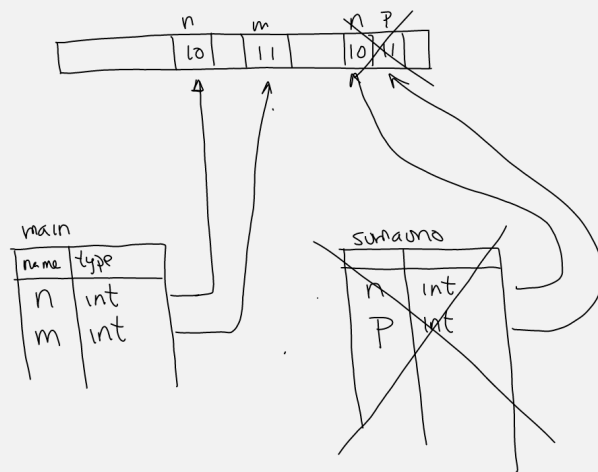


(Versión de fecha 20 de noviembre de 2024)

1 REPASO VARIABLES LOCALES

```
int sumauno(int n) {
    int p = n + 1;
    return p;
}

int main() {
    int n = 10;
    int m = sumauno(n);
    printf("%d\n", m);
}
```



Este ejemplo es un repaso de otros ejemplos similares que hemos visto en clases anteriores. En la figura se muestra el funcionamiento de las variables locales.

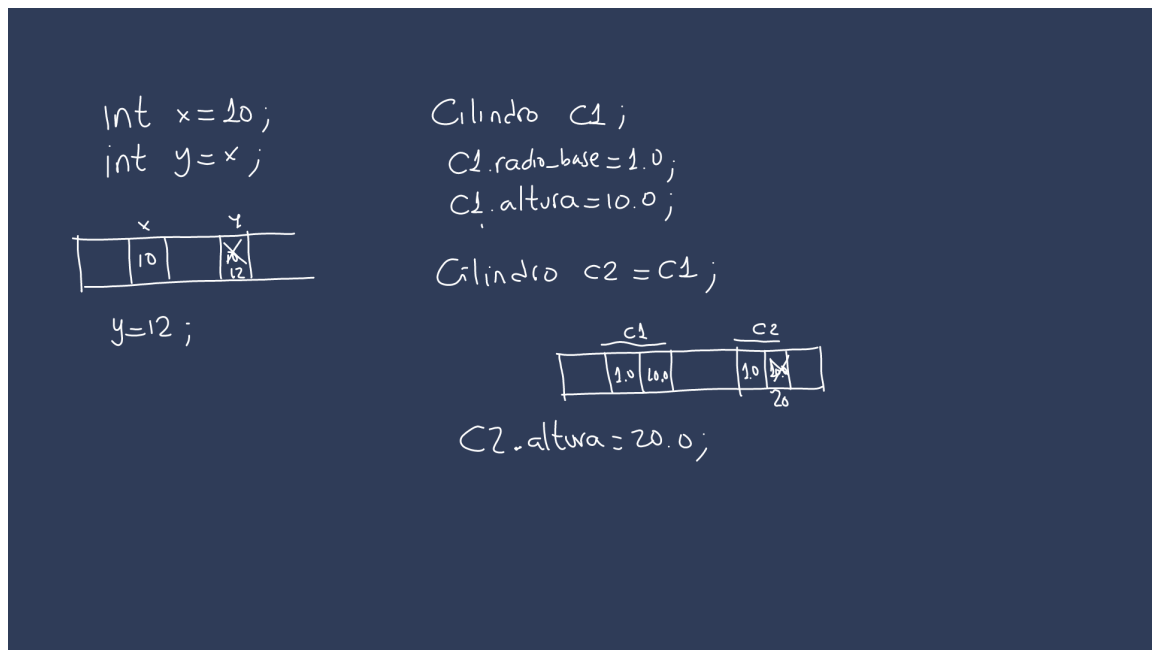
En cada momento, solo son visibles (accesibles) las variables locales de la función que se está ejecutando:

- Cuando la ejecución está dentro de la función *main()*, solo son visibles (accesibles) las variables *n* y *m*.
- Cuando la ejecución está dentro de la función *sumauno()*, solo existen la variable *n* y la variable *p*. ¡Ojo!, la variable *n* de la función *sumauno()* no es la misma variable que la *n* de la función *main()*: cada una tiene su propio espacio en memoria donde guarda su valor asociado.

¡Importante!: cuando acaba la ejecución de una función, las variables locales dejan de existir. Así, cuando acaba la ejecución de *sumauno()*, sus variables dejan de existir. Y, ¿cuando se sale de *main()* durante la llamada a *sumauno()*? Ahí no ha terminado la ejecución de *main()*, simplemente el flujo del programa pasa a *sumauno()*, pero *main()* no ha acabado y su memoria se mantiene. Eso sí, la memoria de *main()* no se destruye, pero desde dentro de *sumauno()* no es posible acceder a las variables de *main()*.

¿Quizas un poco lioso? No, coloca algún *break point* en el programa y juega un poco con el depurador.

2 UTILIZACIÓN DEL OPERADOR DE ASIGNACIÓN CON ESTRUCTURAS



Las estructuras tienen un comportamiento similar al de los tipos primitivos en muchos aspectos. Por ejemplo, se puede usar el operador de asignación para crear una copia de una variable tipo estructura:

Ejemplo 1 Operador de asignación con estructuras

```
#include <stdio.h>

typedef struct {
    double radio_base;
    double altura;
} Cilindro;

int main() {
    printf("Creo la variable c1:\n");
    Cilindro c1;
    c1.radio_base = 1.0;
    c1.altura = 10.0;
    printf("c1(%.1f, %.1f)\n\n", c1.radio_base, c1.altura);

    printf("Creo c2, como copia de c1:\n");
    Cilindro c2 = c1;
    printf("c2(%.1f, %.1f)\n\n", c2.radio_base, c2.altura);

    printf("Hay dos variables cilindro, pero cada\n");
    printf("una tiene su propio radio_base y su\n");
    printf("propia altura.\n\n");

    printf("Ahora cambio la altura de c2:\n");
    c2.altura = 20.0;
    printf("c2(%.1f, %.1f)\n\n", c2.radio_base, c2.altura);

    printf("Pero los campos de c1 no han cambiado:\n");
    printf("c1(%.1f, %.1f)\n\n", c1.radio_base, c1.altura);
}
```

Terminal output:

```
santiago@Ejemplos_20241105$ ./ej2
Creo la variable c1:
c1(1.0, 10.0)

Creo c2, como copia de c1:
c2(1.0, 10.0)

Hay dos variables cilindro, pero cada
una tiene su propio radio_base y su
propia altura.

Ahora cambio la altura de c2:
c2(1.0, 20.0)

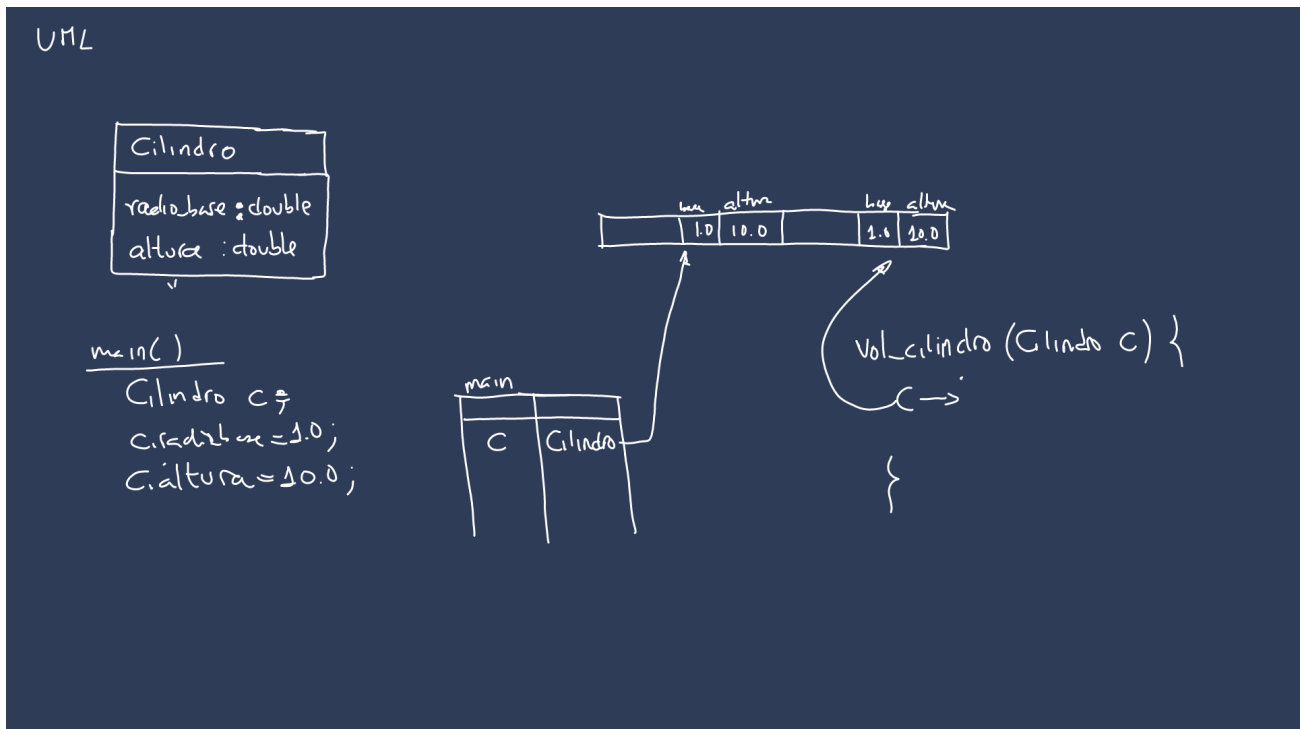
Pero los campos de c1 no han cambiado:
c1(1.0, 10.0)

santiago@Ejemplos_20241105$
```

Con variables de tipo estructura, se puede usar el operador de asignación, pero no se pueden utilizar operadores de comparación. Una expresión como la siguiente no compila:

`c1==c2` ⇒ ¡ERROR!

3 UTILIZACIÓN DE ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES



Cuando las variables de tipo estructura se utilizan como parámetros de una función, pasan por valor. Esto quiere decir que lo que llega a la función es una copia de la variable tipo estructura original, no la variable en sí misma: si dentro de la función cambio algún campo, los cambios no quedan reflejados al salir de la función.

Ejemplo 2 Estructuras como parámetros

```
#include <stdio.h>

typedef struct {
    int x, y;
} Punto;

void f(Punto p) {
    printf("Punto en f(), antes de cambios:\n");
    printf("P = (%d, %d)\n\n", p.x, p.y);
    p.x = 2*p.x;
    p.y = 2*p.y;
    printf("Punto en f(), después de cambios:\n");
    printf("P = (%d, %d)\n\n", p.x, p.y);
}

int main() {
    Punto p;
    p.x = 10;
    p.y = 15;
    printf("Punto en main() antes de ir a f():\n");
    printf("P = (%d, %d)\n\n", p.x, p.y);
    f(p);
    printf("Punto en main() después de f():\n");
    printf("P = (%d, %d)\n\n", p.x, p.y);
}
```

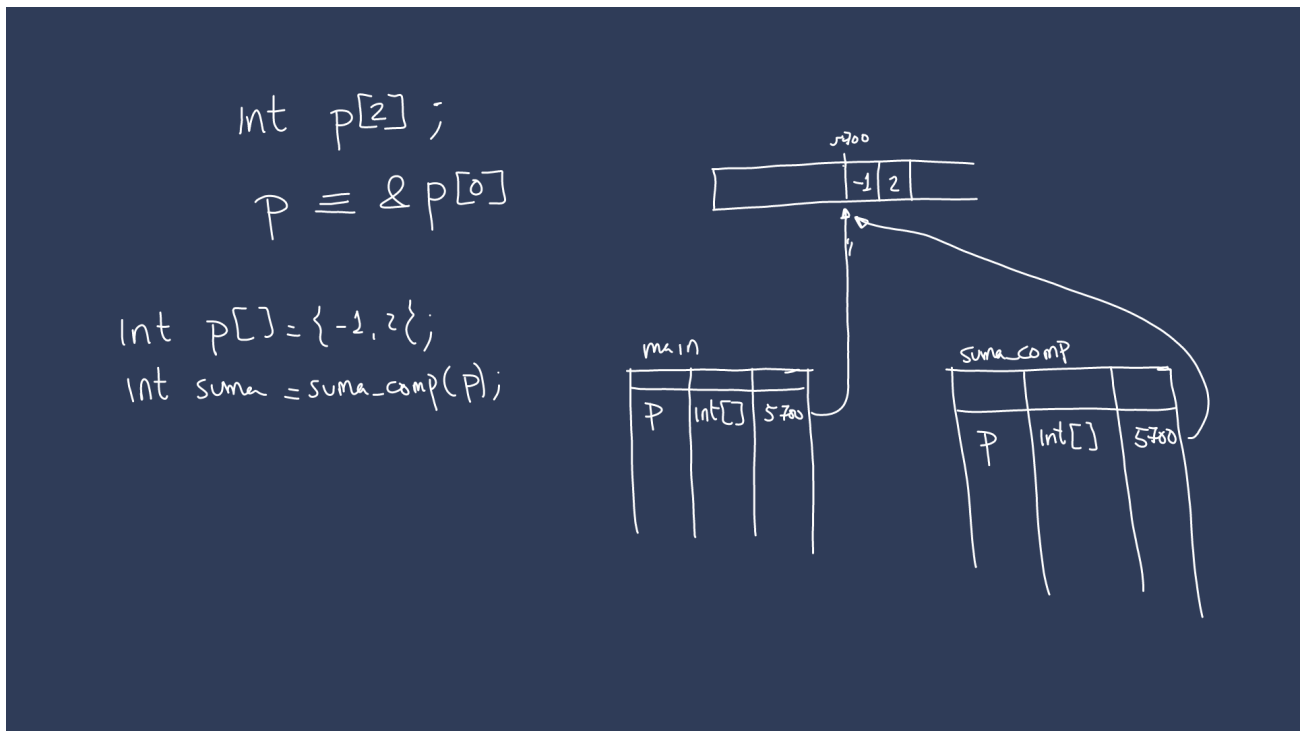
```
santiago@santiago-HP-ProBook-440-14-inch-G10-Notebook-PC: ~/ownCloud/AAA_Tel...
santiago:Pruebas$ ./p1
Punto en main() antes de ir a f():
P = (10, 15)

Punto en f(), antes de cambios:
P = (10, 15)

Punto en f(), después de cambios:
P = (20, 30)

Punto en main() después de f():
P = (10, 15)
santiago:Pruebas$
```

4 EL NOMBRE DE LOS ARRAYS COMO PUNTEROS



La siguiente instrucción declara y asigna valores a un array de números enteros:

```
int vector[] = {1, 2, 3};
```

Para referirnos a los elementos del array, podemos utilizar el nombre del array y un índice entre corchetes:

```
vector[1];
```

La expresión anterior devuelve 2, que es el valor del segundo elemento del array, el de índice 1.

Podríamos declarar un puntero a ese elemento con la siguiente instrucción:

```
int* ptr = &vector[1];
```

Se trata de un puntero normal que apunta a un número entero, la posición de memoria en la que está guardado el valor de la segunda componente del array. Podríamos acceder al valor de esa segunda componente mediante la siguiente expresión:

```
*ptr;
```

La instrucción anterior se lee como “*contenido de la dirección de memoria a la que apunta ptr*”. Esta expresión devolvería 2, que es el valor de la segunda componente de `vector`.

Pues bien, el propio nombre del array equivale a un puntero a la primera componente del array:

```
vector ≡ &vector[0]
```

De esta forma, para acceder al valor de la primera componente del array, se puede utilizar nomenclatura de índices o nomenclatura de punteros:

```
vector[0] ≡ *vector
```

Se muestra a continuación un ejemplo en el que se usan estos conceptos:

Ejemplo 3 Nombre de array como puntero

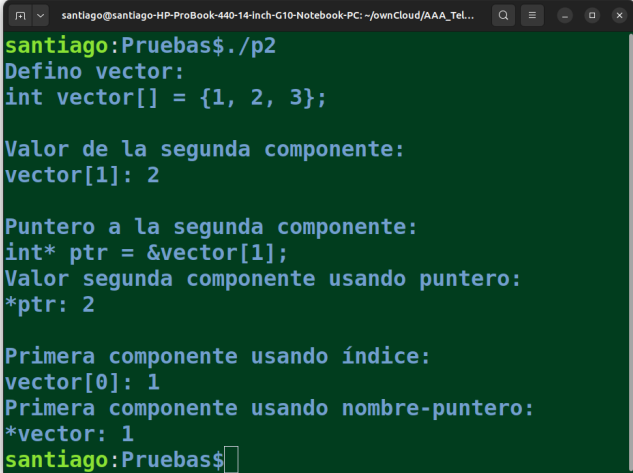
```
#include <stdio.h>

int main() {
    printf("Defino vector:\n");
    printf("int vector[] = {1, 2, 3};\n\n");
    int vector[] = {1, 2, 3};

    printf("Valor de la segunda componente:\n");
    printf("vector[1]: %d\n\n", vector[1]);

    printf("Puntero a la segunda componente:\n");
    printf("int* ptr = &vector[1];\n");
    int* ptr = &vector[1];
    printf("Valor segunda componente usando puntero:\n");
    printf("*ptr: %d\n\n", *ptr);

    printf("Primera componente usando índice:\n");
    printf("vector[0]: %d\n", vector[0]);
    printf("Primera componente usando nombre-puntero:\n");
    printf("*vector: %d\n", *vector);
}
```



```
santiago@Pruebas$ ./p2
Defino vector:
int vector[] = {1, 2, 3};

Valor de la segunda componente:
vector[1]: 2

Puntero a la segunda componente:
int* ptr = &vector[1];
Valor segunda componente usando puntero:
*ptr: 2

Primera componente usando índice:
vector[0]: 1
Primera componente usando nombre-puntero:
*vector: 1
santiago@Pruebas$
```