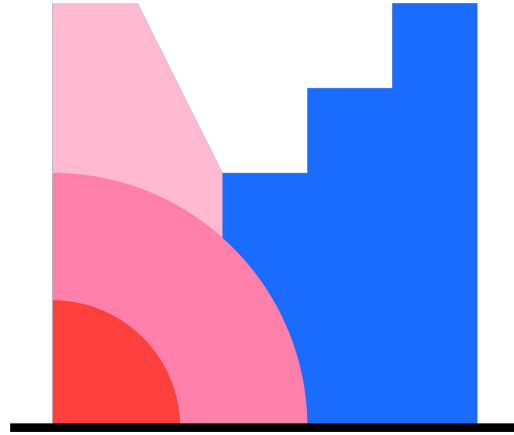


# Introducción a C

---

Lenguaje, Setup y Conceptos (parte 2)

# Momento menti



Mentimeter

# 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**

Stack

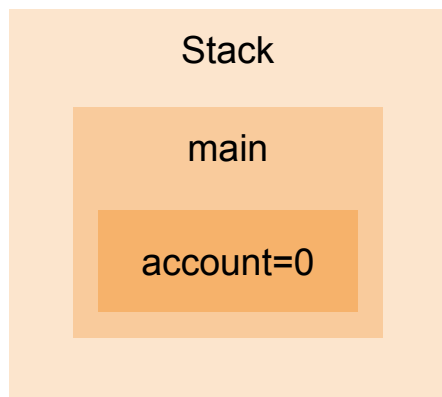
main

account=0

```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account account, int amount){  
    account.balance += amount;  
}
```

```
int main() {  
    Account acc;  
    acc.balance = 0;  
    printf("%d , %p\n", acc.balance, &acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc.balance, &acc);  
}
```

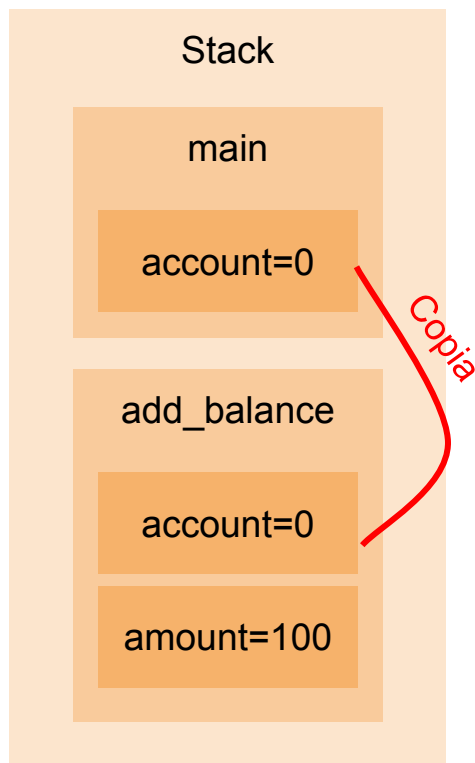


```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account account, int amount){  
    account.balance += amount;  
}
```

Imprime 0 y  
el puntero

```
int main() {  
    Account acc;  
    acc.balance = 0;  
    printf("%d , %p\n", acc.balance, &acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc.balance, &acc);  
}
```



```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account account, int amount){  
    account.balance += amount;  
}
```

```
int main() {  
    Account acc;  
    acc.balance = 0;  
    printf("%d , %p\n", acc.balance, &acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc.balance, &acc);  
}
```

Stack

main

account=0

add\_balance

account=100

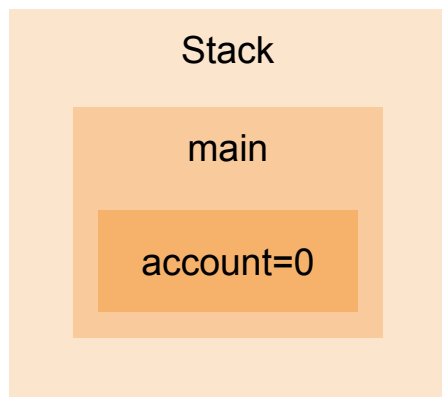
amount=100

```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account account, int amount){  
    account.balance += amount;  
}
```

```
int main() {  
    Account acc;  
    acc.balance = 0;  
    printf("%d , %p\n", acc.balance, &acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc.balance, &acc);  
}
```





```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account account, int amount){  
    account.balance += amount;  
}
```

```
int main() {  
    Account acc;  
    acc.balance = 0;  
    printf("%d , %p\n", acc.balance, &acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc.balance, &acc);  
}
```

Imprime 0 y  
el puntero



Stack

main

account=0xad

Alocar memoria  
(malloc)

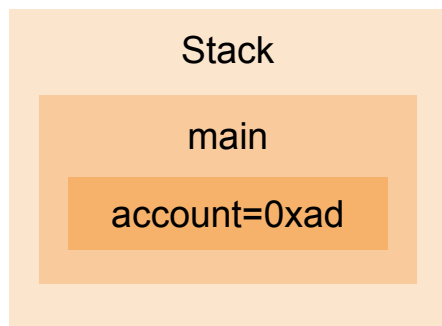
Heap

0xad=0

```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account *account, int amount){  
    account -> balance += amount;  
}
```

```
int main() {  
    Account* acc;  
    acc = malloc(sizeof(Account));  
    acc -> balance = 0;  
    printf("%d , %p\n", acc -> balance, acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc -> balance, acc);  
    free(acc);  
}
```

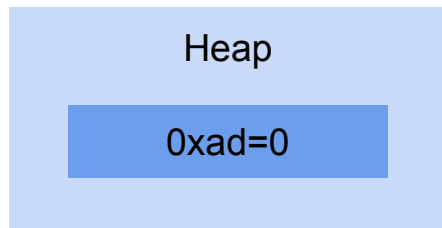


```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account *account, int amount){  
    account -> balance += amount;  
}
```

Imprime 0 y  
el puntero

Alocar memoria  
(malloc)



```
int main() {  
    Account* acc;  
    acc = malloc(sizeof(Account));  
    acc -> balance = 0;  
    printf("%d , %p\n", acc -> balance, acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc -> balance, acc);  
    free(acc);  
}
```

Stack

main

account=0xad

add\_balance

account=0xad

amount=100

Heap

0xad=0

```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account *account, int amount){  
    account -> balance += amount;  
}
```

```
int main() {  
    Account* acc;  
    acc = malloc(sizeof(Account));  
    acc -> balance = 0;  
    printf("%d , %p\n", acc -> balance, acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc -> balance, acc);  
    free(acc);  
}
```

Stack

main

account=0xad

add\_balance

account=0xad

amount=100

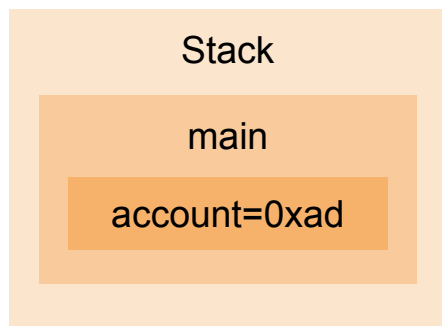
Heap

0xad=100

```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account *account, int amount){  
    account -> balance += amount;  
}
```

```
int main() {  
    Account* acc;  
    acc = malloc(sizeof(Account));  
    acc -> balance = 0;  
    printf("%d , %p\n", acc -> balance, acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc -> balance, acc);  
    free(acc);  
}
```

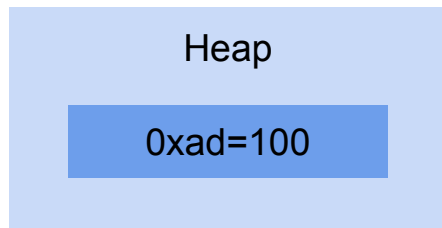


```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account *account, int amount){  
    account -> balance += amount;  
}
```

```
int main() {  
    Account* acc;  
    acc = malloc(sizeof(Account));  
    acc -> balance = 0;  
    printf("%d , %p\n", acc -> balance, acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc -> balance, acc);  
    free(acc);  
}
```

Imprime 100  
y el puntero





¡Tenemos que liberar la memoria del programa!

Heap

0xad=100

```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account *account, int amount){  
    account -> balance += amount;  
}
```

```
int main() {  
    Account* acc;  
    acc = malloc(sizeof(Account));  
    acc -> balance = 0;  
    printf("%d , %p\n", acc -> balance, acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc -> balance, acc);  
    free(acc);  
}
```

Heap

A diagram showing a light blue rectangular area labeled 'Heap'. Inside this area, there is a smaller, darker blue rectangular block, representing a memory allocation on the heap.

```
typedef struct account{  
    int balance;  
} Account;
```

```
void add_balance(Account *account, int amount){  
    account -> balance += amount;  
}
```

```
int main() {  
    Account* acc;  
    acc = malloc(sizeof(Account));  
    acc -> balance = 0;  
    printf("%d , %p\n", acc -> balance, acc);  
    add_balance(acc, 100);  
    printf("%d , %p\n", acc -> balance, acc);  
    free(acc);  
}
```



*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 esté 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
```



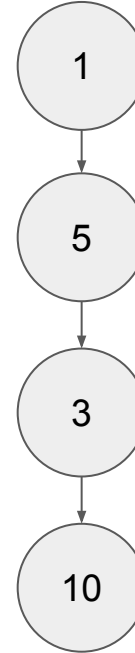
# Memoria

# Listas Ligadas

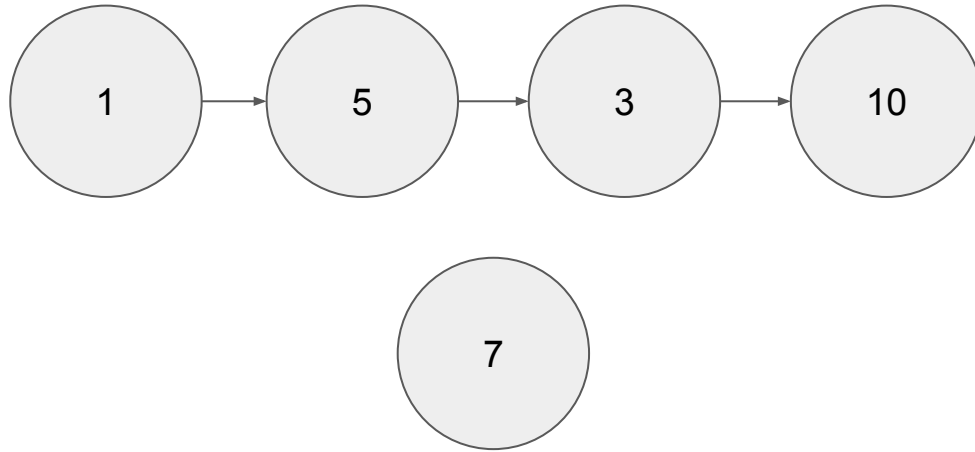


# Listas Ligadas

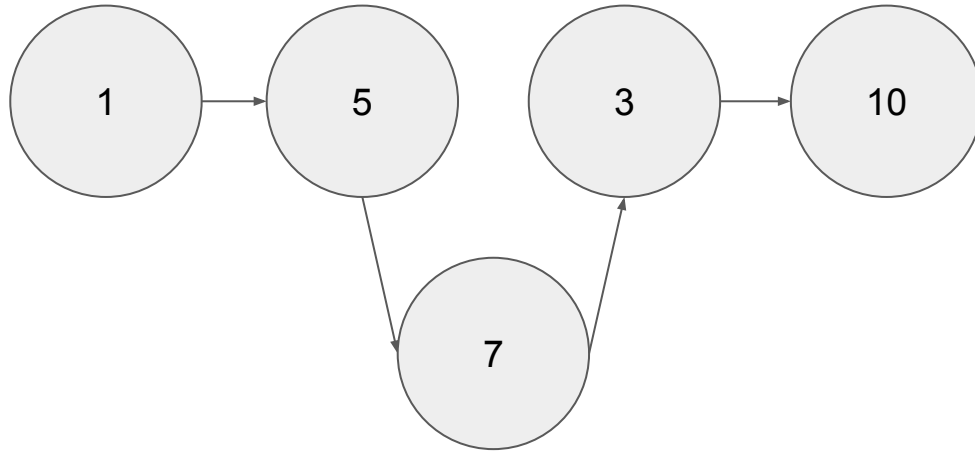
- Es una estructura organizada en nodos.
- Cada nodo guarda una referencia al nodo siguiente y un valor.
- Tiene un largo variable.
- Es fácil insertar datos nuevos.
- Es difícil buscar datos específicos por índice (se deben recorrer todos los nodos anteriores).



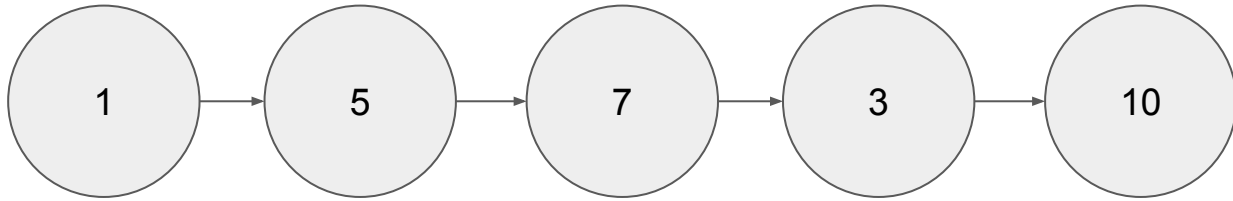
# Inserción en Listas Ligadas



# 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;
}
```

```
int** A = calloc(2, sizeof(int*));  
A[0] = calloc(3, sizeof(int));  
A[1] = calloc(3, sizeof(int));  
A[0][0] = 2;  
A[0][1] = 27;  
A[1][0] = 6;  
A[1][1] = 3;  
A[1][2] = 5;
```

- *int\*\* A es puntero de punteros*
- *A[0] y A[1] son punteros de int*

# Valgrind

# 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 el tiempo de ejecución hasta 40 veces

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

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

**Memcheck**

## ***Uso y comando útiles***

*Valgrind se vale de agregar otros comandos para generar outputs que nos puedan indicar cosas que están pasando en nuestro código que sean de nuestro interés.*

*La forma general de usar Valgrind en la terminal se vería como:*  
***valgrind ./nombre\_ejecutable input.txt output.txt***

*Para memory leaks añadimos:*

***valgrind -- leak-check=full ./nombre\_ejecutable input.txt output.txt***

# **Memcheck**

## ***Uso y comando útiles***

*Notar que también existen otras extensiones, como "-v" que proviene de verbose.*

***valgrind -v ./nombre\_ejecutable input.txt output.txt***

*Si bien esta extensión puede ser útil en otros contextos, en particular para asuntos del curso no la ocuparemos pues nos centraremos en el uso de valgrind para la búsqueda de **no liberación de memoria** al finalizar un programa y **malas** alocaiones de 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



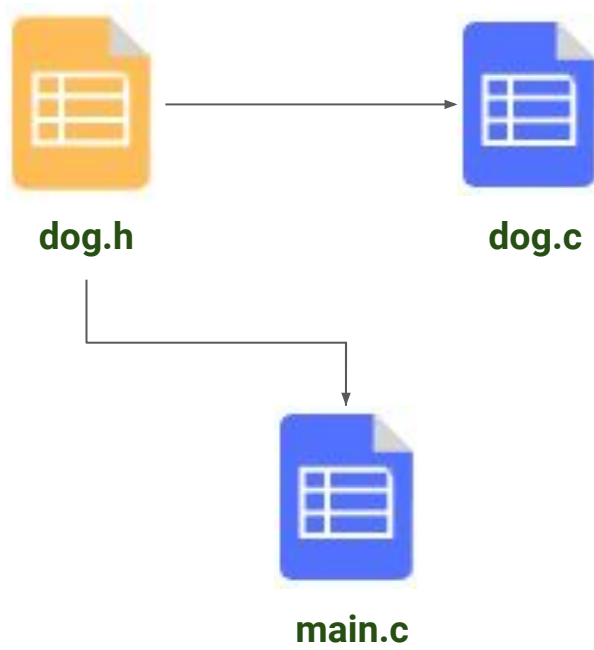
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "dog.h"

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 sera 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
```



Qué es importante que tengan en cuenta?

Cuando nosotros les entregamos un makefile, implica que el proceso de compilado es un **poco** distinto. *(Esto también tómelo como un hint para que no se queden pegados intentando compilar+correr las tareas)*

Veamos las diferencias!



# Compilando y ejecutando mi primer programa!

## Con makefile



```
make clean && make
```

## Sin makefile (gcc)

```
gcc ruta_archivo_main  
-o nombre_ejecutable
```



## Ahora que compilamos, ejecutamos normalmente

```
./nombre_ejecutable input.txt output.txt
```

# Recapitulación Taller C parte II: Qué vimos hoy?

- Uso de memoria de un programa
- Listas ligadas
- Matrices
- Valgrind
- Modularización
- Make, Makefile



# Extra: Ejercicios de práctica

El día lunes 11 de marzo se publicarán ejercicios para facilitar el aprendizaje de C

Orden recomendado de realizarlos:

- 3 problemas introductorios
- Cuando pido memoria como loco
- Termina tu primer ABB

