

# Introducción a C

---

Lenguaje, Setup y Conceptos

**Importante:**

**Ver las cápsulas de C completas**

**Estimamos que este viernes se publicará la  
Tarea 0**

# ¿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 *orientado a estructuras*, Python es *orientado a objetos*
2. C es *compilado*, Python es *interpretado*
3. C tiene *punteros*
4. C es *fuertemente tipado*

*helloworld.c*

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



## Ejecución

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

*helloworld.py*

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



## Ejecución

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

*main.py*




```
def add_floats(x, y):  
    return x + y  
  
def main():  
    x = 2.5  
    y = 2  
  
    print(f"first_number: {x}")  
    print(f"second_number: {y}")  
    print(f"result = {add_floats(x,y)}")  
  
    return
```

## Ejecución

```
> python3 main.py
```

```
first_number: 2.5  
second_number: 2  
result = 5.0
```

## main.c



```
#include <stdio.h>

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

int main() {
    float x = 2.5;
    float y = 2;

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

    return 0;
}
```

## Ejecución

```
> gcc main.c -o main && ./main
first_number: 2.50000001
second_number: 2.0
result = 5.00000001
```

*main.c*



```
#include <stdio.h>

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

int main() {
    float x = 2.5;
    float y = 2;

    printf("first_number %f\n", x);
    printf("second_number %f\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 = 2.5
    y = 2

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

    return
```





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

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

```
int main() {  
    float x = 2.5;  
    float y = 2;  
  
    printf("first_number %f\n", x);  
    printf("second_number %f\n", y);  
    printf("result = %f\n", add_floats(x,y));  
  
    return 0;  
}
```

Se debe incluir la librería que gestiona el input/output



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

Se debe incluir la librería que gestiona el input/output

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

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

```
int main() {  
    float x = 2.5;  
    float y = 2;  
  
    printf("first_number %f\n", x);  
    printf("second_number %f\n", y);  
    printf("result = %f\n", add_floats(x,y));  
  
    return 0;  
}
```



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

Se debe incluir la librería que gestiona el input/output

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

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

```
int main() {  
    float x = 2.5;  
    float y = 2;  
  
    printf("first_number %f\n", x);  
    printf("second_number %f\n", y);  
    printf("result = %f\n", add_floats(x,y));  
  
    return 0;  
}
```

Las variables deben declarar su tipo



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

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

```
int main() {  
    float x = 2.5;  
    float y = 2;
```

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

```
    return 0;  
}
```

Se debe incluir la librería que gestiona el input/output

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



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

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

```
int main() {  
    float x = 2.5;  
    float y = 2;
```

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

```
    return 0;  
}
```

Se debe incluir la librería que gestiona el input/output

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

***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*



```
#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;
}
```

*int*



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



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



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

# Funciones

***funciones con retorno*** - Funciones en C que poseen un valor de retorno junto con un tipo explícito del mismo


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

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

***declarar*** - Indicar el nombre de la función y sus tipos.

# 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

# Punteros

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

# Punteros

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

*¿Hay una solución más eficiente?*

**Ejemplo**

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

*Nosotros no sabemos en qué lugar del  
guardarropía está nuestro polerón.*

**Ejemplo**

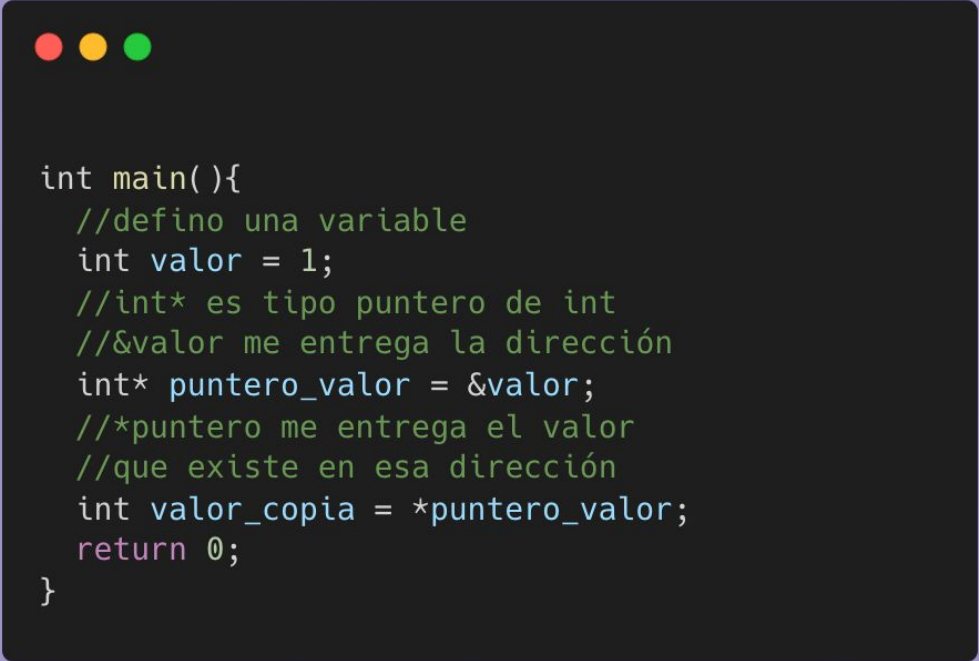
*En este ejemplo el polerón es el **valor** y el  
papelito es la **dirección**.*

**Ejemplo**

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

**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

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



## Ejecución

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

# Arreglos

# Arreglos

- Es una lista donde todos los datos están consecutivos en memoria.
- Tiene un largo definido→ **Inmutable**
- 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

# 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.
- Se pueden declarar implícitamente

```
#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"};

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

    return 0;
}
```

# Structs

# Structs

- Es una **colección de variables** (pueden ser de diferentes tipos) bajo un sólo nombre.
- A semeja una **clase** de python, pero sin las funciones propias (piensa en los **atributos de la clase**)
  - Disclaimer: **NO** existen las clases en C

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

typedef struct dog {
    char* name;
    char* breed;
    bool good_boy;
    int age;
} Dog;

int main(){

    Dog my_dog;
    my_dog -> name = "Frodo";
    my_dog -> breed = "Jack Russell";
    my_dog -> good_boy = true;
    my_dog -> age = 4;

    return 0;
}
```



# Memoria

## ***Un programa utilizar 2 tipos de memoria:***

- ***Stack:*** *Es el sector de memoria asignado para el programa. En este van variables, funciones, contextos, etc. Es de tamaño fijo.*
- ***Heap:*** *A diferencia del Stack, no posee ninguna estructura de asignación de espacios. Es una colección de bloques de memoria que fueron solicitados por el programa. Estos bloques pueden ser de distinto tamaño.*

# **Memoria**

*Para interactuar con el HEAP, existe la librería `<stdlib.h>` que trae las funciones:*

- *malloc*
- *calloc*
- *free*

# Memoria

## ***malloc: Memory Allocation***

*Recibe la cantidad de bytes a pedir al HEAP y retorna un puntero.*

```
int a = 4;  
int* b = malloc(a * sizeof(int));  
printf("%p\n", b);
```

```
$ gcc main.c -o main  
$ ./main  
0xfa20fc
```

# Memoria

## ***calloc: Clear Memory Allocated***


*Recibe la cantidad de elementos y su tamaño para pedir esa cantidad al HEAP y retorna un puntero.*

***(RECOMENDADO)***

```
int a = 4;  
int* b = calloc(a, sizeof(int));  
printf("%p\n", b);
```

```
$ gcc main.c -o main  
$ ./main  
0x522d040
```

# Memoria



```
if (pais.ciudad) {  
    pais.ciudad -> alcalde = "Joaquín Lavín";  
}
```

Si se inicializó la clase (struct) con un malloc y no con un calloc, podría pasar que **pais.ciudad** no este vacío ya que habrá valores random rellenandolo

# Memoria

## ***free: Memory Deallocation***

*Cuando terminamos de usar los bloques del HEAP, tenemos que liberarlos manualmente usando la función free.*

```
int a1 = 4;
int* b1 = malloc(a1 * sizeof(int));

int a2 = 4;
int* b2 = calloc(a2, sizeof(int));

free(b1);
free(b2);
```

# Memoria

## Arreglos en el HEAP

- Podemos crear arreglos con malloc y calloc en el HEAP.

```
int* A = malloc(3 * sizeof(int));  
A[0] = 1;  
A[1] = 4;  
A[2] = 3;  
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

?

# Memoria



## Arreglos en el HEAP

- Podemos crear arreglos con `malloc` y `calloc` en el HEAP.

```
int* A = malloc(3 * sizeof(int));  
A[0] = 1;  
A[1] = 4;  
A[2] = 3;  
printf("%p\n%p\n%p\n", A, &A, &A[0]);
```

```
$ gcc main.c -o main  
$ ./main  
0x22d040  
0x01fd2b  
0x22d040
```

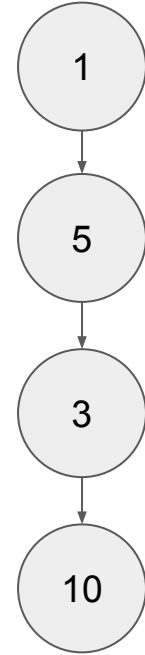


# Memoria

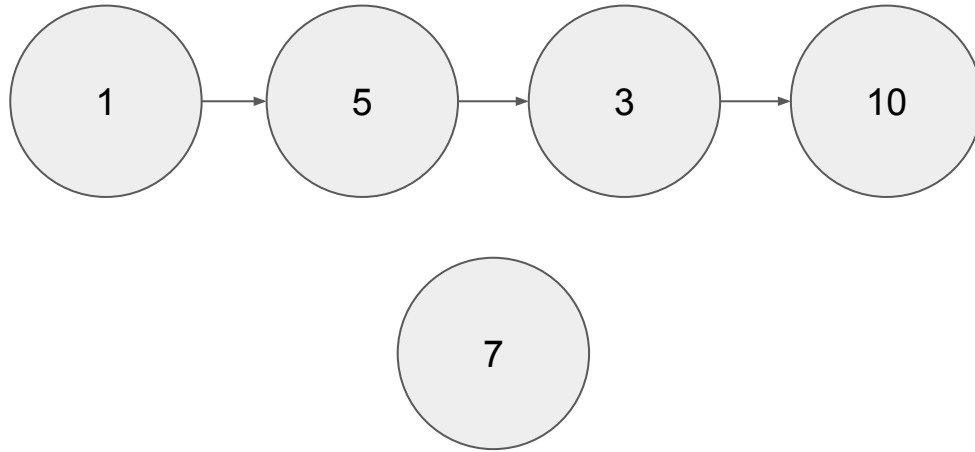
# Listas Ligadas

# Listas Ligadas

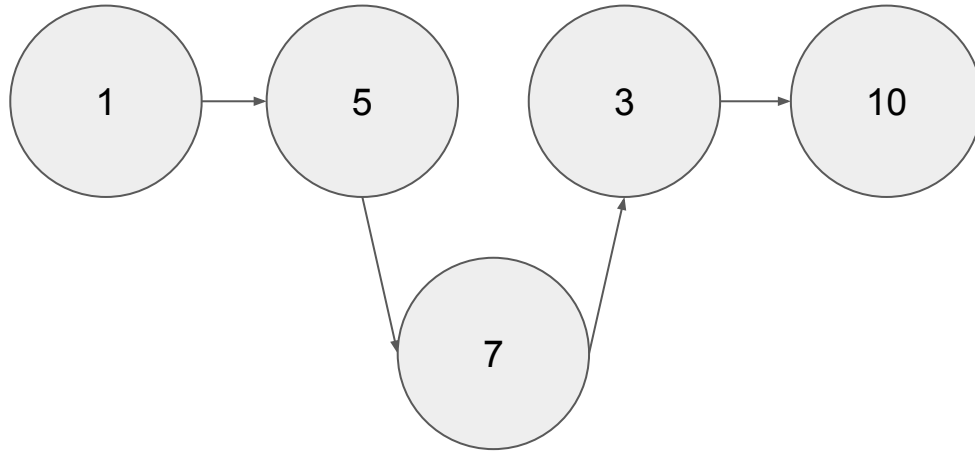
- Es una estructura organizada en nodos.
- Cada nodo guarda una referencia al nodo siguiente y un valor.
- Tiene un largo variable.
- Es fácil insertar datos nuevos.
- Es difícil buscar datos específicos por índice (se deben recorrer todos los nodos anteriores.)



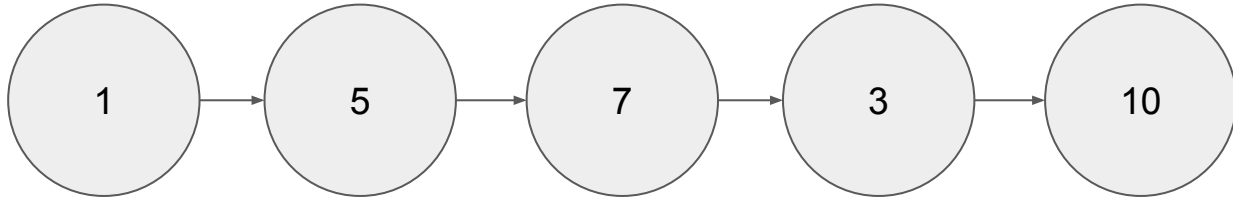
# Inserción en Listas Ligadas



# Inserción en Listas Ligadas



# Inserción en Listas Ligadas



# Listas Ligadas

```
#include <stdio.h>

typedef struct node {
    int value;
    struct node *next;
} Node;

void print_linked_list(Node* head) {
    while (head != NULL) {
        printf("Value in node %d \n", head ->value);
        head = head -> next;
    }
}

int main(){
    // Inicializamos los nodos

    Node* one = NULL;
    Node* two = NULL;
    Node* three = NULL;
```

```
// Allocar memoria
one = calloc(1, sizeof(Node));
two = calloc(1, sizeof(Node));
three = calloc(1, sizeof(Node));

// Asignamos valores

one -> value = 1;
two -> value = 2;
three -> value = 3;

// Conectamos los nodos

one -> next = two;
two -> next = three;
three -> next = NULL;

print_linked_list(one);

return 0;
}
```

# Matrices



# Matrices

- ¡Es un **array de arrays**!
- Podemos usar **Arreglos** para guardar punteros.
- Como los **Arreglos** son punteros, un **Arreglo de Arreglos** no es más que un **Arreglo de punteros**.

# Matrices

```
#include <stdio.h>

int main(){
    // Una forma de inicializar una matriz
    int matrix[3][2] = {{1, 1}, {2, 2}, {3, 3}};
    // Matriz 3x2

    // Otra forma de inicializar una matriz con punteros

    int* matrix[3];
    for(int i; i<3; i++){
        int array[2] = {i+1, i+1};
        matrix[i] = array;
    };

    // Matrix 3x2
```

# Matrices

```
// Otra forma de inicializar una matriz con punteros a punteros en el heap
int matrix_size = 3;
int** matrix = calloc(matrix_size, sizeof(int*));

for(int row = 0; row < matrix_size; row++) {
    matrix[row] = calloc(matrix_size, sizeof(int*));

    for(int column = 0; column < matrix_size; column++) {
        matrix[row][column] = row + 1;
    }
}
// Matrix 3x3

return 0;
}
```

# Valgrind

## ***¿Qué es?***

- *Un set de herramientas para analizar y monitorear el uso de recursos de un programa.*
- *Ejecuta el programa en un ambiente virtual.*
- *Aumenta la ejecución hasta 40 veces.*

# **Valgrind**

## ***Herramienta principal de Valgrind***

- *Permite visualizar el estado de la memoria.*
- *Revisar leaks y errores en el uso de la memoria.*

**Memcheck**

# Modularizar

## Importación de módulos

### **#include <global.h>**

Para librerías o módulos globales instalados en el PC que compila el programa.

### **#include "local.h"**

Para módulos locales que se encuentran en la misma carpeta que el archivo que se importa. Ruta relativa



dog.h



```
#include <stdbool.h>

typedef struct dog {
    char* name;
    char* breed;
    bool good_boy;
    int age;
} Dog;

Dog* dog_init(char* name, char* breed, int age);
void dog_bark(Dog* dog);
```

1. Definición del struct
2. Declaración de funciones del struct

dog.c



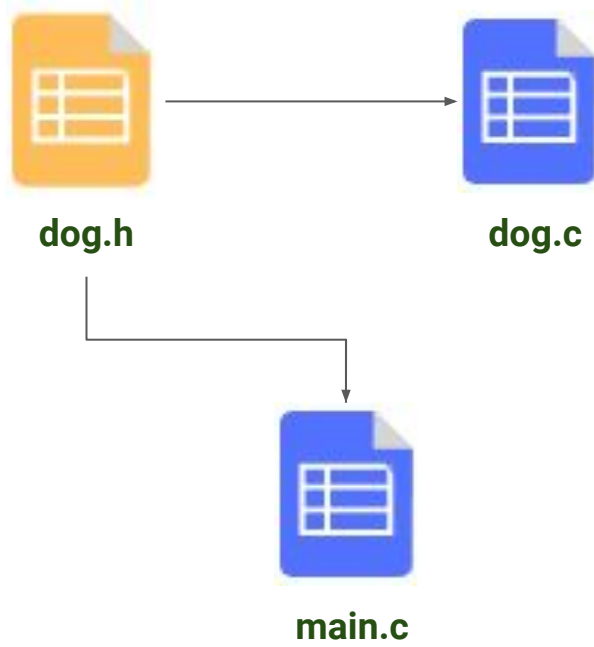
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "dog.h"

Dog* dog_init(char* name, char* breed, int age) {
    Dog* dog = calloc(1, sizeof(Dog));
    *dog = (Dog) {
        .name = name,
        .breed = breed,
        .good_boy = true,
        .age = age,
    };

    return dog;
}

void dog_bark(Dog* dog) {
    printf("My name is %s\n", dog->name);
}
```

Definición de funciones



```
#include <stdbool.h>

typedef struct dog {
    char* name;
    char* breed;
    bool good_boy;
    int age;
} Dog;

Dog* dog_init(char* name, char* breed, int age);
void dog_bark(Dog* dog);
```

dog.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "dog.h"

Dog* dog_init(char* name, char* breed, int age) {
    Dog* dog = calloc(1, sizeof(Dog));
    *dog = (Dog) {
        .name = name,
        .breed = breed,
        .good_boy = true,
        .age = age,
    };
    return dog;
}

void dog_bark(Dog* dog) {
    printf("My name is %s\n", dog->name);
}
```

dog.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "dog.h"

int main(){
    Dog* my_dog = dog_init("Frodo", "Jack Russel", 4);
    dog_bark(my_dog);
    return 0;
}
```

main.c

```
#include <stdbool.h>

typedef struct dog {
    char* name;
    char* breed;
    bool good_boy;
    int age;
} Dog;

Dog* dog_init(char* name, char* breed, int age);
void dog_bark(Dog* dog);
```

dog.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "dog.h"

Dog* dog_init(char* name, char* breed, int age) {
    Dog* dog = calloc(1, sizeof(Dog));
    *dog = (Dog) {
        .name = name,
        .breed = breed,
        .good_boy = true,
        .age = age,
    };
    return dog;
}

void dog_bark(Dog* dog) {
    printf("My name is %s\n", dog->name);
}
```

dog.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "dog.h"

int main(){
    Dog* my_dog = dog_init("Frodo", "Jack Russel", 4);
    dog_bark(my_dog);
    return 0;
}
```

main.c

# Make

C, es un lenguaje compilado, esto quiere decir que **se debe traducir al código de máquina para producir un programa ejecutable.**

Make es el comando encargado de hacerlo, mientras que un archivo Makefile es el encargado de tener las instrucciones por defecto que podría tener la compilación de un programa.

A ustedes se le entregará en sus tareas un Makefile, hecho por el cuerpo docente.

En caso de querer cambiar la optimización del código (para que sea más eficiente) al compilar, pueden cambiar el archivo **SOLO** entre las líneas 20 y 24 inclusive.

La línea que quede des-comentada será la cual se considere para compilar.

```
20  OPT=-g # Guardar toda la información para poder debugear. No optimiza
21  # OPT=-O0 # No optimiza.
22  # OPT=-O1 # Optimiza un poquito
23  # OPT=-O2 # Optimiza bastante
24  # OPT=-O3 # Optimiza al máximo. Puede ser peor que -O2 según tu código
```