



Escuela de  
Ciencia y Tecnología  
ECyT\_UNSAM

# Programación (en C)

## Primer Cuatrimestre 2025

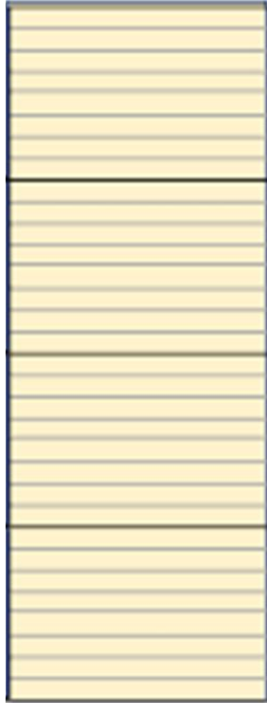
[programacionbunsam@gmail.com](mailto:programacionbunsam@gmail.com)



# Memoria y Punteros



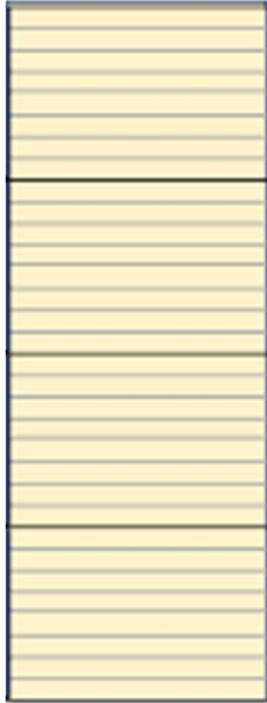
# Memoria, para programadores



- La memoria en el contexto de la programación es un conjunto de espacios / cajas / baldes que contienen números, caracteres o cualquier otro tipo de dato, siempre en formato bits.



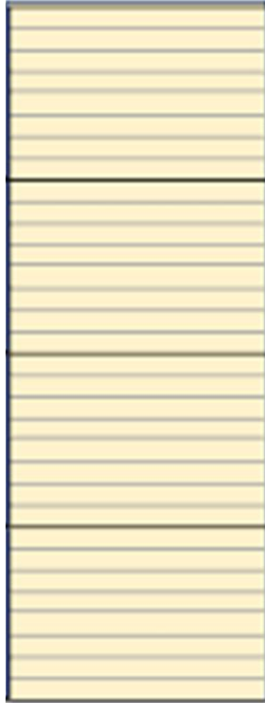
# Memoria, para programadores



- La memoria en el contexto de la programación es un conjunto de espacios / cajas / baldes que contienen números, caracteres o cualquier otro tipo de dato, siempre en formato bits.
- Cada uno de estos espacios tiene una dirección de memoria que lo identifica, en general expresada en hexadecimal.



# Memoria, para programadores



- La memoria en el contexto de la programación es un conjunto de espacios / cajas / baldes que contienen números, caracteres o cualquier otro tipo de dato, siempre en formato bits.
- Cada uno de estos espacios tiene una dirección de memoria que lo identifica, en general expresada en hexadecimal.
- El tamaño de cada uno de estos espacios es de 1 byte.



- La memoria en el contexto de la programación es un conjunto de espacios / cajas / baldes que contienen números, caracteres o cualquier otro tipo de dato, siempre en formato bits.
- Cada uno de estos espacios tiene una dirección de memoria que lo identifica, en general expresada en hexadecimal.
- El tamaño de cada uno de estos espacios es de 1 byte.



# Estructura

<b>Segmento de código</b>	<ul style="list-style-type: none"><li>+ Código del programa</li><li>- Solo lectura</li></ul>
<b>Segmento de datos</b>	<ul style="list-style-type: none"><li>+ Variables globales</li><li>+ Variables estáticas</li></ul>
<b>Pila (pila)</b>	<ul style="list-style-type: none"><li>+ Variables locales</li></ul>
<b>Heap (montón)</b>	<ul style="list-style-type: none"><li>+ Memoria dinámica</li><li>- Administrable (malloc, free)</li></ul>



# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**





# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**

En general, los usaremos para guardar las direcciones de memoria de alguna variable.



# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**

```
int var = 10;
```

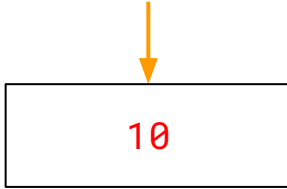
En general, los usaremos para guardar las direcciones de memoria de alguna variable.



# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**

```
int var = 10;
```



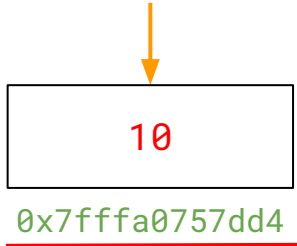
En general, los usaremos para guardar las direcciones de memoria de alguna variable.



# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**

```
int var = 10;
```



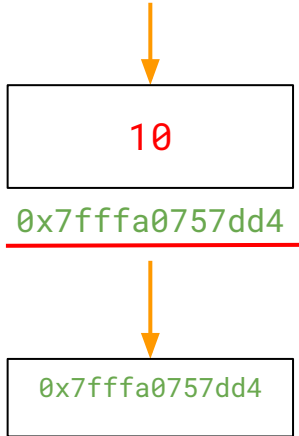
En general, los usaremos para guardar las direcciones de memoria de alguna variable.



# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**

```
int var = 10;
```



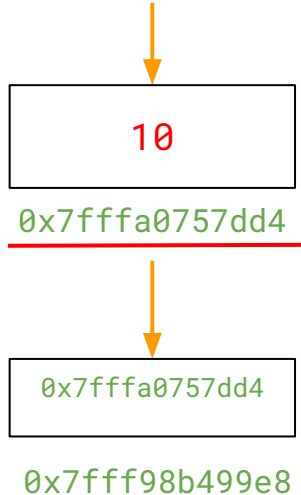
En general, los usaremos para guardar las direcciones de memoria de alguna variable.



# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**

```
int var = 10;
```



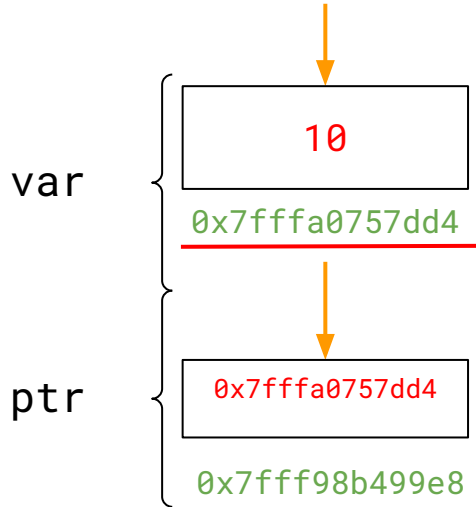
En general, los usaremos para guardar las direcciones de memoria de alguna variable.



# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**

```
int var = 10;
```



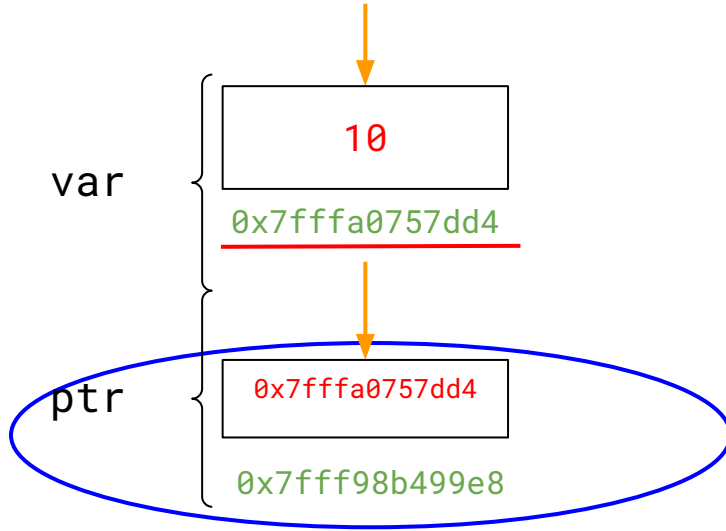
En general, los usaremos para guardar las direcciones de memoria de alguna variable.



# Punteros (1)

**def.** Los punteros son un tipo de dato en los que podemos guardar **direcciones de memoria**

```
int var = 10;
```



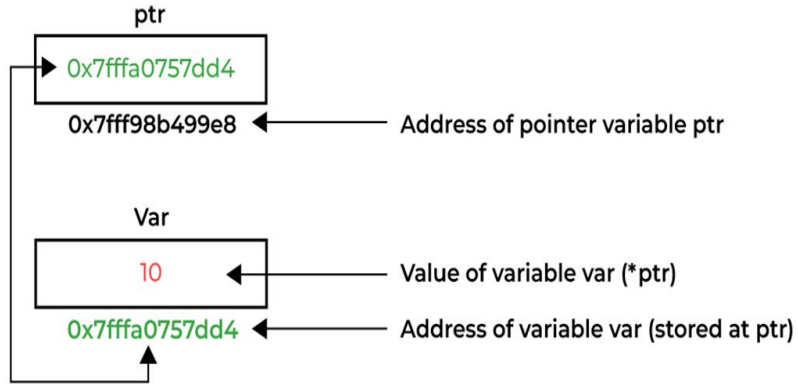
En general, los usaremos para guardar las direcciones de memoria de alguna variable.

Este será nuestro puntero





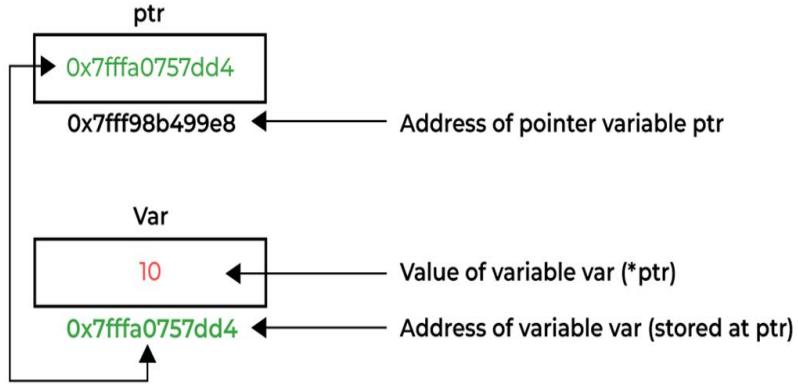
## Punteros (2) - declaración





## Punteros (2) - declaración

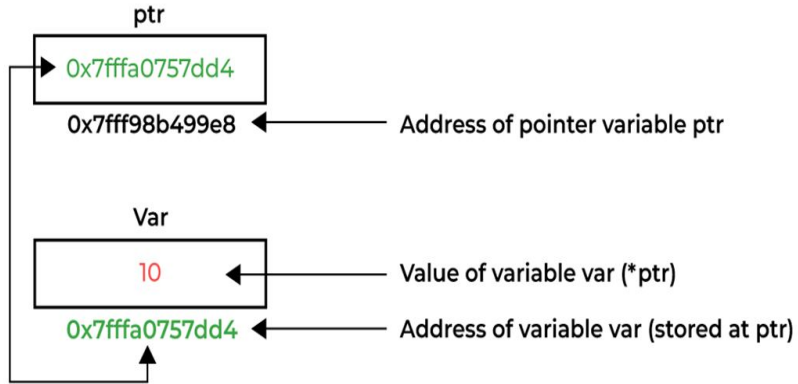
```
data_type* pointer_name = &variable_name;
```






## Punteros (2) - declaración


Tipo de dato al que **data\_type\*** `pointer_name` = `&variable_name`; apunta

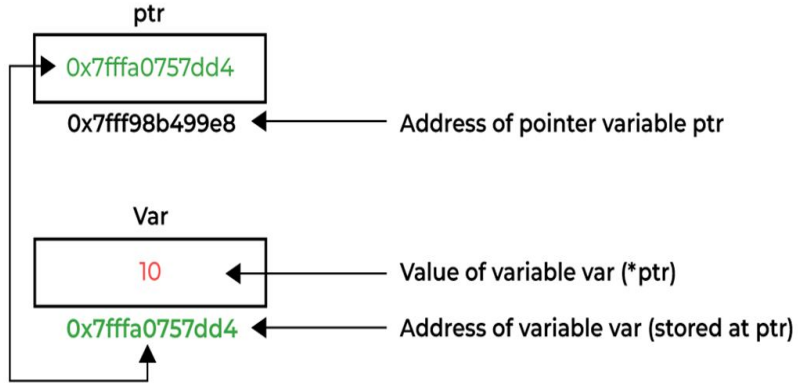




## Punteros (2) - declaración

Tipo de dato al que apunto  `data_type*` `pointer_name` = `&variable_name`;

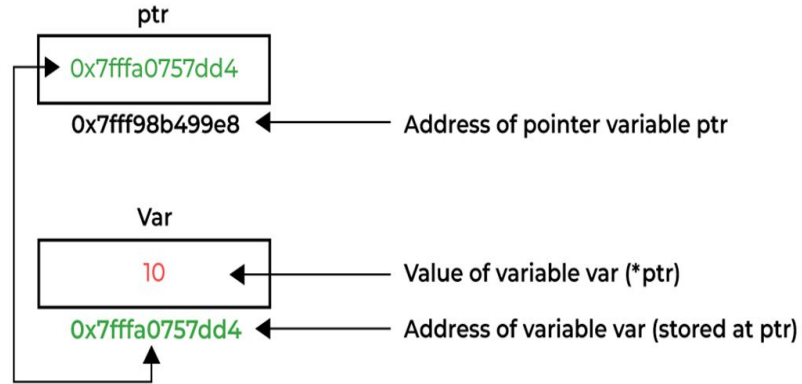
 variable a la que quiero apuntar





## Punteros (2) - declaración

Tipo de dato al que apunto  $\leftarrow$  `data_type*` `pointer_name` = `&variable_name;`



dirección de memoria de la variable a la que quiero apuntar

variable a la que quiero apuntar

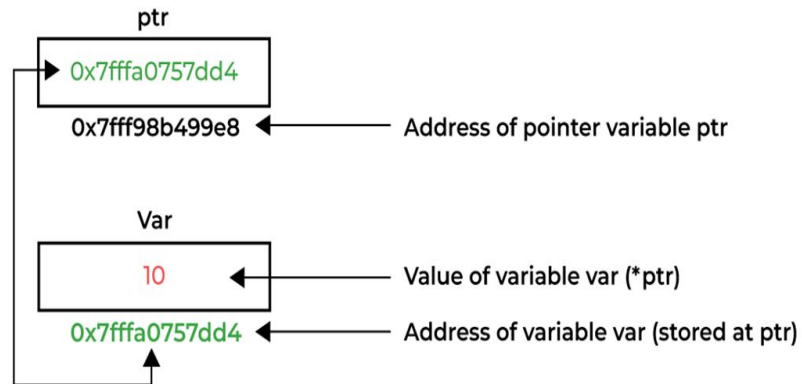


## Punteros (2) - declaración

Tipo de dato al que  
apunto

`data_type*`

`pointer_name = &variable_name;`



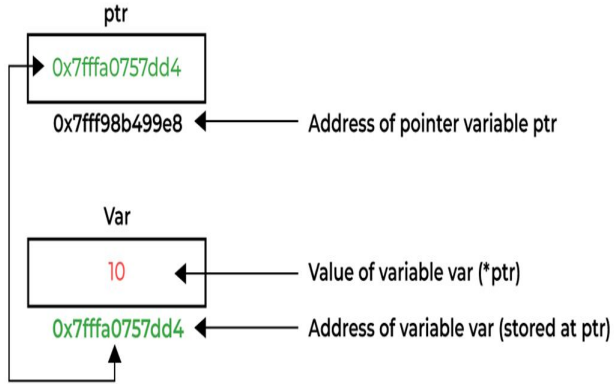
dirección de memoria de la  
variable a la que quiero  
apuntar

variable a la  
que quiero  
apuntar

```
int  var = 10;    /* guardado en 0x7fffa0757dd4 */
int* ptr = &var; /* guarda 0x7fffa0757dd4 en 0x7fff98b499e8 */
```



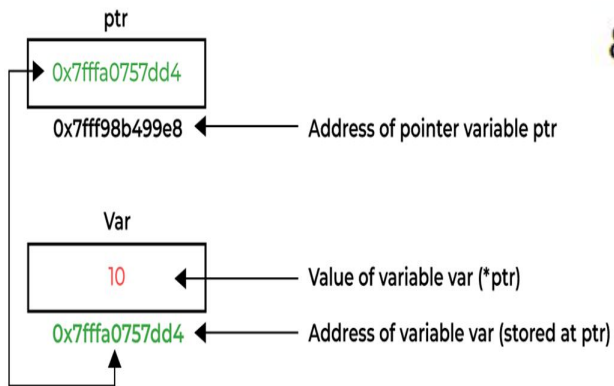
# Punteros (3) - operadores





## Punteros (2) - operadores

. Address-of (&) → devuelve la dirección de memoria de su operando



`&var = 0x7ffa0757dd4`

. Pointer indirection (\*) → devuelve el valor guardado en la dirección de memoria a la cual el operando apunta

`*ptr = 10`





# Operador & - Consideraciones

→ El operando de & debe ser 'fijo'



# Operador & - Consideraciones

→ El operando de & debe ser 'fijo'

```
int x = 10;  
int *ptr = &x;      /* Ok, la direccion de x es 'fija' */
```



# Operador & - Consideraciones

→ El operando de & debe ser 'fijo'

```
int x = 10;  
int *ptr = &x;          /* Ok, la direccion de x es 'fija' */  
  
int y = x + 5;  
int *ptr2 = &(x + 5); /* Error: 'x + 5' es un valor temporal */
```



# Operador & - Consideraciones

→ El operando de & debe ser 'fijo'

```
int x = 10;  
int *ptr = &x;          /* Ok, la direccion de x es 'fija' */
```

```
int y = x + 5;  
int *ptr2 = &(x + 5); /* Error: 'x + 5' es un valor temporal */
```

```
const int z = 150;  
int *ptr3 = &z;          /* Error 'z' puede estar guardad en una direccion de memoria 'read only' */
```



## Operador \* - Consideraciones

→ El operando de \* debe ser una dirección de memoria válida



# Operador \* - Consideraciones

→ El operando de \* debe ser una dirección de memoria válida

```
int x = 10;
int *ptr = &x;      /* Ok, la direccion de x es 'fija' */
int y = *ptr;        /* Ok, ptr guarda una direccion de memoria valida */
```



# Operador \* - Consideraciones

→ El operando de \* debe ser una dirección de memoria válida

```
int x = 10;
int *ptr = &x;      /* Ok, la direccion de x es 'fija' */
int y = *ptr;        /* Ok, ptr guarda una direccion de memoria valida */

int *ptr2;
int y = *ptr2;       /* Error: Comportamiento no definido, 'ptr2' no fue inicializada */
```



# Operador \* - Consideraciones

→ El operando de \* debe ser una dirección de memoria válida

```
int x = 10;
int *ptr = &x;      /* Ok, la direccion de x es 'fija' */
int y = *ptr;        /* Ok, ptr guarda una direccion de memoria valida */

int *ptr2;
int y = *ptr2;       /* Error: Comportamiento no definido, 'ptr2' no fue inicializada */

int *ptr3 = NULL;
int y = *ptr3;       /* Runtime error! Esta prohibido des-referenciar un NULL */
```





# Operador \* - Consideraciones

→ El operando de \* debe ser una dirección de memoria válida

```
int x = 10;
int *ptr = &x;      /* Ok, la direccion de x es 'fija' */
int y = *ptr;        /* Ok, ptr guarda una direccion de memoria valida */

int *ptr2;
int y = *ptr2;       /* Error: Comportamiento no definido, 'ptr2' no fue inicializada */

int *ptr3 = NULL;
int y = *ptr3;       /* Runtime error! Esta prohibido des-referenciar un NULL */

int *ptr4 = (int*)malloc(sizeof(int));
free(ptr4);
int y = *ptr4;       /* Error: Comportamiento no definido, la memoria fue liberada */
```



# Operador \* - Consideraciones

→ El operando de \* debe ser una dirección de memoria válida

```
int x = 10;
int *ptr = &x;      /* Ok, la direccion de x es 'fija' */
int y = *ptr;        /* Ok, ptr guarda una direccion de memoria valida */

int *ptr2;
int y = *ptr2;       /* Error: Comportamiento no definido, 'ptr2' no fue inicializada */

int *ptr3 = NULL;
int y = *ptr3;       /* Runtime error! Esta prohibido des-referenciar un NULL */

int *ptr4 = (int*)malloc(sizeof(int));
free(ptr4);
int y = *ptr4;       /* Error: Comportamiento no definido, la memoria fue liberada */

int arr[5] = {1,2,3,4,5};
int *ptr5 = arr + 5; /* Apunta a arr[5] */
int y = *ptr5;       /* Error: Comportamiento no definido, 'out-of'bound' access */
```



# Operador \* - Consideraciones

→ El operando de \* debe ser una dirección de memoria válida

```
int x = 10;
int *ptr = &x;      /* Ok, la direccion de x es 'fija' */
int y = *ptr;        /* Ok, ptr guarda una direccion de memoria valida */

int *ptr2;
int y = *ptr2;       /* Error: Comportamiento no definido, 'ptr2' no fue inicializada */
```

```
int *ptr3 = NULL;
int y = *ptr3;       /* Runtime error! Esta prohibido des-referenciar un NULL */

int *ptr4 = (int*)malloc(sizeof(int)); Estos son los que nos van a molestar cuando
free(ptr4);          trabajamos con listas enlazadas
int y = *ptr4;        /* Error: Comportamiento no definido, la memoria fue liberada */

int arr[5] = {1,2,3,4,5};
int *ptr5 = arr + 5;  /* Apunta a arr[5] */
int y = *ptr5;        /* Error: Comportamiento no definido, 'out-of'bound' access */
```



# Arrays y Pointers

En C, los arrays y los punteros están fuertemente relacionados. El nombre de un array funciona exactamente igual que un **puntero constante** <sup>1</sup>.



# Arrays y Pointers

En C, los arrays y los punteros están fuertemente relacionados. El nombre de un array funciona exactamente igual que un **puntero constante** <sup>1</sup>.

```
long long arr[LEN] = {1E8 , 3.8E6, 5};  
long long *p_arr;  
p_arr = arr;  
p_arr = &arr[0];
```

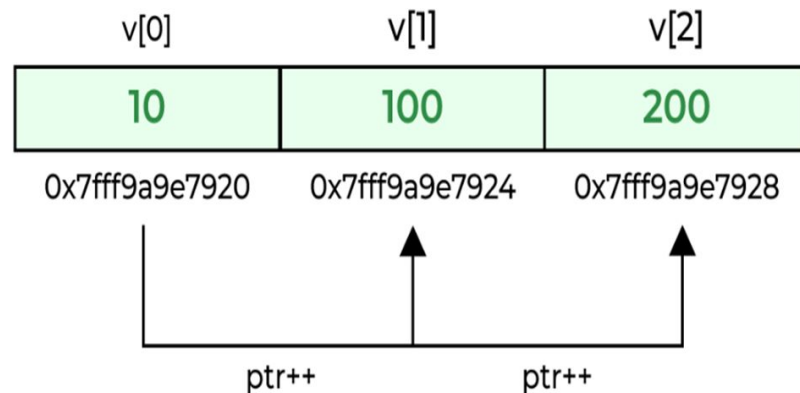


# Arrays y Pointers

En C, los arrays y los punteros están fuertemente relacionados. El nombre de un array funciona exactamente igual que un **puntero constante** <sup>1</sup>.

```
long long arr[LEN] = {1E8 , 3.8E6, 5};  
long long *p_arr;  
p_arr = arr;  
p_arr = &arr[0];
```

} son equivalentes





# Pointer Arithmetic



# Pointer Arithmetic

1. Incremento / Decremento





# Pointer Arithmetic

1. Incremento / Decremento  $\longrightarrow$  Dependenden del tipo de dato a la que apunta el puntero  
ptr++ / ptr--



# Pointer Arithmetic

1. Incremento / Decremento  $\longrightarrow$  Dependenden del tipo de dato a la que apunta el puntero  
ptr++ / ptr--
2. Suma / Resta de integer



# Pointer Arithmetic

1. Incremento / Decremento  $\longrightarrow$  Dependenden del tipo de variable a la que apunta el puntero  
 $\text{ptr}++$  /  $\text{ptr}--$
2. Suma / Resta de integer  $\longrightarrow$   $= \text{ptr} \pm \text{sizeof}(\text{data\_type}) * \text{value}$   
i.e  $\text{ptr} \pm 2$



# Pointer Arithmetic

1. Incremento / Decremento  $\longrightarrow$  Dependenden del tipo de variable a la que apunta el puntero  
 $\text{ptr}++$  /  $\text{ptr}--$
2. Suma / Resta de integer  $\longrightarrow$   $= \text{ptr} \pm \text{sizeof}(\text{data\_type}) * \text{value}$   
i.e  $\text{ptr} \pm 2$
3. Resta de 2 ptrs del mismo tipo



# Pointer Arithmetic

1. Incremento / Decremento  $\longrightarrow$  Dependenden del tipo de variable a la que apunta el puntero  
 $\text{ptr}++$  /  $\text{ptr}--$
2. Suma / Resta de integer  $\longrightarrow$   $= \text{ptr} \pm \text{sizeof}(\text{data\_type}) * \text{value}$   
i.e  $\text{ptr} \pm 2$
3. Resta de 2 ptrs del mismo tipo  $\longrightarrow$   $= \frac{\text{ptr}_a - \text{ptr}_b}{\text{sizeof}(\text{data type})}$   
i.e  $\text{ptr}_a - \text{ptr}_b$



# Pointer Arithmetic

1. Incremento / Decremento  $\longrightarrow$  Dependenden del tipo de variable a la que apunta el puntero  
 $\text{ptr}++$  /  $\text{ptr}--$
2. Suma / Resta de integer  $\longrightarrow$   $= \text{ptr} \pm \text{sizeof}(\text{data\_type}) * \text{value}$   
i.e  $\text{ptr} \pm 2$
3. Resta de 2 ptrs del mismo tipo  $\longrightarrow$   $= \frac{\text{ptr}_a - \text{ptr}_b}{\text{sizeof}(\text{data type})}$   
i.e  $\text{ptr}_a - \text{ptr}_b$
4. Comparación de punteros



# Pointer Arithmetic

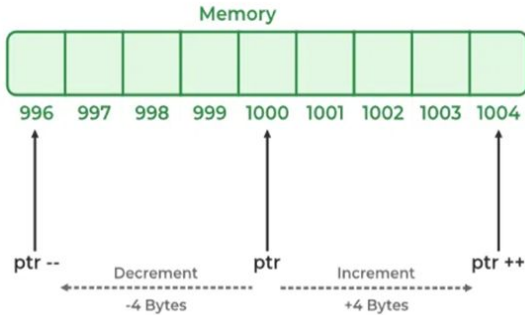
1. Incremento / Decremento  $\longrightarrow$  Dependen del tipo de variable a la que apunta el puntero  
 $\text{ptr}++$  /  $\text{ptr}--$
2. Suma / Resta de integer  $\longrightarrow$   $= \text{ptr} \pm \text{sizeof}(\text{data\_type}) * \text{value}$   
i.e  $\text{ptr} \pm 2$
3. Resta de 2 ptrs del mismo tipo  $\longrightarrow$   $= \frac{\text{ptr}_a - \text{ptr}_b}{\text{sizeof}(\text{data type})}$   
i.e  $\text{ptr}_a - \text{ptr}_b$
4. Comparación de punteros  $\longrightarrow$  igual que con cualquier otra variable  
i.e  $\text{ptr}_a > \text{ptr}_b$



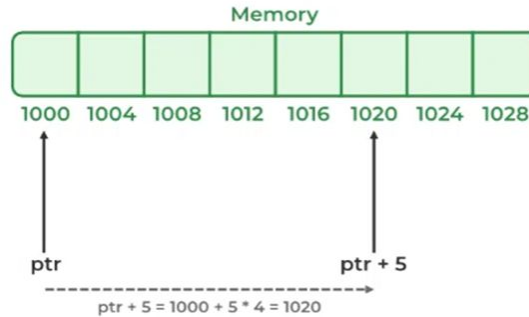
# Pointer Arithmetic

```
int* ptr = &var;
```

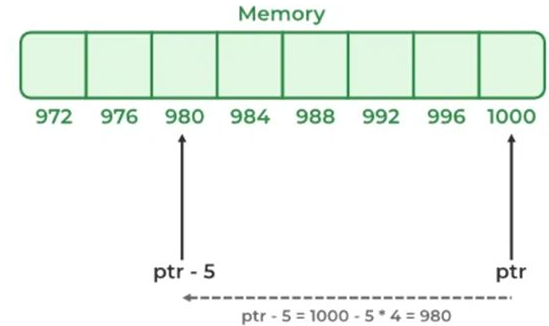
## Pointer Increment & Decrement



## Pointer Addition



## Pointer Subtraction







# Funciones - argumentos por referencia



# Funciones - argumentos por referencia

Los argumentos que uno le pasa a una función son variables locales. Esto implica que, aunque tengan el mismo valor que la variable que se pasó como argumento, su dirección de memoria es distinta.



# Funciones - argumentos por referencia

Los argumentos que uno le pasa a una función son variables locales. Esto implica que, aunque tengan el mismo valor que la variable que se pasó como argumento, su dirección de memoria es distinta.

Por este motivo, hasta ahora, solo era posible modificar los valores de una variable desde una función si:



# Funciones - argumentos por referencia

Los argumentos que uno le pasa a una función son variables locales. Esto implica que, aunque tengan el mismo valor que la variable que se pasó como argumento, su dirección de memoria es distinta.

Por este motivo, hasta ahora, solo era posible modificar los valores de una variable desde una función si:

→ Se usa una variable global



# Funciones - argumentos por referencia

Los argumentos que uno le pasa a una función son variables locales. Esto implica que, aunque tengan el mismo valor que la variable que se pasó como argumento, su dirección de memoria es distinta.

Por este motivo, hasta ahora, solo era posible modificar los valores de una variable desde una función si:

- Se usa una variable global
- Con un return



# Funciones - argumentos por referencia

Los argumentos que uno le pasa a una función son variables locales. Esto implica que, aunque tengan el mismo valor que la variable que se pasó como argumento, su dirección de memoria es distinta.

Por este motivo, hasta ahora, solo era posible modificar los valores de una variable desde una función si:

- Se usa una variable global
- Con un return

```
void argumento_por_referencia(int* a, int* b, int* c){  
    (*a)++;  
    *b = 300;  
    *c = *b - *a;  
}
```

```
void argumento_por_referencia2 (int arr[], int length);
```

```
void argumento_por_referencia3 (int *arr, int length);
```



# Funciones - argumentos por referencia

Los argumentos que uno le pasa a una función son variables locales. Esto implica que, aunque tengan el mismo valor que la variable que se pasó como argumento, su dirección de memoria es distinta.

Por este motivo, hasta ahora, solo era posible modificar los valores de una variable desde una función si:

- Se usa una variable global
- Con un return

```
void argumento_por_referencia(int* a, int* b, int* c){  
    (*a)++;  
    *b = 300;  
    *c = *b - *a;  
}
```

```
void argumento_por_referencia2 (int arr[], int length);
```

```
void argumento_por_referencia3 (int *arr, int length);
```

Usando punteros como argumento, puedo modificar todas las variables que quiera, porque estoy cambiando información contenida en la dirección de memoria de la variable original.



## *sizeof(pointer)*

El espacio que ocupa una variable puntero es el mismo sin importar el tamaño de la variable a la que estemos apuntando. El mismo dependerá del sistema que estemos utilizando (32-bit o 64-bit).

Pueden comprobarlo usando:

```
sizeof(ptr);
```





## Bonus Track - Structs / Unions y Pointers

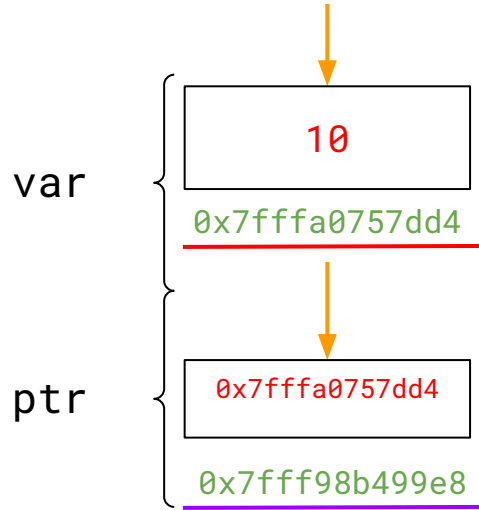
→ Un puntero a una struct / union funciona igual que un puntero a cualquier otro tipo de dato. Me permite, además, acceder a sus miembros. Cambia, un poco, la sintaxis para hacerlo.

```
struct myStruct {  
    int a;  
    char b;  
} s1;  
struct myStruct* ps1 = &s1;  
ps1->a = 50;  
ps1->b = 'h';
```



## Bonus Track - Double pointers

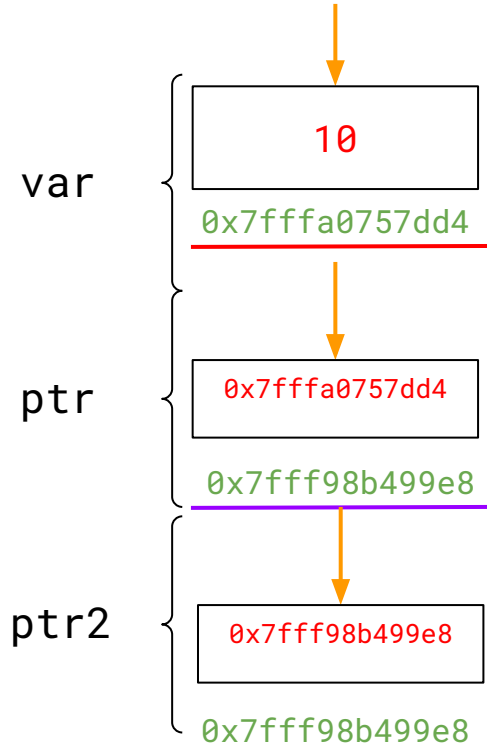
```
int var = 10;
```





## Bonus Track - Double pointers

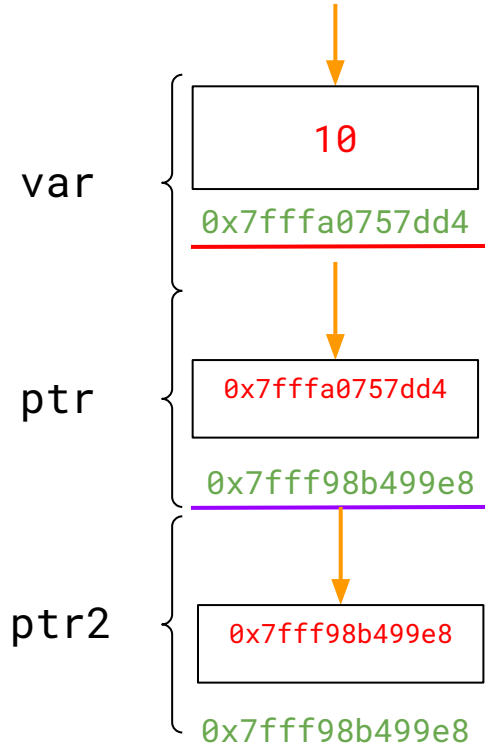
```
int var = 10;
```





## Bonus Track - Double pointers

```
int var = 10;
```



```
int var = 10;
```

```
int* ptr = &var;
```

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

```
printf("var  = \t%d\t\t[en %p]\n", var, &var);
```

```
printf("ptr   = \t%p \t[en %p]\n", ptr, &ptr);
```

```
printf("ptr2  = \t%p \t[en %p]\n", ptr2, &ptr2);
```