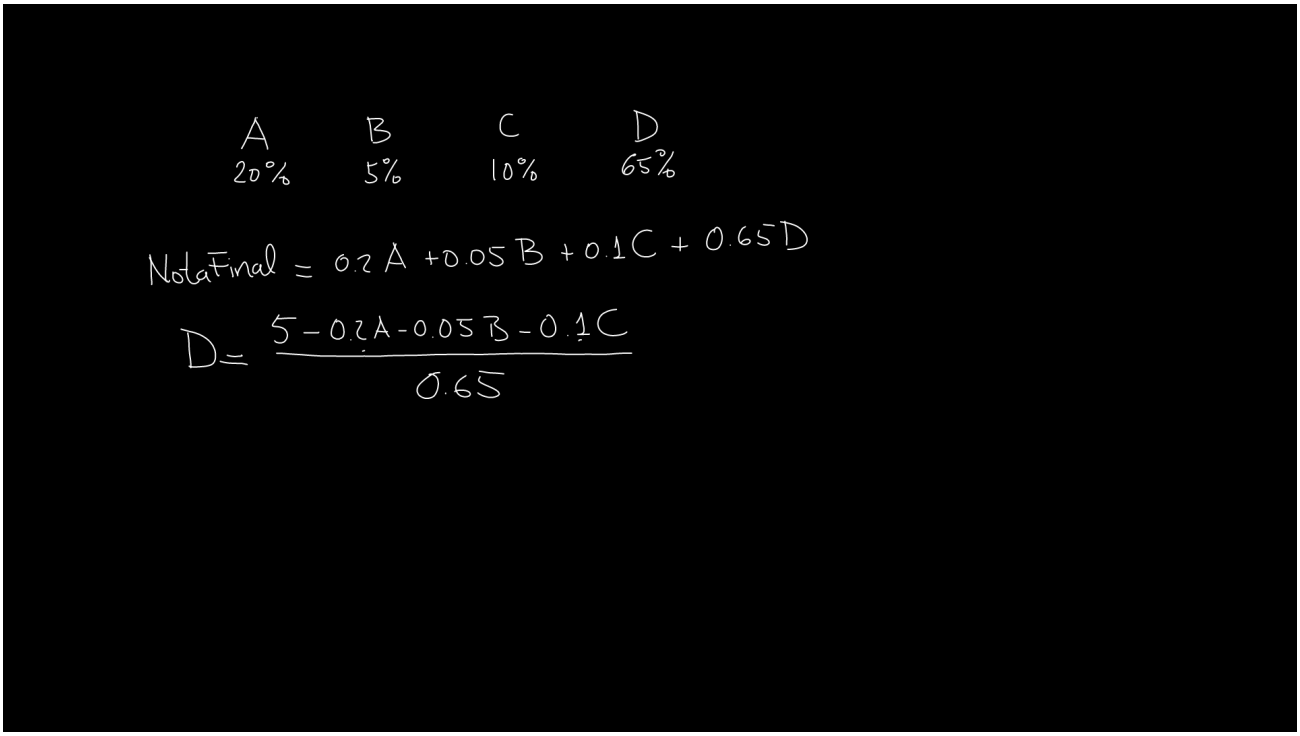
(Versión de fecha 18 de noviembre de 2024)

1 Fórmula para calcular la nota final promediada

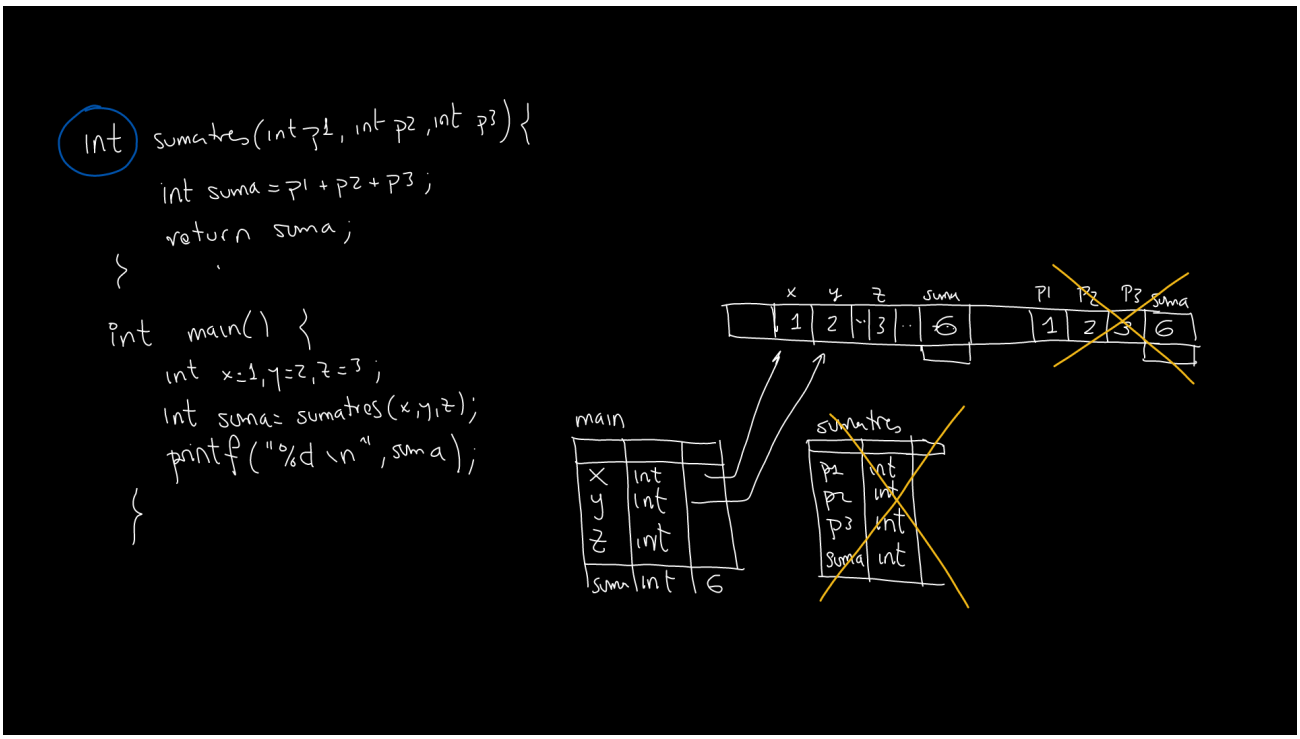


En la imagen se puede ver cómo calcular la nota final en función de las notas A, B, C y D obtenidas en los cuatro exámenes del curso.

También se muestra cómo calcular qué nota se tiene que sacar en el segundo parcial (D) para aprobar, una vez que se saben las notas A, B y C de los tres exámenes anteriores.

2 Comprender las funciones: variables locales

(Ver las diapositivas del tema 4 del curso)

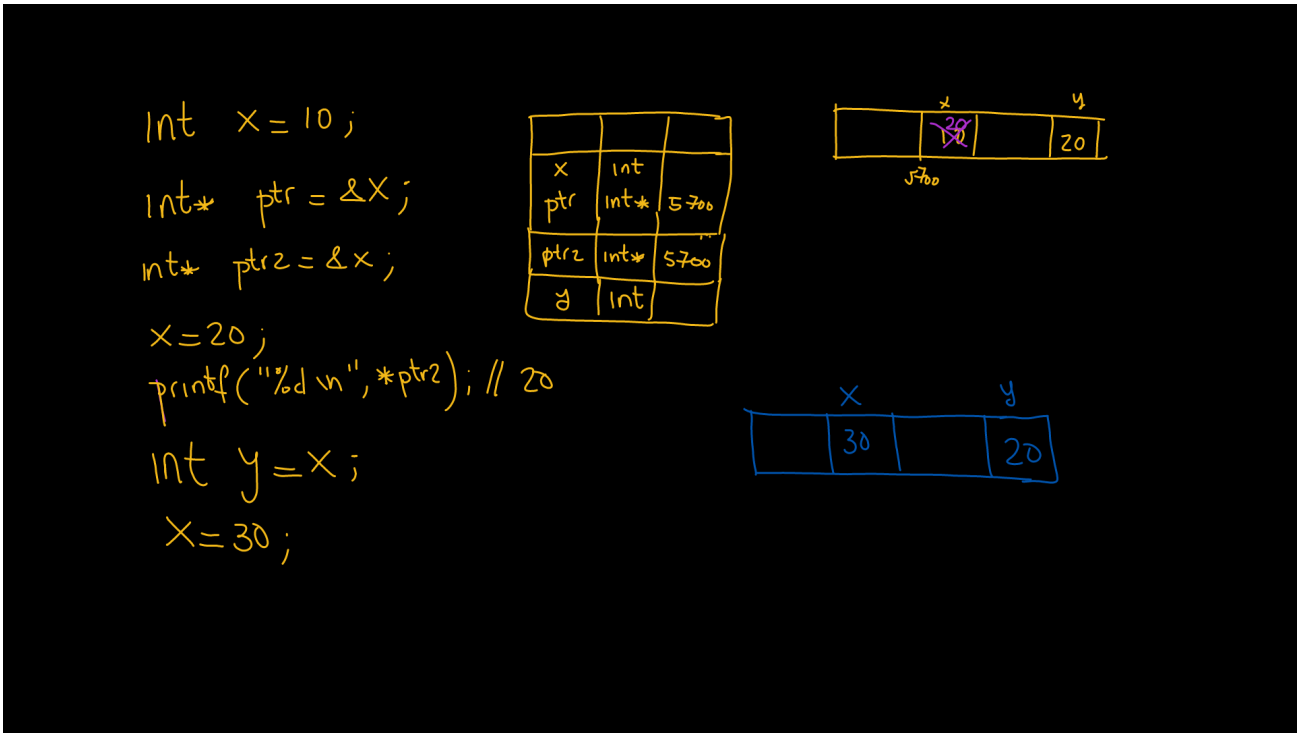


La parte izquierda de la figura anterior muestra el código de un programa que consta de dos funciones (se ha omitido la directiva *#include <stdio.h>*):

- Una función llamada *sumatres()*, que recibe tres números enteros como argumentos y cuyo valor devuelto es la suma de dichos números.
- La función principal del programa, main(). Primero se declaran tres números enteros y se les asigna un valor. A continuación, se declara la variable *suma* y se le asigna el resultado de llamar a la función *sumatres()*, pasándole los tres números anteriores como argumentos. Finalmente, se muestra el valor de la variable *suma* en pantalla.

La parte derecha de la figura muestra el esquema del funcionamiento en memoria de las distintas variables que se van creando en el programa. Las variables locales que crea la función *sumatres()*, *p1, p2 y p3*, se destruyen cuando se sale de la función. Es importante entender lo siguiente:

- Dentro de *main()*, las variables *p1, p2 y p3* no son visibles.
- Dentro de *sumatres()*, las variables *x, y, z y suma* no son visibles.
- La variables *suma* de la función *main()* y la variable *suma* de la función *sumatres()* son distintas variables, aunque se llamen igual. Cada una guarda su valor en un sitio distinto de la memoria.
- La función *sumatres()* devuelve una copia del valor de su variable *suma*, no la propia variable. Cuando termina la ejecución de la función *sumatres()*, se libera la memoria donde estaban guardados los valores de sus variables y dichas variables dejan de ser accesibles.



La figura anterior muestra el funcionamiento de los valores en memoria en dos situaciones diferentes:

```
int x = 10;
int* ptr = &x;
int* ptr2 = &x;
```

En este caso, se declara una variable *x* y se le asigna el valor 10. A continuación se crean dos punteros, *ptr* y *ptr2*, que apuntan a la dirección de memoria del valor de la variable *x*. En esta situación, solo hay un valor en memoria, aunque hay tres variables, *x*, *ptr* y *ptr2* que permiten acceder al mismo. Si, con cualquier procedimiento, se modifica el valor contenido en la memoria, las tres variables reflejarán el cambio. Observa el resultado de las siguientes instrucciones:

```
x = 20;                // Ahora el valor que hay guardado en memoria es 20
printf("%d\n", x);     // Imprime 20
printf("%d\n", *ptr);  // Imprime 20
printf("%d\n", *ptr2); // Imprime 20
*ptr = 30;             // Ahora el valor que hay guardado en memoria es 30
printf("%d\n", x);     // Imprime 30
```

Observa el siguiente código:

```
int x = 20; // Hay una variable y un valor en memoria
int y = x;  // Asigna a y el valor de x. Hay dos variables y dos valores en memoria
printf("%d %d\n", x, y); // Imprime 20 20
x = 30;     // Se modifica el valor en memoria de la variable x
printf("%d %d\n", x, y); // Imprime 30 20
```

El operador de asignación lo que asigna es el valor, las variables siguen siendo independientes. En el ejemplo anterior, hay dos variables, *x* e *y*. También hay dos valores independientes en memoria, el valor de la *x* y el valor de la *y*. La asignación *y=x* copia el valor que tenga la *x* en ese momento en la variable *y*, pero siguen siendo dos variables independientes, cada una con su valor en memoria.

Piensa en la diferencia con el ejemplo anterior basado en punteros: ahí había varias variables, pero todas apuntaban a un mismo valor en memoria. Es otra forma de entender los punteros, como un mecanismo que permite tener varias variables que utilizan el mismo valor guardado en memoria.