

Introducción a C

Lenguaje, Setup y Conceptos

6683 1676



¿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 → `gcc nombre_archivo.c -o nombre_ejecutable`

Correr → `./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.py

```
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 ("
\\n")

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



```
#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.

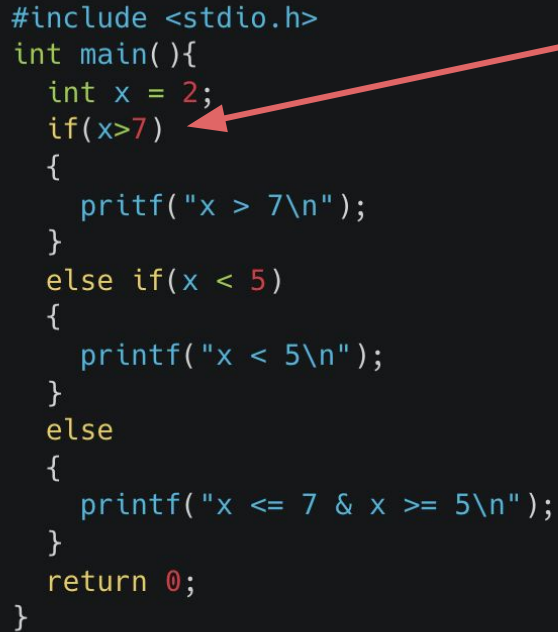
while - ejecución repetida mientras se mantenga la veracidad de la condición.

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 (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**?



*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;`**

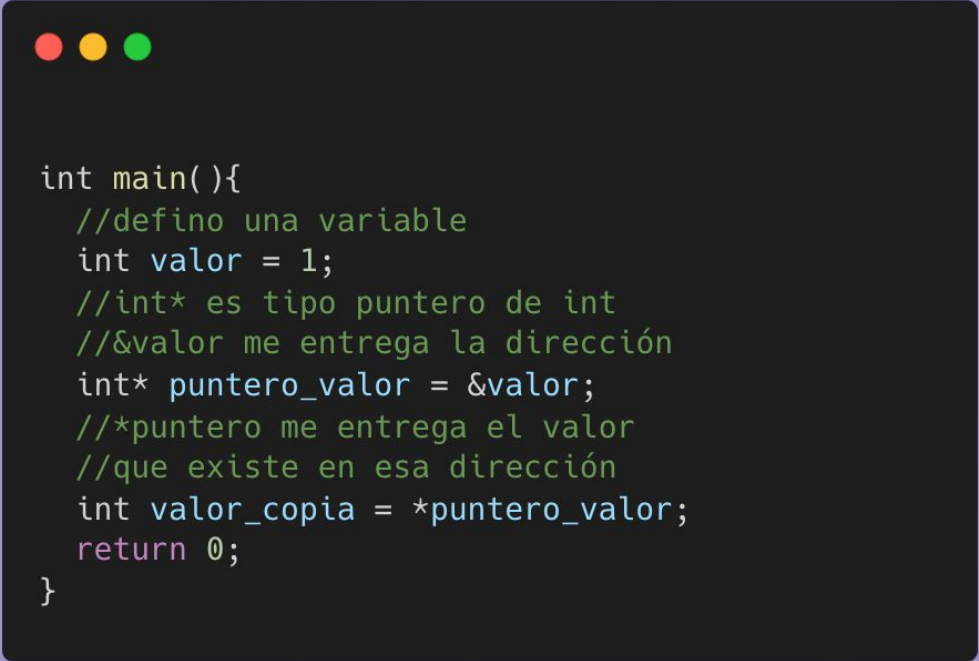
ampersand (&)

- 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

```
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
```

* 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;  
}
```



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;  
}
```



15

¿Para qué sirven los punteros? →

Por ejemplo: para llamar por referencia en una función

```
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;  
}
```



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]

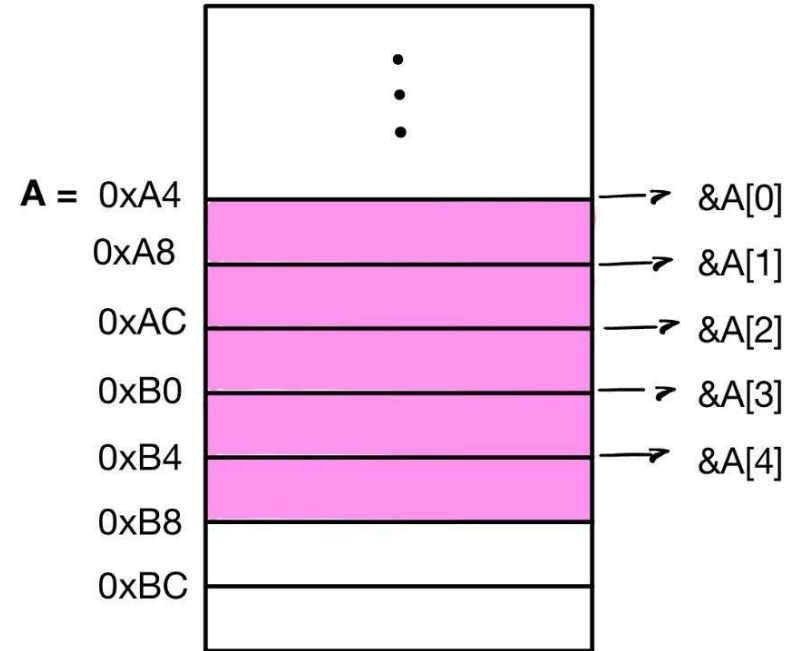
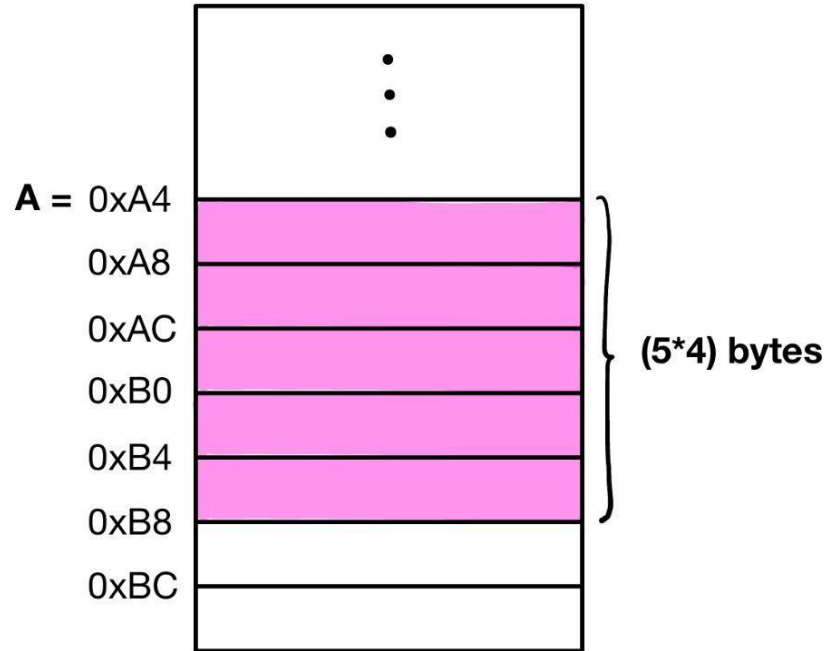
Aritmética de punteros

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

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

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

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[] = {"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;
}
```

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

```
1  #include <stdbool.h>
2  #include <stdlib.h>
3
4  typedef struct gatito {
5
6      char * nombre;
7      bool desordenao;
8      int edad;
9
10 } Gato;
11
12
13 int main() {
14
15     Gato mi_gato;
16
17     mi_gato.nombre = "harry";
18     mi_gato.desordenao = true;
19     mi_gato.edad = 4;
20
21     return 0;
22 }
23
```

Recapitulación Taller C parte I: Qué vimos hoy?

- **Diferencias entre C y python** (destacando bondades de C)
- **Sobre C:**
 - Tipos de datos
 - Flujo
 - Funciones
 - Punteros
 - Arreglos
 - Strings
 - Structs

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. Tiene un límite en su tamaño (stackoverflow).*
- ***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

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

Stack

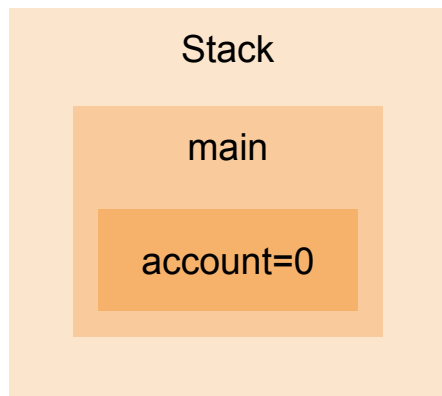
main

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);  
}
```

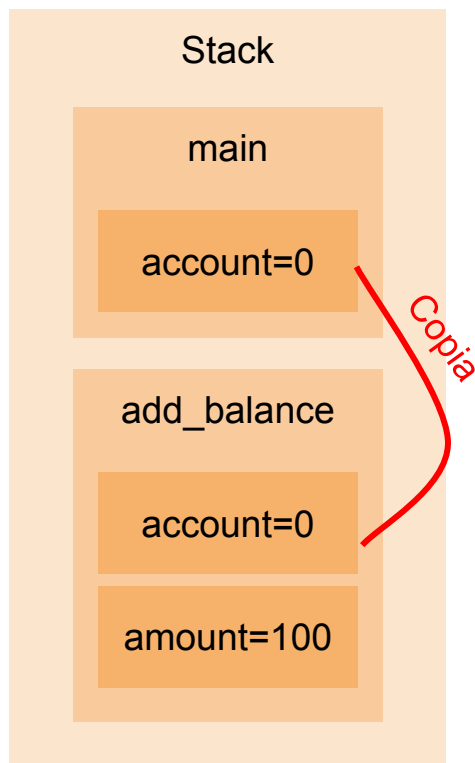


```
typedef struct account{  
    int balance;  
} Account;
```

```
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

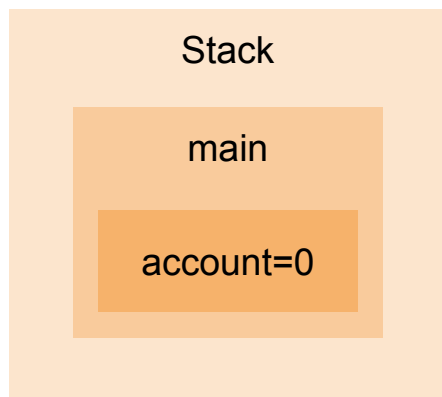
account=100

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)

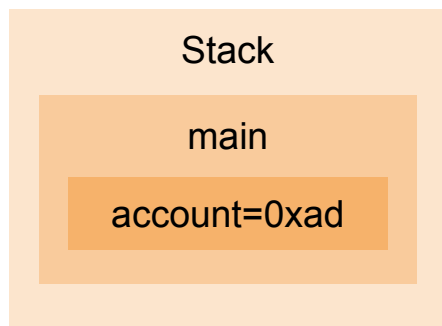
Heap

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);  
}
```

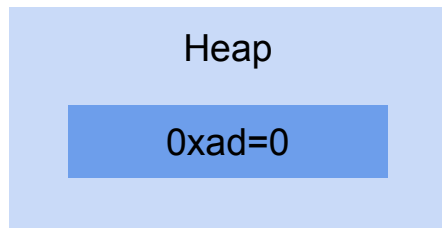


```
typedef struct account{  
    int balance;  
} Account;
```

```
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);  
}
```

Stack

main

account=0xad

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

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

Imprime 100
y el puntero

Heap

0xad=100

```
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);  
}
```

¡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



```
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);  
}
```

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

```
int main() {  
  
    int* A = malloc(3 * sizeof(int));  
    A[0] = 1;  
    A[2] = 3;  
    printf("%d\n%d\n%d\n", A[0], A[1], A[2]);  
  
    return 0;  
}
```

Ojo que malloc inicializa los valores con números random, a diferencia de calloc que los inicializa en 0

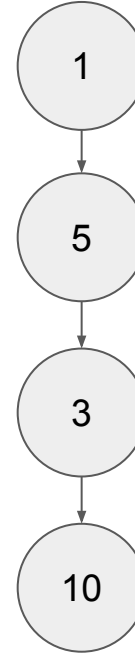
| | | |
|---|-------|---|
| 1 | -1498 | 3 |
|---|-------|---|

Memoria

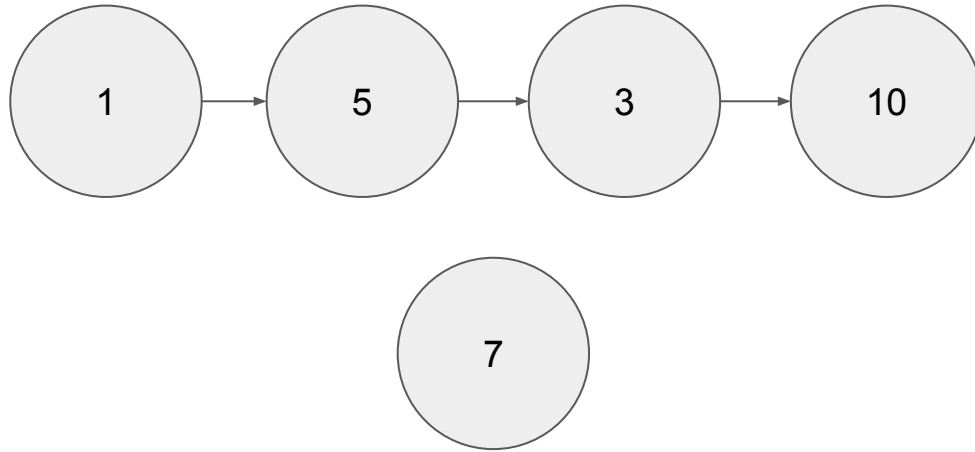
Listas Ligadas

Listas Ligadas

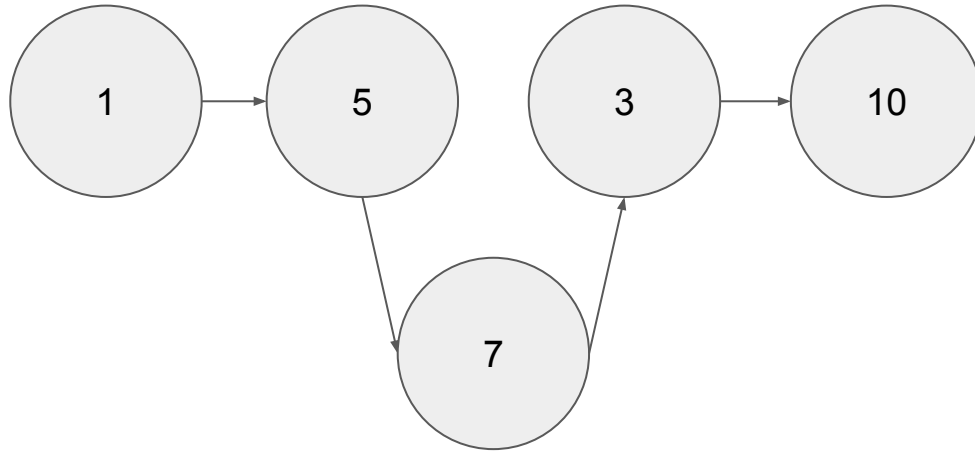
- Es una estructura organizada en nodos.
- Cada nodo guarda una referencia al nodo siguiente y un valor.
- Tiene un largo variable.
- Es fácil insertar datos nuevos.
- Es difícil buscar datos específicos por índice (se deben recorrer todos los nodos anteriores).



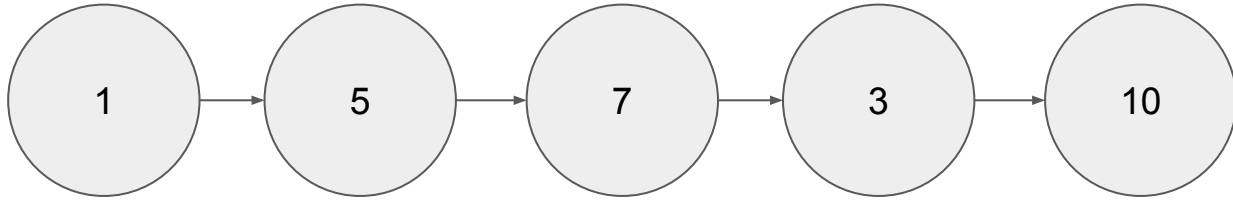
Inserción en Listas Ligadas



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;

// Connectamos 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 a punteros*
- *A[0] y A[1] son punteros a ints*

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



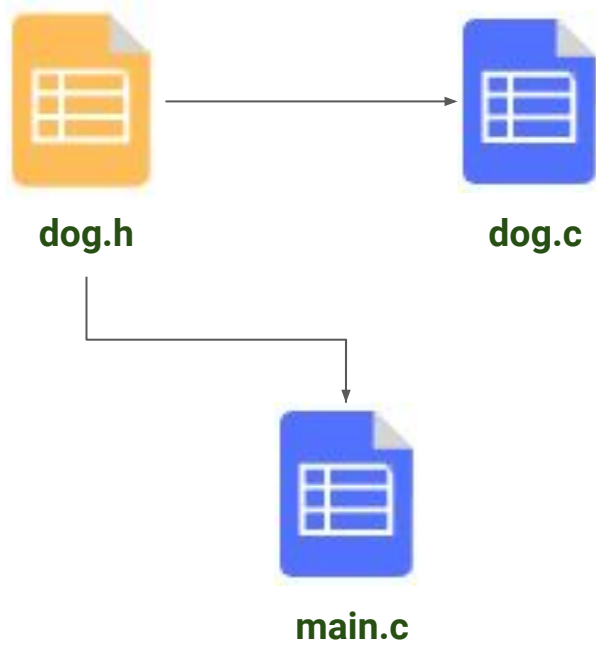
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "dog.h"

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

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 viernes 9 de agosto 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

