



# Lenguaje C

## Algoritmos y programación I

Capra - Chaves - Di Matteo - Lanzillotta -  
Sosa - Villegas - Palavecino



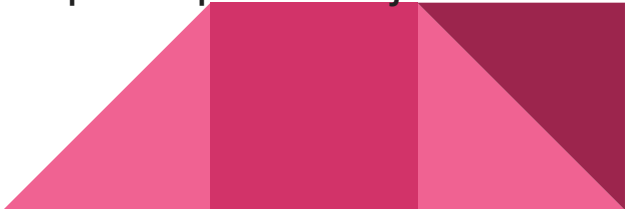
# ¿Qué vamos a ver?

- Funcionamiento de un compilador
- Lo básico de C
- Funciones y procedimientos
- Structs
- Arrays
- Strings
- Archivos (tal vez)



# Un “poco” de historia

“C es un lenguaje de programación de propósito general que ofrece como ventajas economía de expresión, control de flujo y estructuras de datos modernas y un rico conjunto de operadores. Además, no es un lenguaje de muy alto nivel ni grande, y no está especializado en ninguna aplicación particular. Pero su ausencia de restricciones y su generalidad lo hacen más conveniente y efectivo para muchas tareas que otros lenguajes supuestamente más poderosos. Originalmente, C fue diseñado para el sistema operativo Unix. El sistema operativo, el compilador de C y esencialmente todos los programas de aplicación de Unix están escritos en C. El lenguaje C no está ligado a ningún hardware o sistema en particular y es fácil escribir programas que corran sin cambios en cualquier máquina que maneje C.”



# Un poco de historia...



C Nació en 1972, su creador fue Dennis Ritchie.

En 1978 se publicó la primera edición del libro “El lenguaje de programación C” cuyos autores fueron Brian Kernighan y Dennis Ritchie (el mismo que arriba).

Feb 2021	Feb 2020	Change	Programming Language	Ratings	Change
1	2	▲	C	16.34%	-0.43%
2	1	▼	Java	11.29%	-6.07%
3	3		Python	10.86%	+1.52%

# Python está escrito en C



dato super duper  
hiper mega ultra  
archi recontra  
spoileado

o por lo menos su  
implementación estándar...

# Python vs C

## C

- Generalmente usado para aplicaciones relacionadas al hardware
- Bajo nivel
- Paradigma estructural
- Punteros
- Compilado
- Ejecución más rápida que python

## Python

- Propósito general
- Alto nivel
- Paradigma de objetos
- *I don't know her*
- Interpretado
- Ejecución más lenta que C



# Python vs C

## C

- Tipado
- Sintaxis más compleja
- Manejo de memoria a cargo del programador
- Limitadas built-in functions

Programador  
de C



Utilizo el lenguaje mas  
usado en el mundo,  
domino punteros y se  
de que tipo es cada  
una de mis variables

## Python

- No tipado
- Sintaxis sencilla
- Garbage collector
- Extensa built-in library

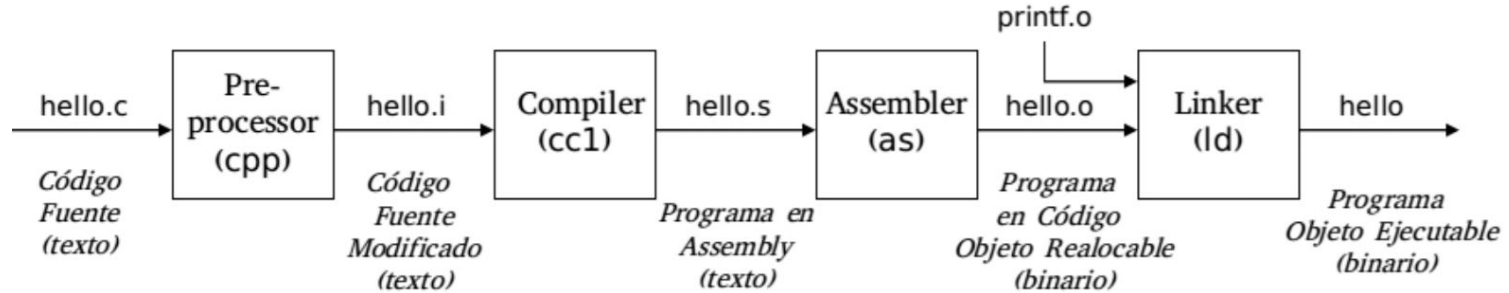
El compilador me  
asusta



Programador de  
python



# El sistema de compilación




1. **Fase de procesamiento:** El preprocesador (cpp) modifica el código de fuente original de un programa escrito en C de acuerdo a las directivas que comienzan con un caracter(#). El resultado de este proceso es otro programa en C con la extensión .i

2. **Fase de compilación:** El compilador (cc) traduce el programa .i a un archivo de texto .s que contiene un programa en lenguaje assembly.

# El sistema de compilación

3. **Fase de ensamblaje:** El ensamblador (as) traduce el archivo .s en instrucciones de lenguaje de máquina completándolas en un formato conocido como programa objeto realocable. Este es almacenado en un archivo con extensión .o

4. **Fase de link edición:** Generalmente los programas escritos en lenguaje C hacen uso de funciones que forman parte de la biblioteca estándar de C que es provista por cualquier compilador de ese lenguaje. Por ejemplo la función printf(), la misma se encuentra en un archivo objeto precompilado que tiene que ser mezclado con el programa que se está compilando, para ello el linker realiza esta tarea teniendo como resultado un archivo objeto ejecutable.



# Estructura básica de un programa en C

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello world\n");
5      return 0;
6  }
```

- `#include` : Estos headers dan la instrucción de inclusión de las bibliotecas indicadas
- `main` : La ejecución del programa comienza por esta función.
- “{” indican el comienzo y final de un bloque de código
- “;” indica el final de una sentencia
- `return 0` : Ordena que el programa termine su ejecución y devuelva 0 al contexto en el que fue ejecutado indicando una ejecución exitosa.

# “Hello world”

Hello\_world.py

Hello\_world.c

```
1 def main():  
2     print("Hello world")  
3  
4 main()  
5
```

Hello\_world.py

Hello\_world.c

```
1 #include <stdio.h>  
2  
3 int main() {  
4     printf("Hello world\n");  
5     return 0;  
6 }  
7
```

# Palabras reservadas

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

## Operadores aritméticos

Operador	Descripción	Ejemplo
*	Multiplicación	(a*b)
/	División	(a/b)
+	Suma	(a+b)
-	Resta	(a-b)
%	Módulo	(a %b)

## Operadores lógicos

Operador	Descripción	Ejemplo
expresion1    expresion2	or	(x>2)    (y<4)
expresion1 && expresion2	and	(x>2) && (y<4)

## Operadores incrementales

Operador	Descripción	Ejemplo	equivalencia
++	incremento	i++	i=i+1
--	decremento	i--	i=i-1

## Operadores relacionales

Operador	Descripción	Ejemplo
<	Menor que	(a<b)
<=	Menor que o igual	(a <= b)
>	Mayor que	(a > b)
>=	Mayor o igual que	(a >= b)
==	Igual	(a == b)
!=	No igual	(a != b)

# Variables

Para definir una variable en C se debe establecer el tipo de dato al cual pertenece seguido del nombre asignado a la variable (identificador).

```
int numero ;  
long int contador ;  
char letra ;  
float raiz ;  
int contador, resultado;
```

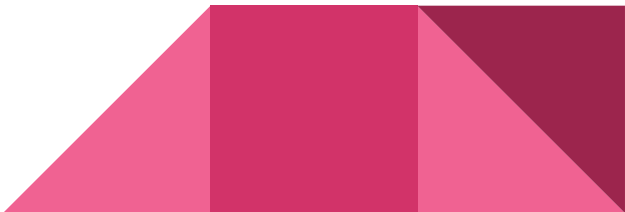
# Comentarios

```
/*Yo, I'll tell you what I want, what I really, really want  
So tell me what you want, what you really, really want  
I'll tell you what I want, what I really, really want  
So tell me what you want, what you really, really want  
I wanna, (ha) I wanna, (ha) I wanna, (ha) I wanna, (ha)  
I wanna really, really, really wanna zigazig ah*/  
  
// oh sh*t, here we go again.
```

*/\*muchas*

*líneas\*/*

*// una sola línea*



# Include

```
#include "pila.h"
#include "testing.h"
#include <stdio.h>|
```

<> busca los ficheros en todos los directorios especificados en la llamada al compilador (normalmente con la opción -I)  
"" busca este fichero primero en el mismo directorio donde está el fichero actualmente compilado

# Constantes

```
const int MAX_LEN = 5;
const char NOMBRE = "Ayllen";|
```

Se declaran variables con la palabra reservada **const** lo cual asegura que esa variable no cambiará su valor a lo largo del programa

# Macros

```
#define PI 3.141592
#define E 2.718281|
```

La directiva #define indica al preprocesador que debe sustituir, en el código fuente del programa, todas las ocurrencias del nombre de la macro por su valor, antes de la compilación.





# Tipos básicos de datos

**char, int ,double , float** y combinaciones (en algunos casos) con **signed unsigned short long**

Estructura  
32 bits

Tipo	Tamaño	Mínimo	Máximo
[signed]char	1 byte	-128	127
unsigned char	1 byte	0	255
short	2 bytes	-32,768	32,767
short int			
signed short			
signed short int			
unsigned short	2 bytes	0	65,535
unsigned short int			
[signed] int	4 bytes	-2,147,483,648	2,147,483,647
unsigned int	4 bytes	0	4,294,967,295
long	4 bytes	-2,147,483,648	2,147,483,647
long int			
signed long			
signed long int			
unsigned long	4 bytes	0	4,294,967,295
unsigned long int			

# Tipos básicos de datos

Estructura  
64 bits

Tipo	Tamaño	Mínimo	Máximo
[signed]char	1 byte	-128	127
unsigned char	1 byte	0	255
short	2 bytes	-32,768	32,767
short int			
signed short			
signed short int			
unsigned short	2 bytes	0	65,535
unsigned short int			
[signed] int	4 bytes	-2,147,483,648	2,147,483,647
unsigned int	4 bytes	0	4,294,967,295
long	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
long int			
signed long			
signed long int			
unsigned long	8 bytes	0	18,446,744,073,709,551,615
unsigned long int			


# Tipos básicos de datos

Tipo	Tamaño	Rango de valores	Precisión
float	4 byte	1.2E-38 to 3.4E+38	6 lugares decimales
double	8 byte	2.3E-308 to 1.7E+308	15 lugares decimales
long double	10 byte	3.4E-4932 to 1.1E+4932	19 lugares decimales

## Booleanos:

No existe tipo de dato como tal, el lenguaje C considera falsa toda expresión que evalúe a 0 y verdadera a cualquier otra.

La biblioteca **stdbool.h** nos permite declarar variables tipo bool y asignar los valores **true** y **false**



# Estructuras de control: Selectivas

## If - Else

```
if (expresion)
    accion;

//-----

if (expresion)
    accion_1;
else
    accion_2;

//-----

if (expresion)
    accion_1;
else if (expresion_N)
    accion_2;

//-----

if (expresion){
    accion_1;
    accion_N;
}

//-----

if (expresion) {
    accion_1 ;
    accion_N ;
} else {
    accion_a;
    accion_M;
}

//-----

if (expresion) {
    accion_1;
    accion_N;
} else if (expresion_N){
    accion_a;
    accion_M;
}
```

# Estructuras de control: Selectivas

## Switch case

```
switch (var) {  
    case 1:  
        accion; //se ejecuta si var == 1  
        break ;  
    case 2:  
        accion;  
        break;  
    case 3:  
        accion;  
        break ;  
    default:  
        accion;  
}
```

- Las variables case pueden ser de tipo int o char
- La sentencia default es opcional
- Si no se pone un break en cada case se ejecutará desde la sentencia que matchee hacia abajo

# Estructuras de control: Iterativas

## Do - while

```
do{  
    accion/es  
}while (expresion);
```

## While

```
while (expresion) {  
    accion/es;  
}
```

El bloque do se ejecutará por única vez siempre antes de entrar o no al ciclo



# Estructuras de control: Iterativas

## For

```
//más utilizado
for (size_t i = 0; i < count; i++) {
    accion;
}

//genérico
for (expresion1 ; expresion2 ; expresion3){
    accion ;
}
```

- **Expresión 1:** Define cuál es la variable de control del ciclo y su valor inicial, esta puede definirse en la misma sentencia.
- **Expresión 2:** Define el valor de corte del ciclo, es una expresión booleana.
- **Expresión 3:** Define cómo se incrementa la variable de control.



# Input y Output

```
1  #include <stdio.h>
2
3  int main () {
4      int var; /* declaracion de la variable */
5
6      printf ("Ingrese un numero entero :"); // escritura
7      scanf (" %d" , &var) ; // lectura
8
9      printf ("numero: %d", var);
10     return 0;
11 }
```

Formatos más usados:

%d o %i	signed int
%u	unsigned int
%c	char
%s	string
%f	float
%lf	double
%.2f	.n (n cant de decimales)



# Funciones y procedimientos

Ejemplo función:

```
int suma(int num1, int num2){  
    int resultado;  
    resultado = num1 + num2;  
    return resultado;  
}
```

Ejemplo procedimiento:

```
void imprimir_emoji(){  
    printf("uwu\n");  
}
```

definimos un  
procedimiento  
usando como tipo  
dato retorno **void**

Estructura genérica:

```
tipo_retorno nombre_funcion (parametros){  
    declaraciones;  
    acciones;  
}
```



# Ejercicio

Pedir al usuario que ingrese números y mostrar su suma. Usar -1 como condición de corte.

Pedir al usuario que ingrese 10 numeros mostrar cuál fue el mayor y el menor ingresados.



# Punteros

Como las funciones devuelven **UN** y solo **UN** tipo de dato, si deseamos modificar más de uno en una función o procedimiento; debemos recurrir a los ***punteros***.



# Pero primero una reflexión...



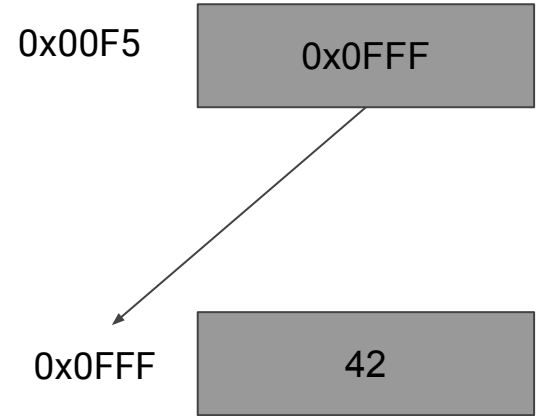
# Punteros

Un puntero es una variable que contiene como valor la ***dirección de memoria*** de otra variable.

Entonces, decimos que la variable puntero ***apunta*** a dicha dirección de memoria.

Declaración:

```
TIPO * nombre_puntero;
```



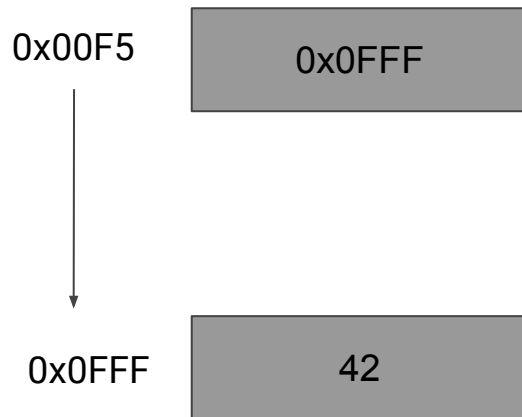
# Punteros

Un puntero es una variable que contiene como valor la ***dirección de memoria*** de otra variable.

Entonces, decimos que la variable puntero ***apunta*** a dicha dirección de memoria.

Declaración:

```
TIPO * nombre_puntero;
```



# Punteros: Operadores

## Operador de dirección

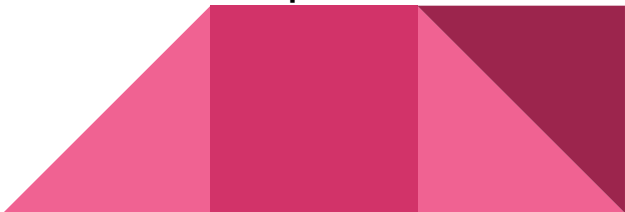
```
int numero;  
int *ptr; // Puntero a un entero  
  
ptr = &numero;
```

El operador de dirección (&) devuelve la dirección de memoria del operando.

## Operador de indirección

```
int numero = 50 ;  
int *ptr; // Puntero a un entero  
ptr = &numero;  
printf("El valor de numero es %d", *ptr);
```

El operador de indirección (\*) devuelve el contenido de la variable apuntada por su operando, el cual debe ser un puntero.



# Punteros: ejemplo

```
void fin_cuarentena(int *fin,int *contador_cuarentenas){  
    *fin=15+*fin;  
    *contador_cuarentenas+=1;  
}
```

```
fin_cuarentena(&dias_faltantes,&cantidad_alargues);|
```





# Punteros: *Ejemplo*

```
void fin_cuarentena(int* fin, int* contador_cuarentenas) {  
  
    *fin = *fin + 15;  
  
    *contador_cuarentenas += 1;  
  
}
```

Procedimiento

```
int main() {  
  
    int dias_faltantes = 0;  
    int cantidad_alargues = 0;  
  
    fin_cuarentena(&dias_faltantes, &cantidad_alargues);  
  
    return 0;  
  
}
```

Main



# Punteros: Pasaje de parámetros por referencia

El lenguaje C pasa los argumentos de funciones por valor, entonces no hay una forma directa de alterar las variables de la función a la que se llama, debido a que utilizaría copias de las mismas.

No tenemos el concepto de datos mutables o inmutables, la única forma de alterar una variable que fue declarada en un scope desde otro es a través del uso de punteros.

Además el lenguaje C, no cuenta con un retorno múltiple. Una forma de manejar esto es enviando parámetros a nuestra función y que esta deposite sus resultados en ellos.


Para visualizarlo mejor, utilizaremos la función swap, que intercambia el valor de dos variables.



# Punteros: Pasaje de parámetros por referencia

```
void swap(int val_1, int val_2){  
    int aux;  
  
    aux = val_1;  
    val_1 = val_2;  
    val_2 = aux;  
  
}
```

Aquí está la función swap, de la manera en la que está escrita los cambios que haga dentro del procedimiento, no serán visibles en el programa principal, ya que los mismo suceden en el scope local de la función.



# Punteros: Pasaje de parámetros por referencia

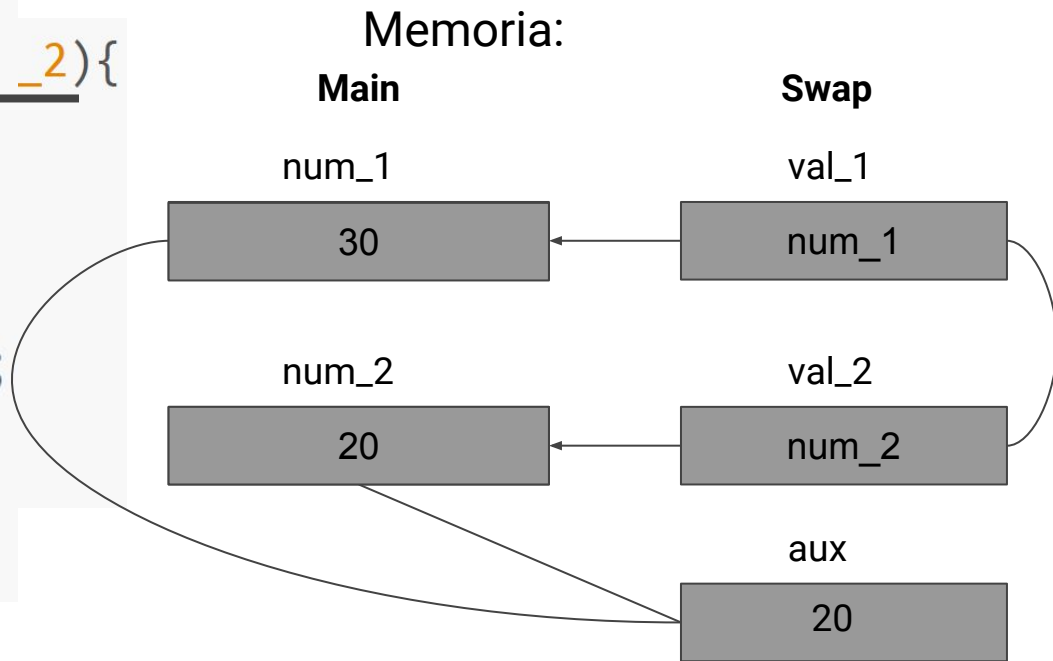
```
void swap(int *val_1, int *val_2){  
    int aux;  
  
    aux = *val_1;  
    *val_1 = *val_2;  
    *val_2 = aux;  
  
}
```

Y aquí se muestran la función modificada para trabajar con punteros y su implementación en el programa principal.

```
int main(){  
  
    int num_1 = 20 ;  
    int num_2 = 30 ;  
  
    swap(&num_1, &num_2);  
  
    return 0;  
}
```

# Punteros: Pasaje de parámetros por referencia

```
int main(){  
    _____  
    ______2){  
    → int num_1 = 20 ;  
    → int num_2 = 30 ;  
  
    → swap(&num_1, &num_2);  
  
    return 0;  
}
```

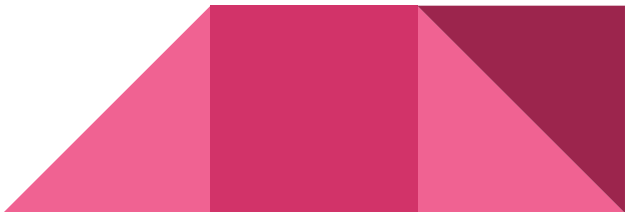


Finalmente, los valores de las variables num\_1 y num\_2 fueron intercambiados

# Análisis del primer tiempo



## Ejercicio 1:

- 1) Crear una **variable** de tipo int
  - 2) Crear un **puntero** a int y almacenar en él, la dirección de memoria de la **variable**.
  - 3) Crear un procedimiento que **reciba** un puntero a int y modifique el **número** almacenado en la dirección de memoria.
  - 4) Pasarle a ese procedimiento; la dirección de memoria de la **variable** creada en (1).
  - 5) Pasarle a ese procedimiento; el **puntero** a int creado en (2).
- 

## Ejercicio 2:

Crear un ***procedimiento*** que sume números ingresados por el usuario (los almacenará utilizando un ***puntero***) y luego haga un print del resultado en el ***main***.





# Abstracción

En ocasiones se requiere representar partes de la realidad pero no se disponen de infinitos detalles por lo que solo deben conservarse los más relevantes. Los resultados de estas simplificaciones sólo pueden entenderse como la versión real mediante la abstracción



Fisicos



# Estructuras (Structs)

Un struct es una colección de una o más variables, que pueden ser de tipos diferentes, agrupadas bajo un solo nombre.

El uso de estructuras ayuda a organizar datos complicados organizando un grupo de variables relacionadas entre sí. Asimismo permite distintos niveles de abstracción.

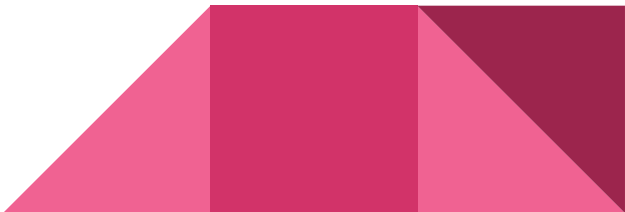


# Structs

Para codear éstos structs, vamos a hacer uso de la palabra reservada *typedef* para poder, luego, utilizar la estructura de una manera más prolija

Cada vez que definimos un struct estamos creando nuestro propio tipo de dato, como convención nombramos a nuestro tipo de dato como **nombre\_t**

```
#include <stdio.h>
#include <stdbool.h>
typedef struct {
    int capacidad;
    char destino[30];
    char empresa[30];
    int as_ocupados;
    bool es_internacional;
} avion_t;
```



# Structs: Inicialización y asignación

Inicialización: En caso de ingresar los valores en orden, sirve la primera sentencia, caso contrario, se deberá utilizar la segunda.


```
avion_t boeing = {200, "Rio de Janeiro", "LATAM", 50, true};
```

```
avion_t airbus = {.destino= "Cordoba", .capacidad=200, .empresa= "Aerolineas Argentinas", .as_ocupados= 20, .es_internacional=false};
```

Asignación:

```
boeing.as_ocupados+=20;  
airbus.as_ocupados+=60;  
if(boeing.as_ocupados == airbus.as_ocupados)  
    printf("El vuelo a %s y a %s han vendido igual cantidad de pasajes", boeing.destino, airbus.destino);
```

C no nos provee de un operador para comparar structs por lo tanto se deben igualar miembro a miembro o crear funciones que realicen esta acción.



# Structs: Punteros


Dado a que estamos definiendo un nuevo tipo de dato, también podemos definir punteros a este tipo de dato.

El operador de indirección puede ser usado utilizando “->” para ingresar a algún miembro del struct.

```
avion_t *ptr_avion;  
ptr_avion = &boeing;
```

```
(*ptr_avion).as_ocupados = 30;  
ptr_avion->as_ocupados = 30;
```

entonces para acceder y/o modificar un campo de un struct usamos “.” si la variable esta en nuestro scope, usamos “->” si estamos trabajando con un puntero a la variable



# Arreglos - Arrays - Vectores

En C hay una fuerte relación entre punteros y arrays. Los arrays son bloques consecutivos de memoria para una cierta variable.

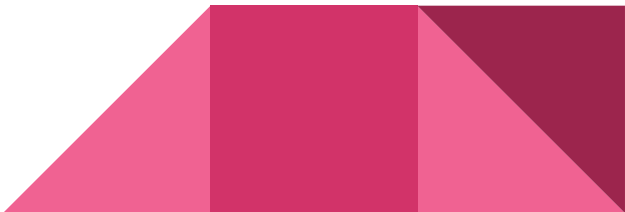
Podemos definirlos de la siguiente manera: `TIPO nombre[#elementos];`

Podemos ver la relación con los punteros definiendo un puntero que apunte a algún elemento del vector.

Casualmente `p_vec+1` apuntará al siguiente elemento del vector de enteros.

```
int vec[10];  
int *p_vec;  
p_vec = &vec[0];
```

En C el nombre el vector es un puntero al primer elemento del mismo



# Vectores: Consideraciones

-Operaciones entre vectores: Se deben realizar elemento por elemento. (No es posible asignar un vector a otro, por ejemplo).

-Definiciones: La longitud de los vectores ES FIJA, por lo que es una buena práctica que sean definidos con constantes.

-Inicialización:

```
int vector[4] = {10, 5, 2, 3};  
int vector[] = {10, 5, 2, 3};  
int vector[4] = {0};
```

# Pasaje de vectores por parámetro

No es necesario aclarar la longitud del vector al momento de llamar a la función.

El pasaje de vectores SIEMPRE se hace por referencia, ya que como dijimos el nombre del mismo es una referencia a la primera posición de este.

```
int suma(int vector[]){  
    /Código  
}  
  
int suma= suma(vector);
```





# Ejercicio

Crear un programa en el cual te permita ingresar edades y, cuando se termine de ingresar, te diga el promedio de las edades.

Todas las edades se deben guardar en un array, el cálculo de promedio debe ser en una función.



# Arrays multidimensionales

Es bastante simple declarar un array de varias dimensiones, un array bidimensional, por ejemplo, es visualizable como una matriz.

```
int matriz[2][2];
```

```
int matriz[2][2]= {{1, 2}, {5, 3}};
```

```
int matriz[2][2]= {1, 2, 5, 3};
```



# Ejercicio

Sean A, B y C arreglos de 2x2 de enteros, realizar las siguientes operaciones y mostrarlas en pantalla:

- $C = A - B$
- $C = B - A$

[Usar como mínimo 2 funciones]



# Strings

## En C un string es un arreglo de datos tipo char

```
int main () {
    char * cadena_salida = "Hello , world !";
    printf ( " %s", cadena_salida ) ;
    return 0;
}
```

Siendo las siguientes expresiones equivalentes:

cada string para estar correctamente declarado debe terminar con el carácter especial “\0”, **por lo tanto SIEMPRE debemos reservar lugar para este!!!**

# Strings - string.h

- **char \* strcat(char \*dest, const char \*src)**

Concatena ambas cadenas (python: dest + src)

- **int strcmp(const char \*str1, const char \*str2)**

Compara dos strings devolviendo 1 si str1 > str2, 0 si str1 = str2, -1 si str1 < str2.

- **char \* strcpy(char \*dest, const char \*src)**

Copia el string apuntado por src al lugar apuntado por dest.

- **size\_t strlen(const char \*str)**

Calcula el largo del string sin incluir el carácter nulo



# Archivos

- **FILE \*fopen(const char\* nombre\_archivo, const char\* modo)**

crear nuevo archivo o abrir uno ya existente

- **int fclose (FILE \*archivo)**

cerrar el archivo para liberar la memoria usada retornando 0 o EOF

**Modos de apertura:** r, w, a, r+, w+, a+



# Escritura

- **int fputc(int character, FILE \*archivo)**

escribe un caracter en el file retorna el carácter escrito o EOF

- **int fputs(const char \*string, FILE \*archivo)**

escribe una cadena terminada en /0, retorna n > 1 o EOF

- **int fprintf(FILE \*archivo, const char \*string, ...)**



# Lectura

- **int fgetc(FILE \*archivo)**

Lee un caracter y lo retorna

- **char \*fgets(FILE \*buffer, int size, FILE \*archivo)**

lee hasta size-1 copia el resultado a buffer agregando un /0, si encuentra un /n o EOF deja de leer y retorna lo que llegó a leer

- **int fscanf(FILE \*archivo, const char \*format, ...)**

Lee siguiendo el formato indicado, se detiene al encontrar un espacio





# Binarios

**size\_t fread(void \*ptr, size\_t size\_elem, size\_t num\_elem, FILE \*archivo);**

lugar donde vamos a guardar lo leído, tamaño del elemento a leer, cantidad de elementos que vamos a leer, archivo.

**size\_t fwrite(const void \*ptr, size\_t size\_elem, size\_t num\_elem, FILE \*archivo;**

estructura en memoria que queremos guardar en el archivo, tamaño del elemento, cantidad de veces que queremos escribir ese elemento, archivo abierto en modo w o a

