, que es el valor devuelto por la función.

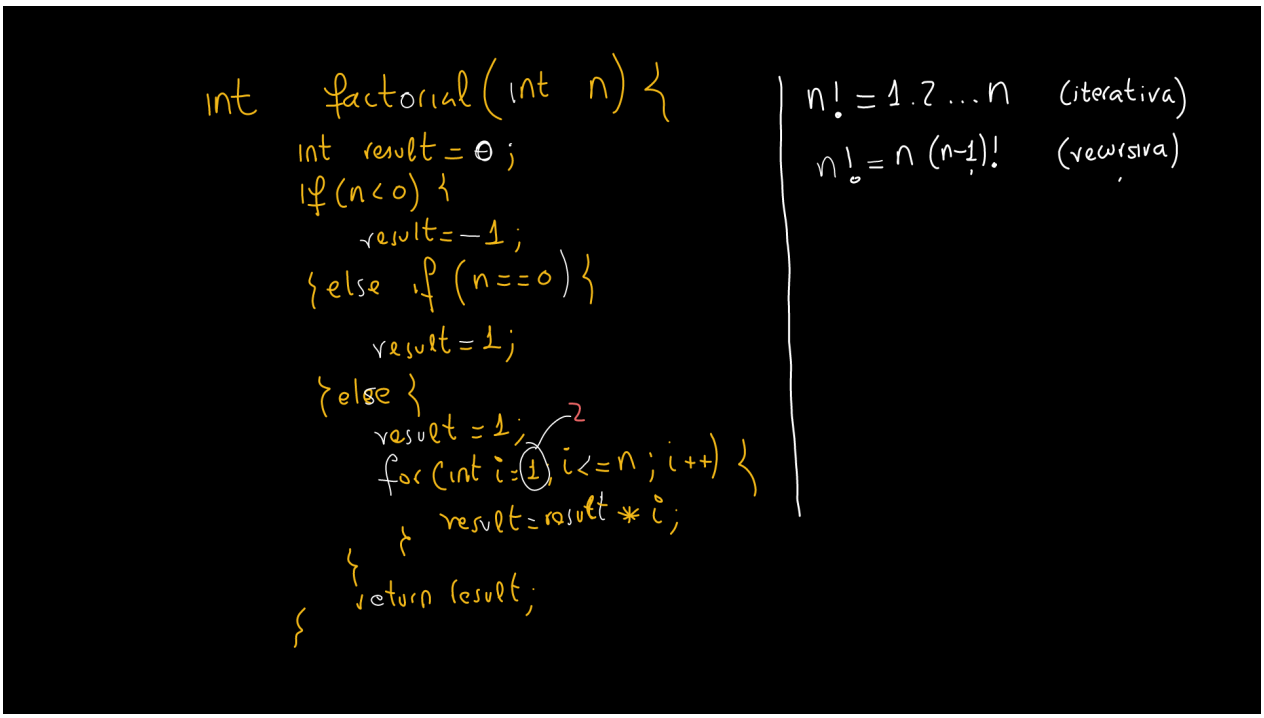

Figura 3: Función que calcula el factorial de un número utilizando un algoritmo iterativo.

Ejemplo 1 Cálculo del factorial utilizando un algoritmo iterativo

```c
#include <stdio.h>

// Prototipos
int factorial(int n);

// Programa Principal
int main() {
   int x = 5;

   printf("%d! = %d\n", x, factorial(x));

   return 0;
}

// Código de funciones
int factorial(int n) {
   int result = 1;
   if(n>0) {
      for(int i=2; i<=n; i++) {
         result = result * i;
      }
   }
   return result;
}
```

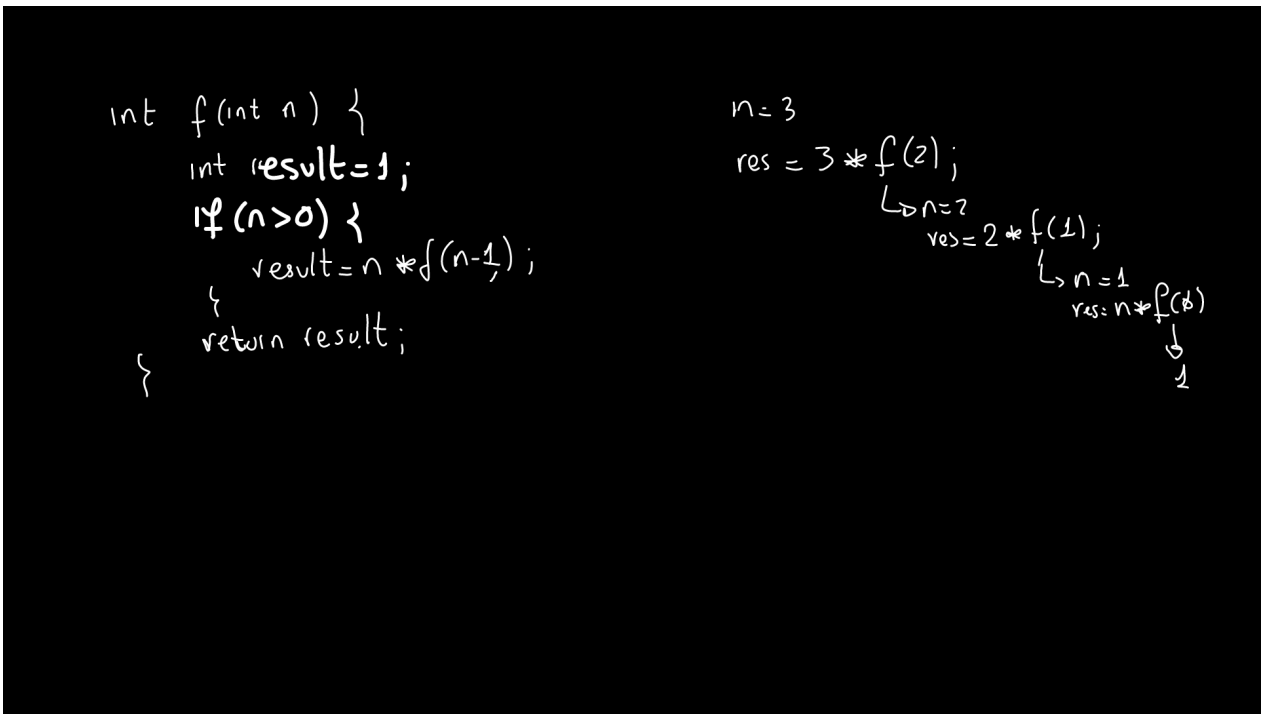

Figura 4: Función que calcula el factorial de un número utilizando un algoritmo recursivo.

Ejemplo 2 Cálculo del factorial utilizando un algoritmo recursivo

```c
#include <stdio.h>

// Prototipos
int factorial(int n);

// Programa Principal
int main() {
   int x = 3;

   printf("%d! = %d\n", x, factorial(x));

   return 0;
}

// Código de funciones
int factorial(int n) {
   int result = 1;
   if(n>0) {
      result = n * factorial(n-1);
   }
   return result;
}
```

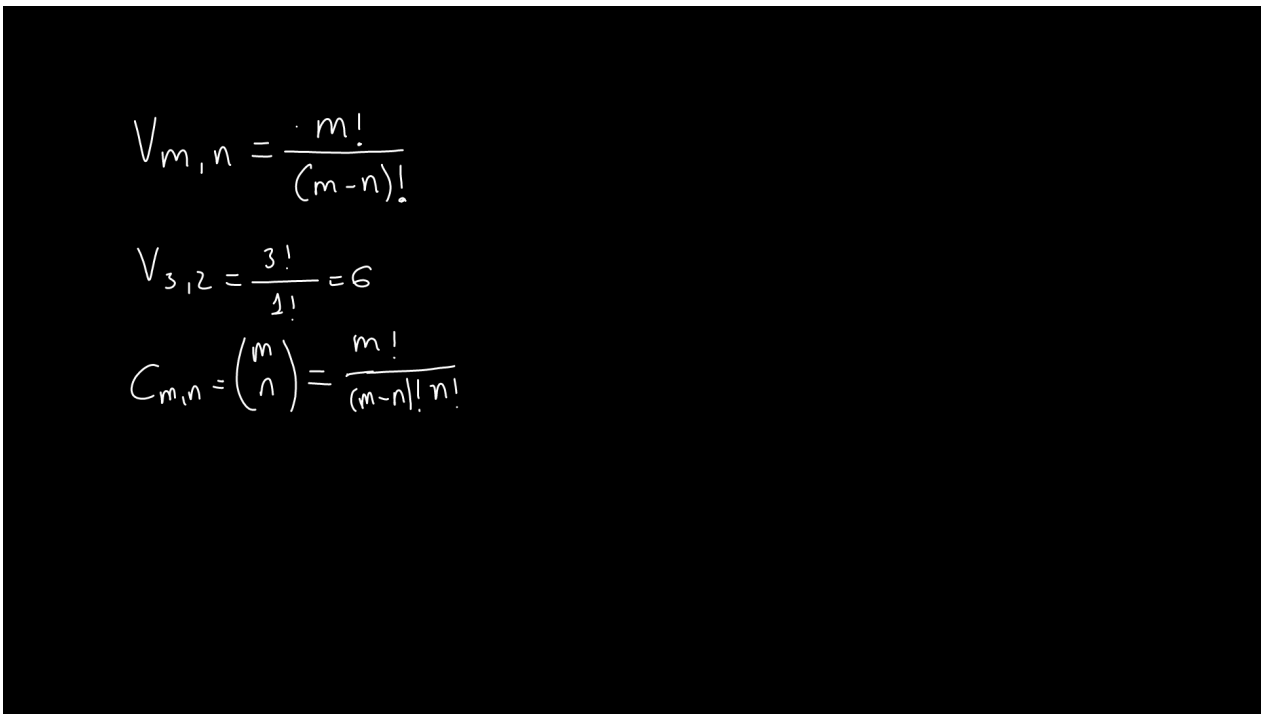

Figura 5: Fórmulas para el cálculo de las *variaciones* y las *combinaciones* de m elementos tomados de n en n.

Ejemplo 3 Cálculo de permutaciones, variaciones y combinaciones

```c
#include <stdio.h>

// Prototipos
int factorial(int n);
int variaciones(int m, int n);
int combinaciones(int m, int n);

// Programa Principal
int main() {
   int x = 0;

   printf("%d! = %d\n", x, factorial(x));
   printf("V(6,2)= %d\n", variaciones(6,2));
   printf("C(6,2)= %d\n", combinaciones(6,2));

   return 0;
}

// Código de funciones
int combinaciones(int m, int n) {
   int v = variaciones(m, n);
   int p = factorial(n);
   int result = v/p;
   return result;
}
int variaciones(int m, int n) {
   int fm = factorial(m);
   int fmn = factorial(m-n);
   int resultado = fm/fmn;
   return resultado;
}
int factorial(int n) {
   int result = 1;
   if(n>0) {
      for(int i=2; i<=n; i++) {
         result = result * i;
      }
   }
   return result;
}
```

2 . Clase 7 noviembre 2025

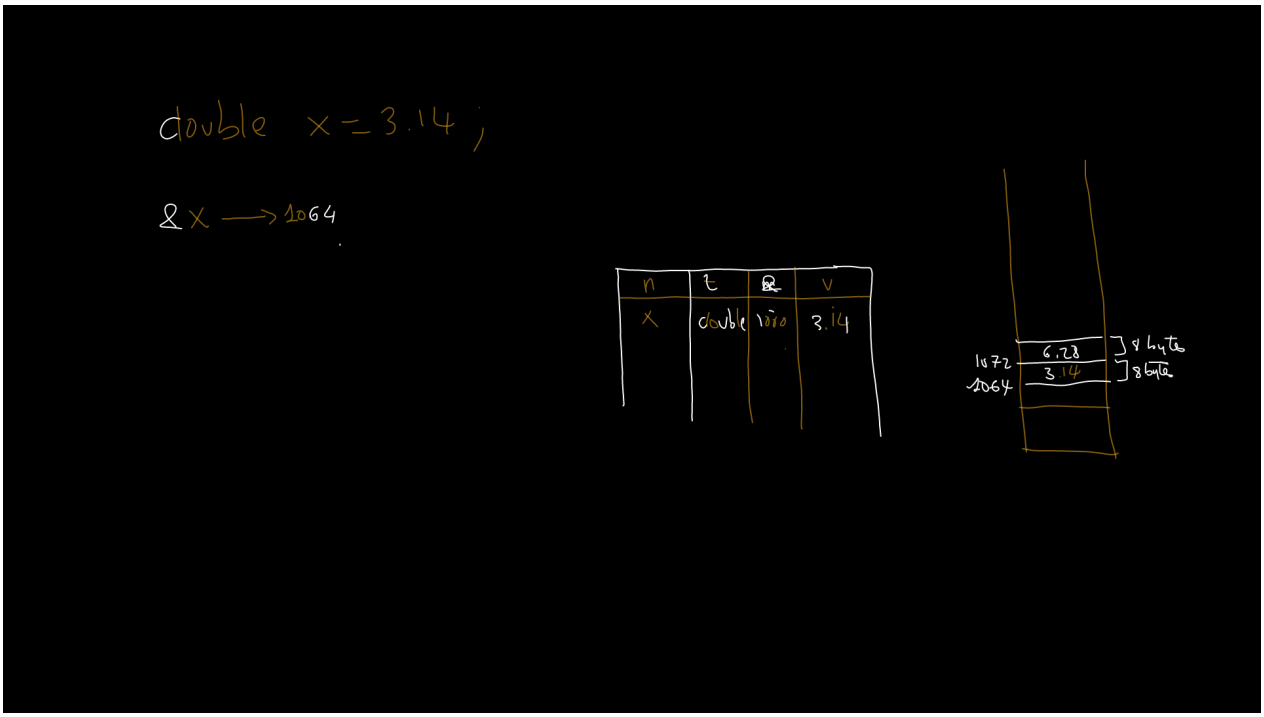

Figura 6: El operador & permite obtener la dirección de memoria de una variable. Se puede leer como *la dirección de memoria de*. Accediendo directamente a la posición de memoria donde está guardado un valor, es posible modificarlo. Es el mecanismo de los punteros.

3 . Clase 14 noviembre 2025

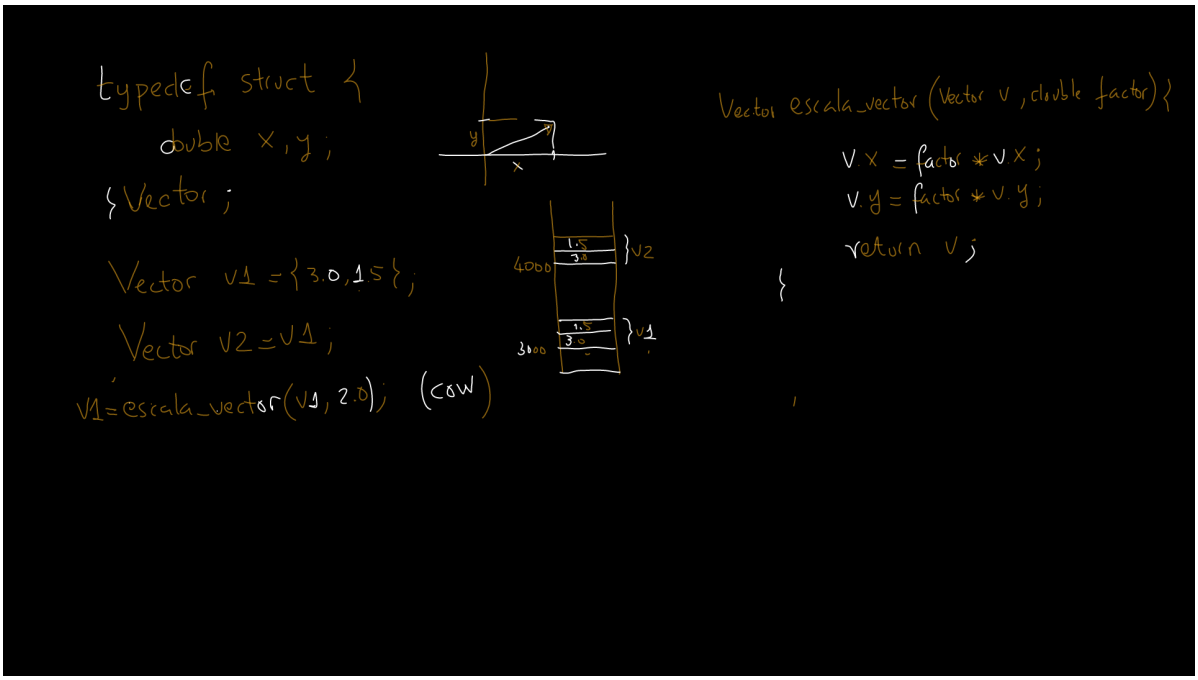

Figura 7: Las estructuras se pueden copiar utilizando el operador de asignación. La copia es por valor: hay dos variables y dos juegos de valores independientes. Cuando una función recibe como argumento una estructura, recibe una copia del valor de la estructura; si, dentro de la función se modifican los campos de la estructura, las modificaciones no quedan reflejadas al salir de la función.

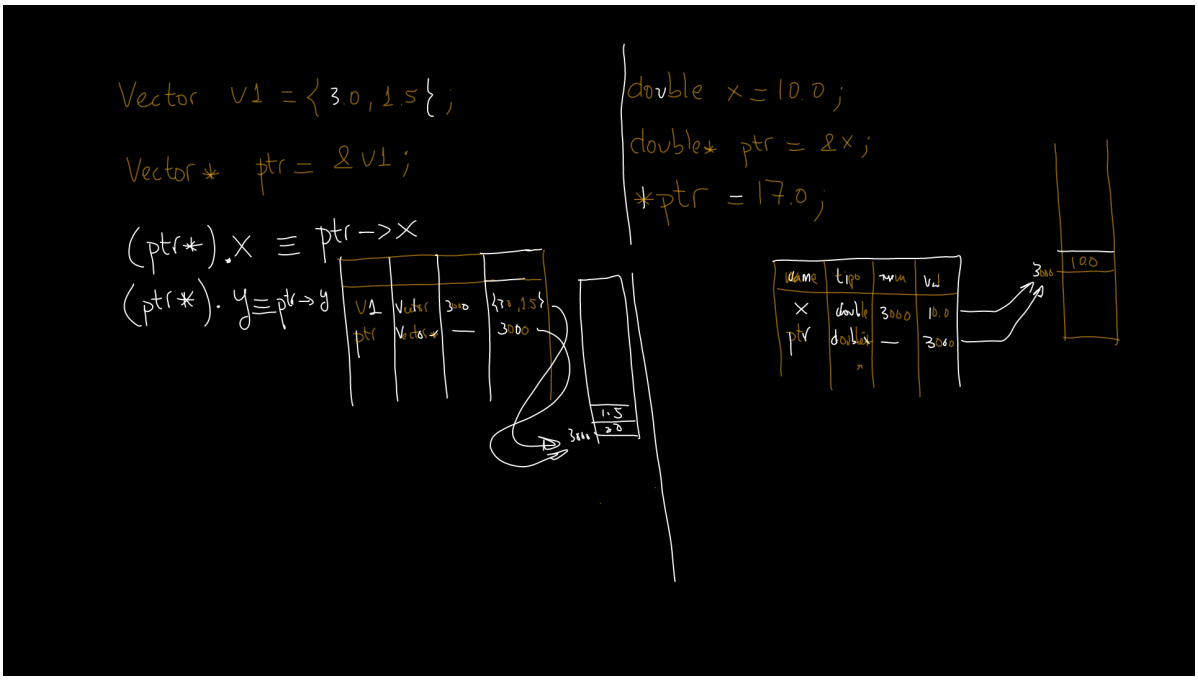

Figura 8: Si se declara un puntero a una variable del tipo estructura, entonces hay un solo juego de valores en memoria, pero hay dos *propietarios* que pueden acceder a dicho valor: la variable y el puntero. Es lo mismo que sucede cuando se declara un puntero a una variable de un tipo primitivo. Para acceder a los campos de una estructura utilizando un puntero, se puede utilizar el *operador flecha* (->), en sustitución de la *desreferenciación + operador punto*, con una sintaxis más farragosa.

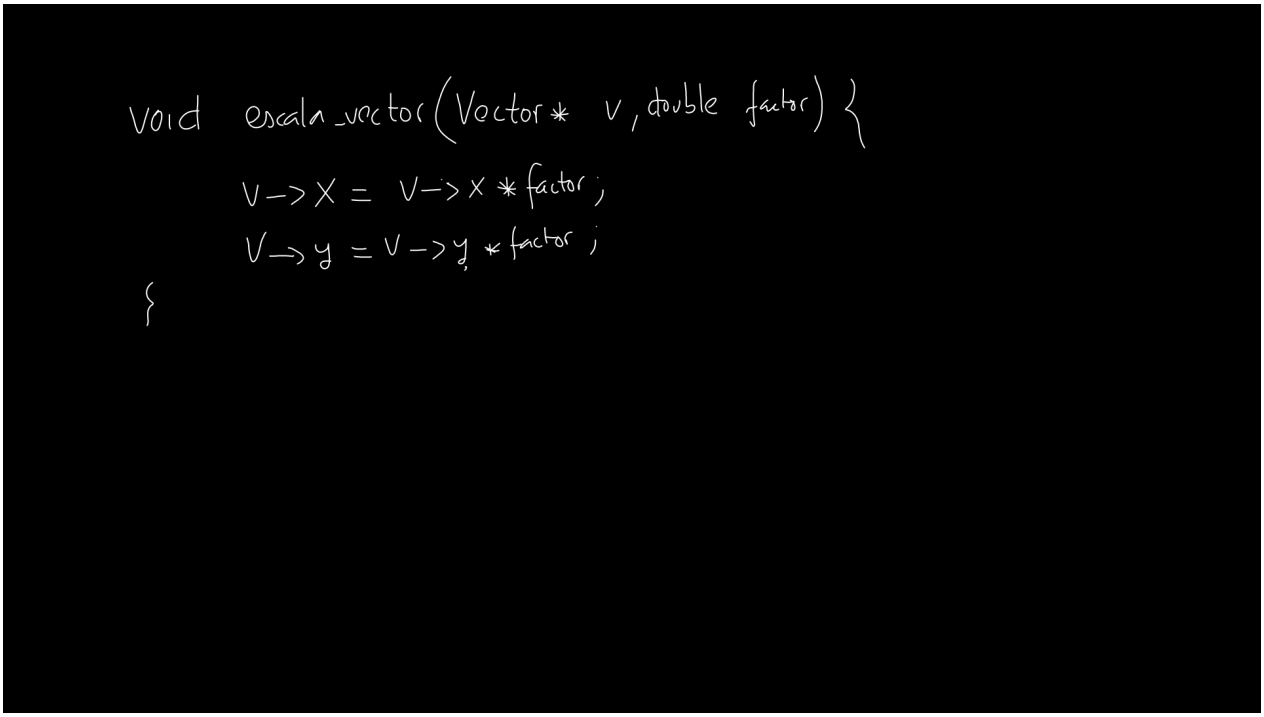

Figura 9: Cuando el parámetro de una función es un puntero a una estructura, los cambios que se hagan en los campos de la estructura dentro de la función serán permanentes y se mantendrán al salir de la misma.

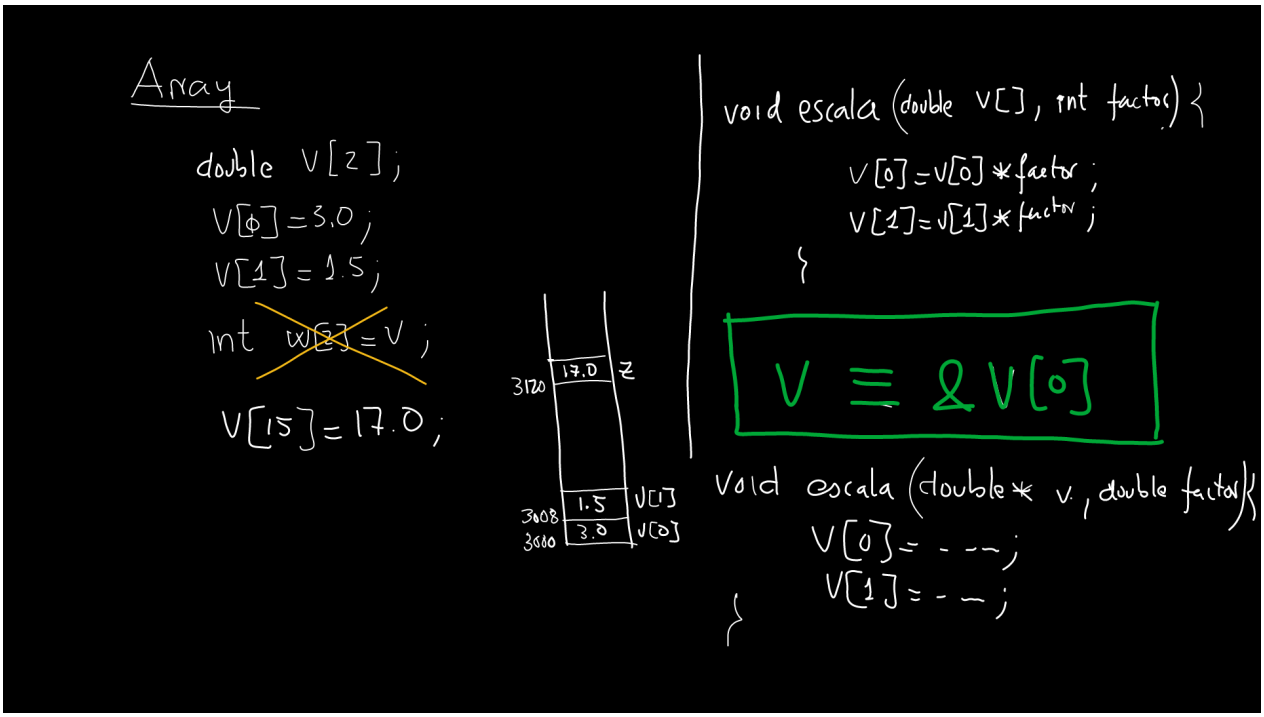

Figura 10: Los arrays no se pueden copiar utilizando el operador de asignación. El nombre de un array se comporta como un puntero al primer elemento del array. Cuando el parámetro de una función es un array, la función recibe un puntero al primer elemento del array y las modificaciones que se hagan de los valores de las componentes del array dentro de la función serán permanentes. Es importante comprender que, cuando una función recibe como argumento un array, no sabe su tamaño. Además, cuando se intenta acceder a un elemento de un array mediante un índice entre corchetes, el compilador de C no comprueba si el índice sobrepasa los límites del array. Esa situación da lugar a un comportamiento indefinido: puede dar un error de ejecución o dar lugar a otros tipos de errores difíciles de detectar. Hay dos formas equivalentes de definir un parámetro como array: poniendo el nombre y corchetes vacíos o usando nomenclatura de punteros.

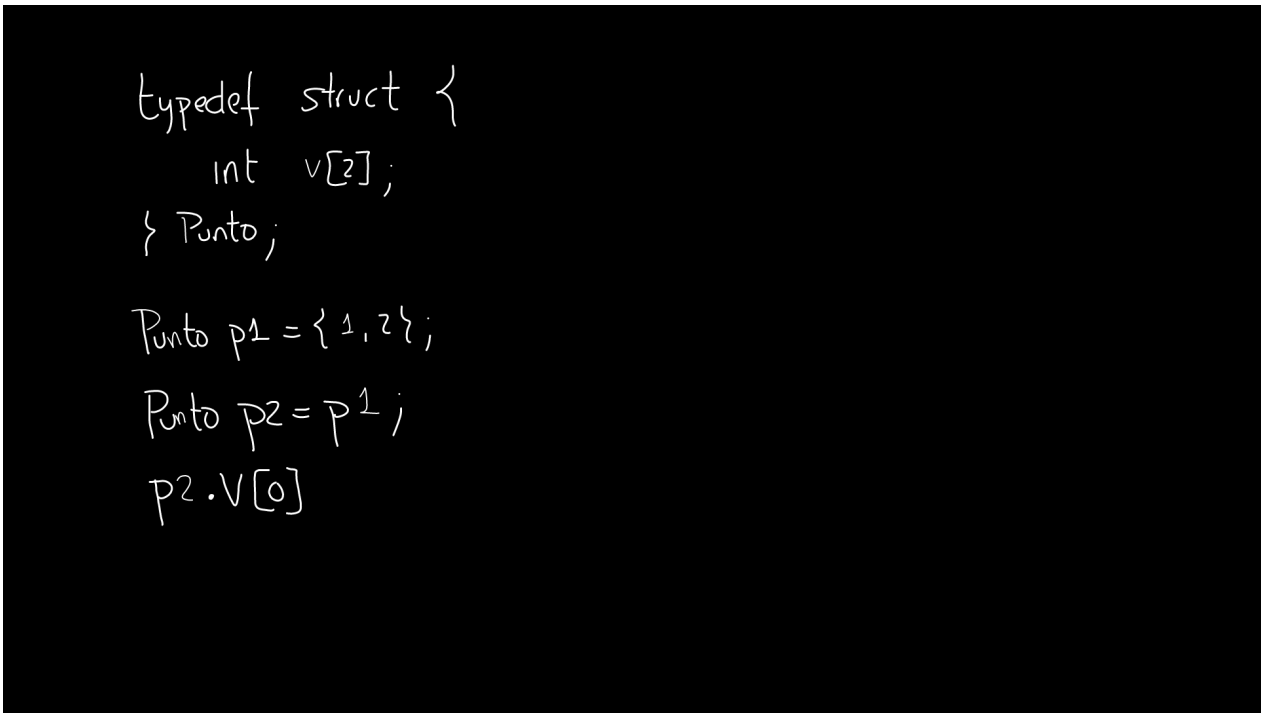

Figura 11: Los campos de una estructura pueden ser arrays. Se puede seguir utilizando el operador de asignación para las variables de tipo estructura, aunque algún campo sea un array.

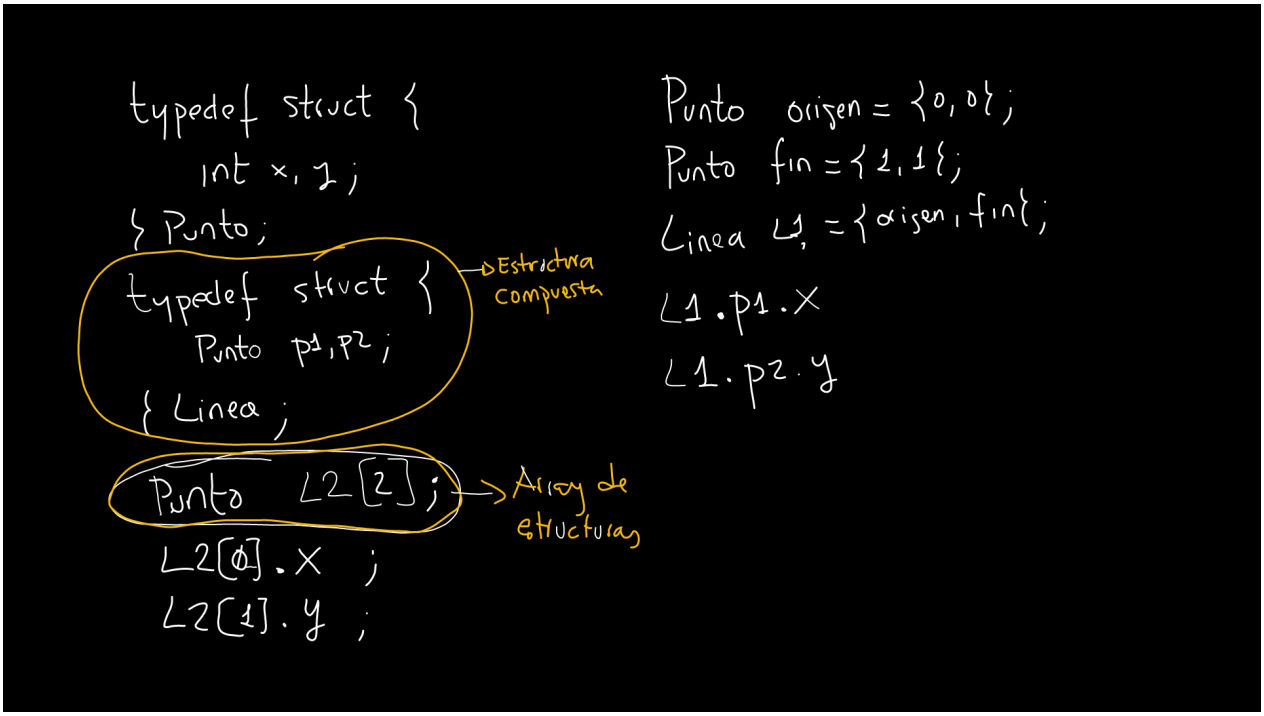

Figura 12: Los campos de una estructura pueden ser otras estructuras. Observa en la imagen la sintaxis de doble operador punto para acceder a un campo de una estructura que es, a su vez, campo de otra estructura. También es posible crear arrays cuyos elementos sean estructuras.

Hay dos técnicas para modificar los campos de una estructura dentro de una función: la técnica denominada *Copy On Write*, en la cual las modificaciones se reciben a través del valor devuelto y la técnica de utilizar como parámetro de la función un puntero a la estructura. El Ejemplo 4 muestra cómo se pueden utilizar ambas técnicas.

Ejemplo 4 Ejemplo de funciones que modifican estructuras

```c
#include <stdio.h>

// Definiciones globales de tipos
typedef struct {
   double x, y;
} Vector;

// Prototipos de funciones
Vector escalar_cow(Vector v, double factor);
void escalar_insitu(Vector* v, double factor);

// Programa principal
int main() {
   Vector v1 = {3.0, 1.5};
   printf("v1={%.2f, %.2f}\n", v1.x, v1.y);

   // Modificación de un Vector utilizando
   // la técnica de Copy On Write
   v1 = escalar_cow(v1, 2.0);
   printf("v1={%.2f, %.2f}\n", v1.x, v1.y);

   // Modificación de un Vector utilizando
   // parámetros de tipo puntero
   escalar_insitu(&v1, 2.0);
   printf("v1={%.2f, %.2f}\n", v1.x, v1.y);

   // Se puede crear primero un puntero y utilizarlo
   // como argumento, en vez de utilizar el operador
   // directamente el operador &
   Vector* ptr = &v1;
   escalar_insitu(ptr, 2.0);
   printf("v1={%.2f, %.2f}\n", v1.x, v1.y);

   return 0;
}

// Código de las funciones

// escalar_cow (cow = Copy On Write)
// El parámetro es por valor. Las modificaciones en los campos
// de la estructura se trasmiten a través del valor devuelto
Vector escalar_cow(Vector v, double factor) {
   v.x = v.x * factor;
   v.y = v.y * factor;
   return v;
}

// escalar_insitu
// El parámetro es un puntero y, por tanto, las modificaciones
// en los campos de la estructura son permanentes
void escalar_insitu(Vector* v, double factor) {
   v->x = v->x * factor;
   v->y = v->y * factor;
}
```