

# Introducción a C

---

Lenguaje, Setup y Conceptos

# **¿Cómo estamos?**

- Se acabaron las vacaciones
- ¿Vieron las cápsulas?
- ¿Han visto C antes?



# ¿Cómo estamos?

- Se acabaron las vacaciones
- ¿Vieron las cápsulas?
- ¿Han visto C antes?



Es muy importante que las vean antes de la To

# Importante:

- Ver las cápsulas de C completas
- La Tarea 0 se va a publicar próximo lunes.



# ¿Por qué C (y no Python)?

---

- En este curso nos enfocamos en dos cosas: Eficiencia y Algoritmos.
- C es mucho mucho más rápido que python
- La implementación de estructuras de datos en C es mucho más natural en relación a los contenidos del curso
- El aprender a manejar memoria permite mejor familiarización con los contenidos del curso
- **Al equipo docente le gusta C** 😊

# ¿En qué se diferencian?

1. C es un lenguaje **procedural** (*orientado a estructuras -> funciones*), Python es *orientado a objetos*
2. C es **compilado**, Python es *interpretado*
3. C tiene **punteros**
4. C es fuertemente **tipado**
5. C no requiere indentación



```
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

# ¿Cómo compilar y correr?

---

Compilar  $\longrightarrow$  `gcc nombre_archivo.c -o nombre_ejecutable`

Correr  $\longrightarrow$  `./nombre_ejecutable` argumentos

*\***gcc** es el compilador utilizado en el curso*

*helloworld.c*

```
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```



## Ejecución

```
> gcc helloworld.c -o helloworld
> ./helloworld
Hello, World!
```

*helloworld.p  
y*

```
print("Hello, World!")
```



## Ejecución

```
> python3 helloworld.py
Hello, World!
```



# **¿Notaron algo más?**

Al finalizar cada instrucción\* se  
agrega un punto y coma ;

\* responderemos cuándo no poner ; en breve

*main.py*

```
def add_floats(x, y):  
    return x + y  
  
def main():  
    x = 0.1  
    y = 2  
  
    print(f"first_number: {x}")  
    print(f"second_number: {y}")  
    print(f"result = {add_floats(x,y)}")  
  
    return
```

## Ejecución

```
$ python3 floats.py  
first_number: 0.1  
second_number: 2  
result = 2.1
```

## main.c

```
#include <stdio.h>

float add_floats(float x, float y) {
    return x + y;
}

int main() {
    float x = 0.1;
    float y = 2;

    printf("first_number: %f\n", x);
    printf("second_number: %d\n", y);
    printf("result = %f\n", add_floats(x,y));

    return 0;
}
```

## main.py

```
def add_floats(x, y):
    return x + y

def main():
    x = 0.1
    y = 2

    print(f"first_number: {x}")
    print(f"second_number: {y}")
    print(f"result = {add_floats(x,y)}")

    return
```

*main.c*

```
#include <stdio.h>

float add_floats(float x, float y) {
    return x + y;
}

int main() {
    float x = 0.1;
    float y = 2;

    printf("first_number: %f\n", x);
    printf("second_number: %d\n", y);
    printf("result = %f\n", add_floats(x,y));

    return 0;
}
```

## Ejecución

```
$ gcc floats.c -o floatsC && ./floatsC
first_number: 0.100000
second_number: 2.000000
result = 2.09999999
```

```
#include <stdio.h>
```

```
float add_floats(float x, float y) {  
    return x + y;  
}
```

```
int main() {  
    float x = 0.1;  
    float y = 2;
```

```
    printf("first_number: %f\n", x);  
    printf("second_number: %d\n", y);  
    printf("result = %f\n", add_floats(x,y));
```

```
    return 0;
```

```
}
```

Se debe incluir la librería que gestiona el input/output (necesaria para printf)

Las funciones deben declarar explícitamente el tipo de su input y output

Las variables deben declarar su tipo

Los prints no agregan automáticamente el salto de línea ("  
")

Los argumentos del print se indican después, de la siguiente forma

# Tipos

***int*** - Número Entero

***float*** - Número decimal (precisión de 6 decimales)

***double*** - Número decimal de doble precisión

***char*** - Caracter / Letra

# Tipos

```
#include <stdio.h>
int main(){
    int n_entero;
    n_entero = 10;

    int n_entero_2 = -10;
    printf("Números: %i, %d\n",n_entero,n_entero_2);

    return 0;
}
```

Se debe declarar el tipo de variable

*int*



```
#include <stdio.h>
int main(){
    int n_entero;
    n_entero = 10;

    int n_entero_2 = -10;
    printf("Números: %i, %d\n", n_entero, n_entero_2);

    return 0;
}
```

Se debe declarar el tipo de variable

Se puede declarar y después inicializar  
o hacerlo en la misma línea

*int*

```
#include <stdio.h>
int main(){
    int n_entero;
    n_entero = 10;

    int n_entero_2 = -10;
    printf("Números: %i, %d\n",n_entero,n_entero_2);

    return 0;
}
```

Se debe declarar el tipo de variable

Se puede declarar y después inicializar  
o hacerlo en la misma línea

Para imprimir se puede usar %i o %d

*int*

```

int main()
{
    #include <stdio.h>
    int n_entero;
    n_entero = 10;

    int n_entero_2 = -10;
    printf("Números: %i, %d\n",n_entero,n_entero_2);

    return 0;
}

```

*int*

```

int main()
{
    #include <stdio.h>
    float n_decimal;
    n_decimal = 3.1415;

    float n_decimal_2 = -2.789;
    printf("Números: %f, %f\n",n_decimal,n_decimal_2);

    return 0;
}

```

*float*

```

int main()
{
    #include <stdio.h>
    double n_double;
    n_double = 3.12355363;

    double n_double_2 = -2.78234;
    printf("Números: %lf, %lf\n", n_double, n_double_2);

    return 0;
}

```

*double*

```

int main()
{
    #include <stdio.h>
    char caracter;
    caracter = 'd';

    char letra = 'e';
    printf("String: %c, %c\n", caracter, letra);

    return 0;
}

```

*char*



```
#include <stdio.h>
int main(){
    int n_entero = 1;
    float n_decimal = 0.1;
    double n_double = 3.12314534534;
    char letra = 'a';
    printf('Tipos: %i, %f, %lf, %c', n_entero, n_decimal, n_double, letra);

    return 0;
}
```

*Resumen de Tipos*

# Tipos

# Flujo

***if*** - ejecución condicionada a veracidad.

***while*** - ejecución repetida mientras se mantenga la veracidad de la condición.

***for*** - ejecución repetida sujeta a número de sucesos.

***continue*** - salto a la siguiente ejecución del loop o bucle en cuestión.

***break*** - quiebre instantáneo del bucle en cuestión.

# Flujo

```
#include <stdio.h>
int main(){
    int x = 2;
    if(x>7)
    {
        printf("x > 7\n");
    }
    else if(x < 5)
    {
        printf("x < 5\n");
    }
    else
    {
        printf("x <= 7 & x >= 5\n");
    }
    return 0;
}
```

La condición va entre paréntesis

*If*

```
#include <stdio.h>
int main(){
    int x = 2;
    if(x>7)
    {
        printf("x > 7\n");
    }
    else if(x < 5)
    {
        printf("x < 5\n");
    }
    else
    {
        printf("x <= 7 & x >= 5\n");
    }
    return 0;
}
```

La condición va entre paréntesis

No llevan ; al final ✨

*If*




```
#include <stdio.h>
int main(){
    int x = 2;
    if(x>7)
    {
        printf("x > 7\n");
    }
    else if(x < 5)
    {
        printf("x < 5\n");
    }
    else
    {
        printf("x <= 7 & x >= 5\n");
    }
    return 0;
}
```

La condición va entre paréntesis

No llevan ; al final

Se usa "else if"

*If*



```
#include <stdio.h>
int main(){
    int i = 0;
    while(i<5)
    {
        printf("Ciclo %i!\n", i);
        i += 1;
    }

    return 0;
}
```

*While*



```
#include <stdio.h>
int main(){
    for(int i=0;i<5;i+=1)
    {
        printf("Ciclo %i!\n", i);
    }

    return 0;
}
```

*for*



```
#include <stdio.h>
int main(){

    while(alive)
    {
        //Codigo

        if (must_kill) break;
    }

    return 0;
}
```

*break*



```
#include <stdio.h>
int main(){
    for (int i=0; i<10;i+=1)
    {
        for(int j=0; j<10;j+=1)
        {
            if (i==j) continue;
            printf("%d, %d\n",i,j);
        }
    }
    return 0;
}
```

*continue*

# Funzione

## S

***funciones con retorno*** - Funciones en C que poseen un valor de retorno junto con un tipo explícito del mismo (int, float, char, etc)

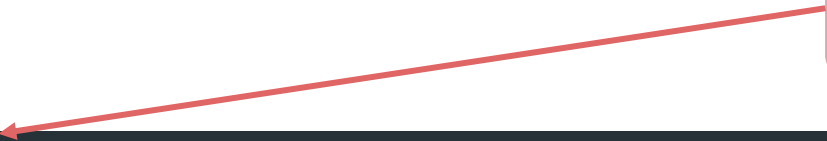
***funciones sin retorno*** - Funciones en C que no poseen ningún valor de retorno (void)

***declarar*** - Indicar el nombre de la función y sus tipos (de inputs/outputs)

***definir*** - Explicitar el contenido de la función

# Funciones

Funciones con retorno debe especificar el tipo de dato de salida



```
out_type name(arg_type1 arg1, ..., argn_type argn)
{
    // code
    return ... ;
}

void name(...)
{
    // no return code
}
```

Funciones con retorno debe especificar el tipo de dato de salida

```
out_type name(arg_type1 arg1, ..., argn_type argn)
{
    // code
    return ... ;
}

void name(...)
{
    // no return code
}
```

Debe incluir explícitamente el retorno

Funciones con retorno debe especificar el tipo de dato de salida

```
out_type name(arg_type1 arg1, ..., argn_type argn)
{
    // code
    return ... ;
}

void name(...)
{
    // no return code
}
```


Debe incluir explícitamente el retorno

Para funciones sin retorno se antecede el tipo de dato `void`



```
// declarar  
int suma(int a, int b);
```

Indicamos tipos de entradas y salidas



```
// definir  
int suma(int a, int b)  
{  
    return a + b;  
}
```

```
// declarar  
int suma(int a, int b);
```

Indicamos tipos de entradas y salidas

```
// definir  
int suma(int a, int b)  
{  
    return a + b;  
}
```

Explicitamos contenido de la función

# Veamos un ejemplo...

```
#include <stdio.h>

void printFibonacci(int depth) {
    int prev = 0, current = 1, next;

    printf("Fibonacci Series up to depth %d:\n", depth);
    printf("%d\n", prev);

    for (int i = 1; i <= depth; ++i) {
        printf("%d\n", current);
        next = prev + current;
        prev = current;
        current = next;
    }

    return;
}

int main() {
    int depth = 10;

    printFibonacci(depth);

    return 0;
}
```

Función imprime en cada iteración y retorna void

```
#include <stdio.h>

int returnFibonacci(int depth) {
    int prev = 0, current = 1, next;

    for (int i = 1; i <= depth; ++i) {
        next = prev + current;
        prev = current;
        current = next;
    }

    return prev;
}

int main() {
    int depth = 10;

    int fibonacci10 = returnFibonacci(depth);
    printf("fibonacci 10: %d\n", fibonacci10);

    return 0;
}
```

Se Imprime en el main el entero retornado por la función

# Punteros

*ojito (suele confundir bastante,  
pero la idea es simple)*

***puntero*** - variable cuyo **valor** es la **dirección** de memoria de otra variable

*“un **puntero** es una variable que **apunta** a (donde se aloja) otra variable”*

# Punteros

**puntero** - variable cuyo **valor** es la **dirección** de memoria de otra variable

**Ejemplo:** ¿Qué pasaría si tuviera el polerón en la mano todo el día?

¿Hay una solución más eficiente **para no perderlo**?



shutterstock

MADE BY: DIMITRIOS  
WWW.SHUTTERSTOCK.COM

# Punteros

*Podríamos tener una **guardarropía** y solo tener el **papelito con el número de perchero**.*



**Ejemplo**

*Podríamos tener una **guardarropía** y solo tener el **papelito con el número de perchero**.*

*En este ejemplo el polerón es el **valor** y el papelito es la **dirección**.*

# Ejemplo



Para el manejo de punteros, usaremos **asterisco (\*)** y **ampersand (&)**

**Sintaxis**

## **asterisco (\*)**

- Se utiliza al **declarar** una variable de tipo puntero, ej:

***int\* foo int \*foo int \* foo***

(todas estas formas son equivalentes)

- Se utiliza para **acceder al valor** que está almacenado en la dirección que contiene el puntero, ej:

***int valor;  
valor = \*foo;***

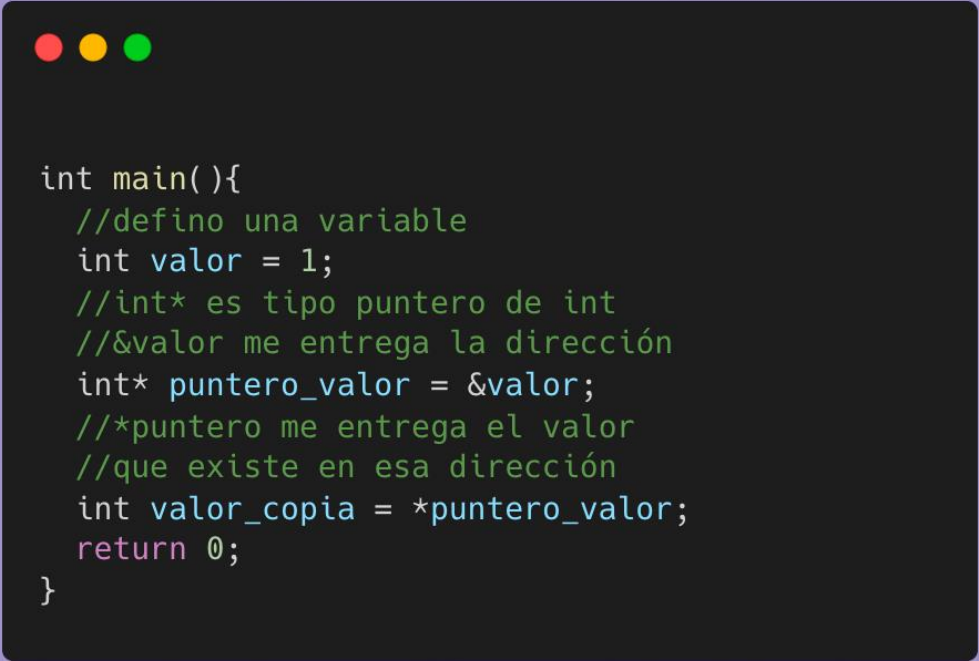
## **ampersand (&)**

- Se utiliza para **acceder a la dirección de memoria** de cualquier variable, ej:

***int\* foo = &valor;***

\*Muy importante notar que al definir el puntero, estamos entregando el tipo de la variable a la que apuntamos

# Sintaxis



```
int main(){  
    //defino una variable  
    int valor = 1;  
    //int* es tipo puntero de int  
    //&valor me entrega la dirección  
    int* puntero_valor = &valor;  
    //*puntero me entrega el valor  
    //que existe en esa dirección  
    int valor_copia = *puntero_valor;  
    return 0;  
}
```

# Sintaxis

## Ejecución

```
int A = 1;  
int* p = &A;  
printf("%i\n", A);  
printf("%i\n", *p);  
printf("%p\n", p);  
printf("%p\n", &A);
```



```
1  
1  
0x7ffeea2e023c  
0x7ffeea2e023c
```

\* Para imprimir una dirección de memoria, esta se debe especificar con el tipo 'p', de "pointer"

# ¿Para qué sirven los punteros?

```
void reiniciar(int* numero){  
    *numero = 0;  
}
```

```
int main() {  
    int n = 15;  
    reiniciar(&n);  
    printf("%d \n", n);  
    return 0;  
}
```

```
void reiniciar(int numero){  
    numero = 0;  
}
```

```
int main() {  
    int n = 15;  
    reiniciar(n);  
    printf("%d \n", n);  
    return 0;  
}
```

*¿Qué va a imprimir cada caso?*

# ¿Para qué sirven los punteros?

```
void reiniciar(int* numero){  
    *numero = 0;  
}
```

```
void reiniciar(int numero){  
    numero = 0;  
}
```

```
int main() {  
    int n = 15;  
    reiniciar(&n);  
    printf("%d \n", n);  
    return 0;  
}
```

```
int main() {  
    int n = 15;  
    reiniciar(n);  
    printf("%d \n", n);  
    return 0;  
}
```

*¿Qué va a imprimir cada caso?*

# ¿Para qué sirven los punteros?

```
void reiniciar(int* numero){  
    *numero = 0;  
}
```

```
int main() {  
    int n = 15;  
    reiniciar(&n);  
    printf("%d \n", n);  
    return 0;  
}
```



\* En C, las variables creadas dentro de una función sólo interactúan dentro del scope de la función

```
void reiniciar(int numero){  
    numero = 0;  
}
```

```
int main() {  
    int n = 15;  
    reiniciar(n);  
    printf("%d \n", n);  
    return 0;  
}
```



# ¿Para qué sirven los punteros? → *Por ejemplo: para llamar por referencia en una función*

```
void reiniciar(int* numero){  
    *numero = 0;  
}
```

```
int main() {  
    int n = 15;  
    reiniciar(&n);  
    printf("%d \n", n);  
    return 0;  
}
```

↓

0

```
void reiniciar(int numero){  
    numero = 0;  
}
```

```
int main() {  
    int n = 15;  
    reiniciar(n);  
    printf("%d \n", n);  
    return 0;  
}
```

↓

15



# Arreglos

Otro caso de uso de  
punteros

# Arreglos

- Es una lista donde todos los datos están consecutivos en memoria.
- Tiene un largo definido → **Inmutable** (muy importante)
- Para agregar o quitar un dato hay que **mover todos los elementos**.
- Todos los elementos son del **mismo tipo**

4
13
2
5
2
1

# Arreglos

Al definir un arreglo **type A[10]**, estamos reservando **10 espacios** en memoria de tamaño del **type**, luego la variable **A** es un puntero al primer elemento.

¿cómo accedemos a los elementos?

**A[offset]**

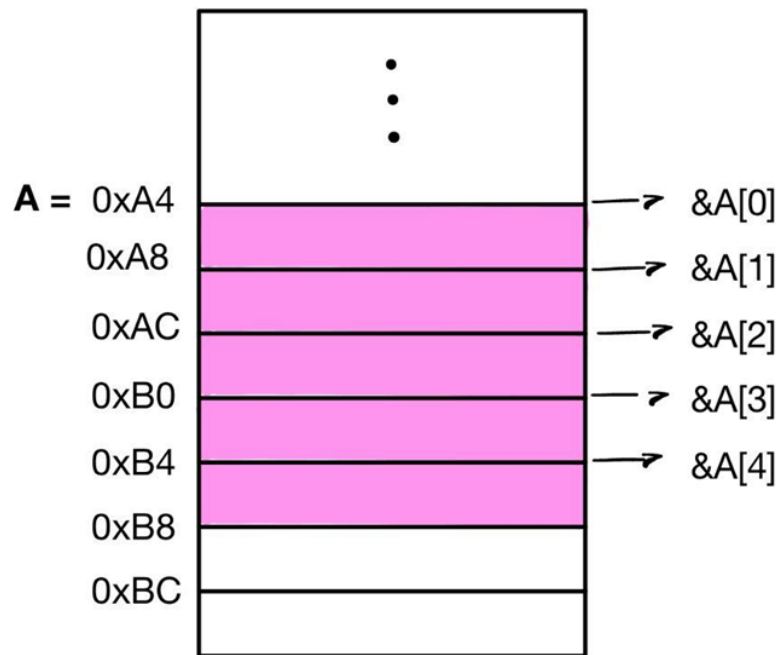
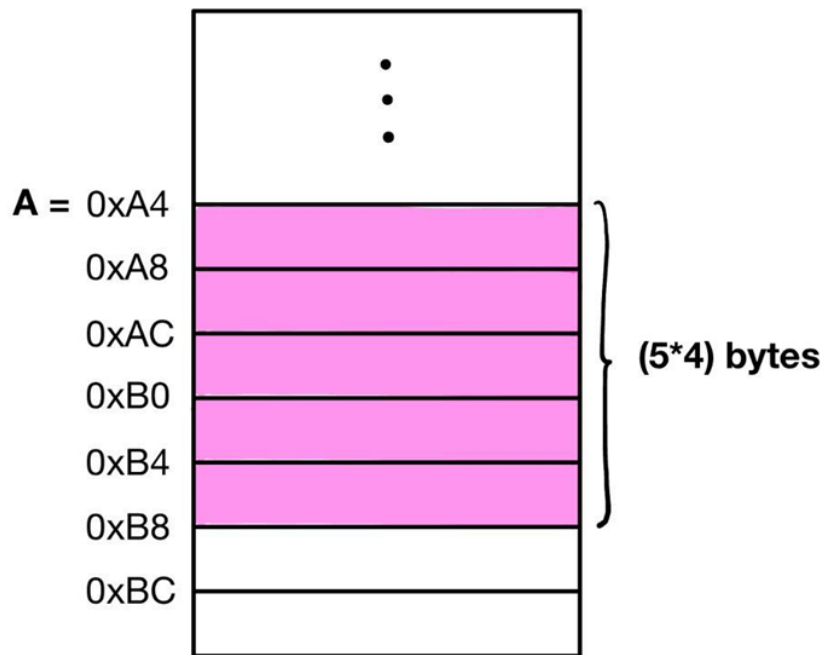
# Aritmética de punteros

$A[\mathbf{0}] = \text{dirección del primer elemento} + \mathbf{0} * \text{sizeof}(\text{type})$

$A[\mathbf{1}] = \text{dirección del primer elemento} + \mathbf{1} * \text{sizeof}(\text{type})$

$A[\mathbf{n}] = \text{dirección del primer elemento} + \mathbf{n} * \text{sizeof}(\text{type})$

# Ejemplo: int A[5]



\*El tamaño de 'int' es 4 bytes

# Arreglos

```
int A[10];  
printf("A = %p\n &A[0] = %p\n &A = %p\n", A, &A[0], &A);
```



```
A = 0xfc  
&A[0] = 0xfc  
&A = 0xfc
```

# Arreglos

Formas de definir arreglos:

- `type A[] = {elem_0, elem_1, elem_2, ..., elem_n}`
- `type A[10];`
- `type A[10] = {elem_0, elem_1, elem_2, ..., elem_9}`

\*Notar que en cada caso, el tamaño del arreglo está definido

# Strings



# Strings

- No existen los strings como tal en C, son arreglos de **char** o caracteres.
- Todos los strings terminan (es decir, el último elemento del arreglo debe ser) un null terminator (carácter de control, el cero).
- Se pueden declarar implícitamente (como en los primeros ejemplos)

```
#include <stdio.h>
#include <stdbool.h>

int main(){

    // Distintas formas de inicializar un string

    char hello[] = "Hello, World";

    char hello[20] = "Hello, World";

    char* hello = "Hello, World";

    char hello[] = {"H", "e", "l", "l", "o", ",", " ", "W", "o", "r", "l", "d", "\0"};

    char hello[20] = {"H", "e", "l", "l", "o", ",", " ", "W", "o", "r", "l", "d", "\0"};

    return 0;
}
```