

Transparencias de la asignatura

INTRODUCCIÓN A LA PROGRAMACIÓN

CURSO 2024/2025

Autores:

Diego Hortelano, Gerardo Reyes, Manuel Rubio



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

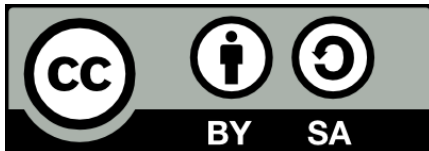
Índice de contenidos

Tema	Título	Página
1	Introducción a la Programación	3
2	Fundamentos del Lenguaje C	88
3	Operadores y Expresiones	144
4	Estructuras de Control	179
5	Arrays y Cadenas de Caracteres	214
6	Punteros	265
7	Subprogramas y Recursividad	367
8	Memoria Dinámica	442
9	Estructuras y Tipos de Datos Enumerados	502
10	Ficheros	540

Tema 1: Introducción a la Programación

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

Índice

- Resolución de problemas
- Algoritmos
- Lenguajes de programación
- Compiladores e interpretes
- Entorno de programación
- Pasos para la creación de un programa
- Pseudocódigo

Resolución de problemas

Para resolver un problema en programación debemos seguir una serie de pasos clave:

Resolución de problemas

Para resolver un problema en programación debemos seguir una serie de pasos clave:

1. Entender el problema.

Resolución de problemas

Para resolver un problema en programación debemos seguir una serie de pasos clave:

1. Entender el problema.
2. Dividir el problema.

Resolución de problemas

Para resolver un problema en programación debemos seguir una serie de pasos clave:

1. Entender el problema.
2. Dividir el problema.
3. Diseñar un algoritmo.

Resolución de problemas

Para resolver un problema en programación debemos seguir una serie de pasos clave:

1. Entender el problema.
2. Dividir el problema.
3. Diseñar un algoritmo.
4. Implementar la solución.

Resolución de problemas

Para resolver un problema en programación debemos seguir una serie de pasos clave:

1. Entender el problema.
2. Dividir el problema.
3. Diseñar un algoritmo.
4. Implementar la solución.
5. Depurar.

Resolución de problemas

Para resolver un problema en programación debemos seguir una serie de pasos clave:

1. Entender el problema.
2. Dividir el problema.
3. Diseñar un algoritmo.
4. Implementar la solución.
5. Depurar.
6. Optimizar.

Algoritmo

Definición

- Conjunto **ordenado** y **finito** de instrucciones **realizables** y **precisas** para resolver un problema en un tiempo **finito**.

Algoritmo

Definición

- Conjunto **ordenado** y **finito** de instrucciones **realizables** y **precisas** para resolver un problema en un tiempo **finito**.
- La palabra **algoritmo** viene de *Abu Abdallah Muḥammad ibn Mūsā al-Jwārizmī*, matemático, astrónomo y geógrafo persa del siglo IX.

Algoritmo

Características

- **Finitud:** el número de instrucciones debe ser finito, y el algoritmo debe finalizar en algún momento.
- **Claridad y precisión:** no puede haber ambigüedad en las instrucciones.
- **Orden secuencial:** las instrucciones deben organizarse de manera lógica y secuencial.

Algoritmo

Características

- **0-N entradas y 1-N salidas:** los algoritmos actúan sobre los datos de entrada para producir resultados (datos de salida).
- **Efectividad:** las instrucciones deben ser comprensibles y ejecutables.
- **Determinismo:** partiendo de los mismos datos de entrada, un algoritmo debe producir siempre los mismos resultados.

Algoritmo

Ejecución

- Dado un algoritmo, **no es necesario entender** sus principios para poder ejecutarlo.
- Sólo necesitamos ejecutar las instrucciones.

Algoritmo

Ejemplo: ¿es un algoritmo?

1. Haz una lista con todos los enteros positivos.
2. Ordena la lista de mayor a menor.
3. Extrae el primer entero de la lista ordenada.

Algoritmo

Ejemplo: ¿es un algoritmo?

**!!! NO REALIZABLE EN
TIEMPO FINITO !!!**



1. Haz una lista con todos los enteros positivos.
2. Ordena la lista de mayor a menor.
3. Extrae el primer entero de la lista ordenada.

Algoritmo

Ejemplo: ¿es un algoritmo?

1. Comienza a contar desde el número 8
2. Suma una unidad al valor del contador.
3. Si el valor del contador es mayor que 4, vuelve al paso 1.

Algoritmo

Ejemplo: ¿es un algoritmo?

!!! NO TERMINA NUNCA !!!



1. Comienza a contar desde el número 8
2. Suma una unidad al valor del contador.
3. Si el valor del contador es mayor que 4, vuelve al paso 1.

Algoritmo

Ejemplo: extraer el valor mínimo

Entrada: lista de n números enteros

Salida: valor mínimo de la lista

1. Toma el primer elemento de la lista y considéralo el mínimo.
2. Para cada elemento de entre los $n - 1$ restantes:
 - Compáralo con el mínimo actual.
 - Si el elemento actual es menor que el mínimo almacenado, actualiza el mínimo con el nuevo valor.

Algoritmo

Ejemplo: extraer el valor mínimo

- Entrada: [7, 2, 5, 4, 9]

Mínimo

7

2

5

4

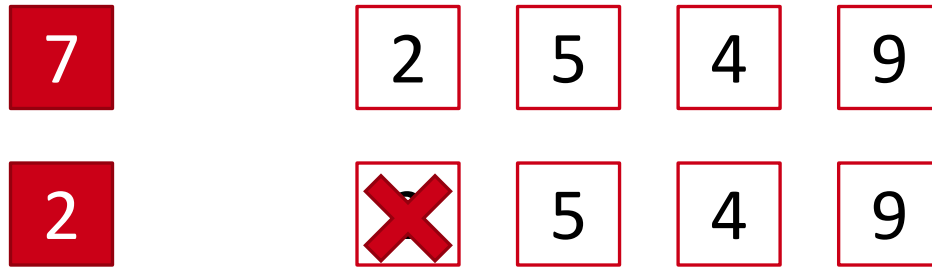
9

Algoritmo

Ejemplo: extraer el valor mínimo

- Entrada: [7, 2, 5, 4, 9]

Mínimo



Algoritmo

Ejemplo: extraer el valor mínimo

- Entrada: [7, 2, 5, 4, 9]

Mínimo

7	2	5	4	9
2	X	5	4	9
2	X	X	4	9

Algoritmo

Ejemplo: extraer el valor mínimo

- Entrada: [7, 2, 5, 4, 9]

Mínimo

7	2	5	4	9
2	X	5	4	9
2	X	X	4	9
2	X	X	X	9

Algoritmo

Ejemplo: extraer el valor mínimo

- Entrada: [7, 2, 5, 4, 9]

Mínimo

7	2	5	4	9
2	✗	5	4	9
2	✗	✗	4	9
2	✗	✗	✗	9
2	✗	✗	✗	✗

Algoritmo

Ejemplo: Búsqueda Lineal

Entradas: lista de n números enteros y número a buscar

Salida: posición del número en la lista

1. Comienza con el primer elemento de la lista.
2. Compara el elemento actual con el número buscado.
 - Si el elemento actual coincide con el número buscado, retorna la posición del elemento actual y finaliza.
 - Si no, pasa al siguiente elemento y repite el paso 2.
3. Si ningún elemento coincide, retornar -1.

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

Número

5

6

1

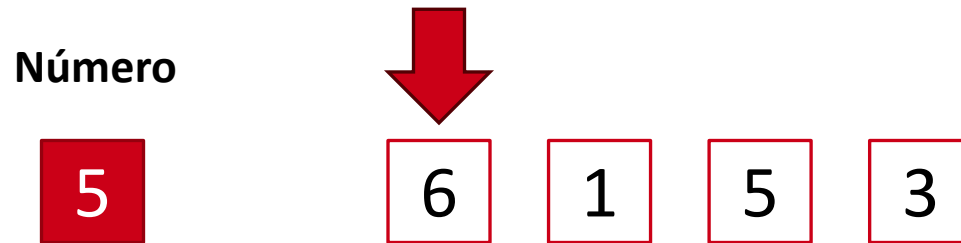
5

3

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

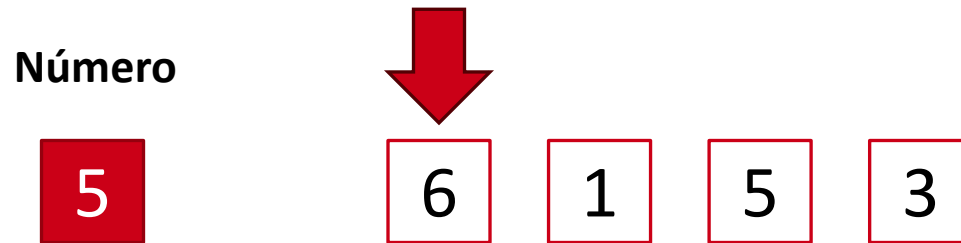
- Entradas: [6, 1, 5, 3] y 5



Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

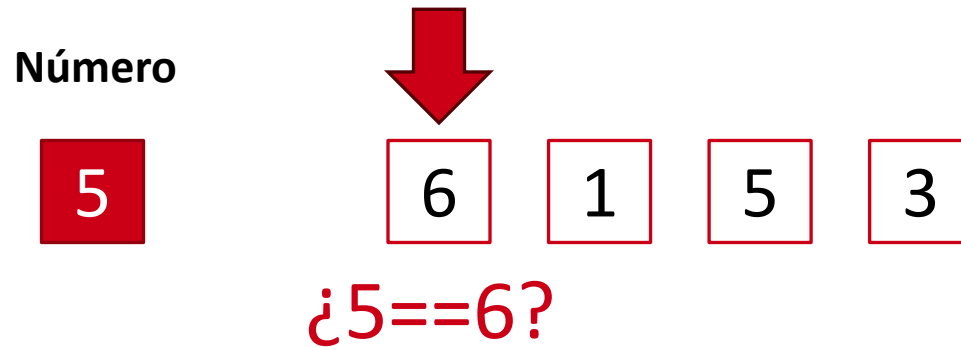


Posición = 0

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

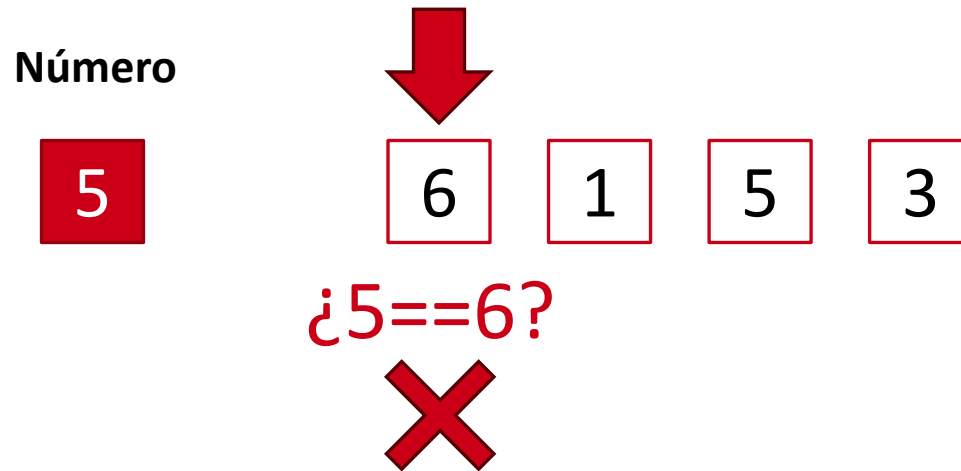


Posición = 0

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

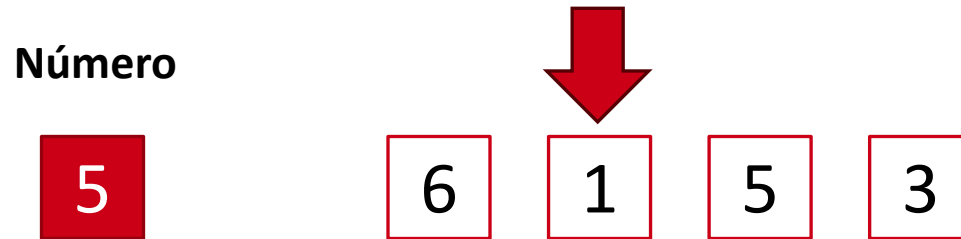


Posición = 0

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5



Posición = 1

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

Número

5

6 1 5 3

¿5==1?

Posición = 1

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

Número

5

6 1 5 3

¿5==1?



Posición = 1

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

Número

5

6

1

5

3



Posición = 2

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

Número

5

6

1

5

3

¿5==5?

Posición = 2

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5

Número

5

6

1

5

3

¿5==5?

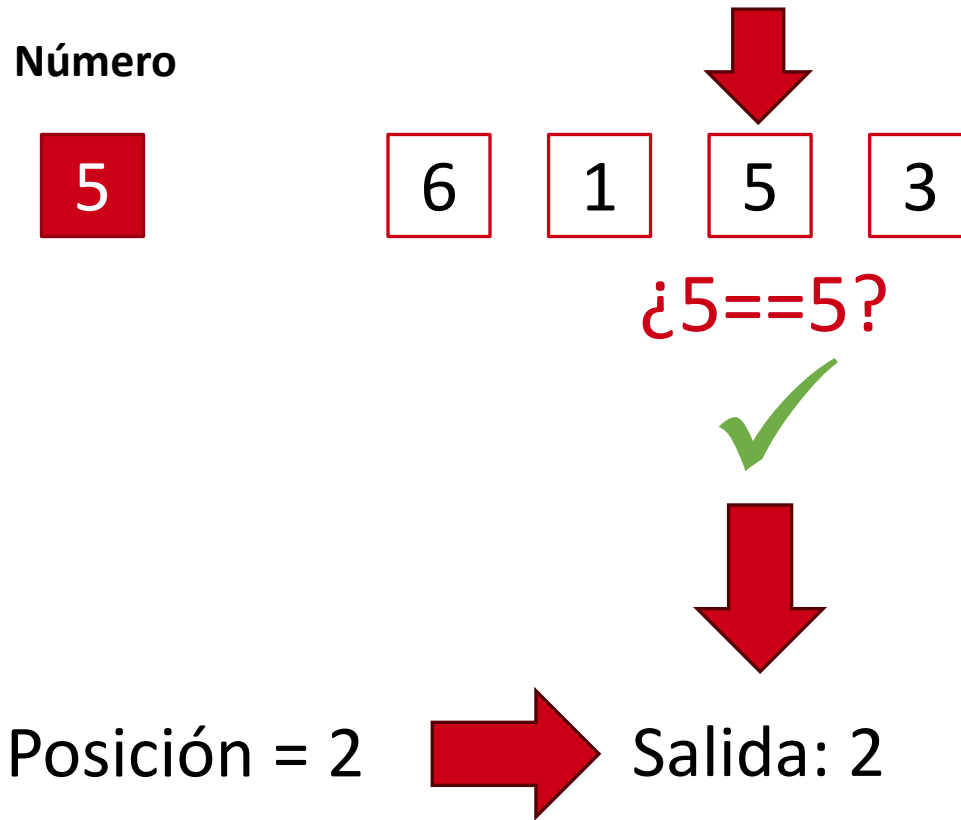


Posición = 2

Algoritmo

Ejemplo: Ejemplo: Búsqueda Lineal

- Entradas: [6, 1, 5, 3] y 5



Algoritmo

Ejemplo: Ordenamiento Burbuja

Entrada: lista de n números enteros

Salida: lista de n números enteros ordenada

1. Para cada elemento de la lista (excepto el último):
 - Compara el elemento actual con el siguiente elemento de la lista.
 - Si el elemento actual es mayor que el elemento siguiente, intercámbialos.
2. Si se ha realizado algún cambio al recorrer la lista completa, repetir el paso 1.

Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

[3, 5, 2]

Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

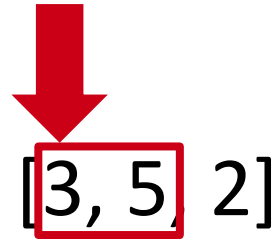


[3, 5, 2]

Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]



Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]



Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

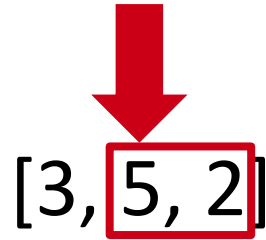


[3, 5, 2]

Algoritmo

Ejemplo: Ordenamiento Burbuja

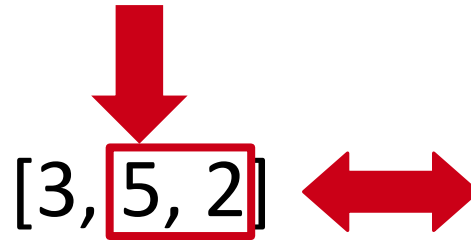
- Entrada: [3, 5, 2]



Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

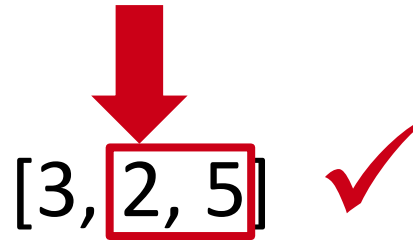


Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

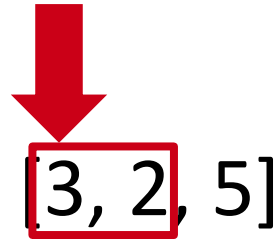
[3, 2, 5] ✓

A diagram illustrating a step in the bubble sort algorithm. It shows the array [3, 2, 5]. A large red arrow points downwards to the element 2, which is enclosed in a red rectangular box. To the right of the array, there is a red checkmark, indicating that the current element is in its correct sorted position.

Algoritmo

Ejemplo: Ordenamiento Burbuja

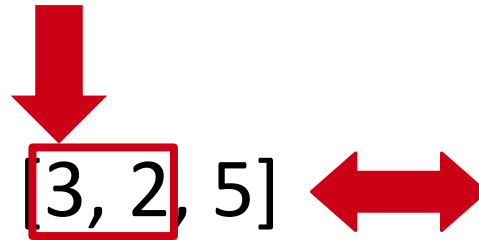
- Entrada: [3, 5, 2]



Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]



Algoritmo

Ejemplo: Ordenamiento Burbuja


- Entrada: [3, 5, 2]

 [2, 3, 5] ✓

Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

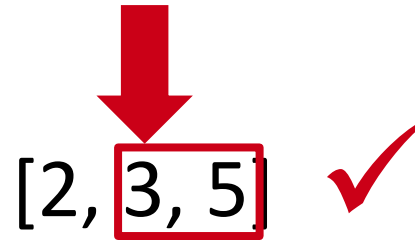

[2, 3, 5]

Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

[2, 3, 5] ✓

A diagram illustrating the result of a sorting step. The array [2, 3, 5] is shown. A large red arrow points down to the number 3, which is enclosed in a red rectangular box. To the right of the array is a red checkmark, indicating that the array is now sorted.

Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

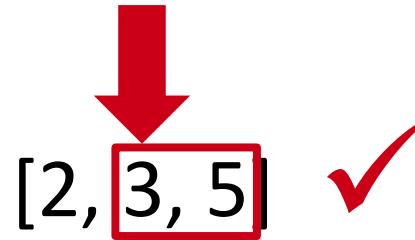
 [2, 3, 5] ✓

Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

[2, 3, 5] ✓

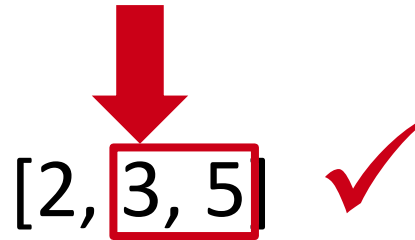
A diagram illustrating the result of a sorting step. The array [2, 3, 5] is shown. A red arrow points down to the number 3, which is enclosed in a red box. A red checkmark is positioned to the right of the array, indicating that the current state is correct or sorted.

Algoritmo

Ejemplo: Ordenamiento Burbuja

- Entrada: [3, 5, 2]

[2, 3, 5] ✓



- Salida: [2, 3, 5]

Lenguajes de programación

¿Qué son?

- Un lenguaje de programación es el **conjunto de instrucciones, signos y reglas** que nos permite codificar instrucciones de manera que puedan ser comprendidas y ejecutadas por un ordenador.
- También puede definirse como la **sintaxis** que los computadores pueden interpretar y ejecutar, lo que permite crear herramientas software, aplicaciones...

Lenguajes de programación

Tipos de lenguajes de programación

- Existen diferentes tipos de lenguajes de programación, según su nivel de abstracción y generación.

Lenguajes de programación

Tipos de lenguajes de programación

Primera generación: código máquina

- Codifican las instrucciones como secuencias de 0s y 1s que entienden directamente los procesadores.
- Cada familia de ordenadores tiene su propio repertorio de instrucciones.

Lenguajes de programación

Tipos de lenguajes de programación

Primera generación: código máquina

```
00000000  B4 03 CD 10 B0 01 B3 0A B9 0E 00 BD 13 7C B4 13 .....|..
00000010  CD 10 F4 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0D ...Hello World!
00000020  0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
--- ex          --0x0/0x200-----
```

Luiz Felipe [https://pt.wikipedia.org/wiki/C%C3%B3digo_de_m%C3%A1quina]

Lenguajes de programación

Tipos de lenguajes de programación

Segunda generación: ensamblador

- Consiste en notaciones mnemotécnicas que representan instrucciones del código máquina.
- Un programa (ensamblador) traduce estas notaciones a código máquina.

Lenguajes de programación

Tipos de lenguajes de programación

Segunda generación: ensamblador

```
1  section .text
2      global _start
3
4  _start:
5      mov  edx,len
6      mov  ecx,msg
7      mov  ebx,1
8      mov  eax,4
9      int  0x80
10
11     mov  eax,1
12     int  0x80
13
14  section .data
15  msg db 'Hola, mundo!', 0xa
16  len equ $ - msg
```

John Harold Belalcazar Lozano [<https://es.quora.com/C%C3%B3mo-se-ve-un-c%C3%B3digo-de-lenguaje-de-m%C3%A1quina>]

Lenguajes de programación

Tipos de lenguajes de programación

Tercera generación: lenguajes de alto nivel

- Permiten escribir programas ejecutables en varios ordenadores.
- Reduce la complejidad en el desarrollo de programas.

Lenguajes de programación

Tipos de lenguajes de programación

Tercera generación: lenguajes de alto nivel

```
#include <stdio.h>

int main() {
    printf(format: "Hello world!!");
    return 0;
}
```


Lenguajes de programación

Tipos de lenguajes de programación

Cuarta generación: lenguajes declarativos y herramientas de desarrollo

- También se conocen como lenguajes de muy alto nivel o lenguajes de consulta.
- Se centran en qué se hace, y no en el cómo.
- La sintaxis se asemeja al lenguaje natural.
- Permiten programación en entornos visuales muy sencillos, acceso a bases de datos, etc.

Lenguajes de programación

Tipos de lenguajes de programación

Cuarta generación: lenguajes declarativos y herramientas de desarrollo

SELECT nombre, edad

FROM estudiantes

WHERE fecha_matriculacion = 2024;

Lenguajes de programación

Tipos de lenguajes de programación

Quinta generación: lenguajes de programación lógica

- También conocidos como lenguajes basados en IA o en la resolución de problemas.
- Se centran en problemas de inteligencia artificial, como el PLN y el razonamiento automático.

Lenguajes de programación

Tipos de lenguajes de programación

Quinta generación: lenguajes de programación lógica

Programa lógico

```
animal(perro).  
animal(gato).  
animal(canguro).  
arbol(palmera).  
flor(margarita).
```

Hechos

```
vegetal(X) :- arbol(X).  
vegetal(X) :- flor(X).
```

Reglas

Invocación

```
> vegetal(perro)  
> false
```

```
> animal(perro)  
> true
```

```
> vegetal(palmera)  
> true
```

Josep Silva Galiana, UPV

Lenguajes de programación

Tipos de lenguajes de programación

Otra posible clasificación basada en los paradigmas de programación es:

- Lenguajes imperativos: se incluyen las instrucciones a seguir para resolver un problema.
- Lenguajes declarativos: se describe el resultado, sin especificar los pasos para alcanzarlo.
- Lenguajes funcionales: uso de funciones matemáticas.
- Lenguajes orientados a objetos: uso de clases y objetos para representar datos.
- Lenguajes lógicos: uso de reglas lógicas para expresar relaciones y realizar inferencias.

Compiladores e intérpretes

¿Qué son?

- Los lenguajes de alto nivel deben traducirse a código máquina para poder ejecutarse.

Compiladores e intérpretes

¿Qué son?

- Los lenguajes de alto nivel deben traducirse a código máquina para poder ejecutarse.

Compilador

- Lee **completamente** un programa en un lenguaje de alto nivel y lo traduce a código máquina.
- El programa resultante se puede ejecutar varias veces sin necesidad de traducir de nuevo el programa original.
- Si modificamos el programa es necesario volver a compilarlo completamente.

Compiladores e intérpretes

¿Qué son?

- Los lenguajes de alto nivel deben traducirse a código máquina para poder ejecutarse.

Intérprete

- Lee un programa escrito en un lenguaje de alto nivel **instrucción a instrucción**, traduciendo cada una de ellas a código máquina y ejecutándola.
- El proceso de traducción no está separado del de ejecución.
- Cada vez que ejecutamos el programa, se repite la traducción y ejecución.

Entorno de programación

IDE – Integrated Development Environment

- Aplicación informática que facilita el desarrollo de software.
- Existen numerosos entornos para cada lenguaje.
- En C, los más utilizados son:
 - Eclipse, Visual Studio, Netbeans, Dev-C++, Qt Creator, CodeBlocks, CLion, etc.

Entorno de programación

Herramientas típicas

- **Editores de texto:** permiten escribir programas.
- **Compiladores o intérpretes:** traducen el código a código máquina.
- **Depuradores:** ayudan a detectar errores.
- **Gestor de proyectos:** facilita la organización de ficheros y directorios de nuestro programa.
- **Consola integrada:** permite interactuar con la ejecución de nuestro programa.

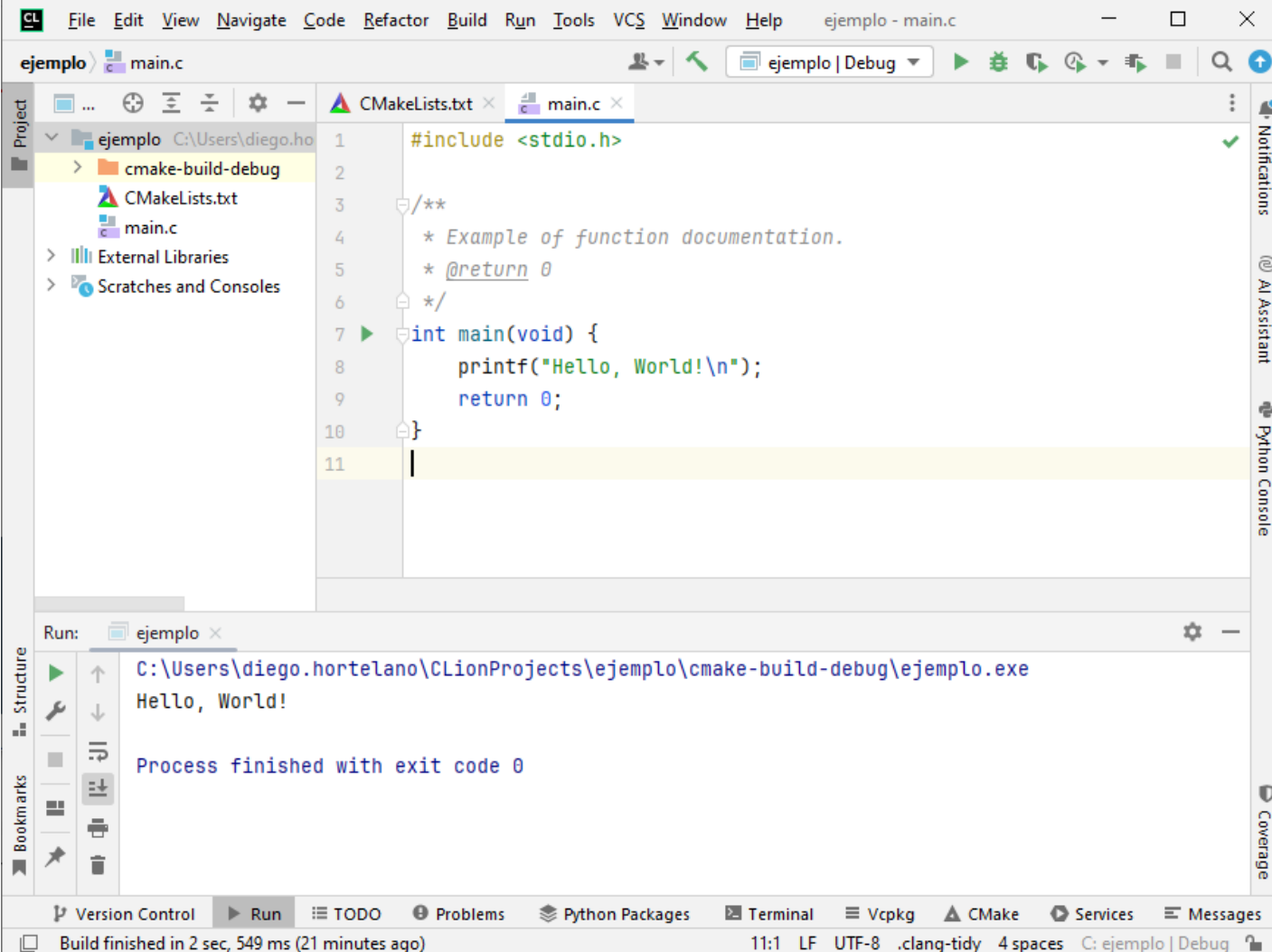
Entorno de programación

Herramientas típicas

- **Analizador en tiempo de ejecución:** estudia la eficiencia de los programas desarrollados.
- **Herramientas de pruebas:** nos permiten verificar que el código desarrollado funciona correctamente.
- **Generadores de documentación:** construyen la documentación a partir de los comentarios.
- **Sistemas de control de versiones:** facilita la colaboración, recuperación de código, etc.

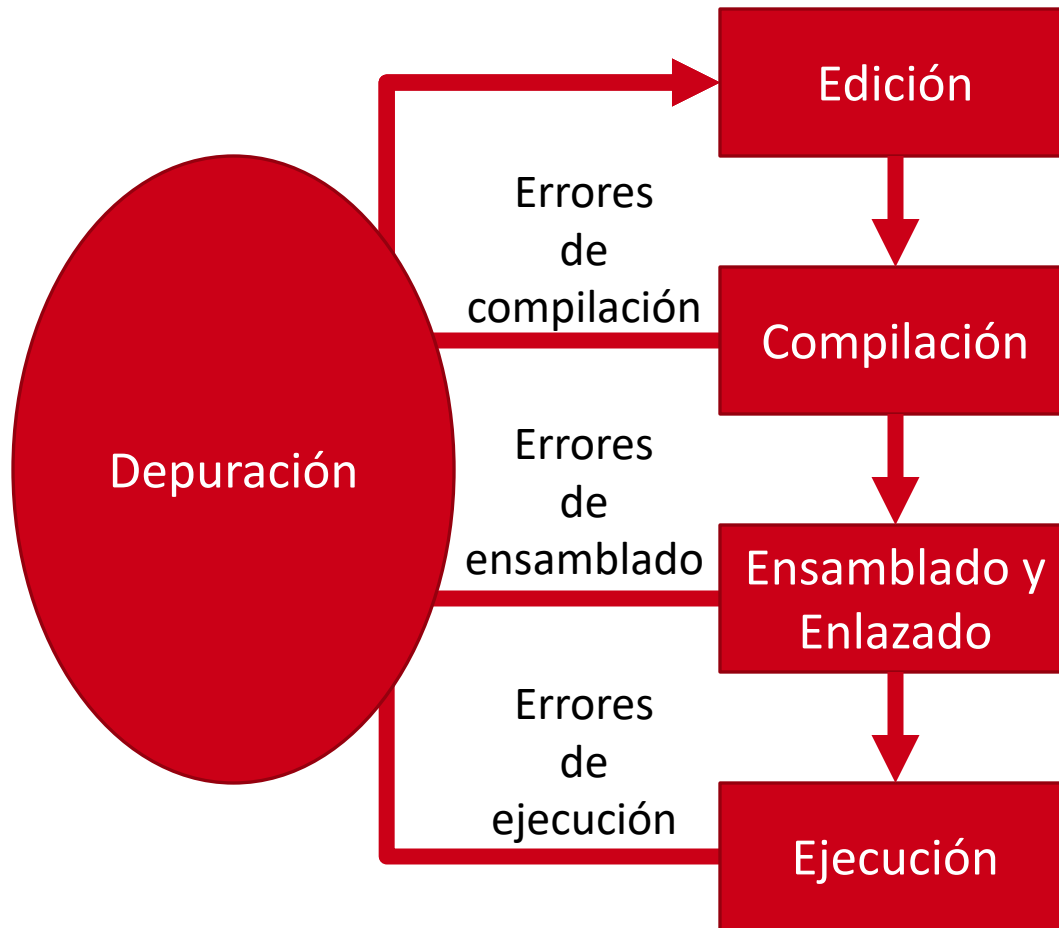
Entorno de programación

CLion

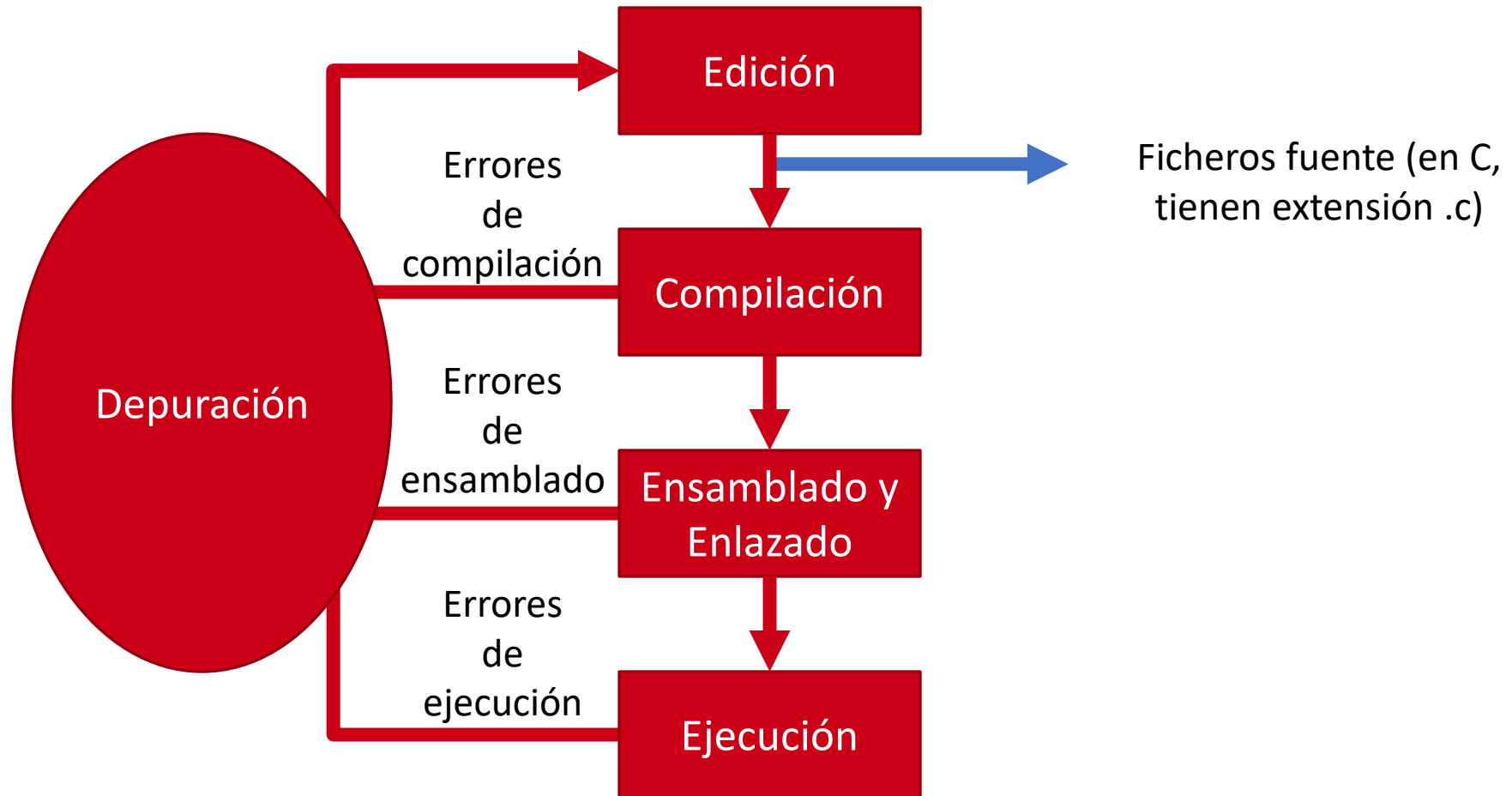


```
CL File Edit View Navigate Code Refactor Build Run Tools VCS Window Help ejemplo - main.c
ejemplo main.c ejemplo | Debug
Project ejemplo C:\Users\diego.ho
  cmake-build-debug
  CMakeLists.txt
  main.c
  External Libraries
  Scratches and Consoles
1 #include <stdio.h>
2
3 /**
4  * Example of function documentation.
5  * @return 0
6  */
7 int main(void) {
8     printf("Hello, World!\n");
9     return 0;
10 }
11 |
Run: ejemplo x
C:\Users\diego.hortelano\CLionProjects\ejemplo\cmake-build-debug\ejemplo.exe
Hello, World!
Process finished with exit code 0
Version Control Run TODO Problems Python Packages Terminal Vcpkg CMake Services Messages
Build finished in 2 sec, 549 ms (21 minutes ago) 11:1 LF UTF-8 .clang-tidy 4 spaces C: ejemplo | Debug
```

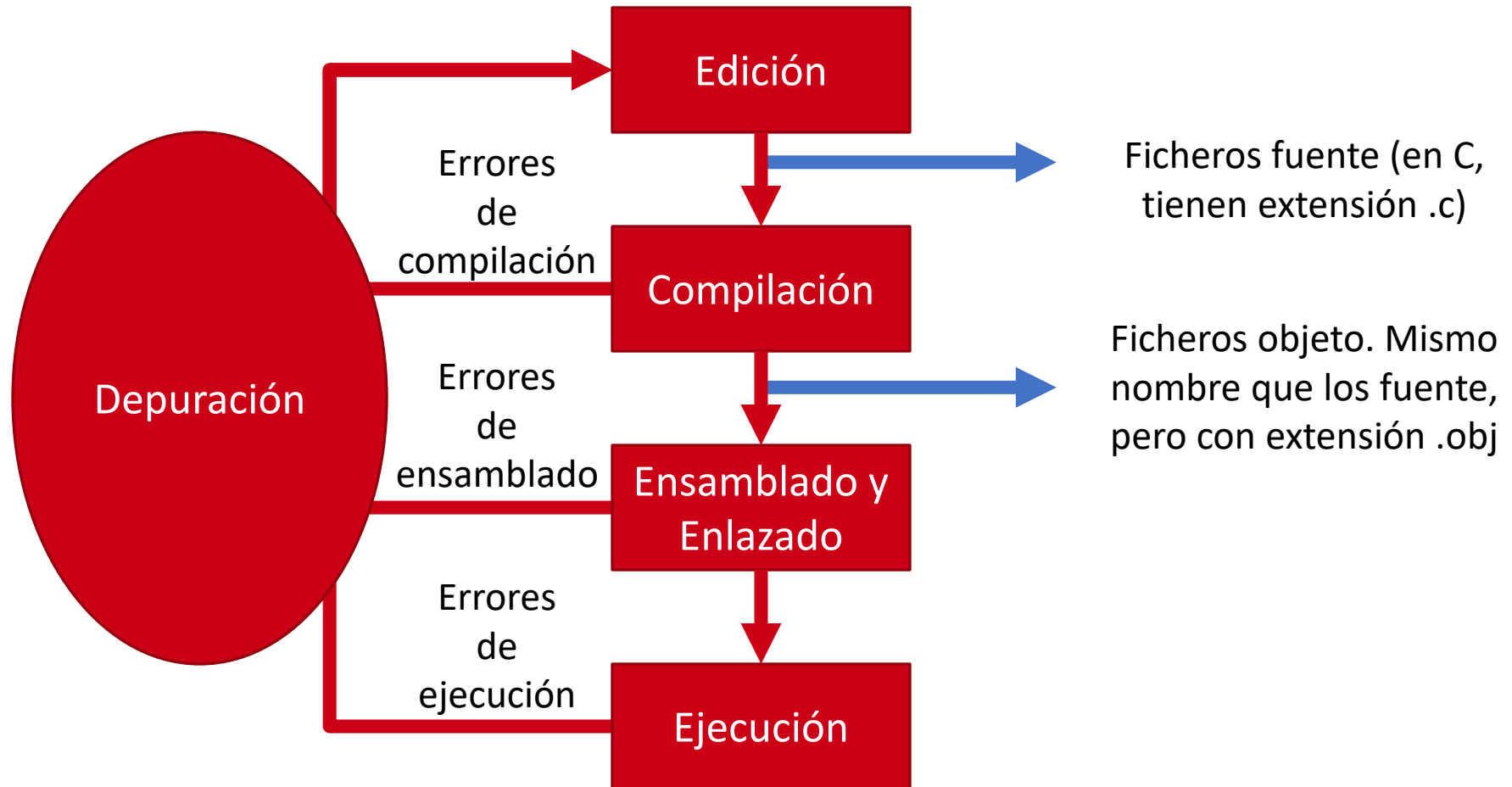
Pasos para la creación de un programa



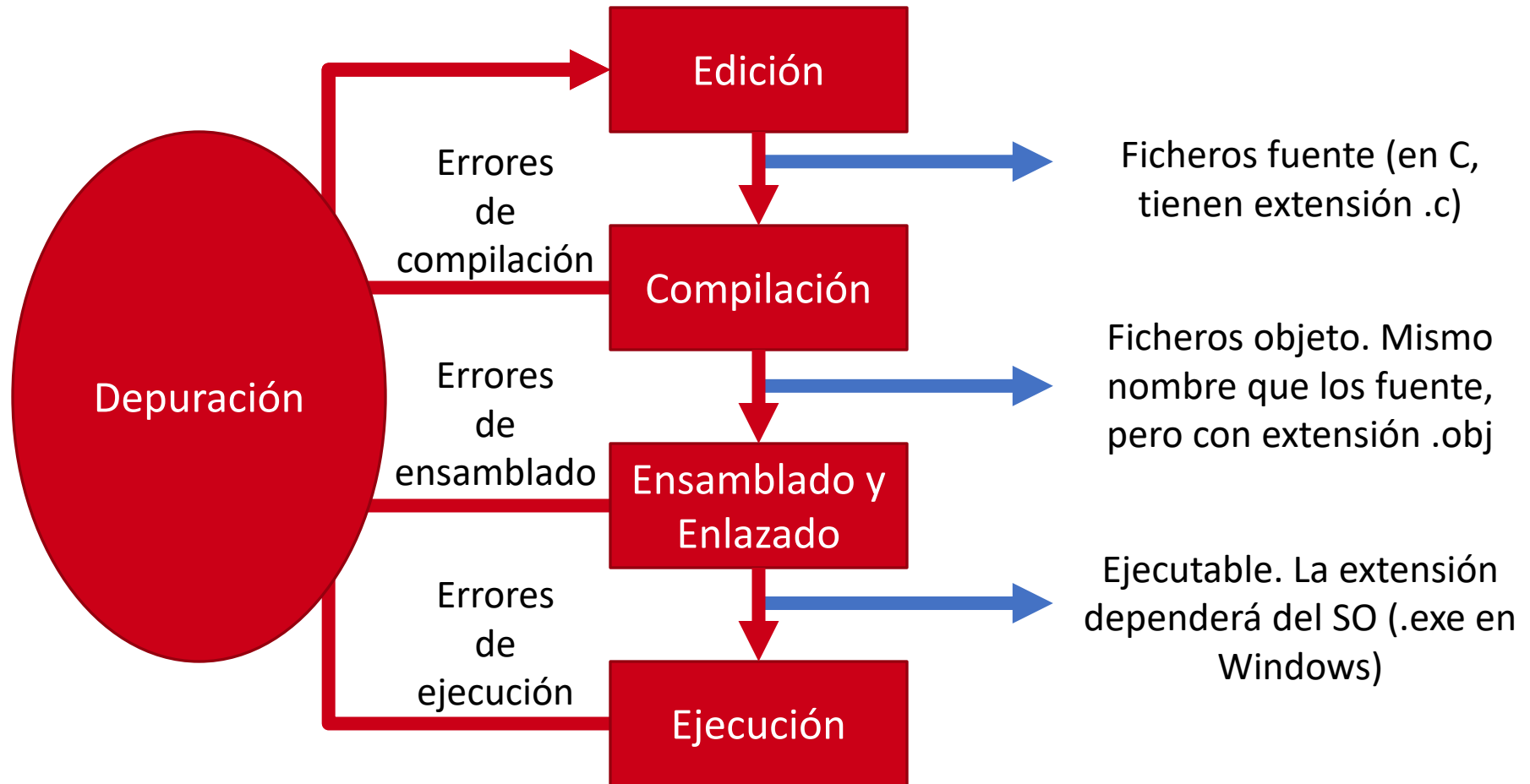
Pasos para la creación de un programa



Pasos para la creación de un programa



Pasos para la creación de un programa



Pseudocódigo

Definición

- Lenguaje de **descripción** algorítmico de **alto nivel**.
- Utiliza las convenciones estructurales de un lenguaje de programación real, pero prima la **lectura humana** (frente al lenguaje máquina).
- Es **independiente** de los lenguajes de programación.
- Muy utilizado para **describir algoritmos** de manera comprensible, sin necesidad de que todos los programadores conozcan el mismo lenguaje.

Pseudocódigo

Sintaxis

- No obedece a las reglas de sintaxis de ningún lenguaje, ni tiene definido un estándar, aunque puede verse influenciado por algún lenguaje de programación.
- A continuación, se muestra como ejemplo una convención para pseudocódigo.

Pseudocódigo

Sintaxis

- Asignar valores:

$x = y$

$x \leftarrow y$

- Operaciones matemáticas:

$res = num1 \cdot num2$

$res = y \% 3$

- Operaciones lógicas:

$x < 5$

$y == 1$

$x \neq 5 \text{ AND } y \geq 2$

- Estructuras condicionales:

Si condicion Entonces

instrucciones

Fin Si

Si condicion Entonces

instrucciones

Si no Entonces

instrucciones

Fin Si

Pseudocódigo

Sintaxis

- Estructuras iterativas o de repetición:

Mientras condicion Hacer
instrucciones
Fin Mientras

Repetir
instrucciones
Hasta Que condicion

Para var = 1 Hasta n Hacer
instrucciones
Fin Para

Pseudocódigo

Sintaxis

- Estructuras iterativas o de repetición:

Mientras condicion Hacer
instrucciones
Fin Mientras

Mientras y Repetir pueden funcionar igual si invertimos la condición!

Repetir
instrucciones
Hasta Que condicion

Para var = 1 Hasta n Hacer
instrucciones
Fin Para

Pseudocódigo

Ejemplo: pseudocódigo del algoritmo para extraer el valor mínimo

Algoritmo EncontrarMinimo

leer lista

longitud = longitud(lista)

mínimo = lista[0]

Para i=1 hasta longitud - 1 Hacer:

 elemento_actual = lista[i]

 Si elemento_actual < mínimo Entonces:

 mínimo = elemento_actual

 Fin Si

Fin Para

escribir mínimo

Fin Algoritmo EncontrarMinimo

Pseudocódigo

Ejemplo: pseudocódigo del algoritmo de búsqueda lineal

Algoritmo BúsquedaLineal

leer lista y leer número_buscar

encontrado = falso longitud = longitud(lista)

posición_actual = 0

Mientras posición_actual < longitud AND encontrado == falso Hacer:

 elemento_Actual = lista[posición_actual]

 Si elemento_Actual == número_buscar Entonces:

 encontrado = cierto

 Si no Entonces

 posición_actual = posición_actual + 1

 Fin Si

Fin Mientras

Si encontrado == cierto Entonces:

 escribir posición_actual

Si no Entonces

 escribir “No se ha encontrado”

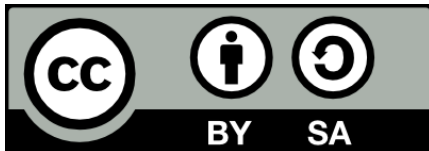
Fin Si

Fin Algoritmo BúsquedaLineal

Tema 2: Fundamentos del Lenguaje C

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.















Universidad
Rey Juan Carlos

- El lenguaje C
- Estructura y elementos de un programa en C
- Comentarios
- Directivas del Preprocesador
- Identificadores
- Palabras reservadas
- Tipos de datos
- Variables
- Tamaño de tipos y variables
- Constantes
- Lectura y escritura

El lenguaje C

Historia

- Ampliamente utilizado en la actualidad.
- <https://www.tiobe.com/tiobe-index/>

Sep 2024	Sep 2023	Change	Programming Language	Ratings	Change
1	1		 Python	20.17%	+6.01%
2	3	▲	 C++	10.75%	+0.09%
3	4	▲	 Java	9.45%	-0.04%
4	2	▼	 C	8.89%	-2.38%
5	5		 C#	6.08%	-1.22%
6	6		 JavaScript	3.92%	+0.62%
7	7		 Visual Basic	2.70%	+0.48%
8	12	▲▲	 Go	2.35%	+1.16%
9	10	▲	 SQL	1.94%	+0.50%
10	11	▲	 Fortran	1.78%	+0.49%
11	15	▲▲	 Delphi/Object Pascal	1.77%	+0.75%
12	13	▲	 MATLAB	1.47%	+0.28%

El lenguaje C

Historia

- Ligado al desarrollo de los sistemas operativos Unix.
- Inicialmente en el centro de investigación Bell Labs, con Ken Thompson y Dennis Ritchie utilizando ensamblador.
- Thompson creó una versión reducida del lenguaje BCPL acercándolo a la sintaxis de SMALGOL, que denominó B.
- Con el lanzamiento de los sistemas PDP-11, Ritchie adaptó el lenguaje B a las características de estos nuevos minicomputadores, denominándolo NB.
- Tras intentar migrar Unix a un lenguaje de alto nivel como NB, y fracasar, decidió añadir nuevas características a este lenguaje, lo que supuso el nacimiento de C.

El lenguaje C

Historia



Ken Thompson y Dennis Ritchie, extraída de la página web del Computer History Museum

- Estandarizado en 1983 por el ANSI, dando lugar al ANSI C: ISO/IEC 9899:1990 (C89 o C90).
- Posteriores revisiones del estándar:
 - ISO/IEC 9899:1999 (C99)
 - ISO/IEC 9899:2011 (C11)
 - ISO/IEC 9899:2018 (C17)
 - ISO/IEC 9899:2024 (C23)

El lenguaje C

Características

- **Eficiente y rápido**, al presentar combinar características de alto y bajo nivel.
- **Portable**, al existir compiladores para diferentes plataformas.
- Lenguaje muy **pequeño pero completo**, con estructuras de control y funciones que permiten estructurar y modular el código.
- Uso de **punteros**, proporcionando gran control sobre la memoria.
- Amplia gama de **bibliotecas** estándar.
- Gran variedad de **tipos de datos** básicos y posibilidad de crear datos complejos.

El lenguaje C

Utilidad

- Lenguaje ampliamente usado en el desarrollo de aplicaciones informáticas: sistemas operativos, sistemas distribuidos, sistemas empotrados, redes, telemática y transmisión de datos.
- Además, sigue siendo un lenguaje de propósito general que se usa también en **aplicaciones profesionales** de otros campos, como robótica, visión artificial, optimización, etc.

Estructura y elementos de un programa en C

- En general, un programa en C puede incluir los siguientes elementos:
 - Directivas del preprocesador
 - Funciones
 - Destaca la función `main()`, de uso obligatorio
 - Declaraciones de variables
 - Diferentes ámbitos
 - Comentarios

Estructura y elementos de un programa en C

```
1  /*
2     * Nombre: Código de ejemplo
3     * Autor: Diego
4     */
```

Todo lo encerrado entre
/* */ es un comentario

```
6  /* Directivas del preprocesador */
7  #include <stdio.h>
```

Directivas del Preprocesador

```
9  /* Definición de viariables globales */
10 int variable_global;
```

Declaración de variables globales

```
12 /* Función que imprime el valor de la variable global y retorna 1. */
13 int funcion()
14 {
15     // imprimimos el valor de la variable global
16     printf("valor variable_global en funcion: %d\n", variable_global);
17     return 1;
18 }
```

Función definida

Estructura y elementos de un programa en C

Comentarios con //

Función main (programa principal)

```
20 //Función main
21 int main(void) {
22     // Definición de variables locales
23     int variable_local = 2;    Declaración de variables locales
24
25     variable_global = variable_local;
26
27     // imprimimos por pantalla la variable local y la variable global
28     printf("valor variable_global: %d\n", variable_global);
29     printf("valor variable_local: %d\n", variable_local);
30
31     funcion();
32
33     return 0;
34 }
```

Comentarios

¿Qué son?

- Permiten **documentar** el programa:
 - Facilita el mantenimiento del código.
 - Permite a otros desarrolladores (y a nosotros mismos) comprender rápidamente el código.
- Son **ignorados** por el compilador.
 - No se traducen a código máquina.
 - No influirán en la ejecución del programa.
- Se **recomienda** su uso:
 - En expresiones que no sean obvias.
 - Para comentar aspectos importantes del código.

Comentarios

Tipos y ejemplos

//

- Comentarios de una línea.
- La línea que comience por doble barra se trata como un comentario.
- Solo comenta esa única línea.

/* */

- Comentarios de bloque.
- El texto introducido entre /* y */ se trata como un comentario.
- Puede ocupar más de una línea.

Directivas del Preprocesador

¿Qué son?

- En C existe una etapa de preprocesamiento anterior a la compilación.
- En esta etapa se prepara el código fuente antes de ser compilado.
- Es posible incluir directivas al procesador, las cuales comienzan por # y se escriben al comienzo del código fuente.

Directivas del Preprocesador

Tipos de directivas

#include

- Permite incluir código fuente del archivo indicado.

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

#define

- Define constantes y macros, que pueden utilizarse en el código en lugar de los valores literales, mejorando la legibilidad. Suelen ir en mayúsculas.

```
#define PI 3.14  
#define CUADRADO(x) ((x) * (x))
```

Directivas del Preprocesador

Tipos de directivas

#if, #ifdef, #ifndef, #else, #elif y #endif

- Permiten controlar la compilación de bloques de código en función de si se cumplen o no determinadas condiciones.

```
#include <stdio.h>

// Definir DEBUG
#define DEBUG

int main() {
#ifdef DEBUG
    printf("Modo Debug\n");
#endif

    printf("Modo Normal\n");
    return 0;
}
```

Identificadores

¿Qué son?

- Son nombres que podemos asignar a diferentes elementos de nuestro programa, como variables, funciones o tipos. Estos nombres hacen referencia a la dirección de memoria donde se almacena el elemento representado.

Algunos ejemplos:

- `int variable;`
- `int main(void) {...}`

Identificadores

Normas

- Cada lenguaje tiene sus **propias normas** para los identificadores.
- En C, debe ser una **cadena de caracteres** que puede contener únicamente:
 - Letras minúsculas y mayúsculas.
 - Dígitos numéricos (0-9).
 - El carácter '_' (guion bajo).
- **No** pueden aparecer espacios ni otros símbolos de puntuación.

Identificadores

Normas

- El **primer carácter** debe ser obligatoriamente una letra.
- Se recomienda que los identificadores no tengan más de 31 caracteres.
- Se **distingue** entre minúsculas y mayúsculas.
 - *Resultado* y *resultado* son identificadores diferentes.
- Como buena práctica, debemos elegir identificadores descriptivos para facilitar la **legibilidad y mantenimiento** del código.

Identificadores

Ejemplos

- No pueden utilizarse las **palabras reservadas** como identificadores.
- Ejemplos:
 - **Válidos:**
`media, precioProducto, resultado1, prueba, ejemplo2, PI, VELOCIDAD_LUZ, numero_alumnos,`
 - **No válidos:**
`2valor, tiempo-total, dolar$, %final`

Palabras reservadas

- Palabras que tienen un **significado especial** para el lenguaje, por lo que **NO** pueden utilizarse como identificadores.
- En ANSI C hay 32 palabras reservadas:
`auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.`

Tipos de datos

Tipos de datos en C

- Cada tipo de dato se define por:
 - Representación interna en memoria.
 - Rango.
 - Operaciones que pueden realizar.
- En C hay tres tipos de datos básicos: enteros (**int**), reales (**float**) y carácter (**char**).

Tipos de datos

Tipos de datos básicos

Enteros - int

- Números enteros positivos y negativos (sin decimales)
- En C dependen de la arquitectura del sistema, y pueden ocupar 2 o 4 bytes en memoria, por lo que podemos representar $2^{16} = 65.536$ o $2^{32} = 4.294.967.296$ valores diferentes.
- Rango:
 - $-32768 \dots + 32767$ (2 bytes)
 - $-2147483648 \dots + 2147483647$ (4 bytes)
- Ejemplos: 1000, 4, -523 , 1187, ...

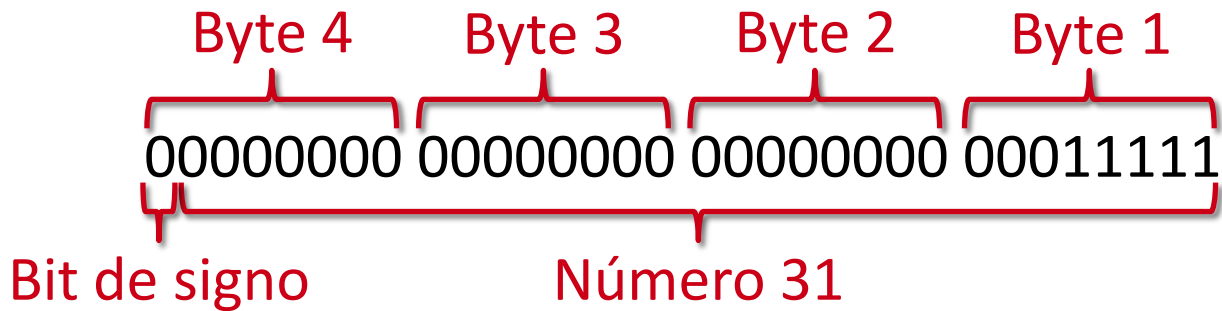
Tipos de datos

Tipos de datos básicos

Enteros – int (almacenamiento en memoria)

Ejemplo:

$$(31)_{10} = (11111)_2$$



Tipos de datos

Tipos de datos básicos

Enteros – int negativos (almacenamiento en memoria)

Ejemplo: -31

MSB

10000000 00000000 00000000 00011111

Bit de signo Número 31

Complemento a 1

11111111 11111111 11111111 11100000

Bit de signo Complemento a 1 de 31

Complemento a 2

11111111 11111111 11111111 11100001

Bit de signo Complemento a 2 de 31

Tipos de datos

Tipos de datos básicos

Reales - float

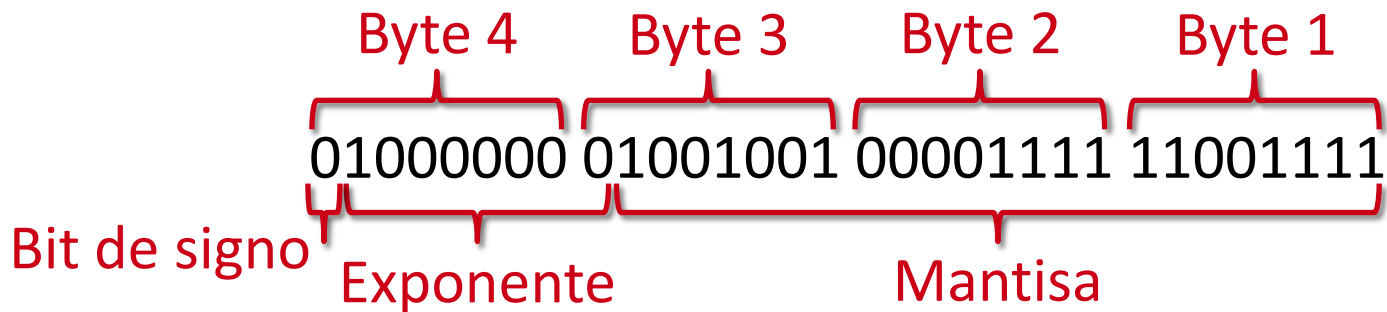
- Números reales con signo y con parte decimal.
- Suelen ocupar 4 bytes en memoria.
- Rango: $-3.4 \times 10^{38} \dots + 3.4 \times 10^{38}$
- Ejemplos: -50.6 , 20.0 , -3.2 , 3.14 , $10e + 9$,
 $-45e - 13$, ...

Tipos de datos

Tipos de datos básicos

Reales – float (almacenamiento en memoria)

$$(3.141590)_{10} = (0011.0010010000111111001111)_2$$



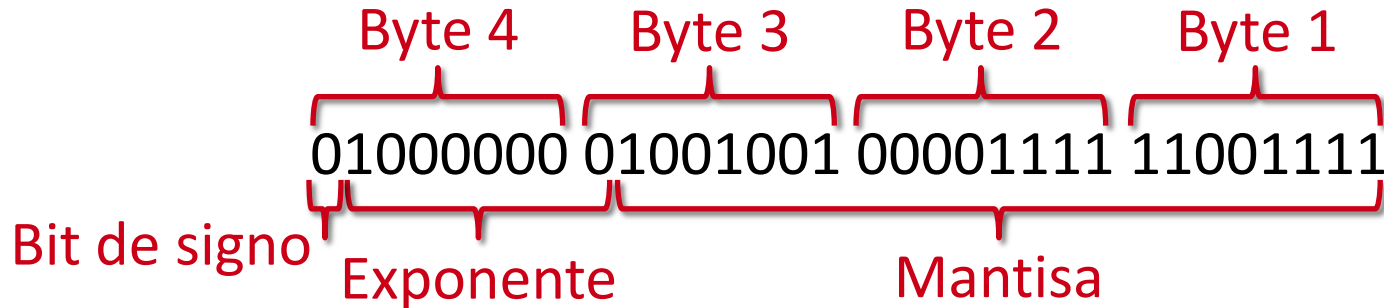
Tipos de datos

Tipos de datos básicos

Reales – float (almacenamiento en memoria)

$$(3.141590)_{10} = (0011.0010010000111111001111)_2$$

- Transformación a $numero \cdot 2^{exponente} \rightarrow 1.100100 \dots \cdot 2^1$
- $exp = 1$; $exponente\ normalizado = (128)_{10} = (10000000)_2$
- Parte decimal a mantisa
- Bit de signo



Tipos de datos

Tipos de datos básicos

Reales – float (almacenamiento en memoria)

- ¡Cuidado con las posibles pérdidas de precisión!
- ¿Qué mostrará este código?

```
#include <stdio.h>

int main ()
{
    float a, b, c;

    a = 1.345f;
    b = 1.123f;
    c = a + b;

    if (c == 2.468)
        printf("Son iguales.\n");
    else
        printf("NO son iguales\n");
}
```

Tipos de datos

Tipos de datos básicos

Reales – float (almacenamiento en memoria)

- ¿Y este?

```
#include <stdio.h>

int main ()
{
    float a, b, c;

    a = 1.345f;
    b = 1.123f;
    c = a + b;
    if (c == 2.468)
        printf("Son iguales.\n");
    else
        printf("NO son iguales. El valor de C es %13.10f or %f",c,c);
}
```

Tipos de datos

Tipos de datos básicos

Reales – float (almacenamiento en memoria)

- Uso de un épsilon para comparar cantidades float.
- Un épsilon es el valor más pequeño que produce un valor diferente cuando se suma a un número representado en punto flotante.
- Podemos elegir un épsilon adecuado en función de nuestra aplicación, o podemos hacer uso de los definidos en la biblioteca `<float.h>`, que debemos incluir en nuestro código:

Constante	Precisión	Valor aproximado
FLT_EPSILON	float	$1,19 \cdot 10^{-7}$
DBL_EPSILON	double float	$2,22 \cdot 10^{-16}$
LDBL_EPSILON	long double	$1,08 \cdot 10^{-19}$

Tipos de datos

Tipos de datos básicos

Reales – float (almacenamiento en memoria)

- ¿Qué resultado produce este otro programa?

```
#include <stdio.h>

#define TOLERANCIA 0.0001 // Definimos la tolerancia

int main ()
{
    float a, b, c;

    a = 1.345f;
    b = 1.123f;
    c = a + b;
    if ((c - TOLERANCIA < 2.468) && c + TOLERANCIA > 2.468)
        printf("Son iguales.\n");
    else
        printf("NO son iguales. El valor de C es %13.10f or %f",c,c);
}
```

Tipos de datos

Tipos de datos básicos

Carácter - char

- Caracteres de texto.
- Suelen ocupar 1 byte.
- Su rango es el de los caracteres ASCII:
 - De 0 a 127 (7 bits) para el código ASCII.
 - De 0 a 255 (8 bits) para el código ASCII extendido.
- Se almacena el valor entero del carácter representado.
- Podemos encontrar todos los caracteres ASCII junto con sus correspondientes valores en:

<https://www.ascii-code.com/>

Tipos de datos

Tipos de datos derivados (int)

- Se indican añadiendo los modificadores short, long y unsigned.

Tipo en C	Descripción	Rango	Nº bytes
short int short	Entero corto con signo	−32.768 ... 32.767	2
unsigned short int unsigned short	Entero corto sin signo	0 ... 65535	2
int	Entero con signo	−32.768 ... 32.767 −2.147.483.648 ... 2.147.483.647	2 4
unsigned int	Entero sin signo	0 ... 65535 0 ... 4.294.967.295	2 4
long int long	Entero largo con signo	−2.147.483.648 ... 2.147.483.647	4
unsigned long int unsigned long	Entero largo sin signo	0 ... 4.294.967.295	4

Tipos de datos

Tipos de datos derivados (float)

- Se indican con diferentes palabras clave que modifican su tamaño y rango: `float`, `double` y `long double`:

Tipo en C	Descripción	Rango	Nº bytes
float	Real	$-3,4 \cdot 10^{38} \dots 3,4 \cdot 10^{38}$	4
double	Real con doble precisión	$-1,7 \cdot 10^{308} \dots 1,7 \cdot 10^{308}$	8
long double	Real de precisión extendida	$-3,4 \cdot 10^{4932} \dots 3,4 \cdot 10^{4932}$	16

Tipos de datos

Tipo de datos lógico

- En ciertos lenguajes existe un tipo de datos denominado **lógico** o **booleano**. Este tipo de datos permite almacenar los valores lógicos true o false.
- En C **no existe** este tipo, por lo que se utiliza una convención con el tipo int para almacenar valores lógicos:
 - El **valor 0** es falso.
 - Cualquier valor **diferente** de 0 es verdadero.
- Posibilidad de utilizar la biblioteca `<stdbool.h>`.

Variables

¿Qué son?

- Las variables son espacios de almacenamiento que tienen un nombre asociado que permite referirnos a ellas. Vienen dadas por:
 - Un **identificador**, el nombre de la variable.
 - El **tipo de dato** que almacena.
 - El **valor** almacenado, que puede cambiar durante la ejecución.
- Cuando se utiliza el identificador en una expresión, se toma como **operando su valor**.

Variables

Declaración

- Proceso de reserva de memoria para la variable en el momento de compilar. Es necesario indicar:
 - Tipo de dato
 - Identificador
- En C es necesario **declarar** las variables antes de utilizarlas, **reservándose** automáticamente la memoria necesaria.

- Ejemplos:

```
int a;
```

```
float pi = 3.14;
```

```
char inicial = 'J';
```

Variables

Inicialización

- Tras declarar una variable, ésta tendrá un valor **desconocido**.
 - En otros lenguajes el valor inicial es 0, pero **esto no sucede en C**.
- La **inicialización** es la asignación de un valor inicial a la variable.
- Ejemplos:

```
char letra = 'C';
```

```
float altura;
```

```
altura = 1.83;
```

Variables

Tipos de variables

- En función de dónde se declare una variable podemos distinguir tres tipos de variables:
 - **Variables globales:** declaradas fuera de todas las funciones, accesibles desde cualquier parte del código. Se recomienda **evitar** su uso como buena práctica.
 - **Variables locales:** declaradas dentro de una función, accesibles únicamente desde dicha función.
 - **Variables de bloque:** declaradas dentro de un bloque de código, accesibles únicamente desde dicho dentro de dicho bloque.

Tamaño de tipos y variables

Tamaño de los tipos de datos

- El número de bytes que ocupa cada dato en memoria **no es fijo**, dependiendo de la arquitectura del ordenador y del compilador.
- Para dotar de portabilidad a nuestros programas se recomienda utilizar el operador **sizeof** (), que permite conocer el **tamaño exacto** que ocupa un tipo de datos o una variable:
 - A partir de un tipo: **sizeof** (tipo)
 - A partir de una variable: **sizeof** (nombreVar)

Constantes

¿Qué son?

- Datos cuyo valor no cambia durante la ejecución de nuestro programa. Existen dos tipos en C:

Literales

- Su valor se escribe directamente en el código.

No literales o simbólicas

- Se representan con un identificador:

- Mediante directiva del preprocesador:

```
#define PI 3.14159
```

- Mediante la palabra clave `const` en la declaración:

```
const float pi = 3.14159;
```


Constantes

Ventajas

- Mejoran la **legibilidad** del código.
- Evitan **errores de escritura** al evitar la repetición.
- Si necesitamos modificar su valor en el futuro solo debemos cambiarlo en la **declaración**.

Lectura y escritura

Entrada/Salida estándar

- Uso de la librería **estándar** de C `<stdio.h>` (*standard input/output*), que ofrece funciones predefinidas para:
 - **Lectura** de datos por teclado.
 - **Escritura** de datos por pantalla.
- Para utilizar estas funciones es necesario incluir previamente la biblioteca utilizando la directiva del preprocesador:

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

Lectura y escritura

Códigos de control

- Caracteres que permiten especificar el tipo de datos en funciones de lectura y escritura.
- Comienzan por el carácter '%'.
• Si el tipo especificado no coincide con el tipo del dato al que hacemos referencia, C intentará transformar el dato.

Lectura y escritura

Códigos de control

Carácter	Nombre
%i	Entero
%u	Entero sin signo
%d	Entero en base decimal
%x	Entero en base hexadecimal
%f	Número en punto flotante
%lf	Flotante de doble precisión
%Lf	Datos de tipo long double
%c	Carácter
%s	Cadena de caracteres
%h %hu	Entero corto y con y sin signo
%ld %lu	Entero largo con y sin signo
%e	Notación científica con e para el exponente
%E	Notación científica con E para el exponente
%Le %LE	Datos de tipo long double con notación científica
%p	Puntero

Lectura y escritura

Secuencias de escape

- Combinaciones de caracteres que representan a caracteres especiales que no podrían indicarse de otra forma dentro de una cadena.
- Comienzan por el carácter '`\`':

Carácter	Nombre
<code>\n</code>	Nueva línea
<code>\t</code>	Tabulación
<code>\r</code>	Retorno de carro al inicio de la línea actual
<code>\b</code>	Retroceso
<code>\\</code>	Barra invertida (' <code>\</code> ')
<code>\'</code>	Comilla simple
<code>\"</code>	Comilla doble
<code>\0</code>	Carácter nulo

Lectura y escritura

Funciones de salida

- Una de las funciones más utilizadas es `printf()`, que proviene de "print formatted", y permite escribir texto con formato en la **consola**.
- Permite mostrar texto con el valor de nuestras variables, para lo que es necesario definir una **cadena de control** que indique dónde y cómo se mostrarán las variables utilizando **códigos de control**.

Lectura y escritura

Funciones de salida

- Sintaxis:

```
printf("Cadena de control", dato1, ..., datoN);
```

- La cadena de control contiene:

- La **cadena de texto** a mostrar (puede incluir caracteres de escape).
- Los **códigos de control** necesarios para indicar los **tipos de los datos** a mostrar.
- La **forma** de mostrarlos.

Lectura y escritura

Funciones de salida

- La función `printf()` permite definir el **tamaño** con el que cada variable se mostrará en la consola, así como su alineación.
- Para ello, podemos escribir el **número de dígitos** que queremos mostrar entre el `%` y el carácter del código de formato. Podemos indicar una alineación a la izquierda si el número es negativo.

Lectura y escritura

Funciones de salida

Ejemplos:

```
int i = 516;
float x = 1024.251;
printf(":%6d: \n", i);
// se escribe -> : 516:
printf(":%-6d: \n", i);
// alineación izq -> :516 :
printf(":%2d: \n", i);
// no cabe, formato por defecto -> :516:
printf(":%12.4f:\n", x);
// anchura 12 con 4 decimales -> : 1024.2510:
printf(":%12e: \n", x);
// anchura 12 caracteres -> :1.024251e+03:
printf(":%12.4e:", x);
// anchura 12 con 4 decimales -> : 1.0243e+03:
```

Lectura y escritura

Funciones de salida

- La biblioteca `<stdio.h>` incluye también otras funciones de salida para mostrar texto:

- `puts()`, que permite mostrar una cadena de caracteres con un salto de línea:

```
puts("Texto a imprimir");
```

- `fputs()`, similar a la anterior, pero indicando el buffer de salida:

```
fputs("Texto a imprimir", stdout);
```

- `putchar()`, imprime un único carácter por consola:

```
putchar('A');
```

Lectura y escritura

Funciones de entrada

- Una de las funciones más utilizadas para entrada de datos es `scanf()`, que proviene de "scan formatted", y permite especificar el formato del texto de entrada.
- Al igual que `printf()`, requiere una cadena de control con los códigos de control que indiquen los **tipos de datos** a leer.

```
scanf ("Cadena de control", &var1, ..., &varN);
```

- Las variables leídas deben ir precedidas del **carácter &** si no son de tipo cadena de caracteres, ya que esta función requiere la dirección de memoria de la variable (más adelante se profundizará sobre esto).

Lectura y escritura

Funciones de entrada

- Cuando se lee una cadena con %s se lee una **palabra** que debe ser almacenada en una cadena de caracteres (con []).
- En ese caso se lee hasta que aparezca un espacio o fin de línea.
- Si deseamos leer **más de una palabra** (incluyendo el espacio), podemos sustituir %s por % [^ \n] .

Lectura y escritura

Funciones de entrada

- Algunos ejemplos:

```
int main()  
{  
    int varInt1, varInt2;  
    float varFloat;  
    char palabra[10];  
    scanf("%f", &varFloat);  
    scanf("%d %d", &varInt1, &varInt2);  
    scanf("%s", palabra);  
    printf("%s", palabra);  
    return 0;  
}
```

Lectura y escritura

Funciones de entrada

- La biblioteca `<stdio.h>` incluye también otras funciones de entrada para leer cadenas de caracteres:

- `gets()`, que permite leer una cadena de caracteres hasta un salto de línea:

```
gets (palabra) ;
```

- `fgets()`, similar a la anterior, pero indicando el buffer de entrada y el número máximo de caracteres:

```
fgets (palabra, 20, stdin) ;
```

- `getchar()`, lee un único carácter:

```
c = getchar() ;
```

Lectura y escritura

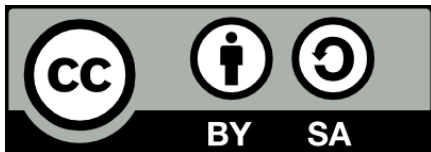
Buffers de entrada y salida

- En C, la entrada y salida de datos se manejan como buffers o streams de datos.
- Los buffers son áreas de memoria temporales que almacenan los datos mientras se transfieren entre el programa y los dispositivos de entrada/salida.
- Mejora la eficiencia de la entrada/salida.
- En C, hay tres:
 - `stdin` (entrada estándar).
 - `stdout` (salida estándar).
 - `stderr` (salida de error estándar).

Tema 3: Operadores y Expresiones

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

Índice

Temas

- Instrucción de asignación.
- Expresiones.
- Operadores y tipos.
 - Operadores aritméticos.
 - Operadores incrementales.
 - Operadores relacionales.
 - Operadores lógicos.
 - Otros operadores.
 - Operador ternario.
- Conversiones entre tipos.
 - Conversión implícita.
 - Conversión explícita.
- Reglas de precedencia.
- Biblioteca math.h

La instrucción de asignación

Definición

- Proporciona el valor de una expresión a una variable.
- Sintaxis:

variable = expresión

Donde:

- ***variable***, es un identificador que hace referencia a un espacio de memoria.
- ***expresión***, es una combinación de valores, variables, operadores, o funciones que devuelve un resultado.
- **Reemplaza** cualquier valor que hubiera en la variable con el valor de la expresión.

La instrucción de asignación

Ejemplo básico

La asignación es una de las operaciones más fundamentales en programación y se utiliza para actualizar el valor almacenado en una variable.

```
int x;
```

```
x = 5; // Se asigna el valor 5 a la variable x
```

La instrucción de asignación

Ejemplo detallado


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

```
int main() {  
    short a, b, resultado;  
    a = 3;  
    b = a*5;  
    a = a+1;  
    resultado = ((a*5)/4)+3-b;  
    printf("a = %d y b = %d \n", a, b);  
    printf("El resultado es: %d \n", resultado);  
    return 0;  
}
```

**¿QUÉ VALORES TENDRÁN
LAS VARIABLES a , b AL FINAL
DE LA EJECUCIÓN?**



**¿QUÉ VALOR
TENDRÁ LA
VARIABLE
resultado?**



La instrucción de asignación

Operadores de asignación

- Operadores compuestos que permiten realizar una operación y asignar el resultado en una sola expresión:

Operador	Descripción	Sintaxis	Equivalente
=	Asignación simple	a = b	a = b
+=	Suma y asignación	a += b	a = a + b
-=	Resta y asignación	a -= b	a = a - b
*=	Multiplicación y asignación	a *= b	a = a * b
/=	División y asignación	a /= b	a = a / b
%=	Resto y asignación	a %= b	a = a % b

La instrucción de asignación

Operadores de asignación

- Sea op cualquiera de los operadores $+=$, $-=$, $*=$, $/=$ ó $\%=$, la instrucción:

*variable **op** expresión*

- Equivale a:

*variable = variable **op** expresión*



Para los operadores aritméticos +, -, *, etc.

La instrucción de asignación

Ejemplos

- Algunos ejemplos:

Expresión	Expresión Equivalente
<code>posicion += 1;</code>	<code>posicion = posicion + 1;</code>
<code>longitud /= 2.0;</code>	<code>longitud = longitud / 2.0;</code>
<code>x *= 3.0*y - 1.0;</code>	<code>x = x * (3.0*y - 1.0);</code>

La instrucción de asignación

Ejemplo completo

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

```
int main() {  
    int x = 10;  
    int y = 5;  
    // Asignación simple  
    x = y; // x ahora es igual a 5  
    // Asignación con suma  
    x += 2; // x ahora es 7 (5 + 2)  
    // Asignación con resta  
    x -= 3; // x ahora es 4 (7 - 3)  
    // Asignación con multiplicación  
    x *= 2; // x ahora es 8 (4 * 2)  
    // Asignación con división  
    x /= 4; // x ahora es 2 (8 / 4)  
    printf("x final: %d\n", x); // Salida: 2  
    return 0;  
}
```


Expresiones

Definición

- Define cómo obtener un resultado a partir de los operandos.
- Para evaluar una expresión se **sustituye** cada identificador por su valor y se **realizan** las operaciones indicadas siguiendo las reglas de precedencia y asociatividad.
- Se pueden utilizar paréntesis para **agrupar** operaciones y espacios en blanco para mejorar la **claridad**.

Operadores

Definición

- Carácter o grupo de caracteres que actúan sobre variables para realizar una **operación** y obtener un **resultado**.
- En C existen diferentes tipos de operadores, siendo posible clasificarlos en diferentes **grupos**: aritméticos, incrementales, relacionales y lógicos.
- Dependiendo del **número de operandos** sobre los que actúen, los operadores pueden ser unarios, binarios, ternarios, etc.

Operadores

Operadores aritméticos

- Permiten realizar operaciones aritméticas, y en C encontramos los siguientes:

Operador	Descripción	Ejemplo
+	Suma	$a+b$
-	Resta	$a-b$
*	Multiplicación	$a*b$
/	División	a/b
%	Resto de la división entera	$a\%b$

Operadores

Operadores aritméticos

- Todos son **binarios**.
- El tipo del resultado depende de los operandos:
 - Si los dos **operandos** son enteros, el **resultado** es entero.
 - Si alguno de los **operandos** es real, el **resultado** es real.
- El operador **resto o módulo (%)** solo se puede aplicar a operandos de **tipo entero**.

Operadores

Ejemplo de operadores aritméticos

En esta expresión, se realiza una suma entre **a** y **b** utilizando el **operador +** y el resultado se multiplica por 2, para ello se usa el **operador ***.

```
int a = 10;
```

```
int b = 20;
```

```
int result = (a + b) * 2; // Expresión aritmética
```

Operadores

Operadores incrementales

- Operadores **unarios** que permiten **incrementar** o **disminuir** en una unidad el valor de la variable a la que afectan. Pueden situarse delante o detrás de la variable:

Operador	Descripción	Ejemplo
++	Pre-incremento (Prefijo)	++a
	Post-incremento (Sufijo)	a++
--	Pre-incremento (Prefijo)	--a
	Post-incremento (Sufijo)	a--

Operadores

Operadores incrementales

- Si preceden a la variable, ésta se incrementa o decrementa **antes** de evaluar la expresión (pre-incremento o pre-decremento).
- Si la variable precede al operador, la variable se incrementa o decrementa **después** de evaluar la expresión (post-incremento o post-decremento).
- Ejemplos:

```
int i = 2;  
int j = 2;  
printf("%d\n", i++);  
printf("%d\n", ++j);
```

Operadores

Ejemplo de operador incremental (++)

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

```
int main() {  
    int x = 5;  
    // Incremento prefijo  
    int y = ++x; // x se incrementa primero, luego se asigna a y  
    printf("Incremento prefijo: x = %d, y = %d\n", x, y);  
    // Incremento sufijo  
    int z = x++; // x se asigna a z primero, luego se incrementa  
    printf("Incremento sufijo: x = %d, z = %d\n", x, z);  
    return 0;  
}
```


Operadores

Ejemplo de operador decremental (--)

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

```
int main() {  
    int x = 5;  
    // Decremento prefijo  
    int y = --x; // x se decrementa primero, luego se asigna a y  
    printf("Decremento prefijo: x = %d, y = %d\n", x, y);  
    // Decremento sufijo  
    int z = x--; // x se asigna a z primero, luego se decrementa  
    printf("Decremento sufijo: x = %d, z = %d\n", x, z);  
    return 0;  
}
```

Operadores

Operadores relacionales

- Permiten **comparar** dos expresiones resultando en un valor **verdadero** o **falso**.
- Como en C **no existe** el tipo booleano, las variables lógicas se declaran como enteros.

Operador	Descripción	Ejemplo
==	Igual que	$a == b$
<	Menor que	$a < b$
>	Mayor que	$a > b$
<=	Menor o igual que	$a <= b$
>=	Mayor o igual que	$a >= b$
!=	Distinto que	$a != b$

Operadores

Ejemplo de operadores relacionales

```
#include <stdio.h>
int main() {
    int a = 10, b = 20, c = 10;
    // Uso de operadores de comparación
    if (a == c) {
        printf("a es igual a c\n"); // a == c es verdadero
    }
    if (a != b) {
        printf("a no es igual a b\n"); // a != b es verdadero
    }
    if (b > a) {
        printf("b es mayor que a\n"); // b > a es verdadero
    }
    if (a <= c) {
        printf("a es menor o igual a c\n"); // a <= c es verdadero
    }
    return 0;
}
```

Operadores

Operadores lógicos

- Los operadores lógicos permiten **combinar** los resultados de expresiones lógicas, por lo que dichos operandos deben ser expresiones lógicas.

Operador	Descripción	Ejemplo
&&	AND	$(a > b) \ \&\& \ (b > 0)$
	OR	$(a > b) \ \ (b > 0)$
!	NOT	$!(a > b)$

Operadores

Tabla de verdad de operadores lógicos

- En función del valor de las entradas, podemos crear una tabla de verdad con los diferentes operadores lógicos:

a	b	!a	a&&b	a b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Operadores

Otros operadores

- Existe otro conjunto de operadores que no pueden clasificarse en el resto de grupos. Estos son:

Operador	Descripción	Ejemplo
+	Identidad	+a
-	Cambio de signo	-a
sizeof	Tamaño en bytes	sizeof(int)
,	Combinar expresiones	a++, b++;
&	Dirección	&a
*	Indirección	*a
?:	Operador condicional	a<0?a:-a

Operador ternario ? :

- Expresión con 3 operandos:

(expresión lógica) ? expresión1 : expresión2

- Se evalúa la expresión lógica.
 - Si es verdadera, se ejecuta la expresión 1.
 - En caso contrario, se ejecuta la expresión 2.
- El siguiente ejemplo asigna el vamos máximo de a y b a la variable max:

```
int a = 2;
```

```
int b = 4;
```

```
int max = (a > b) ? a : b;
```

Conversión de tipos

Conversiones entre tipos

- C **no** es un lenguaje **fuertemente tipado**, por lo que podemos definir variables de un tipo y asignarle valores de otro tipo.
- En general, el valor de la derecha del operador de asignación (=) se convierte al tipo de la variable de la izquierda.
- En C, existen dos tipos principales de conversiones: **implícitas** y **explícitas** (también conocidas como casting).

Conversión de tipos

Conversión implícita

- El valor de una expresión aritmética tiene el **mismo tipo** que sus operandos cuando son todos del mismo tipo.
- Si hay diferentes tipos, el tipo de menor tamaño se convierte al tipo de mayor tamaño, siguiendo este orden de prioridad:

long double > double > float > unsigned long > long > unsigned int > int > unsigned short > short > char

- Se da prioridad a ciertos tipos sobre otros para garantizar que el valor resultante tenga mayor precisión y exactitud.

Conversión de tipos

Ejemplo de conversión implícita

En este caso, la variable **a** es un entero, pero como se está sumando con una variable de tipo **float**, se realiza una conversión implícita del entero a flotante para que la operación se ejecute correctamente.

```
int a = 10;  
float b = 5.5;  
float result = a + b;
```

Conversión de tipos

Conversión explícita (casting)

- El programador puede indicar a qué tipo hacer la conversión (***casting***).
- Para ello, basta con **indicar entre paréntesis** el tipo al que queremos convertir una expresión.
- Ejemplo:

```
int a;  
float b;  
a = 3;  
b = (float) a;
```

Conversión de tipos

Riesgos de la conversión explícita

El casting debe usarse con precaución, ya que puede conducir a truncamientos (pérdida de datos) o desbordamientos si se convierte un tipo más grande a uno más pequeño. Por ejemplo, convertir un `double` a un `int` elimina la parte decimal:

```
double d = 3.14159;  
int i = (int) d; // i será igual a 3.
```

Conversión de tipos

Ejemplo completo de conversiones

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

```
int main() {
```

```
    // Conversión implícita
```

```
    int x = 10;
```

```
    float y = 2.5;
```

```
    float sum = x + y; // x se convierte implícitamente a float
```

```
    printf ("Resultado de la suma (conversión implícita): %.2f\n", sum);
```

```
    // Conversión explícita (casting)
```

```
    int a = 5, b = 2;
```

```
    float div = (float) a / b; // Conversión explícita para obtener resultado decimal
```

```
    printf ("Resultado de la división (conversión explícita): %.2f\n", div);
```

```
    return 0;
```

```
}
```

Reglas de precedencia

Definición

- Definen el **orden** en el que se ejecutan los operandos en una expresión.
- **Precedencia**
 - Orden de ejecución entre operadores en función de su prioridad.
 - Puede modificarse con paréntesis.
- **Asociatividad**
 - Define el orden de las operaciones para los operadores con la misma prioridad.

Reglas de precedencia

Precedencia (ordenados de mayor a menor)

Precedencia					Asociatividad
	()	[]	->	.	izquierda a derecha
++	--	!	sizeof()		derecha a izquierda
+ (unario)	- (unario)	* (indir.)	& (dirección)		
	*	/	%		izquierda a derecha
	+	-			izquierda a derecha
<	<=	>	>=		izquierda a derecha
	==	!=			izquierda a derecha
	&&				izquierda a derecha
					izquierda a derecha
	? :				derecha a izquierda
=	+=	-=	*=	/=	derecha a izquierda
	, (operador coma)				izquierda a derecha

Biblioteca math.h

Definición

- En C existen bibliotecas que **amplían** la funcionalidad del lenguaje a través de funciones ya implementadas.
- La biblioteca ***math*** es una de las más utilizadas y proporciona **funciones matemáticas** útiles para trigonometría, raíces, logaritmos, etc.
- Para poder utilizarla, debemos incluirla en nuestro código:

```
#include <math.h>
```

- <http://www.cplusplus.com/reference/cmath/>

Biblioteca math.h

Funciones comunes de la biblioteca *math.h*

Función	Descripción	Ejemplo de Utilización
sqrt(x)	Devuelve la raíz cuadrada de x.	double r = sqrt(25.0);
pow(x, y)	Calcula x elevado a la potencia y.	double p = pow(2.0, 3.0);
sin(x)	Devuelve el seno de x en radianes.	double s = sin(M_PI/2);
cos(x)	Devuelve el coseno de x en radianes.	double c = cos(0);
tan(x)	Devuelve la tangente de x en radianes.	double t = tan(M_PI/4);
log(x)	Devuelve el logaritmo natural de x (base e).	double l = log(2.718);
log10(x)	Devuelve el logaritmo en base 10 de x.	double l10 = log10(100);
exp(x)	Calcula e elevado a la potencia x.	double e = exp(1);
fabs(x)	Devuelve el valor absoluto de x.	double abs_val = fabs(-5.0);
ceil(x)	Redondea x al siguiente número entero mayor o igual.	double up = ceil(2.3);
floor(x)	Redondea x al siguiente número entero menor o igual.	double down = floor(2.7);

Biblioteca math.h

Ejemplo de uso de la biblioteca math

// Ejemplo de cálculo de una raíz cuadrada

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

```
#include <math.h>
```

```
int main() {
```

```
    double num = 16.0;
```

```
    double result = sqrt(num);
```

```
    printf("La raíz cuadrada de %.2f es %.2f\n", num, result);
```

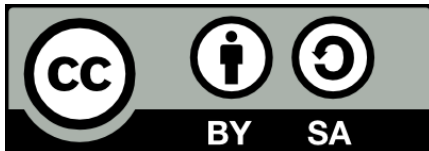
```
    return 0;
```

```
}
```

Tema 4: Estructuras de Control

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

- Estructuras de selección
 - `if`
 - `if-else`
 - `switch-case`
 - Operador ternario ? :
- Estructuras de repetición
 - `for`
 - `while`
 - `do-while`
- Evaluación perezosa

Estructuras de control

Secuencias de instrucciones

- Todos los programas que hemos desarrollado hasta ahora han ejecutado sus instrucciones de forma **secuencial**.
- Como norma general, las instrucciones dispuestas una detrás de otra que se ejecutan en el **orden** en el que aparecen escritas en el código.
- Por ejemplo:

```
int num = 25;  
int resultado;  
num += 24;  
resultado = num * 3;  
printf("resultado: %d\n", resultado);
```

Estructuras de control

¿Qué son?

- Sirven para controlar el **flujo de ejecución** de las instrucciones de un programa.
- Permiten modificar el orden de las instrucciones a ejecutar, definiendo qué instrucciones hay que ejecutar en cada momento.

Selección

- Permiten elegir si un bloque de instrucciones debe ejecutarse o no, o qué bloque de instrucciones debe ejecutarse.

Repetición

- Permiten repetir un bloque de instrucciones.

Estructuras de selección

¿Qué son?

- Permiten elegir si un bloque de instrucciones debe ejecutarse o no.
- Permiten seleccionar qué bloque de instrucciones ejecutamos entre varias opciones.
- Para ello, evalúan el valor de una expresión, la cual suele combinar operadores relacionales y lógicos.
- En C tenemos: `if`, `if-else`, `switch` y `? :`.

Estructuras de selección

Sentencia `if`

Sentencia `if`

- Permite definir un bloque de instrucciones que solo se ejecutará si la condición es cierta.
- Su sintaxis es:

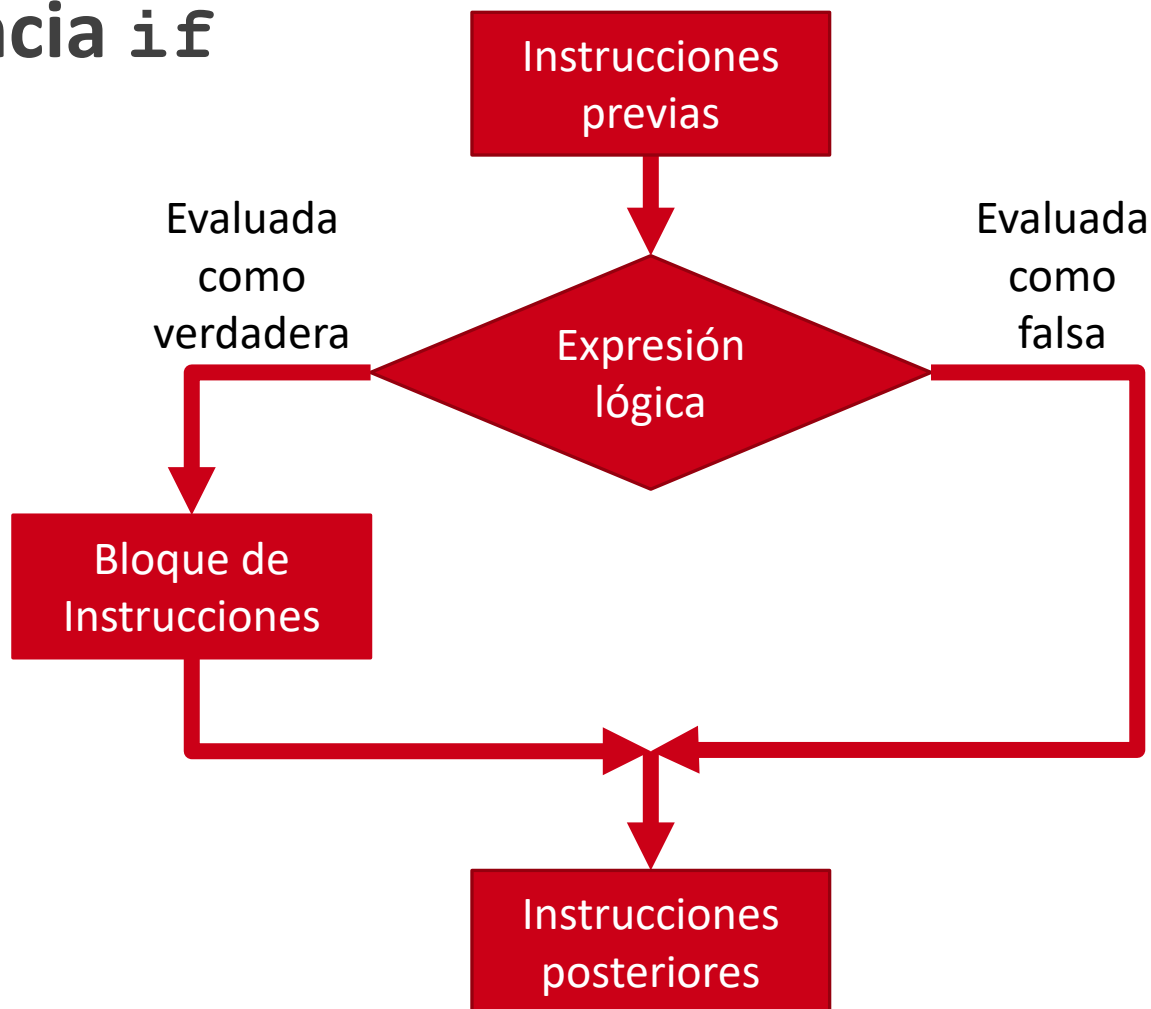
```
if (condición)
{
    // Instrucciones a ejecutar
}
```

- Los paréntesis son obligatorios.
- Si la secuencia es solo una instrucción, podemos omitir las llaves.

Estructuras de selección

Sentencia `if`

Sentencia `if`



Estructuras de selección

Sentencia `if`

Ejemplo de uso:

```
#include <stdio.h>

int main() {
    int temperatura;

    printf("Dame la temperatura actual\n");
    scanf("%d", &temperatura);

    if (temperatura > 25) {
        printf("Hace calor.\n");
        printf("Es un buen día para ir a la playa.\n");
    }

    return 0;
}
```

Estructuras de selección

Sentencia `if-else`

Sentencia `if-else`

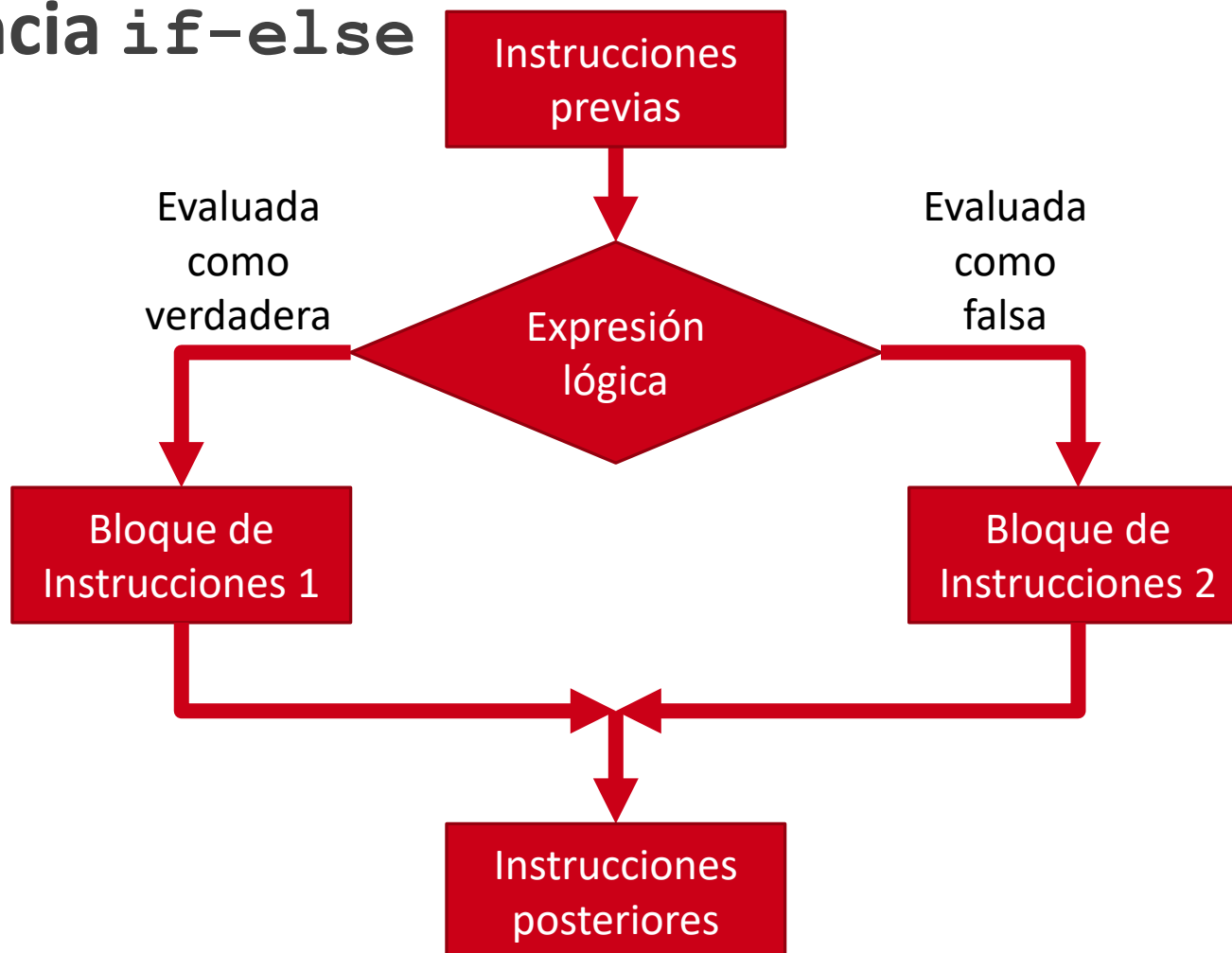
- Permite definir dos bloques de instrucciones, de los cuales solamente se ejecutará uno, en función de si la condición es cierta o no.

```
if (condición)
{
    // Instrucciones a ejecutar si la condición es verdadera
} else {
    // Instrucciones a ejecutar si la condición es falsa
}
```

Estructuras de selección

Sentencia `if-else`

Sentencia `if-else`



Estructuras de selección

Sentencia `if-else`

Ejemplo de uso:

```
#include <stdio.h>

int main() {
    int edad;

    printf("Inserta tu edad\n");
    scanf("%d", &edad);

    if (edad >= 18) {
        printf("Eres mayor de edad.\n");
    } else {
        printf("Eres menor de edad.\n");
    }

    return 0;
}
```

Estructuras de selección

Sentencia `if-else` anidada

- Son estructuras `if` e `if-else` con **múltiples alternativas**.
- Permiten seleccionar entre **varios bloques excluyentes**.
- Se comprueba **cada condición** hasta que una se evalúe como verdadera.
 - En ese momento, **se ejecuta su bloque y se sale** de la estructura de selección.
 - Si ninguna se cumple, se ejecuta el bloque `else` final (si existe).

Estructuras de selección

Sentencia `if-else` anidada

- Permite evaluar múltiples condiciones y mejorar la legibilidad.
- Ejemplo: ¿Cuánto vale `z`?

```
int x = 5;
int z;
if (x > 3) {
    if (x < 5) {
        z = 1;
    } else {
        z = x;
    }
} else {
    z = 0;
}
printf("%d", z);
```


Estructuras de selección

Sentencia `if-else` anidada

```
#include <stdio.h>

int main() {
    int numero;

    scanf("%d", &numero);

    if (numero > 0) {
        printf("El número es positivo.\n");
        printf("Se mostrará si la primera condición es verdadera.\n");
    } else if (numero < 0) {
        printf("El número es negativo.\n");
        printf("Se mostrará si la primera condición es falsa y la
                segunda verdadera.\n");
    } else {
        printf("El número es cero.\n");
        printf("Se mostrará si ambas condiciones son falsas.\n");
    }

    return 0;
}
```

Estructuras de selección

Sentencia switch-case

Sentencia switch-case

- Permite seleccionar uno entre múltiples bloques de instrucciones.

```
switch (expresión) {  
    case valor1:  
        // Código a ejecutar si expresión == valor1  
        break;  
    case valor2:  
        // Código a ejecutar si expresión == valor2  
        break;  
    // Más casos...  
    default:  
        // Código a ejecutar si ningún valor coincide  
}
```

Estructuras de selección

Sentencia `switch-case`

Sentencia `switch-case`

- La expresión debe ser de un tipo **ordinal**.
- El valor de cada sentencia `case` debe ser del **mismo tipo** que el de la expresión.
- Si **ningún** valor **coincide** con el resultado se ejecutará la sentencia `default` (si la incluimos).
- **OJO:** es importante incluir la sentencia `break` al final de cada bloque de instrucciones.
 - ¿Qué pasa si no la incluimos?

Estructuras de selección

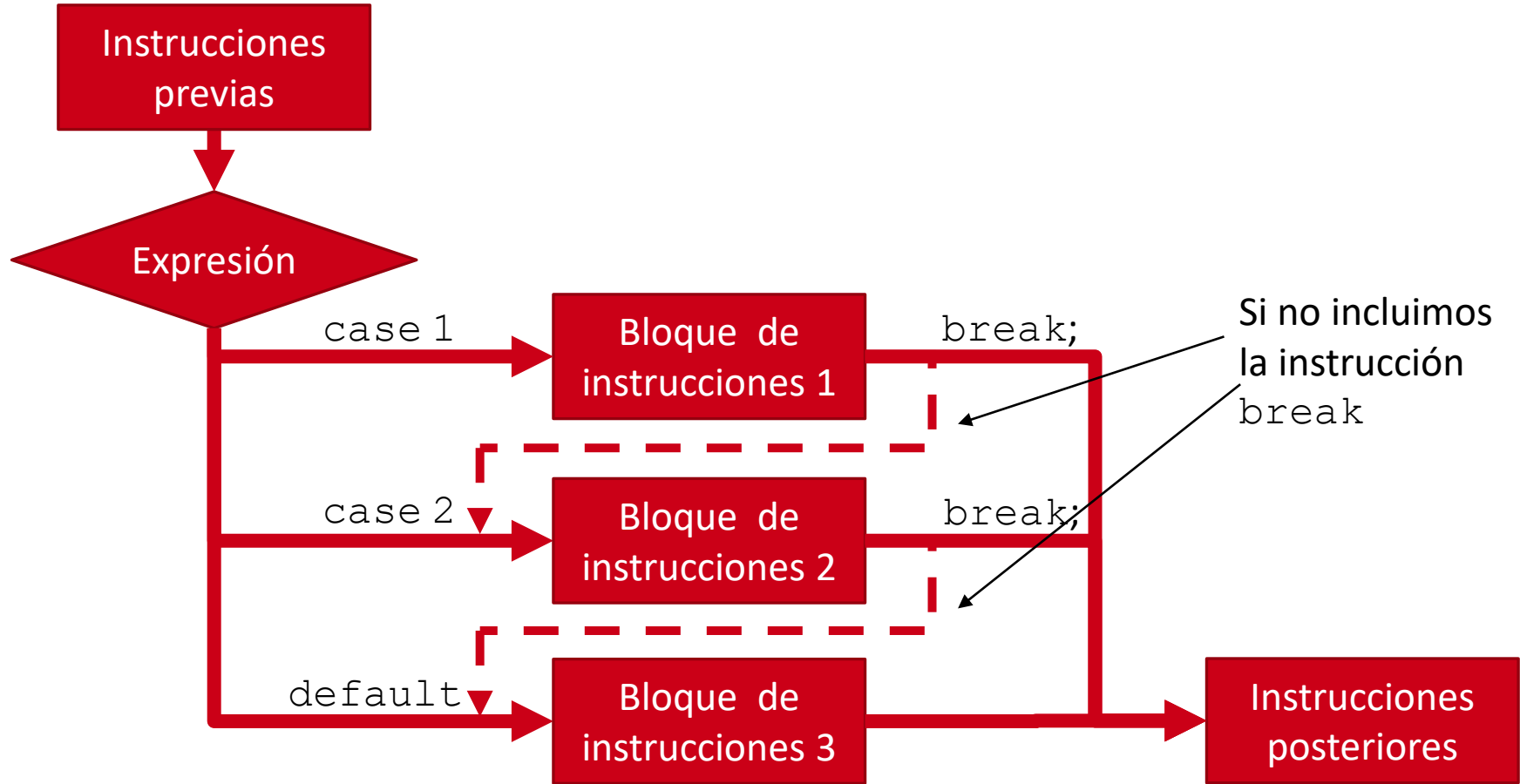
Sentencia `switch-case`

- La sentencia `break` permite que el flujo del programa se detenga y salga del bloque de instrucciones donde se encuentre.
- Se utiliza especialmente dentro de las sentencias `switch-case` para salir del case.
- Su uso **no** está recomendado porque rompe el hilo natural de ejecución de nuestro programa.

Estructuras de selección

Sentencia switch-case

Sentencia switch-case



Estructuras de selección

Sentencia switch-case

```
int dia;

printf("Inserta número del 1 al 5 para indicar un día:\n");
scanf("%d", &dia);

switch (dia) {
    case 1:
        printf("Lunes\n");
        break;
    case 2:
        printf("Martes\n");
        break;
    case 3:
        printf("Miércoles\n");
        break;
    case 4:
        printf("Jueves\n");
        break;
    case 5:
        printf("Viernes\n");
        break;
    default:
        printf("Día no válido\n");
}
```

Estructuras de selección

Operador ternario ? :

Operador ternario ? :

- Expresión con 3 operandos:

```
condición ? expresión_si_verdadera : expresión_si_falsa;
```

- Se evalúa la expresión lógica.
 - Si es verdadera, se ejecuta la expresión 1.
 - En caso contrario, se ejecuta la expresión 2.
- Equivalente a:

```
if (condición)
{
    expresión_si_verdadera;
} else {
    expresión_si_falsa;
}
```

Estructuras de repetición

¿Qué son?

- Permiten repetir la ejecución de un bloque de instrucciones.
- Se llama bucle al conjunto de instrucciones que se repite.
- En C, existen 3 estructuras de repetición:
 - `for`
 - `while`
 - `do-while`

Estructuras de repetición

Sentencia `for`

- El número de repeticiones se controla mediante una variable entera denominada contador.

```
for (inicialización; condición; actualización)
{
    // Código a ejecutar
}
```

- Incluye tres campos:
 - **inicialización**: da un valor inicial a la variable o variables.
 - **condición**: mientras se evalúe como cierta, se continuará ejecutando el bloque de instrucciones.
 - **actualización**: modifica el valor de la variable contador.

Estructuras de repetición

Sentencia `for`

- Los pasos que sigue la sentencia `for` son:
 1. Se ejecuta la sentencia de **inicialización**.
 2. Si se cumple la **condición**, se ejecuta el bloque de instrucciones.
 - Cuando deje de cumplirse, saltará a la instrucción siguiente al cierre de la sentencia `for`.
 3. Tras cada iteración o repetición, se ejecuta la sentencia de **actualización** y se vuelve a evaluar la **condición** (vuelve al paso anterior).
 4. El bucle termina cuando no se cumple la condición.

Estructuras de repetición

Sentencia `for`

Ejemplos (algunos válidos, otros no)

```
for (int i = 1; i < -3; i++) {  
    ...  
}
```

¡Nunca se cumple la condición!

```
for (int i = 1; i < 3; i--) {  
    ...  
}
```

¡Nunca deja de cumplirse la condición, bucle infinito!

```
for (int i = 1; i == 3; i++) {  
    ...  
}
```

¡La primera vez que se evalúa no se cumple la condición!

```
for (int i = 0; i <= 6; i = i + 2) {  
    ...  
}
```

For correcto

Estructuras de repetición

Sentencia for

¿Qué hacen estos ejemplos?

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Estructuras de repetición

Sentencia for

¿Qué hacen estos ejemplos?

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

```
int main() {  
    int suma = 0;  
    for (int i = 1; i <= 5; i++)  
        suma += i;  
    printf("La suma es: %d\n", suma);  
    return 0;  
}
```

Estructuras de repetición

Sentencia `while`

- Útil cuando necesitamos repetir un conjunto de instrucciones, pero se desconoce el número exacto de repeticiones y no se puede calcular evaluando una expresión.

```
while (condición) {  
    // Código a ejecutar  
}
```

Estructuras de repetición

Sentencia `while`

- Los pasos que sigue la sentencia `while` son:
 1. Se evalúa la condición.
 2. Si es falso, no se ejecuta el código del bucle.
 3. Si es cierta, se ejecuta el bloque de instrucciones y se vuelve al paso 1.
- Si es necesario inicializar alguna variable, debe hacerse antes de la sentencia `while`.
- Es necesario que alguna instrucción del bloque que se repite modifique el valor de la condición en algún momento.

Estructuras de repetición

Sentencia `while`

¿Qué hace el siguiente ejemplo?

```
#include <stdio.h>

int main() {
    int suma = 0;
    int i = 1;
    while (i <= 5) {
        suma += i;
        i++;
    }
    printf("La suma es: %d\n", suma);
    return 0;
}
```


Estructuras de repetición

Sentencia `while`

- Cualquier bucle `for` puede transformarse en un bucle `while`.

```
for (int i=0; i <= 5; i++) {  
    // Bloque de instrucciones  
}
```



```
int i = 0;  
while (i <= 5) {  
    // Bloque de instrucciones  
    i++;  
}
```

Estructuras de repetición

Sentencia `do-while`

- En un bucle `while` se comprueba la condición antes de entrar en el bucle.
- La sentencia `do-while` garantiza que el bucle se ejecute al menos una vez antes de comprobar la condición.

```
do {  
    // Código a ejecutar  
} while (condición);
```

Estructuras de repetición

Sentencia do-while

¿Qué hace el siguiente ejemplo?

```
#include <stdio.h>

int main() {
    int num;
    do{
        scanf("%d", &num);
    }while(num != 7);
}
```

Estructuras de repetición

`break` y `continue`

- La sentencia `break` permite interrumpir el flujo de ejecución de un bloque de instrucciones.
- Puede utilizarse para detener la ejecución de un bucle, aunque su uso no está recomendado ya que suele hacer el código más impredecible y difícil de seguir.
- La sentencia `continue` permite interrumpir la ejecución actual del bloque de instrucciones y pasar a la siguiente iteración del bucle.
- Al igual que `break`, su uso puede dificultar la lectura del código.

Evaluación perezosa

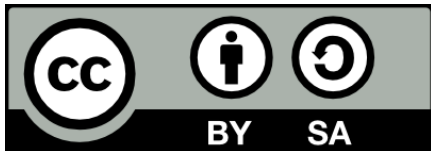
- La evaluación de expresiones lógicas en C se efectúa siguiendo la **evaluación perezosa** o en cortocircuito.
 - Se detiene tan pronto como se sabe su resultado.
 - Ahorra tiempo de ejecución.
 - Influye a la hora de elegir el orden de las expresiones.
- ¿Cuál será la salida de la siguiente expresión, si $a=0$? ¿Se producirá algún error?

```
if (a != 0) && (b % a == 0) {  
    printf("La división es exacta\n");  
}
```

Tema 5: Array y Cadenas de Caracteres

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

- Tipos de datos compuestos.
- Arrays.
- Arrays bidimensionales.
- Arrays multidimensionales.
- Cadenas de caracteres.
- Biblioteca `string.h`.

Introducción

Introducción a los tipos de datos compuestos

- Ya hemos visto los **tipos de datos simples (básicos y derivados)** de C:
 - Tipo entero (`int`, `long`, `short`, ...)
 - Tipo real (`float`, `double`, ...)
 - Tipo carácter (`char`)
- Estos tipos permiten declarar variables con un **único** valor.
- **Pero...**

Existen otros tipos de datos, denominados compuestos, que permiten agrupar múltiples elementos de datos del mismo tipo o de diferentes tipos en una sola estructura.

Introducción

Tipos de datos compuestos

- En C existen varios tipos de datos compuestos que nos permiten manejar colecciones de datos.
- Los tres tipos de datos compuestos en C son:
 - Tipo *array*.
 - Tipo *cadena de caracteres*.
 - Tipo *estructura*.
- Una variable de un tipo de dato compuesto puede almacenar **más de un valor**.

Introducción

Tipos de datos compuestos

Veamos un ejemplo:

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

```
int main() {
```

```
    // Declaración e inicialización de un array de 5 enteros
```

```
    int numeros[5] = {1, 2, 3, 4, 5};
```

```
    // Accediendo y mostrando los elementos del array
```

```
    for (int i = 0; i < 5; i++) {
```

```
        printf("Elemento en la posición %d: %d\n", i, numeros[i]);
```

```
    }
```

```
    return 0;
```

```
}
```

- En este ejemplo, se declara un **array** llamado `numeros` inicializado con cinco enteros.
- Utilizamos un bucle **for** para recorrer cada elemento del array e imprimir su valor.

Arrays

Concepto y sintaxis

- Un *array* (o *arreglo*) es una estructura de datos que permite almacenar múltiples elementos del mismo tipo en una única variable, utilizando un espacio **contiguo** de memoria.
- Los *array* son útiles cuando se necesita almacenar una secuencia de valores relacionados, como una lista de números o una serie de caracteres.

Arrays

Concepto y sintaxis

- Sintaxis:

```
tipo nombre_array[num_elementos] ;
```

Donde:

- **tipo**: es el tipo de datos de los elementos del array (por ejemplo, `int`, `float`, `char`).
- **nombre_array**: es el identificador del array.
- **num_elementos**: es el número de elementos que el array puede contener.

Nota: esta instrucción reserva espacio para **num_elementos * sizeof(tipo)** en memoria principal.

Unidimensionales (**vectores**)

```
#include <stdio.h>

int main() {
    // Declarar un array unidimensional de 5 enteros
    int numeros[5] = {10, 20, 30, 40, 50};
    // Imprimir los elementos del array
    for (int i = 0; i < 5; i++) {
        printf("Elemento %d: %d\n", i, numeros[i]);
    }
    return 0;
}
```

- `int numeros[5] = {10, 20, 30, 40, 50};` declara un **array** de 5 enteros inicializado con valores específicos.
- Un bucle `for` se utiliza para recorrer cada elemento del array y mostrarlo.

Arrays

Clasificación

Multidimensionales, los más habituales son los bidimensionales (**matrices**)

```
#include <stdio.h>
int main() {
    // Declarar un array bidimensional de 3 filas y 3 columnas
    int matriz[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    // Imprimir los elementos de la matriz
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("Elemento [%d][%d]: %d\n", i, j, matriz[i][j]);
        }
    }
    return 0;
}
```

- `int matriz[3][3]` declara un array bidimensional (3 filas y 3 columnas).
- El valor de cada elemento se inicializa con `{1, 2, 3}`, `{4, 5, 6}`, `{7, 8, 9}`.
- Puede verse como un array de arrays.
- Se utilizan dos bucles `for` anidados para recorrer e imprimir todos los elementos de la matriz.

Arrays

Diferencia entre un arreglo unidimensional y bidimensional

Unidimensional (Vector)

0	1	2	...	9
14.5	1.4	5.7	...	121.8

Bidimensional (Matriz)

	0	1	2	...	9
0	54	2	34	...	17
1	87	8	15	...	157
...
5	89	4	56	...	72

Arrays

Usos

- Los arrays permiten almacenar múltiples valores del mismo tipo en una única variable. Esto permite:
 - Organizar los datos de manera ordenada.
 - Acceder a cualquier elemento de manera eficiente.
 - Facilitar la manipulación del conjunto de datos (operaciones como búsqueda, clasificación o modificación).
 - Simplificar el código, al evitar la declaración de múltiples variables individuales.

Arrays

Usos

- Si necesitamos almacenar la nota de 10 estudiantes, podemos:
 - Utilizar 10 variables de tipo float.
 - Utilizar un array de 10 elementos.
- ¿Y si fuesen 100 estudiantes? ¿Cómo podríamos operar con estos datos?

Arrays

Acceso a los datos de un array

- Las declaraciones de **array** reservan espacio para los datos en la memoria principal.
- Se trata de colección de datos **homogénea**.
- Se almacenan en Memoria Principal ocupando posiciones **consecutivas**.
- Para acceder a cada elemento, debemos indicar:
 - **Identificador** del array.
 - **Índice** o posición del elemento, dado por una expresión entera (escrita entre []) que indica la posición buscada en el array.
 - En C, los arrays **comienzan en 0**.

Arrays

Acceso a los datos de un arreglo

- Las posiciones de un array van desde 0 a $n-1$.
- Ejemplos:

```
float notas[7];  
notas[2] = 6.3;  
notas[2] = notas[2] + 1.9;  
notas[2] -= 0.9;  
printf("%f\n", notas[2]);
```

0	1	2	3	4	5	6
4.5	9.3	7.3	3.1	11.8	12.4	29.5

Arrays

Tamaño de un array

- La **declaración** de un array reserva espacio en memoria para almacenar el **número máximo** de elementos que puede contener.
- Se recomienda que su tamaño venga definido por una constante (usualmente, utilizando la instrucción `#define`).
- El uso de una constante permite modificar el tamaño de un array simplemente modificando la propia constante.

Arrays

Inicialización de un array

- Si el contenido del array es conocido en el momento de su declaración, podemos inicializarlo directamente:

```
int array1[6] = {1, 2, 3, 4, 5, 6};
```

```
int array2[] = {1, 2, 3, 4, 5, 6};
```

```
int array3[8] = {1, 2, 3, 4, 5};
```

6 elementos



8 elementos



Arrays

Recorrido de un array

- **No** podemos operar sobre el array completo, debiendo hacerlo sobre **cada uno** de los elementos de manera individual.
- Por ello, para procesar el array completo, es necesario **recorrer** cada uno de sus elementos.
- Con el fin de facilitar este recorrido, se utilizan estructuras de repetición, como **for**, **while**, etc.

Arrays

Accesos incorrectos a los elementos de un array

- Si se accede a una posición **fuera de rango**, se puede producir:
 - Un **error en ejecución** (no en compilación): muchas veces difíciles de detectar.
 - Acceso a una variable **incorrecta**.
 - Acceso a una zona de **memoria protegida**.

```
int calificaciones[10];
```

```
calificaciones[-1]; // No puede haber índices negativos
```

```
calificaciones[10]; // El mayor índice permitido es 9
```

Arrays

Array parcialmente lleno

- En la declaración de un array se indica el número **máximo** de elementos que puede almacenar.
- Sin embargo, es posible que en ciertos momentos de la ejecución el array no esté lleno.
- En ese caso, habrá celdas que no contengan un valor **válido**.
- También es posible que hayamos tenido que **sobredimensionar** el tamaño del array para adaptarnos a los requisitos solicitados.
 - Por ejemplo, si en una clase puede haber hasta 50 matriculados, deberíamos tener una estructura capaz de almacenar 50 notas, aunque el número final de estudiantes sea menor.

Arrays

Array parcialmente lleno

Para saber qué posiciones de un array tienen un valor válido y cuáles no, tenemos dos opciones:

1. **Tope**: variable entera que indica cuántos elementos válidos hay actualmente en el array.

tope = 3

9.6	5.1	7.2	XXX	XXX	XXX	XXX
-----	-----	-----	-----	-----	-----	-----

2. Valor "basura": Asignar un valor **no válido** en el contexto de nuestro programa a las posiciones que aún no tienen un valor válido.

9.5	-1000	3.4	5.8	-1000	8.1	-1000
-----	-------	-----	-----	-------	-----	-------

Arrays bidimensionales (matrices)

¿Qué son?

- Son array que tienen dos dimensiones, denominadas usualmente como filas y columnas.
- Se representan como una matriz del tamaño de sus dimensiones $n \times m$ (número de filas \times número de columnas).

	0	1	2	3	4
0	5.3	2.2	3.3	2.8	5.4
1	9.4	3.6	2.3	9.8	6.7
2	4.6	3.8	6.6	8.9	7.4

Arrays bidimensionales (matrices)

Declaración de un array bidimensional

- La declaración de un array bidimensional es:

```
tipoElemento nombreArray [N] [M] ;
```

- Donde:

- **tipoElemento**: tipo de datos almacenado.

- **N**: el número de filas.

- **M**: es el número de columnas

- Ejemplo de una matriz de números en punto flotante:

```
float matriz[3][6];
```

	0	1	2	3	4	5
0	5.3	2.2	3.3	2.8	5.4	3.4
1	9.4	3.6	2.3	9.8	6.7	1.2
2	4.6	3.8	6.6	8.9	7.4	8.7

Arrays bidimensionales (matrices)

Acceso a un arreglo bidimensional

- El acceso a un elemento de un array bidimensional se realiza indicando la fila y columna donde se encuentra:

nombreArray [fila] [columna]

- Por ejemplo:

matriz[1][3];

matriz[2][4];

matriz[1][1];

matriz[3][1];



	0	1	2	3	4	5
0	5.3	2.2	3.3	2.8	5.4	3.4
1	9.4	3.6	2.3	9.8	6.7	1.2
2	4.6	3.8	6.6	8.9	7.4	8.7

Arrays bidimensionales (matrices)

Inicialización de un array bidimensional

- Los valores de un array bidimensional o matriz se pueden inicializar en la **declaración**, tal y como pasaba con los arrays unidimensionales.
- En este caso debemos pensar en un array bidimensional como un array de arrays
- Ejemplo:

```
int matriz[4][2] = {  
    {11, 12}, {21, 22},  
    {31, 32}, {41, 42}  
};
```

	0	1
0	11	12
1	21	22
2	31	32
3	41	41

Arrays bidimensionales (matrices)

Recorrido de un array bidimensional

- Para **recorrer** la matriz basta con variar el índice de fila y columna del elemento al que estamos accediendo.
- Para ello suelen utilizarse dos bucles `for` anidados.
 - Dependiendo del orden de estos bucles, se recorrerá por **filas** o por **columnas**.

Arrays bidimensionales (matrices)

Recorrido por filas

```
#define N 4 // Número de filas
```

```
#define M 3 // Número de columnas
```

```
int main() {
```

```
    int matriz[N][M]; // Declaración de la matriz
```

```
    for (int i = 0; i < N; i++) { // Recorremos cada una de las filas
```

```
        for (int j = 0; j < M; j++) { // Recorremos cada columna (de cada fila)
```

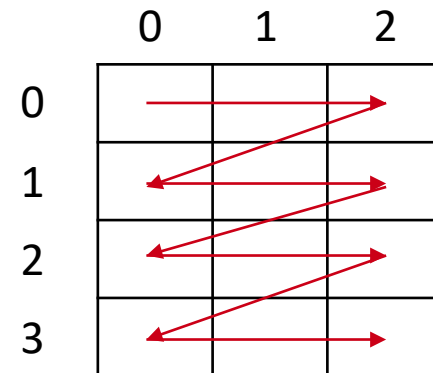
```
            matriz[i][j] = 0; // Ponemos a 0 cada elemento
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```



Arrays bidimensionales (matrices)

Recorrido por columnas

```
#define N 4 // Número de filas
```

```
#define M 3 // Número de columnas
```

```
int main() {
```

```
    int matriz[N][M]; // Declaración de la matriz
```

```
    for (int j = 0; j < M; j++) { // Recorremos cada una de las columnas
```

```
        for (int i = 0; i < N; i++) { // Recorremos cada fila (de cada columna)
```

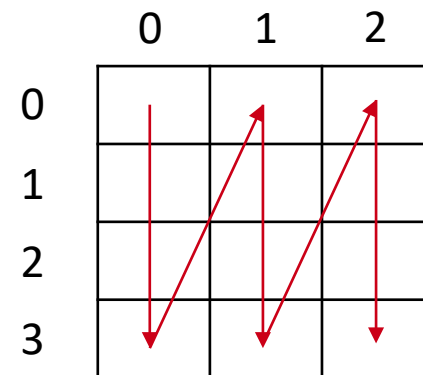
```
            matriz[i][j] = 0; // Ponemos a 0 cada elemento
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```



Arrays bidimensionales (matrices)

Tamaño de vectores y matrices

- Se **recomienda** utilizar la directiva `#define` para definir el tamaño de los arrays.
- De esta forma, cuando queramos modificar su tamaño, basta con modificar el valor en la sentencia `#define` correspondiente.
- En C, siempre es **responsabilidad del programador** comprobar que los índices se encuentran **dentro del rango** del array, debiendo evitando el acceso a posiciones de memoria fuera de dicho rango.

Arrays bidimensionales (matrices)

Almacenamiento en memoria de arrays bidimensionales

- Un array bidimensional se almacena en la memoria del ordenador como una **secuencia continua** de datos.
- Esto significa que todos los elementos del array se guardan uno tras otro, **sin espacios entre ellos**, siguiendo un orden determinado.
- Por ejemplo, para un array bidimensional `int matriz[3][3]`, la memoria se reserva de forma lineal hasta reservar espacio para 9 elementos (3x3).
- En C, el almacenamiento se realiza en lo que se conoce como "**Row-Major Order**" o almacenamiento **por filas**. Esto quiere decir que todos los elementos de una fila se almacenan consecutivamente en la memoria antes de pasar a la siguiente fila.

Arrays bidimensionales (matrices)

Almacenamiento en memoria de arrays bidimensionales

Ejemplo:

```
int matriz[3][3] = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9}  
};
```

Los elementos se organizarán consecutivamente en memoria de la siguiente manera: **1, 2, 3, 4, 5, 6, 7, 8, 9**

- Primero se almacenan todos los elementos de la primera fila: **{1, 2, 3}**.
- Luego se almacenan los elementos de la segunda fila: **{4, 5, 6}**.
- Finalmente, los elementos de la tercera fila: **{7, 8, 9}**.

Arrays bidimensionales (matrices)

Cálculo de la dirección de un elemento

Cada elemento del array bidimensional `matriz[i][j]` se almacena en una posición de memoria calculada mediante la fórmula:

$$\text{dirección_base} + (i * \text{número_de_columnas} + j) * \text{tamaño_del_tipo}$$

Donde:

- **dirección_base**: es la dirección de memoria del primer elemento del array (`matriz[0][0]`).
- **i**: es el índice de la fila.
- **número_de_columnas**: es la cantidad de columnas del array.
- **j**: es el índice de la columna.
- **tamaño_del_tipo**: es el tamaño en bytes del tipo de datos del array (por ejemplo, 4 bytes para `int` en la mayoría de las arquitecturas).

Arrays multidimensionales

¿Qué son?

- Los arrays multidimensionales son estructuras que permiten almacenar datos en **múltiples dimensiones**, lo cual resulta útil para representar tablas, matrices, o cualquier estructura que requiera más de una dimensión.
- Un array multidimensional es una extensión del array unidimensional, donde cada elemento del array puede ser otro array, permitiendo el almacenamiento de datos en forma de tablas o matrices.
- De manera general, un array de n dimensiones se puede visualizar como un conjunto de array anidados.

Arrays multidimensionales

¿Qué son?

- Sintaxis:

```
tipo nombre_array[tamaño1][tamaño2]...[tamañoN];
```

Donde:

- **tipo**: el tipo de datos que almacenará el array (por ejemplo, int, float, char).
- **nombre_array**: el nombre del array.
- **tamaño1, tamaño2, ... tamañoN**: los tamaños de cada dimensión del array.

Ejemplos:

- `tipoElemento nombreArray[dim1][dim2][dim3];`
- `tipoElemento nombreArray[dim1]...[dimN];`
- Para recorrerlos completos, necesitaremos tantos bucles `for` anidados como dimensiones tenga el array.

Arrays multidimensionales

Ejemplo array tridimensional

```
int main() {  
    int main() {  
        // Declaración e inicialización de un array tridimensional  
        int cubo[2][3][4] = {  
            {  
                {1, 2, 3, 4},  
                {5, 6, 7, 8},  
                {9, 10, 11, 12}  
            },  
            {  
                {13, 14, 15, 16},  
                {17, 18, 19, 20},  
                {21, 22, 23, 24}  
            }  
        };  
    }  
};
```

Arrays multidimensionales

Ejemplo array tridimensional

```
// Acceder a los elementos del array tridimensional y mostrarlos
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 4; k++) {
            printf("cubo[%d][%d][%d] = %d\n", i, j, k, cubo[i][j][k]);
        }
    }
}

return 0;
}
```


Cadenas de caracteres

¿Qué son?

- Variables cuyo valor es una **secuencia finita de caracteres** (`string`).
- Los literales de cadenas de caracteres se escriben entre **comillas dobles**.
- En memoria, las variables añaden un carácter especial al final de la variable que indica el **final de cadena**, `'\0'`.
- Este tipo de dato es esencial para trabajar con textos, ya que permite almacenar y manipular secuencias de caracteres

Cadenas de caracteres

¿Qué son?

- En C **no existe** el tipo *String* como en otros lenguajes.
- En su lugar, se utiliza un **array** de elementos tipo **char** para representar un *String*, teniendo en cuenta es necesario **reservar un espacio** en el array para el carácter '`\0`', que indica el final de la cadena.
- La biblioteca **string.h** contiene **operaciones habituales** con cadenas de caracteres.

Cadenas de caracteres

Sintaxis y ejemplos

Para declarar una cadena de caracteres en C, se puede utilizar un array de tipo **char**:

```
char cadena[20];
```

Esto declara un array de 20 caracteres, capaz de almacenar una cadena de hasta 19 caracteres (más el carácter nulo `'\0'`).

Valor de tipo cadena	Contenido almacenado en el array									
"Hola"	'H'	'o'	'l'	'a'	'\0'					
"Buen dia"	'B'	'u'	'e'	'n'	' '	'd'	'i'	'a'	'\0'	
"C/Tulipan"	'C'	'/'	'T'	'u'	'l'	'i'	'p'	'a'	'n'	'\0'
""	'\0'									

Cadenas de caracteres

Sintaxis y ejemplos

- Es importante añadir el carácter fin de cadena `' \0 '` para poder determinar dónde termina nuestra cadena:

```
#include <stdio.h>

int main() {
    char cad[10];
    cad[0] = 'H';
    cad[1] = 'o';
    cad[2] = '!';
    cad[3] = 'a';
    printf("[%s]\n", cad);
    return 0;
}
```

Cadenas de caracteres

Inicialización

- Las cadenas se pueden inicializar en la declaración con su valor entre **comillas dobles**:

```
char texto_ej1[256] = "Esto es una cadena";  
char texto_ej2[] = "Esto es otra cadena";
```

- `texto_ej1` puede contener hasta 255 caracteres (más el carácter nulo).
- `texto_ej2` puede reservar el tamaño que ocupe el valor asignado (teniendo en cuenta el carácter nulo), en este caso, 20 elementos.
- En ambas inicializaciones, al utilizar una cadena literal, se añade **automáticamente** el carácter nulo.

Cadenas de caracteres

Inicialización

- Si una cadena de caracteres no termina en `' \0 '` se considera un array de caracteres, no una cadena.
- Por tanto, en un array de tamaño n solo coge una cadena de $n - 1$ caracteres.
- O, dicho de otra forma, Una cadena de n caracteres ocupa en memoria $n + 1$ caracteres.
- La cadena vacía contiene únicamente `' \0 '`.
- Cuando creamos una cadena de caracteres en C sin inicializarla, se considera una cadena vacía.

Cadenas de caracteres

Inicialización de cadenas

- Las cadenas también se pueden inicializar carácter por carácter, es decir, posición por posición del array:

```
char saludo[5] = {'H', 'o', 'l', 'a', '\0'};
```

En este caso, se especifican explícitamente los caracteres de la cadena, incluyendo el carácter nulo `'\0'`.

Cadenas de caracteres

Biblioteca string.h

- La biblioteca `<string.h>` proporciona una serie de funciones para manipular y operar con **cadenas de caracteres** en C.
- Debemos **incluir** la biblioteca en nuestro código para poder utilizarla.
- A continuación, se explican algunas de las funciones más utilizadas de esta librería: **strcpy**, **strlen**, **strcmp** y **strcat**, aunque contiene muchas otras:

<https://cplusplus.com/reference/cstring/>

Cadenas de caracteres

Biblioteca string.h

strcpy (string copy)

- La asignación sólo puede realizarse en la declaración, si lo hacemos después se produce un error:

```
char cadena[10];  
cadena = "hola";
```

→ No es posible!!!!

- Es necesario copiar el valor de una variable de tipo cadena a otra.

Cadenas de caracteres

Biblioteca string.h

strcpy (string copy)

- Su sintaxis es:

```
char[] strcpy(char[] destino, const char[] origen)
```

- Donde:
 - **origen** es una cadena que no se modificará.
 - **destino** es la cadena que quedará modificada con el valor de origen.

```
char cadena[10];  
strcpy(cadena, "hola");
```

- **IMPORTANTE:** No se comprueban los tamaños, la función supone que hay tamaño suficiente.

Cadenas de caracteres

Biblioteca string.h

strlen (string length)

- Calcula la longitud de una cadena que recibe como argumento.
- Sintaxis:

```
int strlen(char[] cad)
```

- Donde:
 - `cad` es la cadena cuya longitud queremos conocer.
 - Retorna un entero que podemos almacenar para su posterior uso.
- La longitud retornada es el número de caracteres sin contar el `'\0'` (o la posición del carácter nulo en la cadena), no el número máximo de caracteres que la cadena puede contener.

```
char cadena[10] = "hola";  
int len = strlen(cadena);
```

Cadenas de caracteres

Biblioteca string.h

strcmp (string compare)

- Compara dos cadenas de caracteres siguiendo el orden alfabético.
- Sintaxis:

```
int strcmp(char[] cad1, char[] cad2)
```

- Donde:
 - *cad1* y *cad2* son las dos cadenas a comparar.
 - Retorna un entero que podemos utilizar o almacenar para su posterior uso. El valor de este entero puede ser:
 - 0, si los caracteres hasta '`\0`' de ambas cadenas son iguales
 - < 0, si *cad1* es menor alfabéticamente que *cad2*.
 - > 0, si *cad1* es mayor alfabéticamente que *cad2*.

Cadenas de caracteres

Biblioteca string.h

strcmp (string compare)

- Sintaxis:

```
int strcmp(char [] cad1, char [] cad2)
```

```
char cadena1[10] = "hola";  
char cadena2[10] = "adios";  
if(strcmp(cadena1, cadena2)==0)  
{  
    printf("Las cadenas son iguales\n");  
}
```

- **IMPORTANTE:** No es posible comparar dos cadenas utilizando el operador de comparación ==.

Cadenas de caracteres

Biblioteca string.h

strcat (string concatenation)

- Concatena a destino la cadena almacenada en origen.
- Sintaxis:

```
char[] strcat(char[] destino, const char[] origen)
```

- Donde:
 - `destino` es la cadena a la cual se le concatena la otra cadena.
 - `origen` es la cadena concatenada a la primera.

```
char cadena1[50] = "hola";  
char cadena2[] = " que tal";  
strcat(cadena1, cadena2);  
printf("%s\n", cadena1);
```

- **IMPORTANTE:** No se comprueba el tamaño, la función supone que hay tamaño suficiente.

Cadenas de caracteres

Lectura y escritura de cadenas de caracteres (<stdio.h>)

- La función `printf()` permite mostrar cadenas utilizando `%s` en la cadena de control.
- Las funciones `puts()` y `fputs()` escriben en la salida una cadena de caracteres, incluyendo un salto de línea.
- De igual manera, la función `scanf()` lee una palabra hasta encontrar una coma, espacio, tabulador o salto de línea.
 - Podemos utilizar `%[^\\n]` para leer caracteres hasta alcanzar el carácter `'\\n'` (salto de línea).
 - La cadena se utiliza en `scanf()` **sin &**.

```
char nombre[10];  
scanf("%s", nombre);  
scanf("%[^\\n]", nombre);
```

Cadenas de caracteres

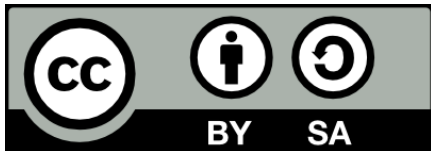
Lectura y escritura de cadenas de caracteres (<stdio.h>)

- La función `gets()` lee todos los caracteres que haya en una línea, incluyendo espacios en blanco, hasta encontrar un salto de línea.
- La función `fgets()` lee todos los caracteres que haya hasta encontrar un salto de línea, marcando un límite de caracteres.
- La función `getchar()` permite leer un carácter de teclado cuando el usuario pulsa intro (`\n`).
- Existen otras funciones de la biblioteca `<conio.h>` (console input/output) como `getch()` o `getche()` para leer caracteres por teclado sin esperar que se pulse intro.

Tema 6: Punteros

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

- Punteros
 - Introducción
 - Definición
 - Usos
 - Declaración
 - Puntero a NULL
 - Puntero a puntero
 - Operadores
 - Errores comunes
 - Punteros void
- Aritmética de punteros
 - Incremento/Decremento
 - Resta de punteros
- Arrays y punteros
 - Relación
 - Acceso
 - Matrices

Punteros

Introducción

- Cuando un programa se ejecuta en nuestro ordenador, requiere memoria para almacenar datos y variables.
- El sistema operativo es el encargado de reservar una zona de memoria cuando vamos a ejecutar nuestro programa.
- Durante la ejecución, los programas reservan y liberan memoria según sus necesidades a lo largo de su ejecución.

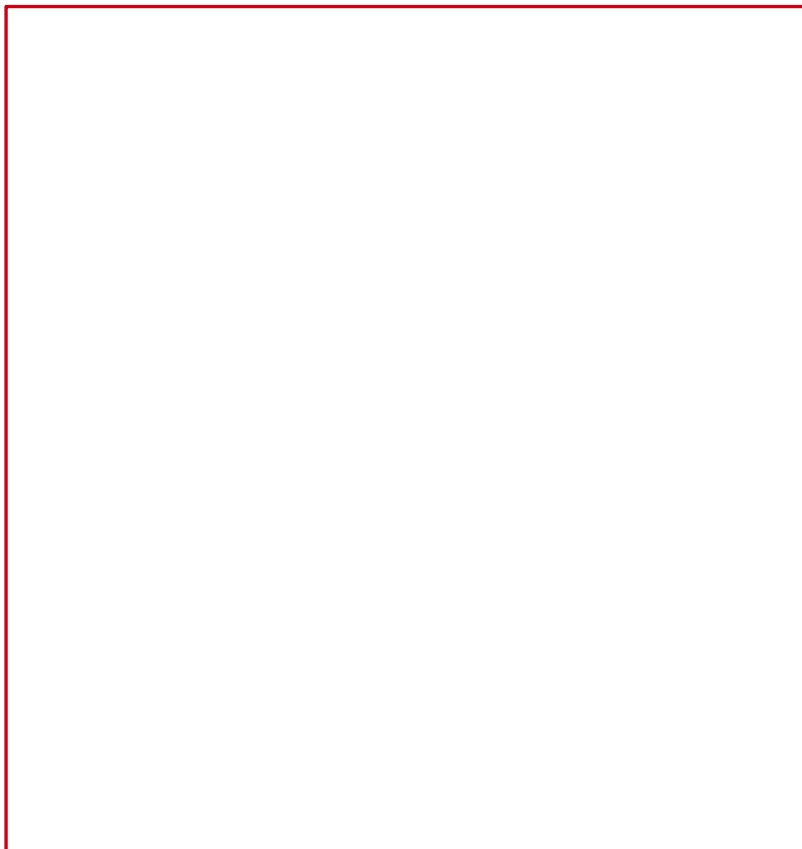
Punteros

Introducción

- Para cada programa tenemos diferentes regiones de memoria:
 - Segmento de código o texto, donde se almacenan las instrucciones que se van a ejecutar.
 - Segmento de datos, donde se almacenan las diferentes variables globales y estáticas de nuestro código.
 - Pila o stack, donde se almacenan las variables locales, parámetros de funciones y registros de llamadas a funciones.
 - Heap, donde se asigna memoria dinámicamente en tiempo de ejecución.

Punteros

Introducción

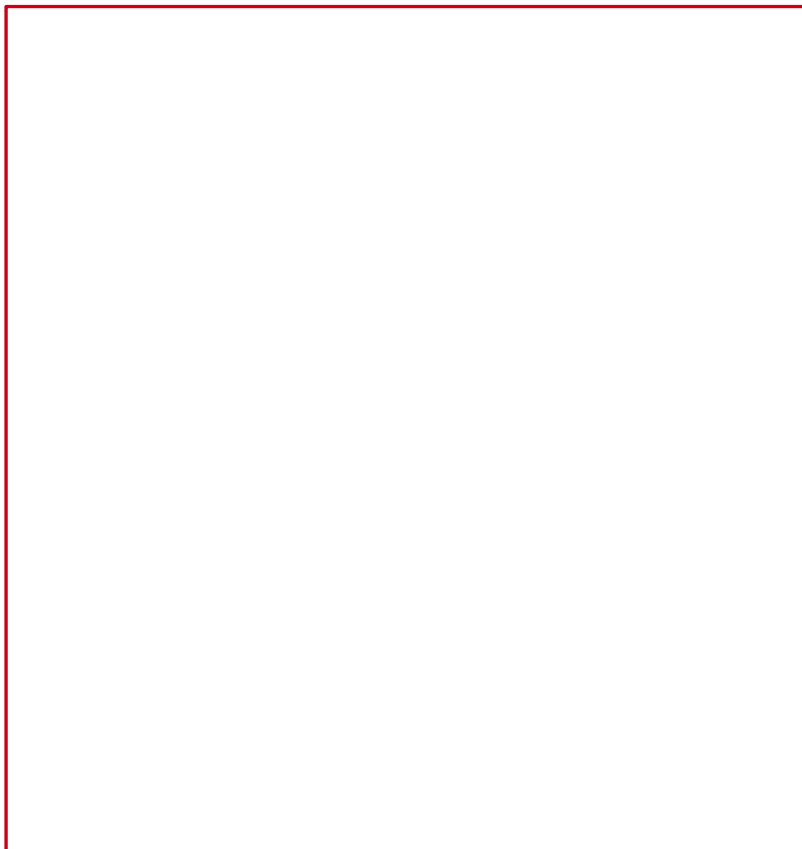


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción



```
#include <string.h>
```

```
int main() {
```

```
    int num_estudiantes;
```

```
    float nota_media;
```

```
    char tipo;
```

```
    char asignatura[100];
```

```
    num_estudiantes = 77;
```

```
    nota_media = 6.2;
```

```
    tipo = 'B';
```

```
    strcpy(asignatura, "Introducción a la Programación");
```

```
    return 0;
```

```
}
```

Punteros

Introducción

num_estudiantes



```
#include <string.h>
```

```
int main() {
```

```
    int num_estudiantes;
```

```
    float nota_media;
```

```
    char tipo;
```

```
    char asignatura[100];
```

```
    num_estudiantes = 77;
```

```
    nota_media = 6.2;
```

```
    tipo = 'B';
```

```
    strcpy(asignatura, "Introducción a la Programación");
```

```
    return 0;
```

```
}
```

Punteros

Introducción

num_estudiantes



```
#include <string.h>
```

```
int main() {
```

```
    int num_estudiantes;
```

```
    float nota_media;
```

```
    char tipo;
```

```
    char asignatura[100];
```

```
    num_estudiantes = 77;
```

```
    nota_media = 6.2;
```

```
    tipo = 'B';
```

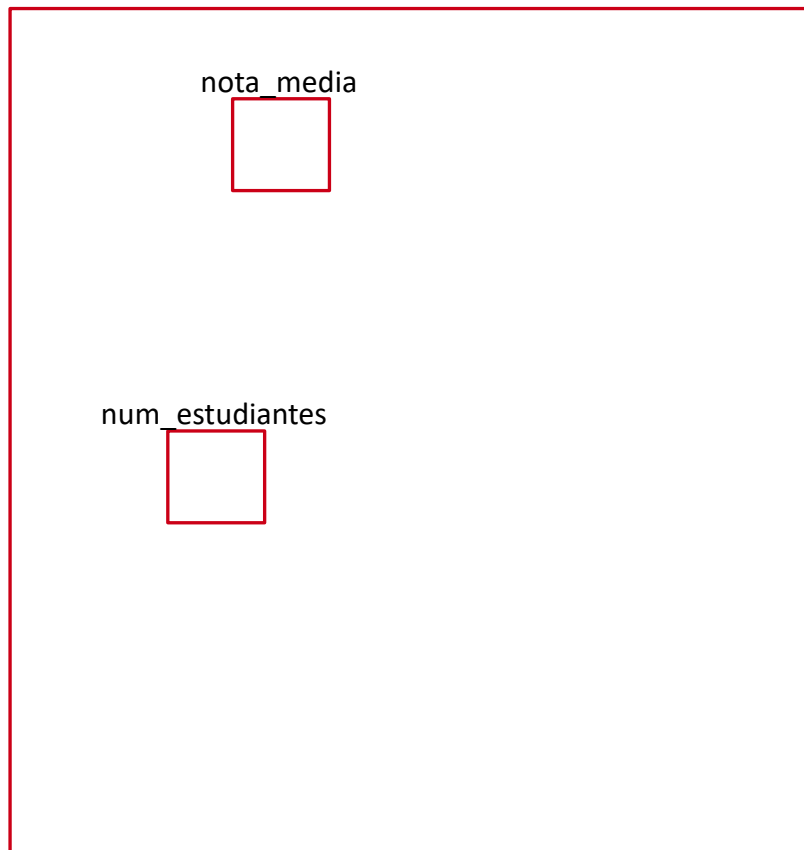
```
    strcpy(asignatura, "Introducción a la Programación");
```

```
    return 0;
```

```
}
```


Punteros

Introducción

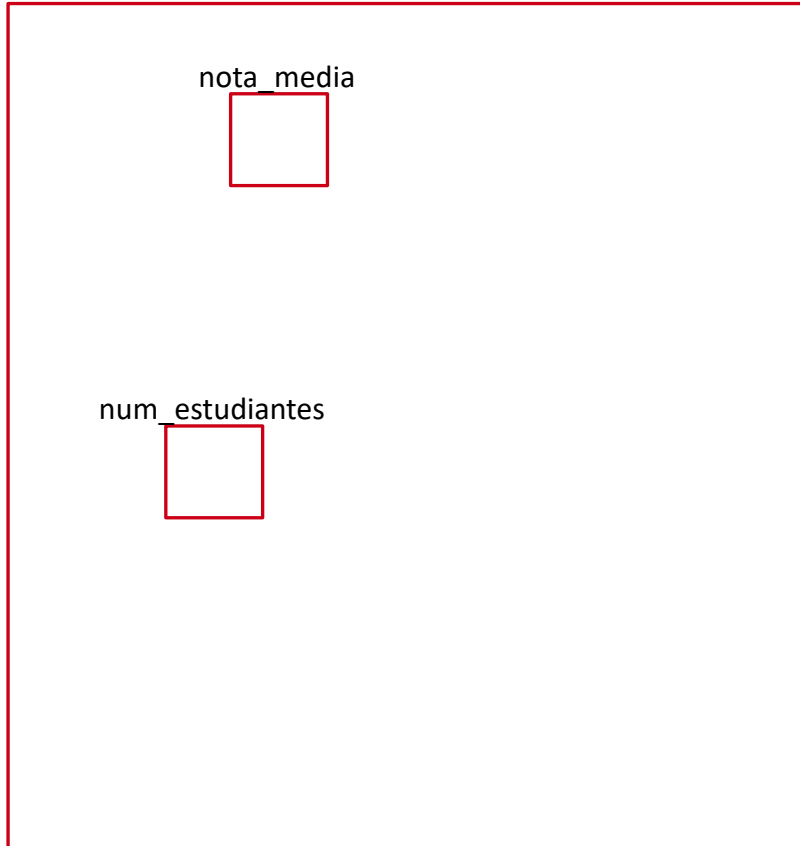


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

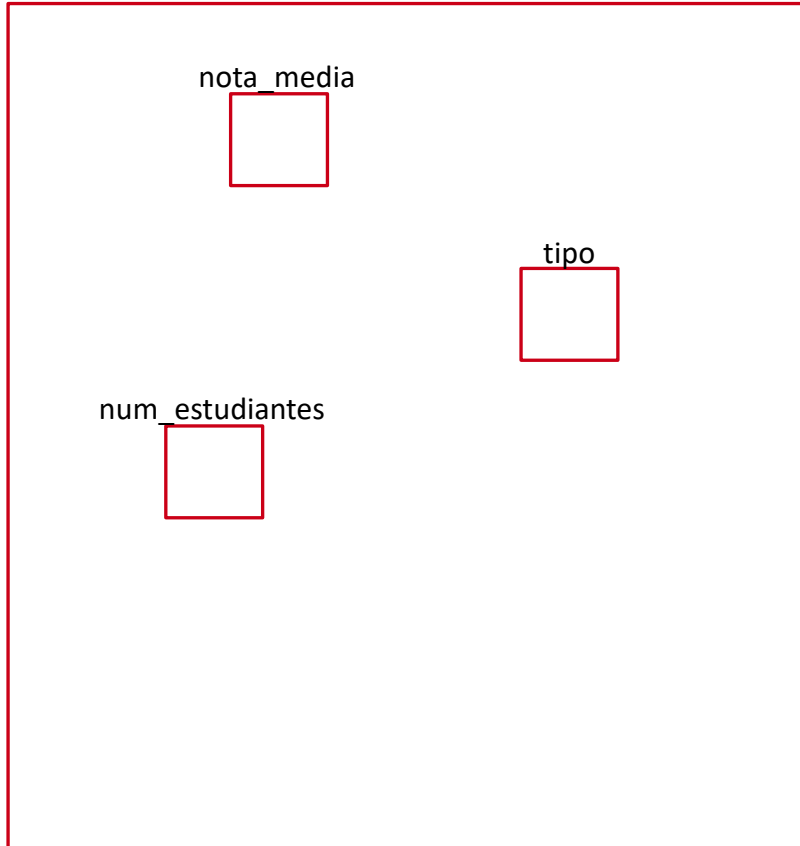


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

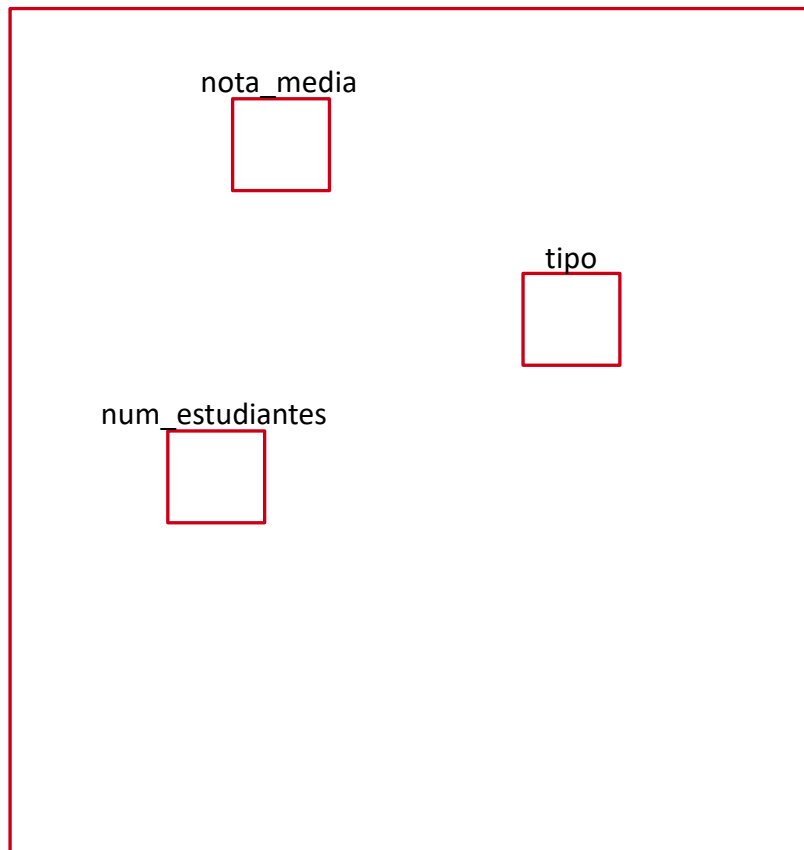


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

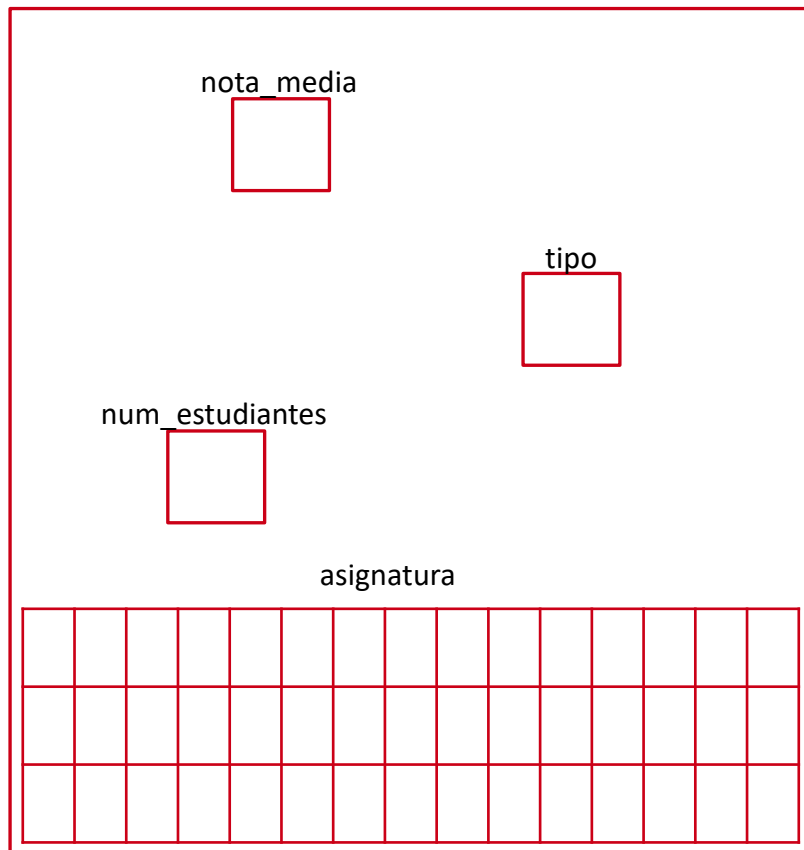


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

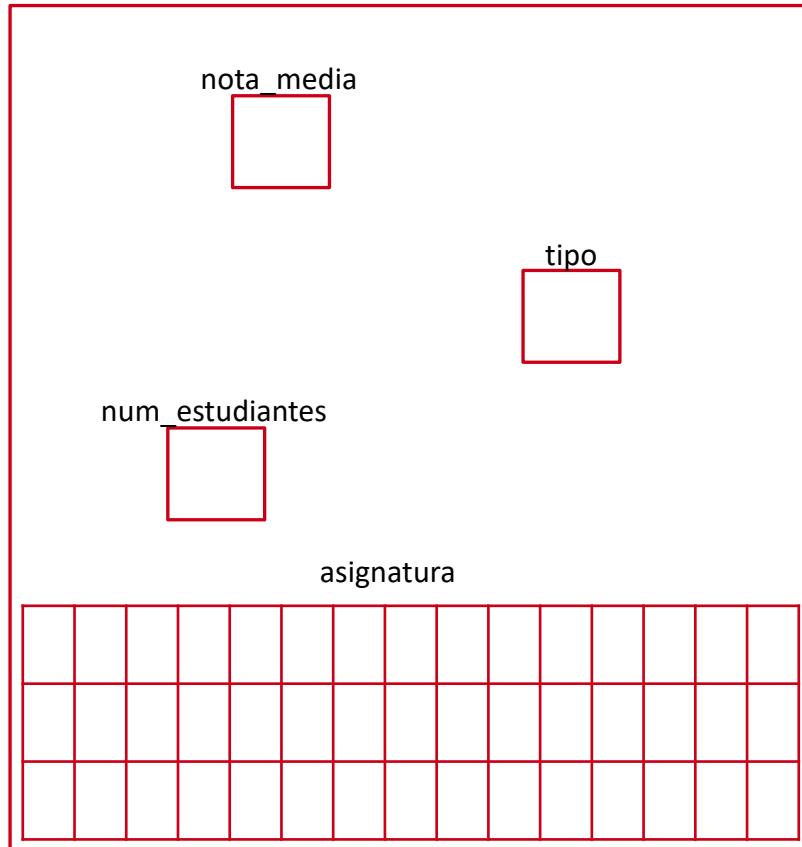


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

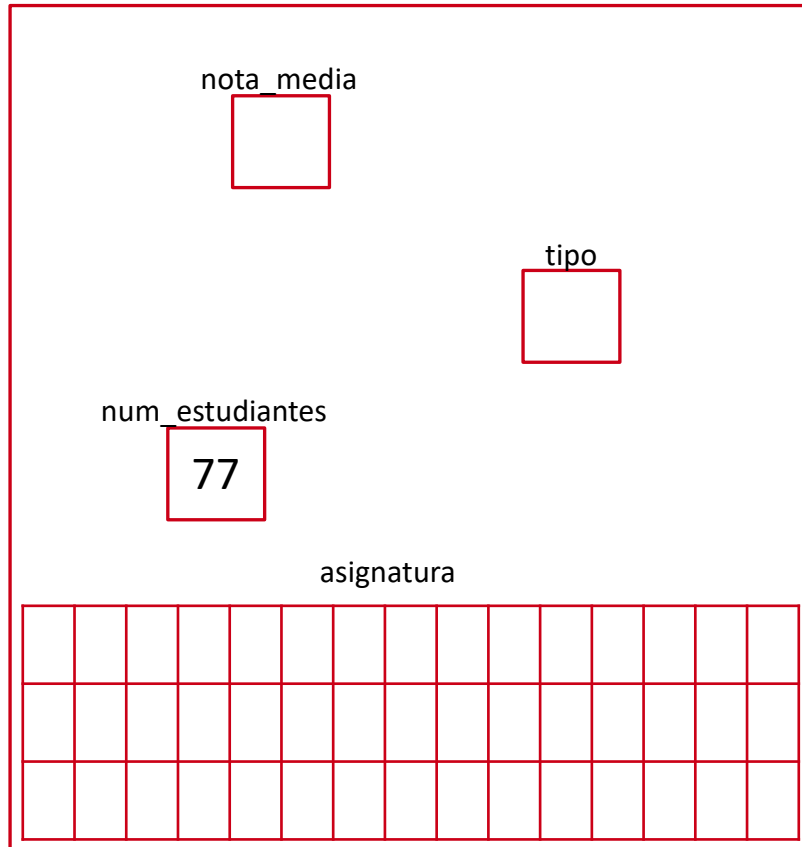


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

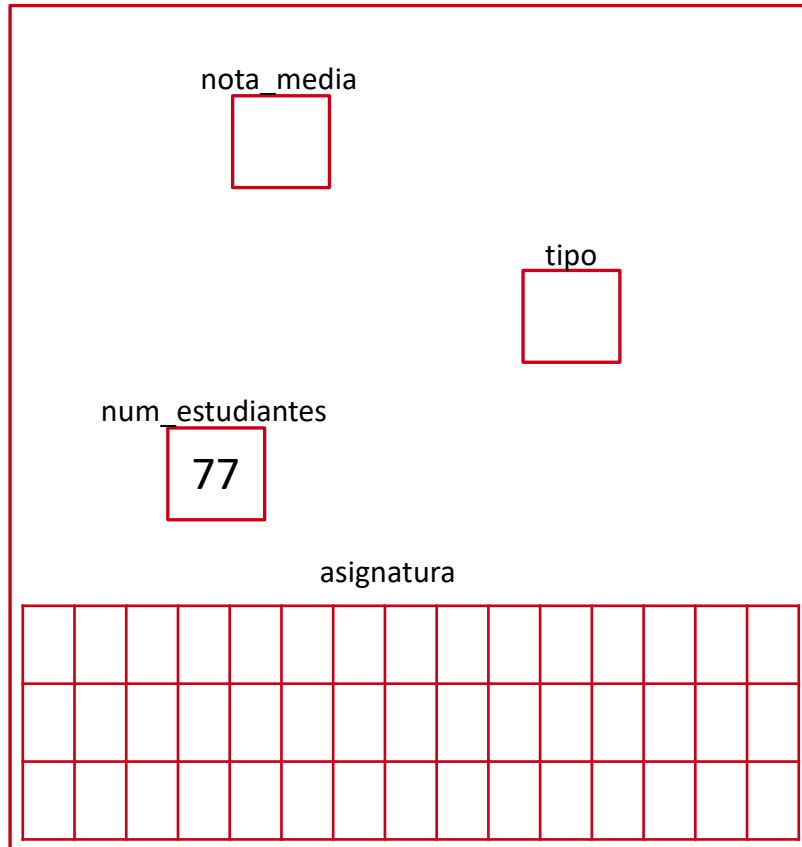


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

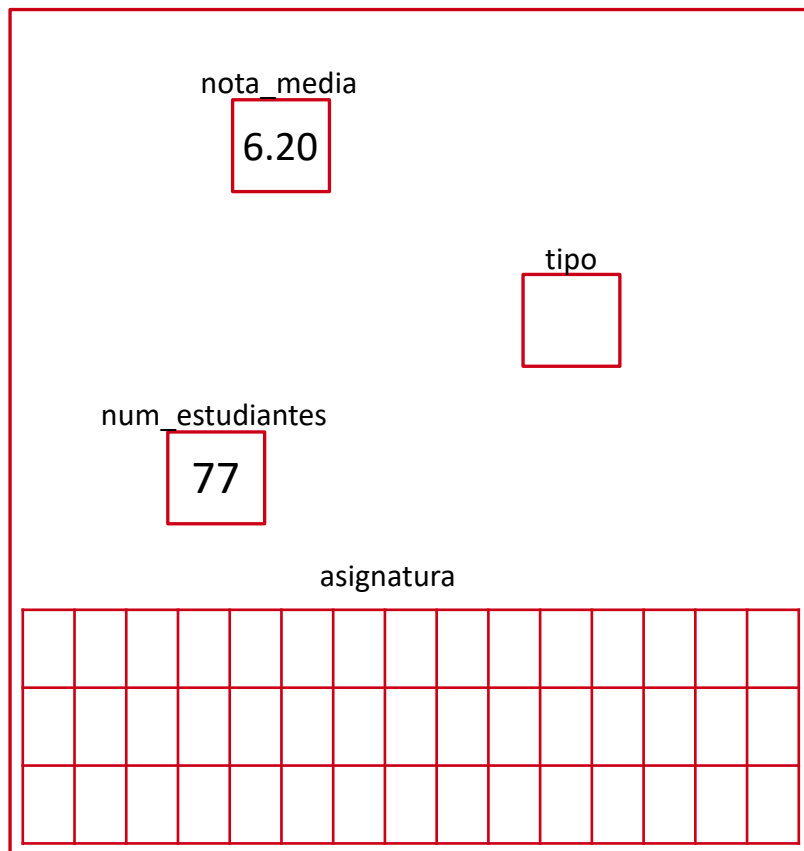


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```


Punteros

Introducción

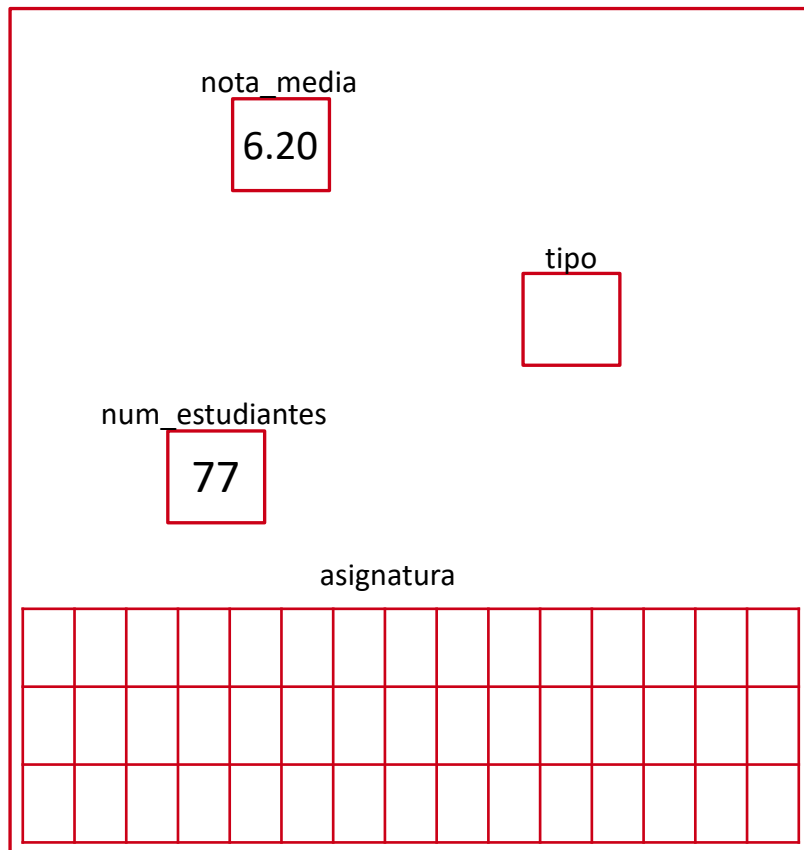


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

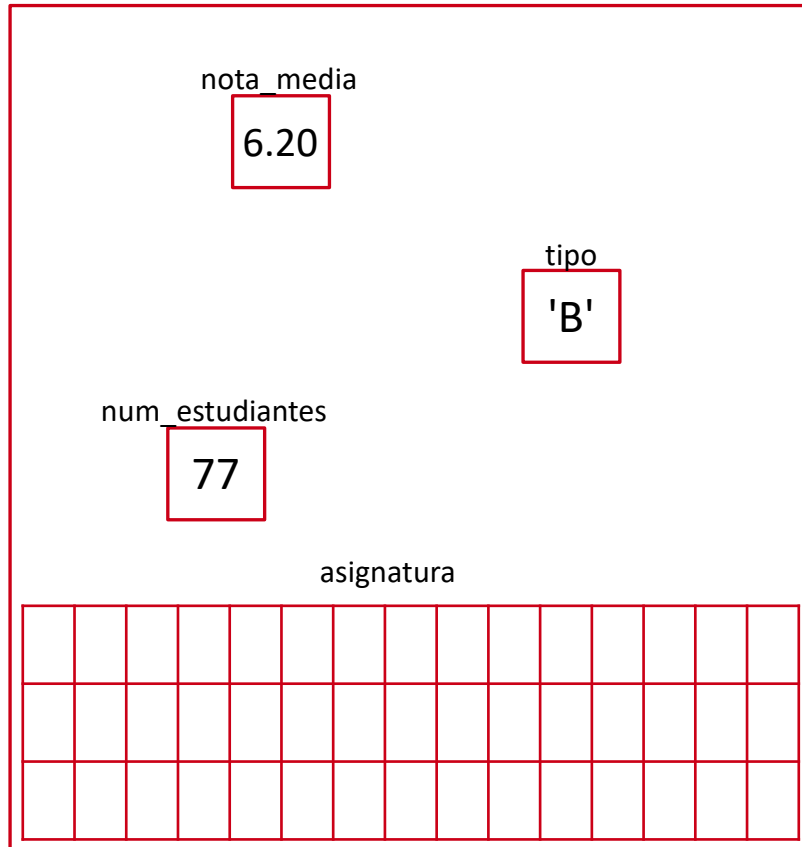


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción

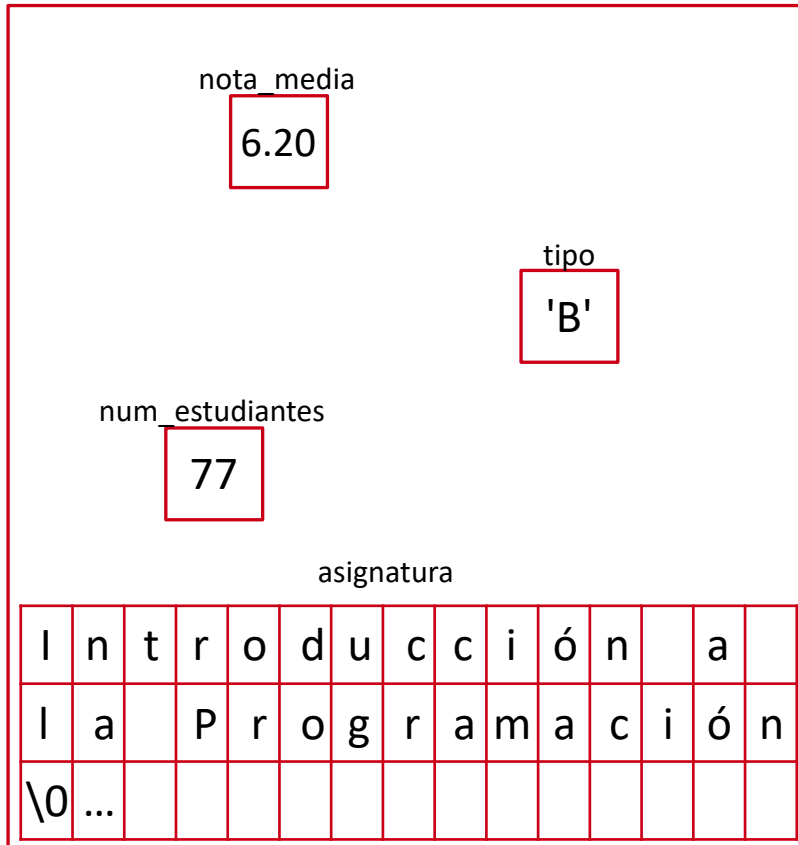


```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

Punteros

Introducción



```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];
```

```
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';
```

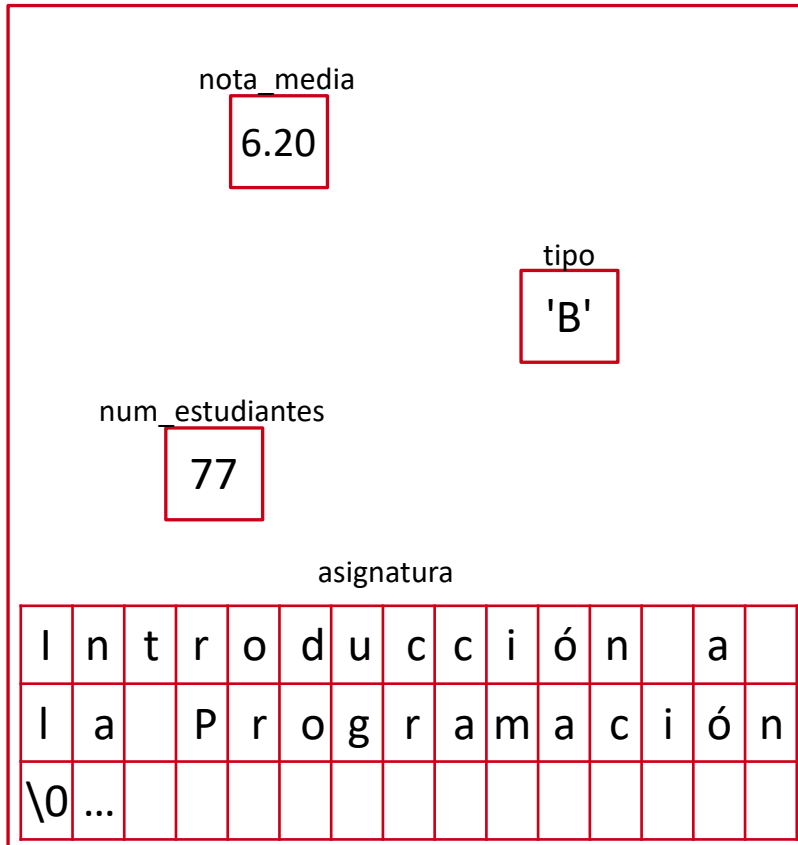
```
    strcpy(asignatura, "Introducción a la Programación");
```

```
    return 0;
```

```
}
```

Punteros

Introducción



```
#include <string.h>
```

```
int main() {  
    int num_estudiantes;  
    float nota_media;  
    char tipo;  
    char asignatura[100];  
  
    num_estudiantes = 77;  
    nota_media = 6.2;  
    tipo = 'B';  
    strcpy(asignatura, "Introducción a la Programación");  
  
    return 0;  
}
```

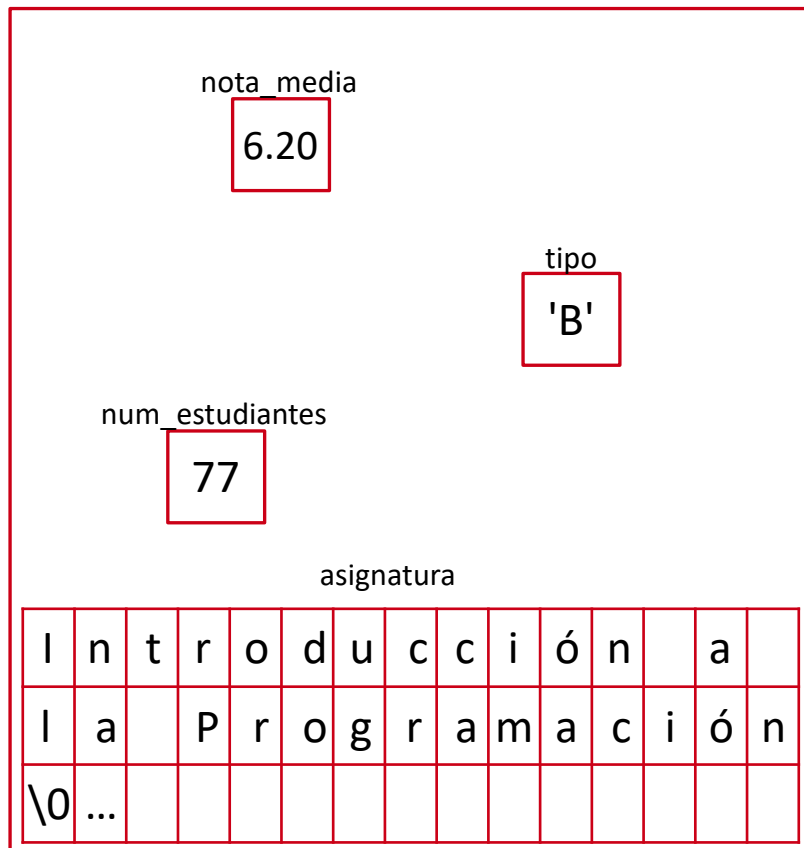
Punteros

Introducción

- Al declarar una variable, reservamos una región de memoria para almacenar un dato del tipo de la variable.
- Después, cuando usamos el identificador de una variable estamos accediendo a la región de memoria que se le asignó en su declaración.
- Por ello, cuando accedemos a una variable, realmente estamos accediendo a su dirección de memoria.

Punteros

Introducción



Variable	Dirección de memoria
<code>num_estudiantes</code>	0x000000a0507ffd1c
<code>nota_media</code>	0x000000a0507ffd18
<code>tipo</code>	0x000000a0507ffd17
<code>asignatura[0]</code>	0x000000a0507ffc00
<code>asignatura[1]</code>	0x000000a0507ffc01
...	...

¡Un nuevo tipo de datos!

Punteros

Definición

- Las variables que hemos utilizado hasta ahora se almacenan en una **posición de memoria**.
- Un puntero es una variable que, al igual que el resto de variables, almacena información. En este caso, contienen **direcciones** de memoria.
- Se utilizan para almacenar **direcciones de memoria**, las cuales pueden ser direcciones de otras variables o de celdas de memoria reservadas dinámicamente.
- Si un puntero contiene una dirección de memoria válida, se dice que "**apunta**" a esa variable o celda de memoria.

Punteros

Usos

- Paso de parámetros por **referencia** en funciones.
- Uso de memoria **dinámica**.
- Construir estructuras **dinámicas** de datos.
- Mejorar el **rendimiento** de algunos algoritmos.
- **Cuidado**: el mal uso de punteros produce errores muy difíciles de detectar.

Punteros

Declaración

- A la hora de declarar un puntero es necesario indicar el tipo de dato apuntado y un asterisco (*) que indique que se trata de un puntero:

```
tipo_dato_apuntado * id_puntero;
```

- Donde:
 - `tipo dato apuntado` es el **tipo de dato al que apunta la variable**, o, dicho de otra forma, el tipo de dato almacenado en la dirección de memoria que contendrá el puntero.
 - `*` indica que la variable es un puntero. Se escribe delante del identificador.
 - `id puntero` es el identificador de la variable declarada, el cual debe seguir las normas vistas para la creación de identificadores en C.

Punteros

Declaración

- Una vez declarado, un puntero se puede utilizar como cualquier otra **variable**:
 - Asignar valores, utilizar en expresiones, etc.
 - La única diferencia es que el dato que contiene es una dirección de memoria.
- Al igual que el resto de variables, C **no** inicializa los punteros al declararlos.
 - Si no están inicializados, apuntan a una dirección **desconocida** (y, por lo tanto, no válida).

Punteros

Puntero a NULL

- Para definir un puntero "vacío", podemos asignarle el valor **NULL**:

```
int * p = NULL;
```

- Se utiliza para indicar que el puntero no apunta a ningún dato.
- Pero cuidado, si accedemos a la dirección de un puntero que apunta a **NULL**, se producirá un **error de ejecución**.

Punteros

Puntero a NULL

- Por ello, si el puntero puede no contener una dirección de memoria válida, es recomendable utilizar la constante **NULL** para comprobar si la dirección de memoria de un puntero es válida o no antes de trabajar con él:

```
if(p != NULL)
{
    printf("El puntero p tiene una dirección de memoria válida\n");
}else{
    printf("El puntero p NO tiene una dirección de memoria válida\n");
}
```

Punteros

Puntero a puntero

- Un puntero sigue siendo una variable, por lo que **puede ser apuntada** por otro puntero.
- Ese nuevo puntero se denomina puntero a puntero.
- Para declarar un puntero que apunte a otros punteros es necesario indicar el tipo de datos apuntado por el puntero apuntado, así como un doble asterisco (*):

```
tipo_dato_apuntado ** id_puntero
```

- `id_puntero` será un puntero a uno o varios punteros que apuntan a datos de tipo `tipo_dato_apuntado`.

Punteros

Operadores

- En C hay dos operadores especialmente relacionados con punteros: **&** y *****.
- El **operador dirección &** permite obtener la dirección de memoria de una variable.
- El **operador indirección *** permite acceder al contenido al que apunta un puntero.

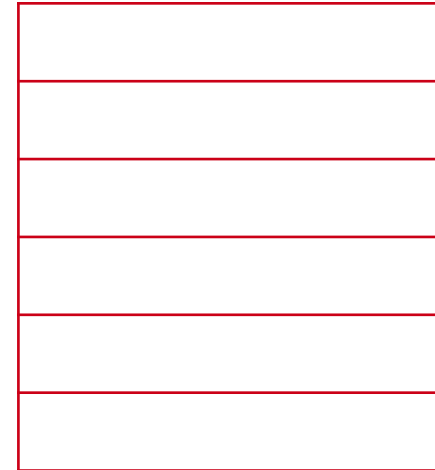
Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;  
**ppNum = **ppNum + 2;  
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num		
pNum		
ppNum		

Punteros

Operadores

```
int num = 3;
```

```
int * pNum = NULL;
```

```
int ** ppNum = NULL;
```

```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num		
pNum		
ppNum		

Punteros

Operadores

```
int num = 3;
```

```
int * pNum = NULL;
```

```
int ** ppNum = NULL;
```

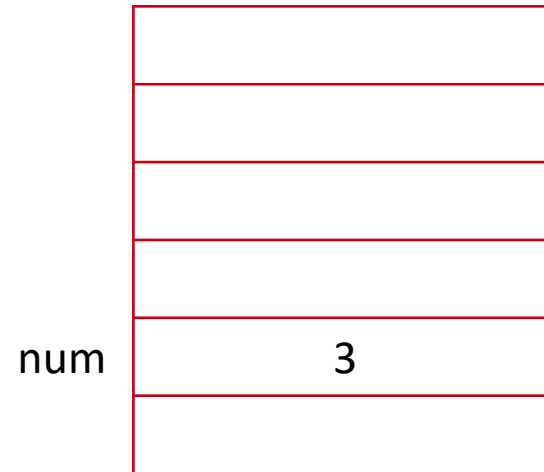
```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum		
ppNum		

Punteros

Operadores

```
int num = 3;
```

```
int * pNum = NULL;
```

```
int ** ppNum = NULL;
```

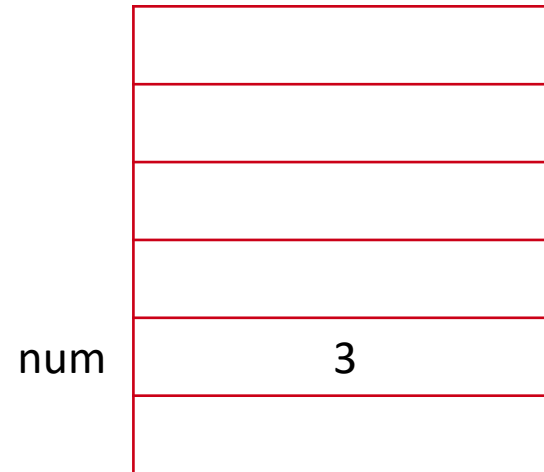
```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
<code>num</code>	0x000000160e1ffc8c	3
<code>pNum</code>		
<code>ppNum</code>		

Punteros

Operadores

```
int num = 3;
```

```
int * pNum = NULL;
```

```
int ** ppNum = NULL;
```

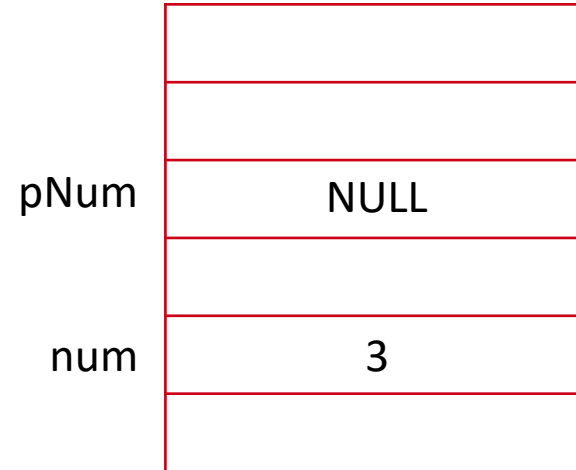
```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	NULL
ppNum		

Punteros

Operadores

```
int num = 3;
```

```
int * pNum = NULL;
```

```
int ** ppNum = NULL;
```

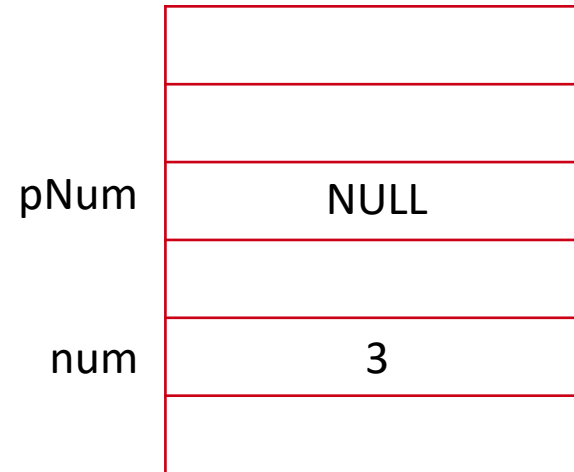
```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	NULL
ppNum		

Punteros

Operadores

```
int num = 3;
```

```
int * pNum = NULL;
```

```
int ** ppNum = NULL;
```

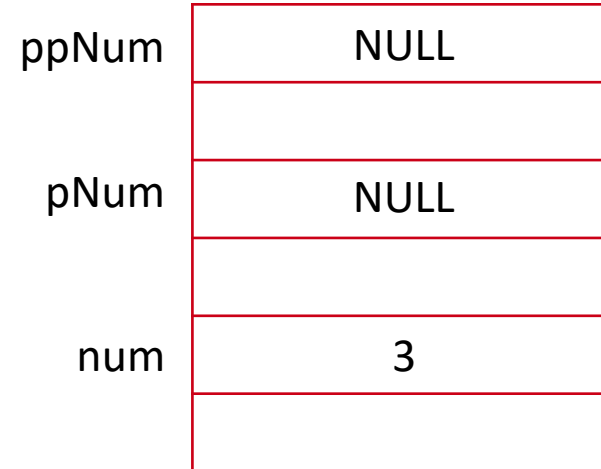
```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	NULL
ppNum	0x000000160e1ffc78	NULL

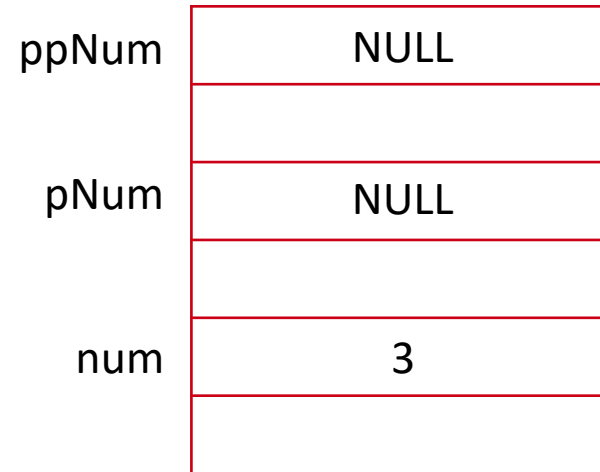
Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;  
**ppNum = **ppNum + 2;  
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	NULL
ppNum	0x000000160e1ffc78	NULL

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

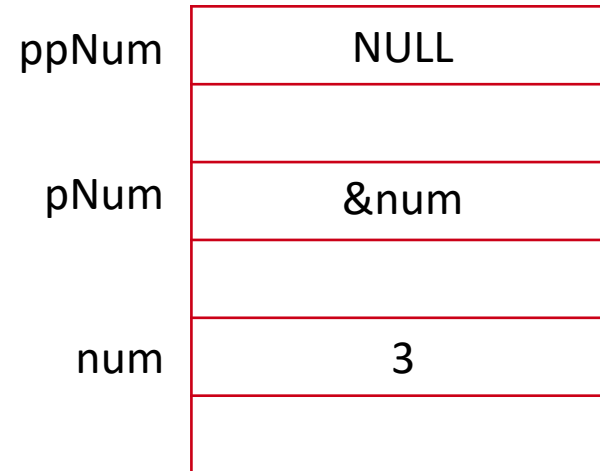
```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	NULL
ppNum	0x000000160e1ffc78	NULL

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```

ppNum	NULL
pNum	0x000000160e1ffc8c
num	3

Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	NULL

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;  
**ppNum = **ppNum + 2;  
**ppNum += 2;
```

ppNum	NULL
pNum	0x000000160e1ffc8c
num	3

Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	NULL

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;
```

```
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```

ppNum	0x000000160e1ffc80
pNum	0x000000160e1ffc8c
num	3

Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;  
**ppNum = **ppNum + 2;  
**ppNum += 2;
```

ppNum	0x000000160e1ffc80
pNum	0x000000160e1ffc8c
num	3

Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

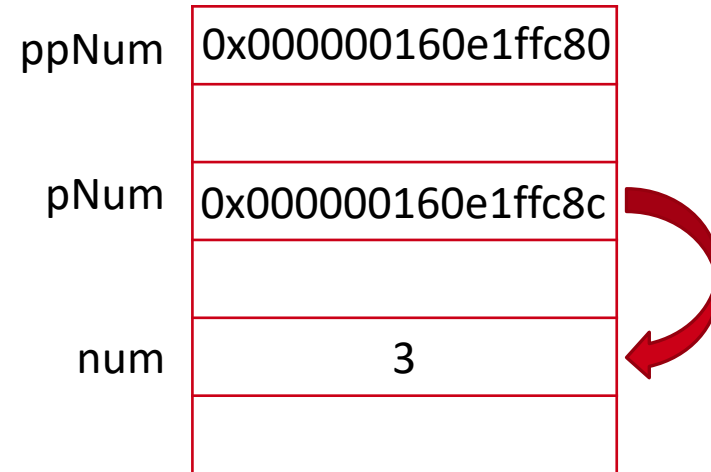
```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	3
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

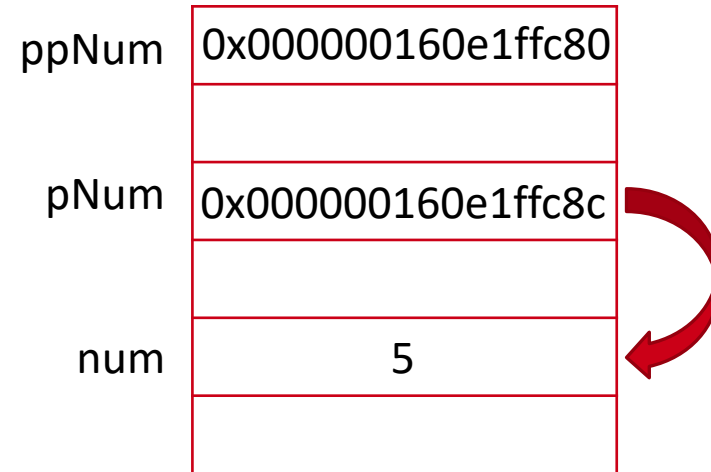
```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	5
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```

ppNum	0x000000160e1ffc80
pNum	0x000000160e1ffc8c
num	5

Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	5
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

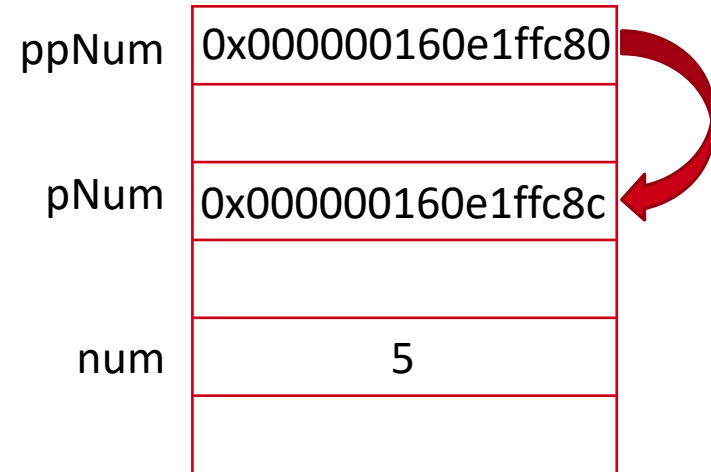
```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	5
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

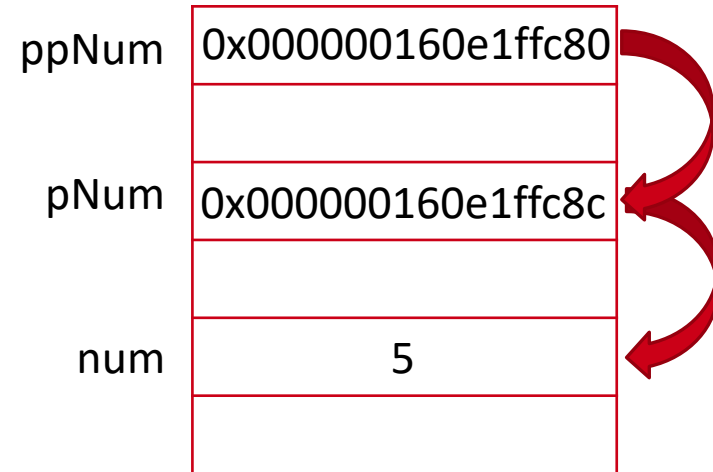
```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	5
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

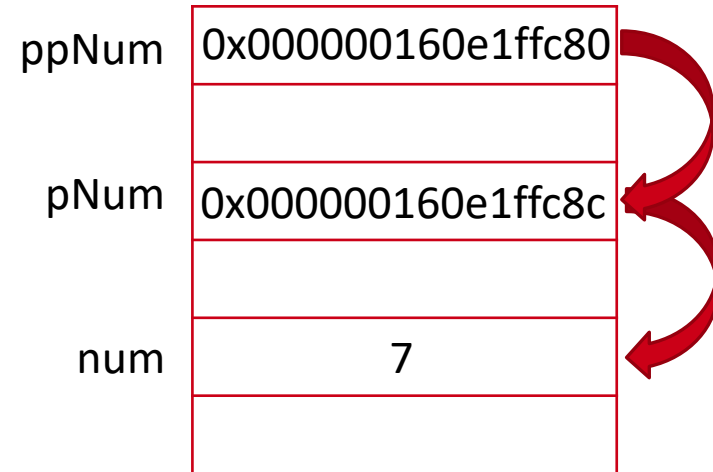
```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;
```

```
**ppNum = **ppNum + 2;
```

```
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	7
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;  
  
pNum = &num;  
ppNum = &pNum;  
  
*pNum = 5;  
**ppNum = **ppNum + 2;  
**ppNum += 2;
```

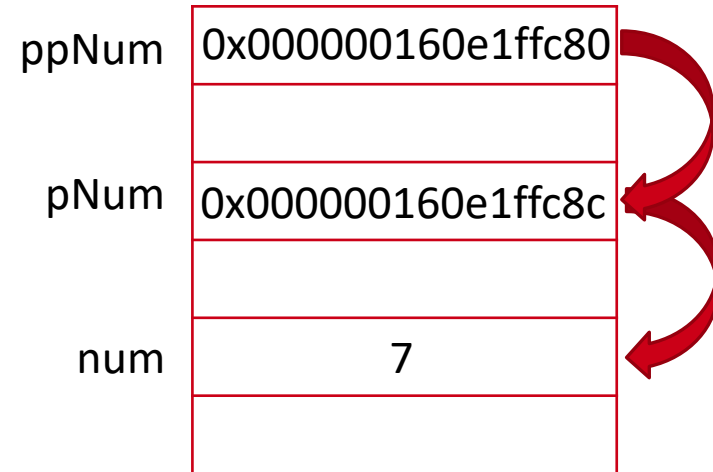
ppNum	0x000000160e1ffc80
pNum	0x000000160e1ffc8c
num	7

Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	7
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;  
  
pNum = &num;  
ppNum = &pNum;  
  
*pNum = 5;  
**ppNum = **ppNum + 2;  
**ppNum += 2;
```

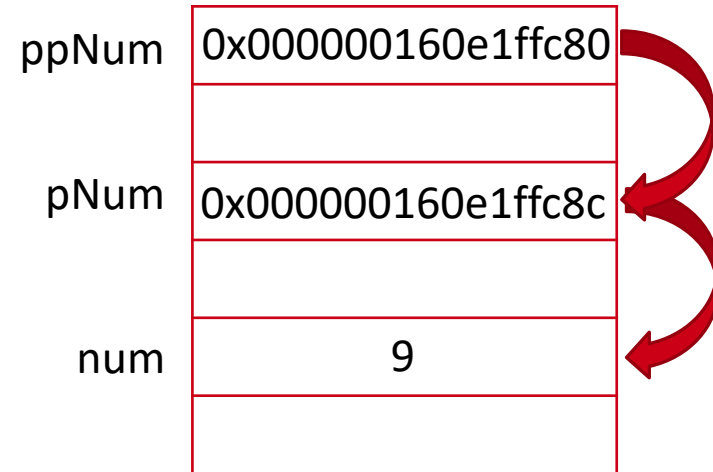


Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	7
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;  
  
pNum = &num;  
ppNum = &pNum;  
  
*pNum = 5;  
**ppNum = **ppNum + 2;  
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	9
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

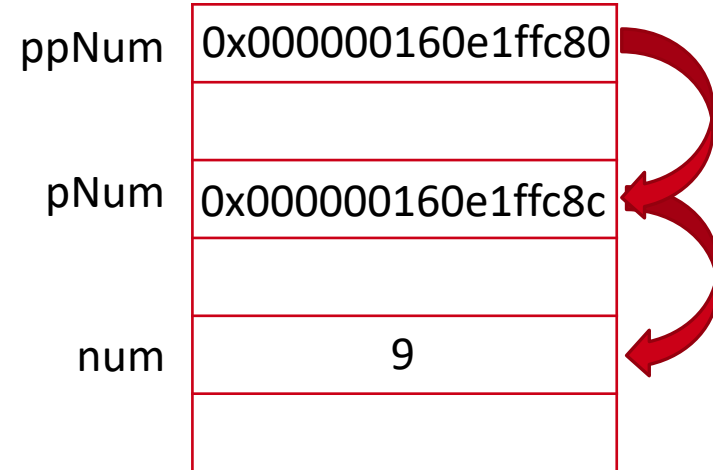
Punteros

Operadores

```
int num = 3;  
int * pNum = NULL;  
int ** ppNum = NULL;
```

```
pNum = &num;  
ppNum = &pNum;
```

```
*pNum = 5;  
**ppNum = **ppNum + 2;  
**ppNum += 2;
```



Identificador	Dirección Memoria	Valor
num	0x000000160e1ffc8c	9
pNum	0x000000160e1ffc80	0x000000160e1ffc8c
ppNum	0x000000160e1ffc78	0x000000160e1ffc80

Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = -32;
```

Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

➔ `int x, y, *p;`

`p = &x;`

`*p = 20;`

`p = &y;`

`*p = -32;`

Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

➔ `int x, y, *p;`

`p = &x;`

`*p = 20;`

`p = &y;`

`*p = -32;`

p



x

y

Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = -32;
```



x

y

Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```



```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = -32;
```

p



x



y



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

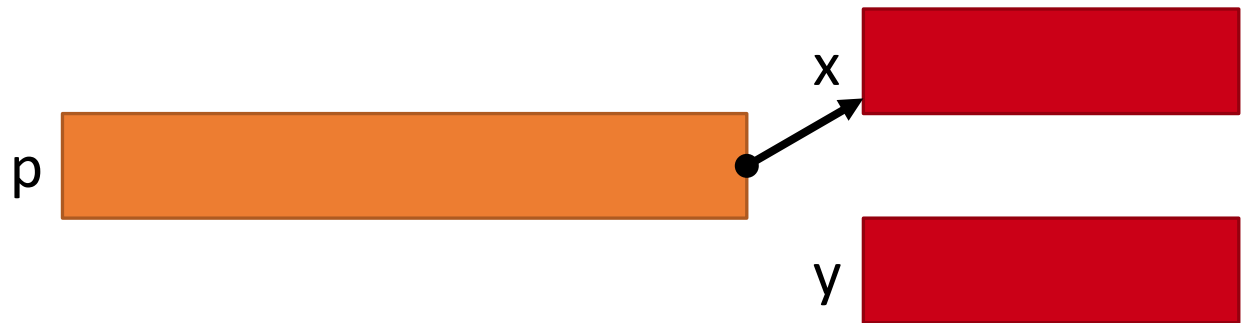


```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```



```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
➔ *p = 20;
```

```
p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
→ *p = 20;
```

```
p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
➔ p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
➔ p = &y;
```

```
*p = -32;
```

p 0x00000000b635ff7a4

x 20

y

Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
➔ p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

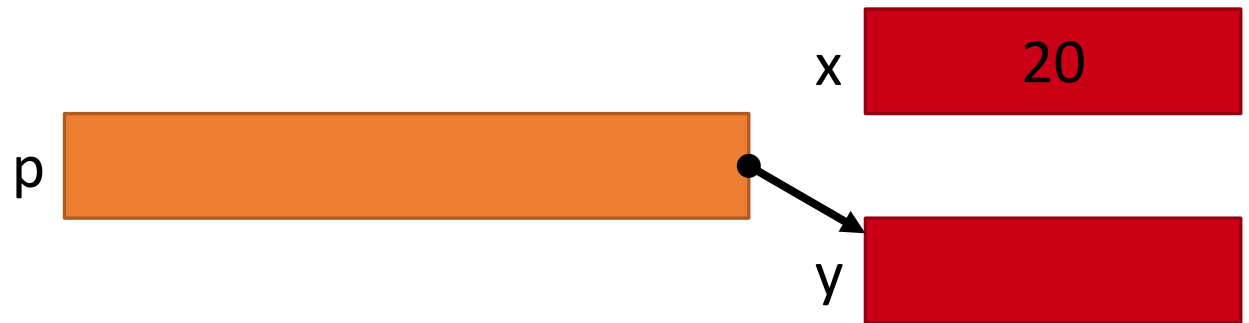
```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
➔ p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

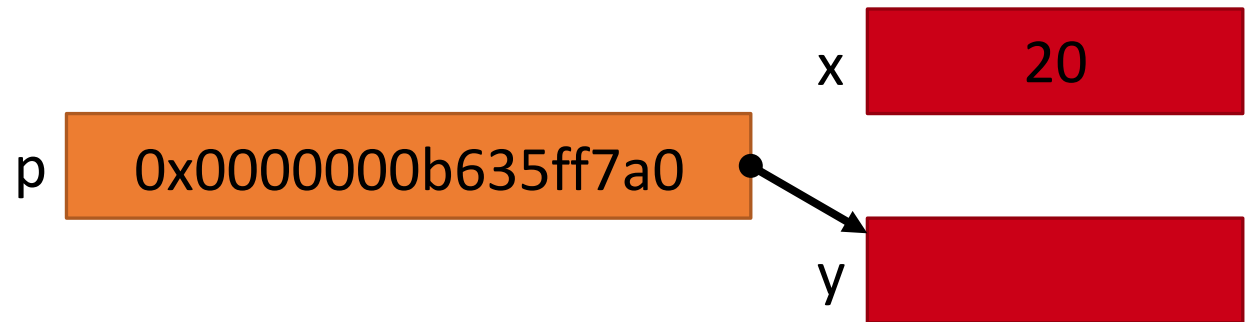
```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
➔ p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

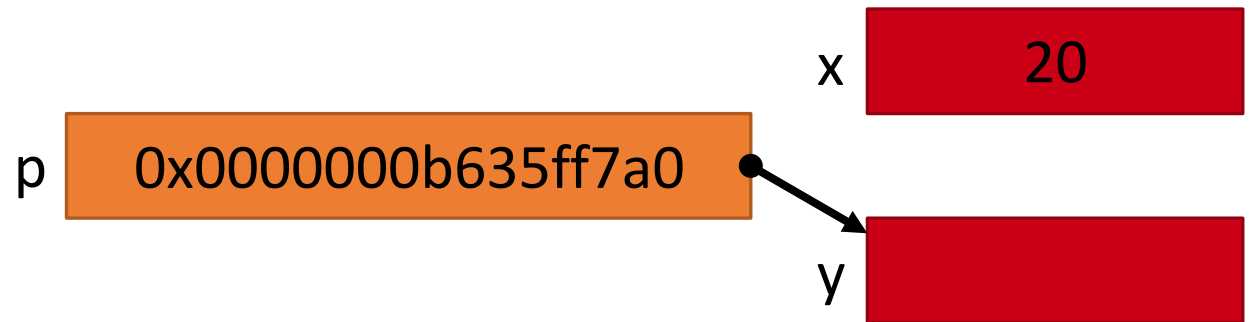
```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
➔ *p = -32;
```



Punteros

Ejemplo

- Un puntero, al igual que cualquier otra variable, puede cambiar su valor (en este caso, la dirección de la variable a la que apunta).

```
int x, y, *p;
```

```
p = &x;
```

```
*p = 20;
```

```
p = &y;
```

```
➔ *p = -32;
```



Punteros

Errores comunes

- Dada la siguiente declaración de variables:

```
int i, *p;
```

```
double *q;
```

- Las siguientes sentencias son incorrectas:

```
&i = p;
```

```
p = 0x0000000b635ff7a0;
```

```
p = q;
```

Punteros

Errores comunes

- Dada la siguiente declaración de variables:

```
int i, *p;
```

```
double *q;
```

- Las siguientes sentencias son incorrectas:

```
&i = p;
```

```
p = 0x0000000b635ff7a0;
```

```
p = q;
```

Punteros

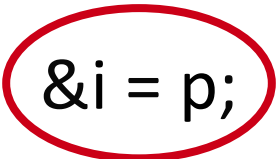
Errores comunes

- Dada la siguiente declaración de variables:

```
int i, *p;
```

```
double *q;
```

- Las siguientes sentencias son incorrectas:

 `&i = p;` ¡¡La dirección de una variable no se puede cambiar!!

```
p = 0x0000000b635ff7a0;
```

```
p = q;
```

Punteros

Errores comunes

- Dada la siguiente declaración de variables:

```
int i, *p;
```

```
double *q;
```

- Las siguientes sentencias son incorrectas:

¡¡La dirección de una variable no
&i = p; se puede cambiar!!

```
p = 0x00000000b635ff7a0;
```

```
p = q;
```

Punteros

Errores comunes

- Dada la siguiente declaración de variables:

```
int i, *p;
```

```
double *q;
```

- Las siguientes sentencias son incorrectas:

¡¡La dirección de una variable no

&i = p; se puede cambiar!!

p = 0x0000000b635ff7a0;

¡¡No puedo asignar una dirección absoluta o un entero a un puntero directamente!!

p = q;

Punteros

Errores comunes

- Dada la siguiente declaración de variables:

```
int i, *p;
```

```
double *q;
```

- Las siguientes sentencias son incorrectas:

¡¡La dirección de una variable no

`&i = p;` se puede cambiar!!

`p = 0x0000000b635ff7a0;`

¡¡No puedo asignar una dirección absoluta o un entero a un puntero directamente!!

`p = q;`

Punteros

Errores comunes

- Dada la siguiente declaración de variables:

```
int i, *p;
```

```
double *q;
```

- Las siguientes sentencias son incorrectas:

ii La dirección de una variable no

```
&i = p; se puede cambiar!!
```

```
p = 0x0000000b635ff7a0;
```

ii No puedo asignar una dirección absoluta o un entero a un puntero directamente!!

```
p = q;
```

ii No se pueden hacer asignaciones directas entre punteros que apuntan a diferentes tipos!!

Punteros

Puntero a `void`

- Los **punteros genéricos** (`void *`) permiten asignarles cualquier tipo de puntero.
- Esto permite que estos punteros apunten a variables de cualquier tipo.

```
int i, *p;
```

```
double *q;
```

```
void *r;
```

```
r = q;
```

```
q = r;
```

```
r = p;
```

Punteros

Puntero a `void`

- Los **punteros genéricos** (`void *`) permiten asignarles cualquier tipo de puntero.
- Sin embargo, aunque no aparezcan los warnings, estos punteros pueden ocasionar problemas. ¿Qué sucede en el siguiente ejemplo?

```
int var[4] = {1,2,3,4};
```

```
int * p = &var[0];
```

```
void * pV;
```

```
double * pd;
```

```
pV = p;
```

```
pd = pV;
```

Aritmética de punteros

Incremento / decremento

- Es posible operar con punteros, admitiendo los operadores aritméticos $+$ y $-$, y los operadores incrementales $++$ y $--$.
- Cuando un puntero (que contiene una dirección de memoria) se modifica en una cantidad N :
 - **No** se modifica la dirección apuntada en N bytes.
 - Se tiene en cuenta el tipo de dato apuntado, modificándose en N elementos del tipo apuntado.
- Es decir, se modifica en `sizeof(tipo)` bytes.

Aritmética de punteros

Resta de punteros

- Uno de los principales usos de la **diferencia** entre punteros es obtener la **distancia** que separa las direcciones a las que apuntamos en memoria.
 - Cuidado, ¡solo se puede hacer entre punteros del **mismo tipo**!!.
- El resultado de esta operación es el **número de elementos** que los separan, **no** el número de bytes.
- Es posible **imprimir** el valor de un puntero por consola utilizando `printf` con el código de control `%p`.

Arrays y punteros

Relación

- En C, un array es un puntero que apunta a una zona de memoria reservada en tiempo de **compilación** (estática) o **ejecución** (si utilizamos arrays de longitud variable o VLA).
- El identificador del array es un puntero al **inicio** (primer elemento) del mismo.
- Sabemos que los elementos del array se disponen de manera **consecutiva** en memoria. Por lo tanto, es posible acceder a un elemento del array utilizando el operador indirección `*`:
 - `array[posición]`
 - `*(array + posición)`

Arrays y punteros

Relación

- El número de elementos de un array debe ser **fijo** durante toda la ejecución del programa.

```
int a[10];
```

- El identificador de la variable representa la **dirección de comienzo** del array en memoria, por lo que:

```
a == &a[0]
```

Arrays y punteros

Relación

- Es posible utilizar **aritmética de punteros** para acceder a los elementos de un array.
- Las expresiones $(a+i)$ y $\&a[i]$ representan la **dirección** de la posición i -ésima del array.
- Por otra parte, las expresiones $*(a+i)$ y $a[i]$ representan el **contenido** del elemento de la posición i -ésima del array.

Arrays y punteros

Relación

```
int a[6];
```


Arrays y punteros

Relación

```
int a[6];
```

4
5
1
3
7
8

Arrays y punteros

Relación

```
int a[6];
```

Acceso elementos

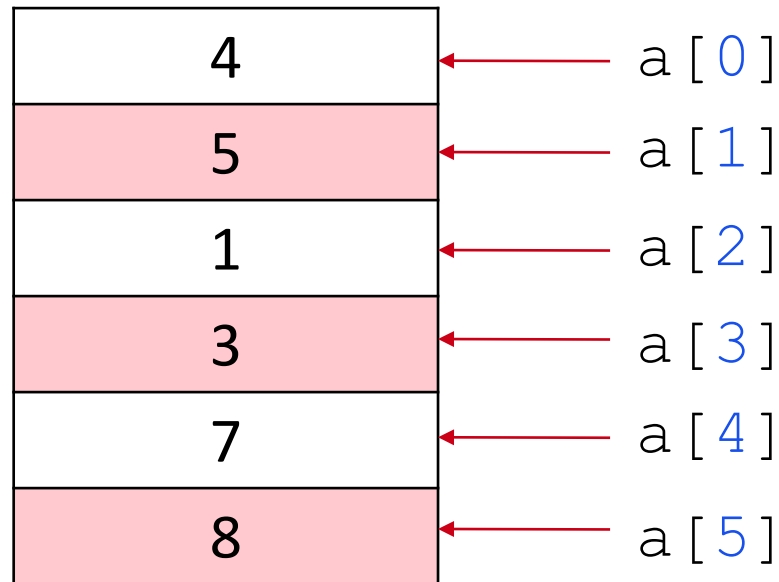
4
5
1
3
7
8

Arrays y punteros

Relación

```
int a[6];
```

Acceso elementos



Indexación

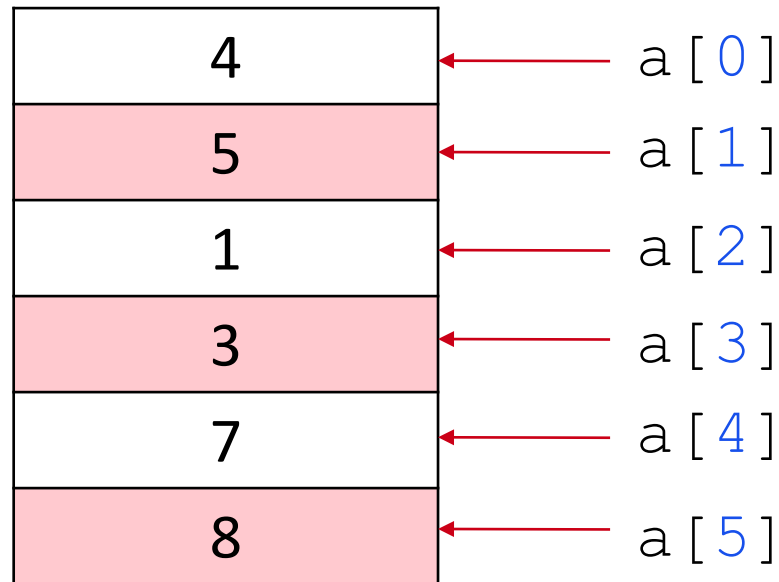
Arrays y punteros

Relación

```
int a[6];
```

Posiciones de memoria

Acceso elementos



Indexación

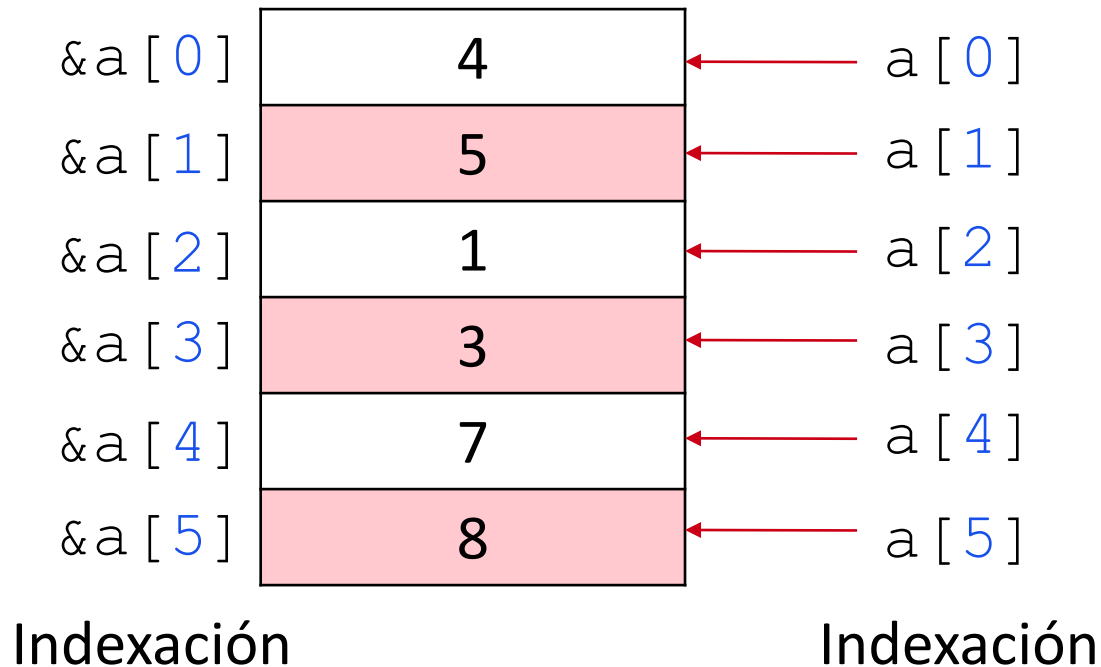
Arrays y punteros

Relación

```
int a[6];
```

Posiciones de memoria

Acceso elementos



Arrays y punteros

Relación

```
int a[6];
```

Posiciones de memoria

Acceso elementos

a	&a[0]	4	←	a[0]
a+1	&a[1]	5	←	a[1]
a+2	&a[2]	1	←	a[2]
a+3	&a[3]	3	←	a[3]
a+4	&a[4]	7	←	a[4]
a+5	&a[5]	8	←	a[5]

Aritmética de punteros
Indexación

Indexación

Arrays y punteros

Relación

`int a[6];`

Posiciones de memoria

Acceso elementos

<code>a</code>	<code>&a[0]</code>	4	<code>a[0]</code>	<code>*a</code>
<code>a+1</code>	<code>&a[1]</code>	5	<code>a[1]</code>	<code>*(a+1)</code>
<code>a+2</code>	<code>&a[2]</code>	1	<code>a[2]</code>	<code>*(a+2)</code>
<code>a+3</code>	<code>&a[3]</code>	3	<code>a[3]</code>	<code>*(a+3)</code>
<code>a+4</code>	<code>&a[4]</code>	7	<code>a[4]</code>	<code>*(a+4)</code>
<code>a+5</code>	<code>&a[5]</code>	8	<code>a[5]</code>	<code>*(a+5)</code>

Aritmética de punteros Indexación

Indexación Aritmética de punteros

Arrays y punteros

Acceso

- Cuando accedemos a la posición i de un array a , podemos hacerlo con la indexación de arrays $a[i]$ o con el operador de indirección $*(a+i)$. Estas instrucciones son equivalentes.
- La indexación de arrays se puede utilizar con cualquier puntero que apunte a un array:

```
int a[N];  
int * pA;  
pA = a;  
pA[3] = 3;  
printf("%d\n", a[3]);
```


Arrays y punteros

Acceso

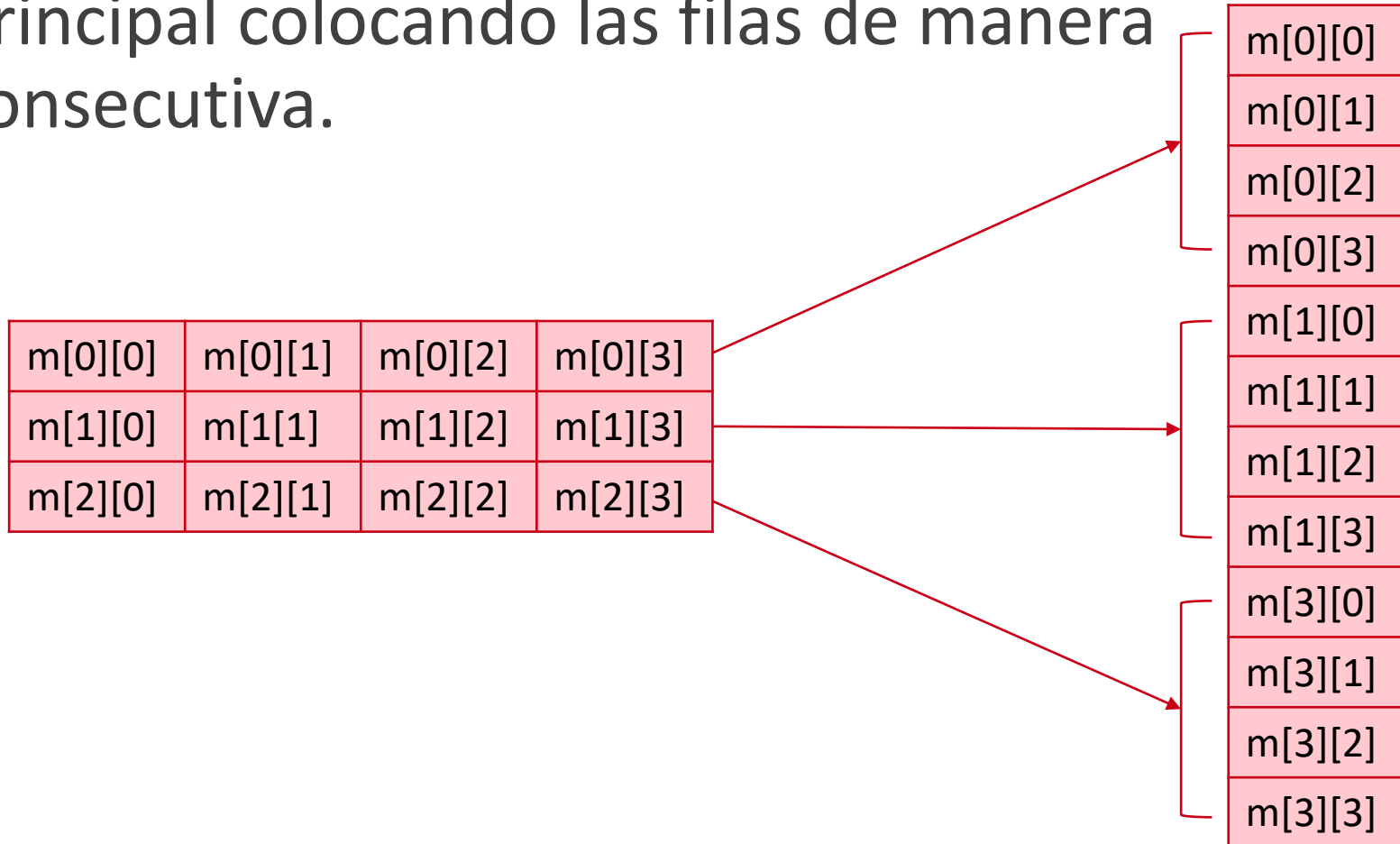
- Si tenemos un array de 100 elementos, y queremos asignar el valor 8 a la posición 2, las siguientes instrucciones son equivalentes:

```
int array[N];  
// Con indexación del array:  
array[2] = 3;  
// Con aritmética de punteros (v1):  
*(array + 2) = 3;  
// Con aritmética de punteros (v2):  
int * puntero_array = array;  
puntero_array += 2;  
*puntero_array = 45;  
puntero_array -= 2;
```

Arrays y punteros

Matrices y punteros

- En C, las matrices se almacenan en memoria principal colocando las filas de manera consecutiva.



Arrays y punteros

Matrices y punteros

- Dada una matriz de $N \times M$ elementos, el acceso a la matriz a través de índices se realiza:

```
matriz[i][j];
```

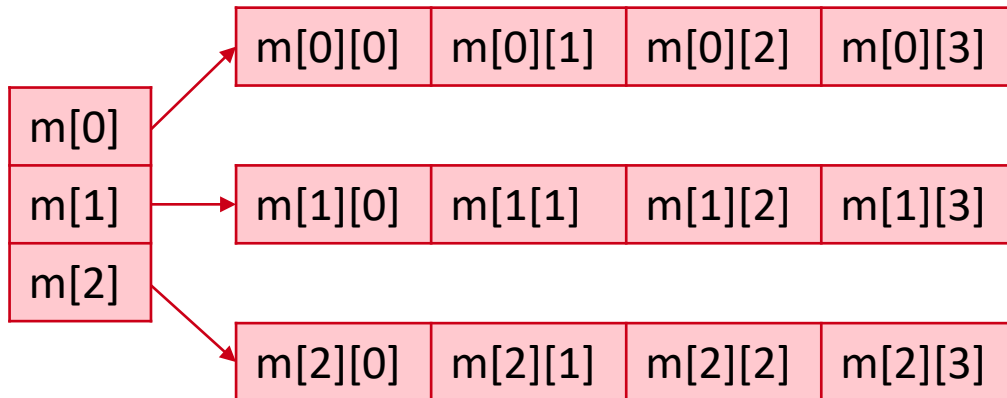
- Además, como las matrices se almacenan de manera **contigua** en memoria, para poder acceder al elemento situado en la fila i , columna j con punteros, podemos utilizar un puntero y la fórmula vista en el tema anterior:

```
int *pMatriz = &matriz[0][0];  
*(pMatriz + (i*M) + j);
```

Arrays y punteros

Matrices y punteros

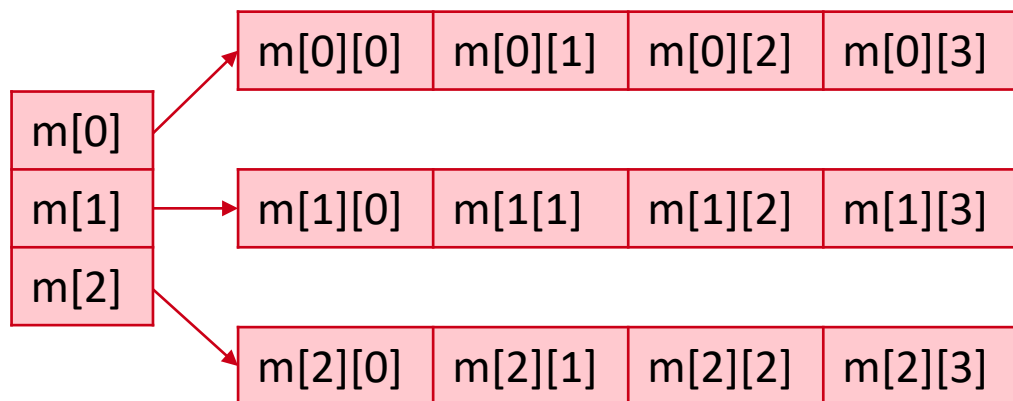
- Por otra parte, también podemos simular una matriz con un puntero a puntero, siendo en este caso nuestra matriz $n \times m$ un vector de tamaño n que contiene punteros a los vectores de cada fila, de longitud m



Arrays y punteros

Matrices y punteros

- Por otra parte, también podemos simular una matriz con un puntero a puntero, siendo en este caso nuestra matriz $n \times m$ un vector de tamaño n que contiene punteros a los vectores de cada fila, de longitud m



```
int matriz[3][4] = {...};
int * fila;
int i, j;
for (i=0;i<N;i++) {
    fila = matriz[i];
    for (j=0;j<M;j++) {
        printf("%2d ", fila[j]);
        printf("\n");
    }
}
```

Arrays y punteros

Matrices y punteros

- En caso de tratar a la matriz como un puntero a puntero también es posible recorrer todos los elementos de la matriz directamente utilizando aritmética de punteros con:

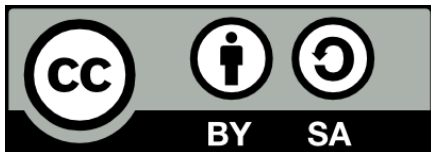
```
* (* (matriz + i) + j) ;
```

- Como hemos visto, en C podemos trabajar con un array bidimensional como si fuera un puntero a puntero. Sin embargo, esto no es bidireccional, ya que un puntero a puntero no tiene por qué ser una matriz.

Tema 7: Funciones

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

- Funciones
 - ¿Para qué y por qué?
 - Subprogramas y módulos
 - Diseño top-down
 - Uso de funciones
 - Ejecución de funciones
 - Declaración
 - Llamada
 - Definición
 - Ámbito
 - Paso de parámetros
- Recursividad
 - ¿Qué es la recursividad?
 - Conceptos clave
 - Programación declarativa
 - Metodología
 - Ejemplo
 - Conclusiones
- Función main
 - Parámetros
- Módulos

Funciones

¿Para qué y por qué?

- Hasta ahora, nuestros programas se encuentran implementados en su totalidad en la **función** `main`.
- Es cierto que la función `main` puede tener tantas líneas de código como necesitemos.
Pero... cuando hay demasiadas líneas, el código se vuelve **difícil de entender y mantener**.
- Además, cuando tenemos un problema **complejo**, **no** suele ser una buena estrategia resolverlo utilizando una única función.

Funciones

¿Para qué y por qué?

Las **funciones** son bloques de código que se encargan de realizar una tarea específica dentro de un programa. Su uso es fundamental en la programación estructurada porque:

- Facilitan la **reutilización** del código.
- Mejoran la **legibilidad** y el mantenimiento al dividir el programa.
- Permiten **modularidad**, permitiendo ver cada función como un "módulo" independiente.
- Facilitan el trabajo en **equipo**.

Funciones

Subprogramas y módulos

- Una aplicación real puede tener **miles** de líneas de código.
- Lo que dificulta enormemente su mantenimiento y actualización.
- Para un mejor mantenimiento del código se suele dividir un programa en:
 - **Subprogramas:** funciones y procedimientos.
 - **Módulos.**

Funciones

Subprogramas

- Un **subprograma** es el resultado de dividir nuestro problema o programa en partes más pequeñas.
- Los **subprogramas** contienen **código** que puede ser ejecutado desde el programa principal o desde cualquier otro subprograma.
- Producen **resultados** a partir de unos datos de entrada que reciben a través de sus **argumentos** (parámetros).

Funciones

Subprogramas

- Los subprogramas pueden dividirse en:
 - Funciones, si pueden devolver un valor.
 - Procedimientos, si no devuelven ningún valor.
- En C, todos los subprogramas se denominan funciones, aunque algunas no devuelvan ningún valor.
- En resumen, una **función** es un conjunto de instrucciones invocado por el programa principal (o por otras funciones) que puede devolver un valor o no, dependiendo de su tipo de retorno.

Funciones

Módulo

- Un **módulo** es una parte del programa que contiene un grupo de funciones relacionadas entre sí (*stdio.h*, *string.h*, etc.).
- El uso de módulos permite dividir el programa en pequeños componentes, donde cada componente tiene una responsabilidad clara.
- El uso de módulos mejora la organización y el mantenimiento del código.

Funciones

Diseño top-down

- La metodología **top-down** es una estrategia de diseño y resolución de problemas que consiste en descomponer un problema complejo en partes más manejables.
- En lugar de abordar todo el problema a la vez, se identifica su estructura general, y después se detallan y desarrollan las partes más pequeñas de manera gradual.
- Partimos de un código complejo que **dividimos** en subproblemas, y estos a su vez en subproblemas más pequeños, etc., siguiendo la estrategia de divide y vencerás.
- En C podemos seguir esta metodología a través del uso de funciones.

Ventajas:

- Ayuda a tener una visión clara del sistema completo desde el inicio.
- Permite una planificación estructurada.
- Facilita la división de tareas y la asignación de funciones.

Ejemplo de diseño top-down:

Imaginemos que queremos crear un programa que calcule el área de varias figuras geométricas. El diseño top-down se enfocaría primero en definir la estructura general del programa y luego ir descomponiendo cada tarea.

Funciones

Uso de funciones

- Cada **subproblema** de nuestro problema inicial se puede implementar en una función en C.
- Una función es un **grupo** de instrucciones que permite resolver una determinada tarea.
- A partir de una **entrada**, la función realiza una serie de **operaciones** y genera una **salida**.
- Cada función tiene una tarea específica que se puede invocar en cualquier parte del código, facilitando la reutilización de código y la simplificación de las tareas complejas.

Funciones

Uso de funciones

- Para **llamar** a una función, sólo debemos saber cuál es su **interfaz**:
 - **Identificador** de la función.
 - Datos o parámetros de **entrada**.
 - Datos o tipo de **salida**.
- El código de la función no debería ser relevante para su uso, lo importante es la funcionalidad de ese "subprograma" del código, así como la(s) entrada(s) y la(s) salida(s).
- Ya hemos utilizado funciones de diferentes bibliotecas sin saber cómo están implementadas.

Funciones

Ejecución de funciones

- Al invocar a una función, se ejecutan sus instrucciones y después se continúa la ejecución del programa en la **instrucción siguiente a la llamada** a la función.

El flujo de ejecución es:

1. El programa principal o la función que realiza la llamada **transfiere el control**.
2. Los **parámetros (argumentos) se pasan** a la función.
3. La función llamada **recibe** los argumentos.
4. La **función ejecuta** sus instrucciones.
5. Si la función tiene un valor de **retorno**, lo devuelve al punto de llamada.

¿Cómo se crean funciones en C?

- **Declaración:** se define el prototipo o firma de la función (tipo de retorno, nombre, y parámetros).
- **Definición:** se escribe el cuerpo de la función, especificando las instrucciones que ejecutará.
- **Llamada:** se invoca la función desde el programa principal o desde otra función.

Funciones

Ejecución de funciones – Ejemplo función *void*

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

```
void saludo();
```

(1) Declaración

```
int main() {  
    printf("Antes de la función\n");  
    saludo();  
    printf("Después de la función\n");  
    return 0;  
}
```

(3) Llamada

```
void saludo() {  
    printf("Buenos días\n");  
}
```

(2) Definición

Funciones

Ejecución de funciones – Ejemplo función *que regresa valores*

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

```
int sumar(int a, int b);
```

(1) Declaración

```
int main() {
```

```
    int resultado = sumar(5, 3);  
    printf( "El resultado es: %d\n", resultado);
```

(3) Llamada

```
    return 0;
```

```
}
```

```
int sumar(int a, int b) {  
    return a + b;  
}
```

(2) Definición

Funciones

Ejecución de funciones – Ejemplo función *que regresa valores*

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

```
int sumar(int a, int b);
```

```
int main() {  
    int resultado = sumar(5, 3);  
    printf( "El resultado es: %d\n", resultado);  
    return 0;  
}
```

```
int sumar(int a, int b) {  
    return a + b;  
}
```

(1) Transfiere el control

(2) Paso de parámetros

(5) Continuación del flujo del programa

(3) Recepción de parámetros

(4) Ejecución y retorno de resultado (si fuera el caso)

- A continuación, revisaremos las etapas principales para ejecutar una función:
 - Declaración.
 - Llamada.
 - Definición.
 - Ámbito.

Funciones

Declaración de funciones

- Toda función debe estar **declarada** antes de poder llamarla.
- Consiste en especificar el **tipo de retorno**, el **identificador de la función**, y los **parámetros** que requiere.
- La declaración se puede hacer en:
 - En la sección de declaraciones globales (antes de la función `main`).
 - En algún fichero cabecera que se incluya mediante la directiva `#include`.

Funciones

Declaración de funciones

- La declaración se denomina **prototipo** de la función, y su sintaxis es:

```
tipo_retorno identificador_funcion(parametro1, ...,  
                                   parametroN);
```

Donde:

- **tipo_retorno**: tipo de dato que la función devuelve (`int`, `float`, `char`, etc.). Si no devuelve nada, se usa `void`.
- **identificador_funcion**: El identificador utilizado para nombrar a la función.
- **parametro1, ... parametroN**: tipos de los parámetros que recibe la función (opcional si no recibe parámetros). Además del tipo, es posible incluir el nombre del parámetro.

Funciones

Declaración de funciones

- Si la función no devuelve ningún valor, el tipo de retorno será **void**.
- Si la función no utiliza parámetros, podemos indicar **void** o dejar vacío ese espacio **()**.
- Los nombres de los parámetros son identificadores válidos en C y se llaman **argumentos formales**.
- La declaración permite que el compilador **compruebe** los argumentos (número y tipo) y el retorno.
- En la declaración podemos **suprimir** los nombres de los argumentos, pero **no sus tipos**:

```
double potencia(double, double);
```

- Aunque es una buena práctica escribir ambos:

```
double potencia(double base, double exp);
```

Funciones

Llamada a una función

- Permite **ejecutar** el código de una función en un momento determinado.
- La llamada se realiza con el **identificador** de la función seguido de paréntesis (). Si la función tiene **argumentos de entrada**, se incluyen entre los paréntesis.
- Sintaxis de la llamada a una función:
`identificador_funcion(arg1, arg2, ...);`
- Ejemplo:
`potencia(5, 3);`

Funciones

Llamada a una función

- Los argumentos actuales son los datos que se **envían** como datos de entrada.
- El número y tipo de argumentos debe **coincidir** con los indicados en la declaración.

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

```
int potencia(int base, int exp);
```

```
int main() {  
    int resultado = potencia(5, 3);  
    printf( "El resultado es: %d\n", resultado);  
    return 0;  
}
```

Funciones

Llamada a una función

- Cuando se llama a una función, se ejecutan sus instrucciones y después se **vuelve** a la **siguiente** instrucción tras la llamada.
- Cuando la función devuelve un valor, se sustituye su llamada por el **valor devuelto** por la función.
- Ejemplo:

```
int resultado = potencia(5, 3);
```

Funciones

Llamada a una función

Parámetros formales

```
int potencia(int base, int exp);
```

```
int main() {
```

```
    int resultado = potencia(5, 3);
```

```
    ...
```

```
}
```

Parámetros actuales

Funciones

Llamada - ¿Qué ocurre al llamar a una función?

1. Se evalúan las expresiones de los argumentos actuales y se obtiene su valor.
2. Se reserva la memoria necesaria para los parámetros formales.
3. Cada parámetro formal toma como valor **una copia del valor** del argumento actual correspondiente.
4. Se ejecuta la función, devolviendo el resultado (si hay retorno).
5. Se sustituye la llamada por el resultado de la función.
6. Se continúa la ejecución en la instrucción siguiente a la llamada.

Funciones

Definición de una función

- La definición de una función es el proceso de **escribir el cuerpo de la función**, detallando lo que hace cuando es llamada.
- Debe incluir el **tipo de retorno**, el **identificador** de la función, los **parámetros formales**, y las **instrucciones que ejecutará**. Es donde se escribe la lógica de la función.

```
tipo_retorno identificador_funcion(tipo param1, ..., tipo paramN) {  
    // Código de la función  
    return valor_retorno;  
}
```

Donde:

- **tipo_retorno**: tipo del valor devuelto por la función.
- **identificador_funcion**: identificador de la función.
- **valor_retorno**: expresión resultante de la función.

Funciones

Definición de una función

```
                                Parámetros formales
                                ┌──────────────────┐
Variables locales { int potencia(int base, int exp) {
  int resultado = 1;
  for(int i = 0; i < exp; i++)
    resultado *= base;
  return resultado; } Retorno de la función
}
```

Funciones

Ámbito de una función

- El **ámbito** de una variable es la zona del código donde ésta puede ser utilizada.
- Una variable puede tener ámbito de programa, de una función o de un bloque.
- El ámbito de una variable viene determinado por su **declaración**.

Funciones

Ámbito de una función

- Las variables con ámbito de **programa** son las de **ámbito global**, visibles desde cualquier punto de nuestro código. Se recomienda **evitar su uso** porque dificultan la reutilización y modulación del código, así como su mantenimiento. Además, también hacen su lectura y seguimiento más complejo.
- Las variables con ámbito de **función** son las variables **locales a una función**, que existen solo dentro de ella.
- Las variables con ámbito de **bloque** son variables declaradas dentro de un bloque `{ }` y solo existen dentro de ese bloque.

Funciones

Ámbito de una función

En este ejemplo, la variable **global** es accesible en todo el programa, mientras que la **local** solo lo es dentro de `main`:

```
#include <stdio.h>

int global = 10; // Variable global

void imprimirGlobal() {
    printf("Valor de la variable global: %d\n", global);
}

int main() {
    int local = 5; // Variable local
    imprimirGlobal();
    printf("Valor de la variable local: %d\n", local);
    return 0;
}
```

Funciones

Paso de parámetros

- Existen dos formas de pasar parámetros:
 - Por **valor**.
 - Por **referencia**.
- En C, el paso de parámetros siempre se realiza por **valor**.
- El paso por referencia se emula con **punteros**.

Funciones

Paso de parámetros por valor

- La asignación de los valores a los parámetros se denomina **paso de parámetros por valor**.
- El valor de los parámetros actuales **se copia** en los respectivos parámetros formales.
 - Dentro de la función se trabaja con **copias**, no con la variable original.
- Las variables que se pasan en la llamada (parámetros actuales) **no se modifican**, aunque se cambie su valor dentro de la función.
- Al terminar la función se vuelve al lugar donde se realizó la llamada, y las variables que intervenían en los parámetros actuales **continúan con el mismo valor**.

Funciones

Paso de parámetros por valor - Ejemplo

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

```
void cambiarValor(int x) {
```

```
    x = 10; // Cambia el valor de la copia local, no el original
```

```
}
```

```
int main() {
```

```
    int numero = 5;
```

```
    cambiarValor(numero);
```

```
    printf("Valor de numero: %d\n", numero); // Imprime 5
```

```
    return 0;
```

```
}
```


Funciones

Paso de parámetros por referencia

- En el paso de parámetros por **referencia**, las variables que se pasan como parámetros sí se ven modificadas durante la ejecución de la función, ya que se pasa la variable **original**, no una copia.
- En C no es posible hacer esto (ya que siempre se realiza una copia), por lo que el paso por referencia se simula mediante **punteros**.

Funciones

Paso de parámetros por referencia

- Lo que se copia es el puntero, pero el contenido al que apunta es el **original**.
- La función tiene así **acceso** al valor del parámetro a través del puntero.
- Si modificamos el contenido a través del puntero, se verá modificado el contenido original.
- Sin embargo, si modificamos el valor del puntero (lo apuntamos a otra dirección de memoria) el puntero original no se verá modificado.

Funciones

Paso de parámetros por referencia

- El paso por referencia permite devolver más de un resultado.
- Si deseamos retornar más de un valor, podemos utilizar **return** para devolver uno de los resultados, y modificamos los parámetros pasados por referencia para completar el resto.

Funciones

Paso de parámetros por referencia - Ejemplo

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

```
void cambiarValor(int * x) {  
    *x = 10; // Cambia el valor del original a través del puntero  
}
```

```
int main() {  
    int numero = 5;  
    cambiarValor(&numero);  
    printf("Valor de numero: %d\n", numero); // Imprime 10  
    return 0;  
}
```

Funciones

Ejemplo de paso de parámetros por valor

```
int main() {  
→ int n = 2;  
  incremento(n);  
  printf("%d\n", n);  
}
```

```
void incremento(int n) {  
  n++;  
}
```

Memoria

	Dirección	Valor
n	0000007bd83ff6dc	2

Funciones

Ejemplo de paso de parámetros por valor

```
int main() {  
    int n = 2;  
    → incremento(n);  
    printf("%d\n", n);  
}
```

```
void incremento(int n) {  
    n++;  
}
```

Memoria

	Dirección	Valor
n	0000007bd83ff6dc	2

Funciones

Ejemplo de paso de parámetros por valor

```
int main() {  
    int n = 2;  
    incremento(n);  
    printf("%d\n", n);  
}
```



```
void incremento(int n) {  
    n++;  
}
```

Memoria

	Dirección	Valor
n	0000007bd83ff6dc	2

	Dirección	Valor
n	0000007bd83ff6b0	2

Funciones

Ejemplo de paso de parámetros por valor

```
int main() {  
    int n = 2;  
    incremento(n);  
    printf("%d\n", n);  
}
```



```
void incremento(int n) {  
    n++;  
}
```



Memoria

	Dirección	Valor
n	0000007bd83ff6dc	2

	Dirección	Valor
n	0000007bd83ff6b0	3

Funciones

Ejemplo de paso de parámetros por valor

```
int main() {  
    int n = 2;  
    incremento(n);  
    printf("%d\n", n);  
}
```



```
void incremento(int n) {  
    n++;  
}
```

Memoria

	Dirección	Valor
n	0000007bd83ff6dc	2

Funciones

Ejemplo de paso de parámetros por referencia

```
int main() {  
→ int n = 2;  
  incremento(&n);  
  printf("%d\n", n);  
}
```

```
void incremento(int * n) {  
  (*n)++;  
}
```

Memoria

	Dirección	Valor
n	000000e743fff9bc	2

Funciones

Ejemplo de paso de parámetros por referencia

```
int main() {  
    int n = 2;  
    → incremento(&n);  
    printf("%d\n", n);  
}
```

```
void incremento(int * n) {  
    (*n)++;  
}
```

Memoria

	Dirección	Valor
n	000000e743fff9bc	3

Funciones

Ejemplo de paso de parámetros por referencia

```
int main() {  
    int n = 2;  
    incremento(&n);  
    printf("%d\n", n);  
}
```



```
void incremento(int * n) {  
    (*n)++;  
}
```

Memoria

	Dirección	Valor
n	000000e743fff9bc	2

	Dirección	Valor
*n	000000e743fff990	000000e743fff9bc

Funciones

Ejemplo de paso de parámetros por referencia

```
int main() {  
    int n = 2;  
    incremento(&n);  
    printf("%d\n", n);  
}
```



```
void incremento(int * n) {  
    (*n)++;  
}
```



Memoria

	Dirección	Valor
n	000000e743fff9bc	3

	Dirección	Valor
*n	000000e743fff990	000000e743fff9bc



Funciones

Ejemplo de paso de parámetros por referencia

```
int main() {  
    int n = 2;  
    incremento(&n);  
    printf("%d\n", n);  
}
```



```
void incremento(int * n) {  
    (*n)++;  
}
```

Memoria

	Dirección	Valor
n	000000e743fff9bc	3

Funciones

Paso de arrays como parámetro

- En C, un array siempre es un **puntero**, por lo que cuando pasamos un array como parámetro a una función siempre se pasa por **referencia**.
- Esto significa que cualquier modificación que se haga al array dentro de la función afectará al array original, ya que se pasa la dirección base (dirección del primer elemento) del array.

Funciones

Paso de arrays como parámetro

- La función sólo recibe un puntero al primer elemento, por lo que no es posible conocer la dimensión del array.
 - Es necesario incluir otro parámetro adicional con el tamaño del array:

```
void funcion(int * vector, int numElementos);
```

- Para llamar a la función, se pasa como parámetro el nombre del array:

```
int v[10];
```

```
funcion(v, 10);
```


Funciones

Ejemplo de paso de un array como parámetro

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

```
void modificarArray(int arr[], int size) {  
    for (int i = 0; i < size; i++) {  
        arr[i] *= 2; // Duplica cada valor del array  
    }  
}
```

```
int main() {  
    int numeros[] = {1, 2, 3, 4, 5};  
    int tamaño = 5;  
    modificarArray(numeros, tamaño);  
    // Imprime el array modificado  
    for (int i = 0; i < tamaño; i++) {  
        printf("%d ", numeros[i]); // Imprime 2 4 6 8 10  
    }  
    return 0;  
}
```

En este ejemplo, se pasa el array **numeros** a la función **modificarArray**.

La función modifica el contenido del array original.

Recursividad

¿Qué es la recursividad y las funciones recursivas?

- Una función se puede llamar desde cualquier lugar:
 - Desde el programa principal.
 - Desde otra función.
 - **Desde dentro de la propia función: recursividad.**
- La recursividad es una técnica en la que una función se **llama a sí misma** directa o indirectamente.
- Permite resolver un problema **dividiéndolo en subproblemas más pequeños.**

Recursividad

¿Qué es la recursividad y las funciones recursivas?

- La recursividad se utiliza para resolver problemas que tienen una naturaleza **repetitiva**, como calcular factoriales, resolver problemas de recorridos o algoritmos de búsqueda.
- Es una **alternativa a los bucles** (a la iteración), para ejecutar instrucciones de manera repetida.

Recursividad

Conceptos clave

- **Descomposición o simplificación** de problemas:
 - Para resolver ciertos problemas, es posible utilizar **soluciones a "subproblemas"** idénticos al original, pero más sencillos o de menor tamaño.
- **Inducción:**
 - Diseñamos nuestra solución suponiendo que **ya sabemos la solución a estos problemas más simples.**
 - No es necesario calcular las "subsoluciones", se obtienen a través de llamadas recursivas.

Recursividad

Paradigma de la programación declarativa

- En general, debemos pensar en **qué** vamos a hacer en lugar de en **cómo** vamos a hacerlo.
- A diferencia del paradigma imperativo, evitaremos pensar en **cómo** se modifican los parámetros y variables en cada iteración.
- Suponemos que conocemos **qué** se resuelve (el subproblema) sin interesarnos en el **cómo**.
- **Salto de fe recursivo:** utilizamos la función recursiva que estamos programando asumiendo que funciona, aunque no hayamos terminado de implementarla.

Recursividad

Metodología para diseñar algoritmos recursivos

1. Identificación del **tamaño del problema** (lo que determinará el número de operaciones a realizar).
2. Establecimiento de los **casos base**. Estos son instancias sencillas (suelen ser las de menor tamaño) que se resuelven sin llamadas recursivas.
3. Descomposición: escoger **instancias del problema de menor tamaño**.
4. Inducción: establecer los **casos recursivos** a partir de las soluciones de las instancias seleccionadas en la fase anterior.
5. Implementación y **pruebas**.

Recursividad

Ejemplo: Factorial

- Ejemplo: cálculo del factorial:

$$F(n) = n! = \prod_{i=1}^n i = 1 \times 2 \times 3 \times \cdots \times (n - 1) \times n$$

Recursividad

Ejemplo: Factorial

- Ejemplo: cálculo del factorial:

$$F(n) = n! = \prod_{i=1}^n i = 1 \times 2 \times 3 \times \cdots \times (n - 1) \times n$$

1. ¿Tamaño del problema?

n

Recursividad

Ejemplo: Factorial

- Ejemplo: cálculo del factorial:

$$F(n) = n! = \prod_{i=1}^n i = 1 \times 2 \times 3 \times \cdots \times (n - 1) \times n$$

2. ¿Casos base?

- $n = 0$. En este caso: $F(0) = 1$
- $n = 1$. En este caso: $F(1) = 1$

Recursividad

Ejemplo: Factorial

- Ejemplo: cálculo del factorial:

$$F(n) = n! = \prod_{i=1}^n i = 1 \times 2 \times 3 \times \cdots \times (n - 1) \times n$$

3. Descomposición:

- Posibles problemas de menor tamaño:
 - $n - 1, n - 2, n/2, n/10...$
- Algunas opciones no conducen a algoritmos sencillos o eficientes
- La opción más sencilla: $F(n - 1)$

Recursividad

Ejemplo: Factorial

- Ejemplo: cálculo del factorial:

$$F(n) = n! = \prod_{i=1}^n i = \underbrace{1 \times 2 \times 3 \times \dots \times (n-1)}_{\text{Subproblema: } (n-1)!} \times n$$

Problema: n!

4. Casos recursivos:

$$F(n) = \begin{cases} n \times (n-1)! & \text{si } n > 1 \\ 1 & \text{si } n \leq 1 \end{cases}$$

Recursividad

Ejemplo: Factorial

```
#include <stdio.h>
int factorial(int n) {
    if (n == 0) {
        return 1; // Caso base
    } else {
        return n * factorial(n - 1); // Llamada recursiva
    }
}
int main() {
    int num = 5;
    printf("El factorial de %d es: %d\n", num, factorial(num));
    return 0;
}
```

En este ejemplo, la función **factorial** se llama a sí misma hasta que alcanza el caso base ($n = 0$), en el cual la función retorna 1 y las llamadas recursivas comienzan a resolverse.

Recursividad

Conclusiones

- Para detener las llamadas recursivas, se introduce un **caso base**, momento en el cual debe terminar la ejecución de la función.
- La recursividad continúa mientras que **no** se produzca el caso base:
 - En el caso del factorial, cuando $n = 1$ o $n = 0$ si consideramos el caso particular del factorial de 0.
- Debemos garantizar que **siempre** se llegará al caso base en algún momento:
 - En el factorial, si disminuimos el valor de n , en algún momento tomará el valor de 1.

Función main

Parámetros de la función main

- En nuestros programas, hemos definido la función principal como:

```
int main() {  
    // Código  
    return 0;  
}
```

- Sin embargo, se puede definir como:

```
int main(int argc, char * argv[]) {  
    // Código  
    return 0;  
}
```

Función main

Parámetros de la función main

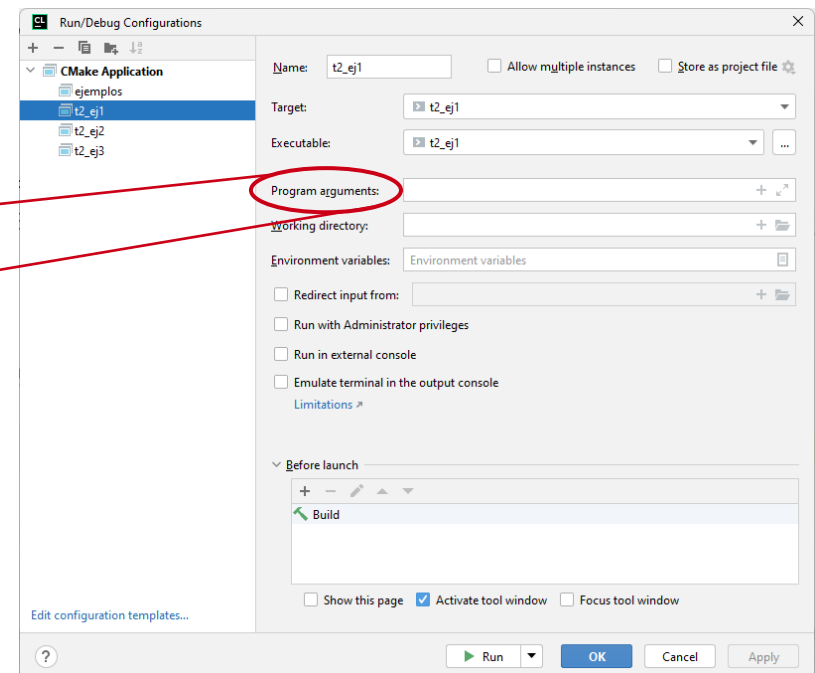
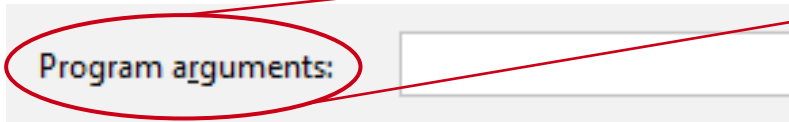
- Nuestra función `main` puede recibir dos parámetros o argumentos:
 - **`argc`**: (argument count) número de argumentos que le pasamos a nuestro programa.
 - **`argv`**: (argument vector) array con los parámetros que le pasamos a nuestro programa.

```
int main(int argc, char * argv[]) {  
    // Código  
    return 0;  
}
```

Función main

Parámetros de la función main

- Para utilizar esos parámetros o argumentos, basta con indicarlos al ejecutar nuestro programa:
 - Por línea de comandos: programa.exe p1 p2 ... pN.
 - A través del IDE (Pestaña "Run" > "Edit Configurations").



Función main

Parámetros de la función main

- El primer parámetro del vector (argv[0]) siempre será el nombre de nuestro programa:

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

```
int main(int argc, char * argv[]) {  
    for (int i = 0; i < argc; i++) {  
        printf("Parametro %d: %s\n", i, argv[i]);  
    }  
}
```

Función main

Parámetros de la función main

- Todos los parámetros o argumentos del vector `argv` siempre serán de tipo **cadena de caracteres**, por lo que necesitaremos convertirlos al tipo de dato necesario.
- **No** es suficiente con hacer un *casting*
- Podemos utilizar funciones ya definidas en C, como las funciones `atoi` (ASCII to `int`), `atof` (ASCII to `float`) definidas en `stdlib.h`.

```
double d = atof(argv[3]);  
int a = atoi(argv[1]);
```

Módulos

Introducción

- Hasta ahora todo el código se encontraba en el fichero `main.c`, lo produce un código **poco modular**, todo en el mismo fichero.
- En C existe la opción de **modularizar** nuestro código.
- Esta modularidad se logra mediante el uso de archivos de código fuente (.c) y archivos de cabecera (.h):
 1. Archivo de cabecera (`[nombre].h`): contiene las declaraciones de las funciones y variables globales que se van a compartir entre varios archivos de código.
 2. Archivo de código fuente (`[nombre].c`): Contiene las definiciones y la implementación de las funciones y variables.

Módulos

Ejemplo

- Vamos a definir una librería `calculadora`.
- Para ello necesitaremos crear dos ficheros:
 - `calculadora.h`: contendrá los prototipos de las funciones disponibles.
 - `calculadora.c`: contendrá el código de nuestra librería.

Módulos

Ejemplo

- Contenido del fichero `calculadora.h`:

Evita incluir
varias veces
la misma librería

```
#ifndef CALCULADORA_H_  
#define CALCULADORA_H_
```

```
int sumar(int a, int b);  
int restar(int a, int b);  
int multiplicar(int a, int b);  
int dividir(int a, int b);  
int potencia(int a, int b);
```

```
#endif /* CALCULADORA_H_ */
```

Módulos

Ejemplo

- El fichero `calculadora.c` comenzará incluyendo el `.h` de nuestra librería, y a continuación definirá todas las funciones:

```
#include "calculadora.h"
```

```
int sumar(int a, int b) {  
    return a+b;  
}
```

```
int restar(int a, int b) {  
    return a-b;  
}
```

```
int multiplicar(int a, int b) {  
    return a*b;  
}
```

Entre ""
en lugar de <>
porque es una
biblioteca de
Nuestro proyecto

Módulos

Ejemplo

- A partir de aquí, podremos utilizar las funciones de nuestra biblioteca si la incluimos en nuestro código:

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

int main() {
    int res_suma = sumar(5, 2);
    printf("Suma de 5 y 2: %d", res_suma);
    return 0;
}
```

Módulos

Ejemplo

- Para compilar nuestro código sin problema, también es necesario añadir nuestra biblioteca al "target" adecuado en el fichero `CMakeList`:

```
cmake_minimum_required(VERSION 3.28)  
project(ejemplos C)
```

```
set(CMAKE_C_STANDARD 11)
```

```
add_executable(ejemplos main.c calculadora.h calculadora.c)
```


Módulos

Ejemplo

- Si comprobamos la salida de la compilación, (pestaña "Messages" en CLion) veremos como también compila el fichero `calculadora.c`:

Código objeto de nuestra biblioteca

```
=====[ Build | ejemplos | Debug ]=====
"C:\Program Files\JetBrains\CLion 2024.1.4\bin\cmake\win\x64\bin\cmake.exe" --build
[1/3] Building C object CMakeFiles/ejemplos.dir/calculadora.c.obj
[2/3] Building C object CMakeFiles/ejemplos.dir/main.c.obj
[3/3] Linking C executable ejemplos.exe
```

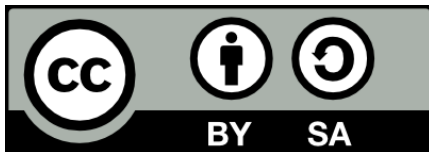
Build finished

El linker carga la biblioteca

Tema 8: Memoria Dinámica

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

- ¿Qué es?
- ¿Por qué?
- Espacio de memoria
- Definición
- Memoria estática vs dinámica
- Funciones para la gestión de memoria
- Arrays dinámicos
- Pasos para crear arrays dinámicos
- Matrices dinámicas

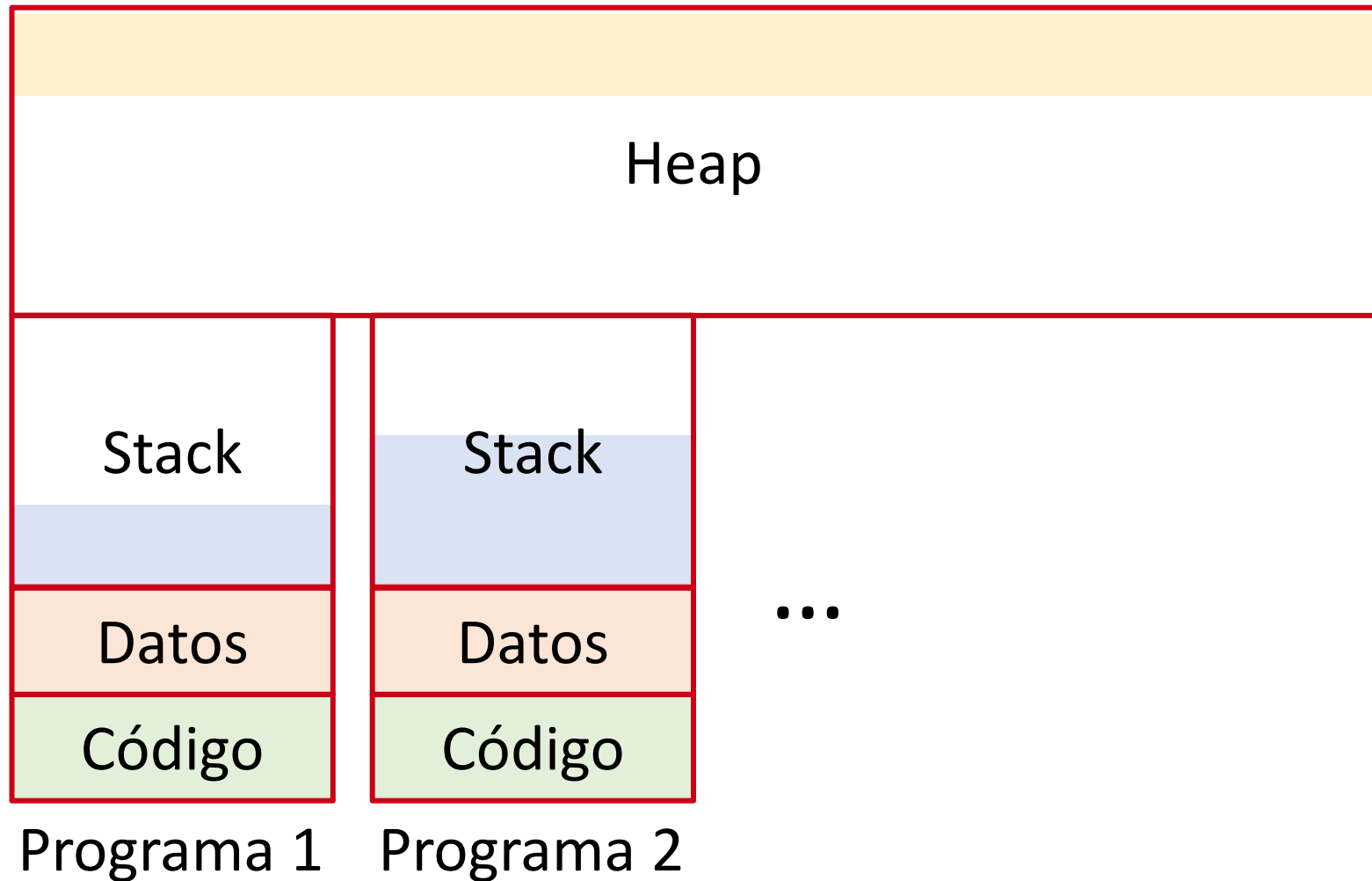
Gestión de memoria dinámica

Espacio de memoria

- La memoria de un programa en C se divide en 4 bloques diferentes de memoria:
 - Segmento de **código**: contiene las instrucciones de nuestro programa. Asignación automática.
 - Segmento de **datos**: contiene variables globales y estáticas, inicializadas y sin inicializar. Asignación automática.
 - **Stack** (pila): contiene las variables locales de las funciones, e información de las llamadas. Asignación automática.
 - **Heap** (montículo): contiene memoria dinámica asignada en tiempo de ejecución. Asignación manual.

Gestión de memoria dinámica

Espacio de memoria



Gestión de memoria dinámica

¿Qué es?

- La **memoria dinámica** es aquella que hace uso del bloque **heap** o montículo.
- La gestión de memoria dinámica permite **reservar** y **liberar** memoria durante la ejecución de un programa.
- Se lleva a cabo mediante el uso de **punteros**.
- Estos punteros permiten referenciar zonas de **memoria dinámica**, de igual manera que los utilizados para referenciar zonas en la memoria estática o pila.
- Las posiciones de memoria reservadas en tiempo de ejecución se conocen como **variables dinámicas**.

Memoria dinámica

¿Por qué?

- Si en tiempo de **compilación** no conocemos el número de datos a almacenar, podemos:
 - Utilizar **memoria estática (pila)**, indicando el tamaño máximo para asegurar que cualquier tamaño será válido, lo que ocasiona un **desperdicio** de memoria.
 - Utilizar arrays de longitud variable en **memoria estática (pila)**, aunque puede no funcionar con todos los compiladores.
 - Utilizar **memoria dinámica**, reservada en tiempo de ejecución.

Memoria dinámica

¿Por qué?

- **El tamaño de la memoria:** la memoria local asignadas a subprogramas se gestiona en la pila (una por proceso) tiene un tamaño limitado, mientras que la memoria dinámica se gestiona en un bloque de gran tamaño.
- **Tiempo de vida de las variables:** utilizando la región stack la gestión de memoria se realiza de manera automática. Sin embargo, al utilizar memoria dinámica el programador decide cuando reservar y liberar memoria.

Memoria dinámica

Diferencias con memoria estática

- **Memoria estática:**
 - Almacenada en el stack (más limitado).
 - Variables disponibles en función de su ámbito.
 - La gestión de memoria automática.
 - Asignación de memoria más rápida
- **Memoria dinámica:**
 - Almacenada en el heap (mayor tamaño).
 - Variables disponibles durante toda la ejecución (hasta que sean liberadas).
 - La gestión de memoria es responsabilidad del programador.
 - Asignación de memoria flexible, que permite redefinir espacios de memoria.

Gestión de memoria dinámica

Funciones para la gestión de memoria

- La gestión de memoria dinámica se realiza mediante **llamadas al sistema** que permiten **reservar, redimensionar y liberar** segmentos de memoria.
- Al trabajar con memoria, estas llamadas devuelven o aceptan como parámetros valores de tipo **puntero**.

Gestión de memoria dinámica

Funciones para la gestión de memoria – `malloc()`

- La función `malloc()` reserva un **bloque de memoria contigua** del tamaño que le pasamos como argumento:

```
void * malloc (int num_bytes)
```

- Donde `num_bytes` es el tamaño del bloque a reservar, medido en bytes.
- Podemos utilizar el operador `sizeof` para calcular el número de bytes necesarios.
- Espacio para `num_elementos` de tipo `int`:

```
num_elementos * sizeof(int)
```

Gestión de memoria dinámica

Funciones para la gestión de memoria – `malloc()`

- La función `malloc()` retorna un **puntero genérico** que apunta a la dirección de **comienzo** del bloque reservado si se ha podido hacer la reserva.
- Devuelve **NULL** si no se ha podido reservar.
- Tenemos que hacer un **casting** al tipo de datos deseado.

Gestión de memoria dinámica

Funciones para la gestión de memoria – malloc()

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

```
int main() {
    int * pl; // Puntero a entero
    pl = (int *) malloc(sizeof(int));

    if(pl == NULL) {
        printf("No se ha podido reservar memoria.\n");
        return -1;
    }

    *pl = 20;
    return 0;
}
```

Stack

Dirección	Valor
0x0000000b871ff6e0	????

Gestión de memoria dinámica

Funciones para la gestión de memoria – malloc()

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

```
int main() {
    int * pl; // Puntero a entero
    pl = (int *) malloc(sizeof(int));

    if(pl == NULL) {
        printf("No se ha podido reservar memoria.\n");
        return -1;
    }

    *pl = 20;
    return 0;
}
```

Stack

Dirección	Valor
0x0000000b871ff6e0	0x000001f6553b1470

Heap

Dirección	Valor
0x000001f6553b1470	????

Gestión de memoria dinámica

Funciones para la gestión de memoria – malloc()

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

```
int main() {
    int * pl; // Puntero a entero
    pl = (int *) malloc(sizeof(int));
```

➔

```
    if(pl == NULL) {
        printf("No se ha podido reservar memoria.\n");
        return -1;
    }
```

```
    *pl = 20;
    return 0;
}
```

Stack

Dirección	Valor
0x0000000b871ff6e0	0x000001f6553b1470

Heap

Dirección	Valor
0x000001f6553b1470	????

Gestión de memoria dinámica

Funciones para la gestión de memoria – malloc()

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

```
int main() {
    int * pl; // Puntero a entero
    pl = (int *) malloc(sizeof(int));
```

```
    if(pl == NULL) {
        printf("No se ha podido reservar memoria.\n");
        return -1;
    }
```

```
    *pl = 20;
    return 0;
```

```
}
```

Stack

Dirección	Valor
0x0000000b871ff6e0	0x000001f6553b1470

Heap

Dirección	Valor
0x000001f6553b1470	20



Gestión de memoria dinámica

Funciones para la gestión de memoria – `calloc()`

- La función `calloc()` reserva un **bloque de memoria contigua** del tamaño especificado en los argumentos:

```
void * calloc (int num_elem, int tam_elem)
```

- Donde:
 - `num_elem` es el número de elementos que queremos reservar.
 - `tam_elem` es el tamaño de cada uno de los elementos.
- **Inicializa** el espacio reservado a cero.
- El valor de retorno es el mismo que en `malloc()`.

Gestión de memoria dinámica

Funciones para la gestión de memoria – `calloc()`

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

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

```
int main() {
```

```
    int * pl; // Puntero a entero
```

```
    pl = (int *) calloc(1, sizeof(int));
```

```
    if(pl == NULL) {
```

```
        printf("No se ha podido reservar memoria.\n");
```

```
        return -1;
```

```
    }
```

```
    *pl = 20;
```

```
    return 0;
```

```
}
```

Gestión de memoria dinámica

Funciones para la gestión de memoria – `realloc()`

- La función `realloc()` **redimensiona** un bloque de memoria reservado previamente con `malloc()` o `calloc()`:

```
void * realloc (void * puntero_anterior, int num_bytes)
```

- **Donde:**
 - `puntero_anterior` es la dirección de memoria inicial del bloque de memoria previamente reservado.
 - `num_bytes` es el nuevo tamaño (en bytes) del bloque.

Gestión de memoria dinámica

Funciones para la gestión de memoria – `realloc()`

- `realloc()` también retorna un puntero genérico.
- En este caso, el puntero retornado puede contener la misma dirección que la pasada como parámetro, una nueva dirección o NULL.
- Puede utilizarse para liberar memoria.

Gestión de memoria dinámica

Funciones para la gestión de memoria – `realloc()`

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

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

```
int main() {
```

```
    int num_elementos = 10;
```

```
    int * pl; // Puntero a entero
```

```
    pl = (int *) calloc(1, sizeof(int)); // Reservamos memoria para un único entero
```

```
    if(pl == NULL) { ... } // Comprobamos que la memoria es válida
```

```
    *pl = 20; // Acceso a la variable dinámica reservada:
```

```
    pl = (int *) realloc(pl, num_elementos * sizeof(int)); // Redimensionamos el bloque
```

```
    if(pl == NULL) { ... } // Comprobamos que el nuevo bloque de memoria es válido
```

```
    // Uso del array...
```

```
    return 0;
```

```
}
```

Gestión de memoria dinámica

Funciones para la gestión de memoria – `free()`

- La función `free()` libera un bloque de memoria previamente reservado de manera dinámica.

```
void free (void * puntero_bloque)
```

- Donde:
 - `puntero_bloque` es la dirección de memoria inicial del bloque a liberar.
- Cualquier bloque reservado **debe ser liberado** en algún momento de nuestro programa.
- `free()` libera la memoria, pero **no cambia** la dirección apuntada por el puntero. Una buena práctica es establecer el puntero a `NULL` después de utilizar esta función.

Gestión de memoria dinámica

Funciones para la gestión de memoria

```
int main() {  
    int num_elementos = 10;  
    int * pl; // Puntero a entero  
  
    pl = (int *) malloc(sizeof(int)); // Reservamos memoria para un único entero  
    if(pl == NULL) { ... } // Comprobamos que la memoria es válida  
  
    // Aquí podríamos utilizar el puntero pl, que apunta a una variable dinámica de tipo int  
  
    // Liberamos memoria y ponemos el puntero a NULL:  
    free(pl);  
    pl = NULL;  
  
    return 0;  
}
```

Gestión de memoria dinámica

Funciones para la gestión de memoria

```
int main() {  
    int num_elementos = 10;  
    int * pl; // Puntero a entero  
  
    pl = (int *) malloc(sizeof(int)); // Reservamos memoria para un único entero  
    if(pl == NULL) { ... } // Comprobamos que la memoria es válida  
  
    // Aquí podríamos utilizar el puntero pl, que apunta a una variable dinámica de tipo int  
  
    // Liberamos memoria y ponemos el puntero a NULL:  
    free(pl);  
    pl = NULL;  
  
    return 0;  
}
```

El puntero ahora apunta a una dirección no válida, por lo que no podemos utilizarlo para acceder a ella

Gestión de memoria dinámica

Arrays dinámicos

- En ciertas ocasiones, podemos **no conocer** el tamaño de los arrays que necesitamos utilizar en tiempo de compilación. En ese caso tenemos las siguientes opciones:
 1. Crear un array del **tamaño máximo** que puede pedir el usuario. Ocasiona un desperdicio de memoria.
 2. Utilizar un **array de tamaño variable (VLA)**. No disponibles en todas las versiones de C.
 3. Crear un array en **memoria dinámica** tras saber cuál es el tamaño.

Gestión de memoria dinámica

Arrays dinámicos

- Para declarar y utilizar un array en memoria debemos seguirlos siguientes pasos:
 1. Declarar un puntero al tipo de datos del array.
 2. Reservar un bloque de memoria con `malloc()` o `calloc()`.
 3. Comprobar que el bloque de memoria se ha reservado correctamente.
 4. Utilizar el array como hasta ahora (con indexación o aritmética de punteros).
 5. Liberar la memoria con `free()`.

Gestión de memoria dinámica

Arrays dinámicos

```
int main() {
    int num_elementos; int * p; float media = 0;
    scanf("%d", &num_elementos);

    p = malloc(num_elementos * sizeof(int));
    if(p == NULL) {
        printf("No se ha podido asignar memoria. Finalizando ejecución\n");
        return -1;
    }

    for(int i = 0; i < num_elementos; i++) {
        scanf("%d", (p+i));
    }

    free(p);
    p = NULL;
    return 0;
}
```

Gestión de memoria dinámica

Arrays dinámicos

```
int main() {  
    int num_elementos; int * p; float media = 0;  
    scanf("%d", &num_elementos);
```

1. Declaración del puntero

```
    p = malloc(num_elementos * sizeof(int));  
    if(p == NULL) {  
        printf("No se ha podido asignar memoria. Finalizando ejecución\n");  
        return -1;  
    }
```

```
    for(int i = 0; i < num_elementos; i++) {  
        scanf("%d", (p+i));  
    }
```

```
    free(p);  
    p = NULL;  
    return 0;  
}
```

Gestión de memoria dinámica

Arrays dinámicos

```
int main() {  
    int num_elementos; int * p; float media = 0;  
    scanf("%d", &num_elementos);  
  
    p = malloc(num_elementos * sizeof(int));  
    if(p == NULL) {  
        printf("No se ha podido asignar memoria. Finalizando ejecución\n");  
        return -1;  
    }  
  
    for(int i = 0; i < num_elementos; i++) {  
        scanf("%d", (p+i));  
    }  
  
    free(p);  
    p = NULL;  
    return 0;  
}
```

1. Declaración del puntero

2. Reserva de memoria

Gestión de memoria dinámica

Arrays dinámicos

```
int main() {  
    int num_elementos; int * p; float media = 0;  
    scanf("%d", &num_elementos);
```

1. Declaración del puntero

```
    p = malloc(num_elementos * sizeof(int));
```

2. Reserva de memoria

```
    if(p == NULL) {  
        printf("No se ha podido asignar memoria. Finalizando ejecución\n");  
        return -1;  
    }
```

3. Comprobación de la reserva

```
    for(int i = 0; i < num_elementos; i++) {  
        scanf("%d", (p+i));  
    }
```

```
    free(p);  
    p = NULL;  
    return 0;  
}
```

Gestión de memoria dinámica

Arrays dinámicos

```
int main() {  
    int num_elementos; int * p; float media = 0;  
    scanf("%d", &num_elementos);
```

1. Declaración del puntero

```
p = malloc(num_elementos * sizeof(int));
```

2. Reserva de memoria

```
if(p == NULL) {  
    printf("No se ha podido asignar memoria. Finalizando ejecución\n");  
    return -1;  
}
```

3. Comprobación de la reserva

```
for(int i = 0; i < num_elementos; i++) {  
    scanf("%d", (p+i));  
}
```

4. Uso del array (con aritmética o indexación)

```
free(p);  
p = NULL;  
return 0;
```

Gestión de memoria dinámica

Arrays dinámicos

```
int main() {  
    int num_elementos; int * p; float media = 0;  
    scanf("%d", &num_elementos);
```

1. Declaración del puntero

```
    p = malloc(num_elementos * sizeof(int));
```

2. Reserva de memoria

```
    if(p == NULL) {  
        printf("No se ha podido asignar memoria. Finalizando ejecución\n");  
        return -1;  
    }
```

3. Comprobación de la reserva

```
    for(int i = 0; i < num_elementos; i++) {  
        scanf("%d", (p+i));  
    }
```

4. Uso del array (con aritmética o indexación)

```
    free(p);  
    p = NULL;  
    return 0;
```

5. Liberación de la memoria

Gestión de memoria dinámica

Arrays bidimensionales dinámicos

- De igual manera, también es posible declarar un array bidimensional en memoria dinámica.
- En este caso, el array bidimensional será realmente un puntero a puntero. Los pasos a seguir son:
 1. Declarar un puntero a puntero al tipo de dato de la matriz.
 2. Reservar memoria para las direcciones de cada fila (array de punteros).
 3. Reservar memoria para cada fila de la matriz.

Gestión de memoria dinámica

Matrices dinámicas

➔ `int ** matriz;`

`int num_filas = 3;`

`int num_cols = 2;`

`matriz = (int **) malloc(num_filas * sizeof(int *));`

`for (int i = 0; i < num_filas; i++) {`

`matriz[i] = (int *) malloc((num_cols) * sizeof(int));`

`}`

Gestión de memoria dinámica

Matrices dinámicas

→ `int ** matriz;`

`int num_filas = 3;`

`int num_cols = 2;`

`matriz = (int **) malloc(num_filas * sizeof(int *));`

`for (int i = 0; i < num_filas; i++) {`

`matriz[i] = (int *) malloc((num_cols) * sizeof(int));`

`}`

?????

m

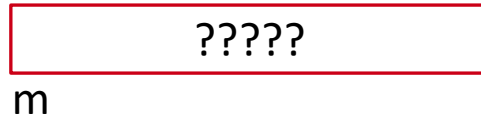
Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;  
int num_filas = 3;  
int num_cols = 2;
```

➔ `matriz = (int **) malloc(num_filas * sizeof(int *));`

```
for (int i = 0; i < num_filas; i++) {  
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));  
}
```



Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;  
int num_filas = 3;  
int num_cols = 2;
```

➔ `matriz = (int **) malloc(num_filas * sizeof(int *));`

```
for (int i = 0; i < num_filas; i++) {  
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));  
}
```

0x000001a934281470

m

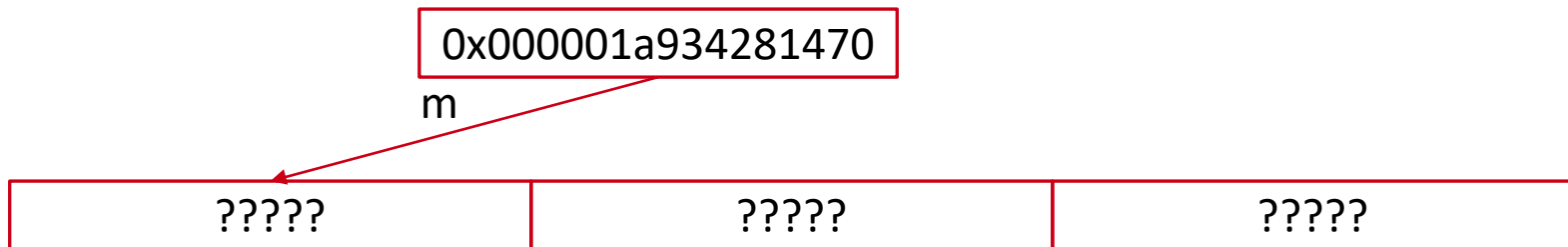
Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;  
int num_filas = 3;  
int num_cols = 2;
```

➔ `matriz = (int **) malloc(num_filas * sizeof(int *));`

```
for (int i = 0; i < num_filas; i++) {  
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));  
}
```



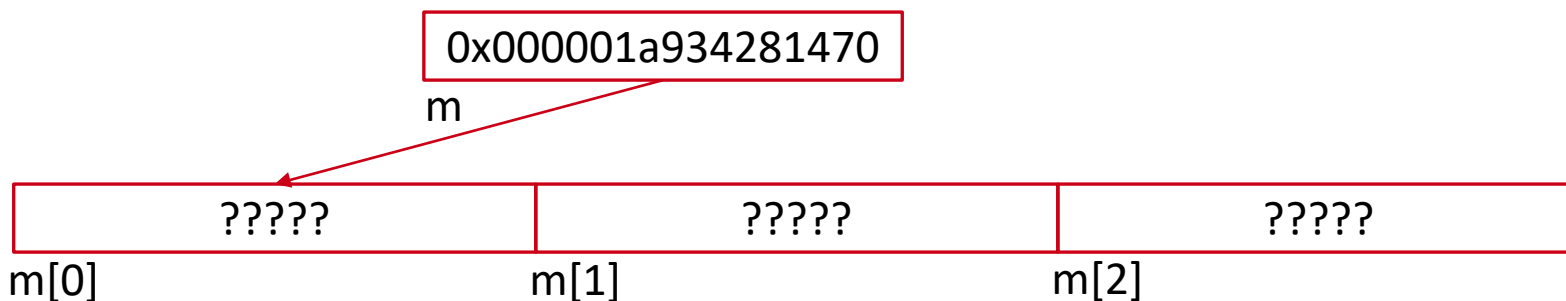
Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;  
int num_filas = 3;  
int num_cols = 2;
```

➔ `matriz = (int **) malloc(num_filas * sizeof(int *));`

```
for (int i = 0; i < num_filas; i++) {  
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));  
}
```



Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;
```

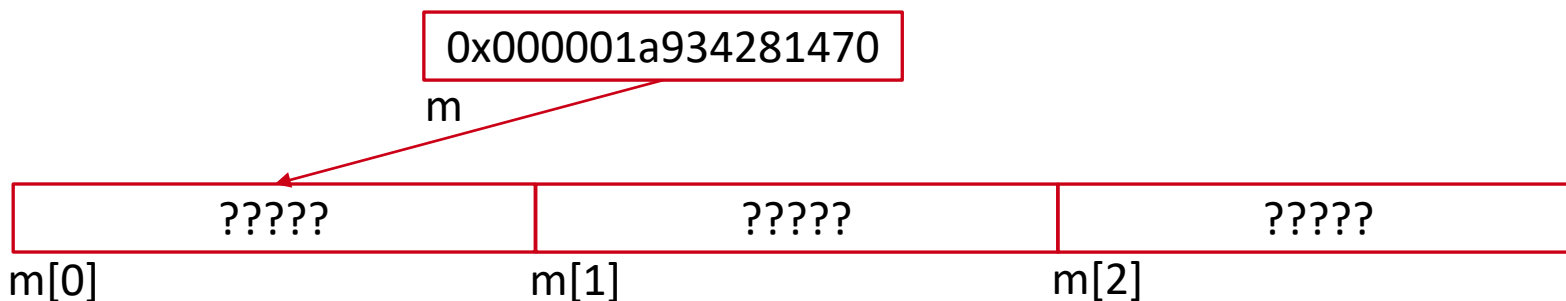
```
int num_filas = 3;
```

```
int num_cols = 2;
```

```
matriz = (int **) malloc(num_filas * sizeof(int *));
```

```
for (int i = 0; i < num_filas; i++) {
```

```
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));  
}
```



Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;
```

```
int num_filas = 3;
```

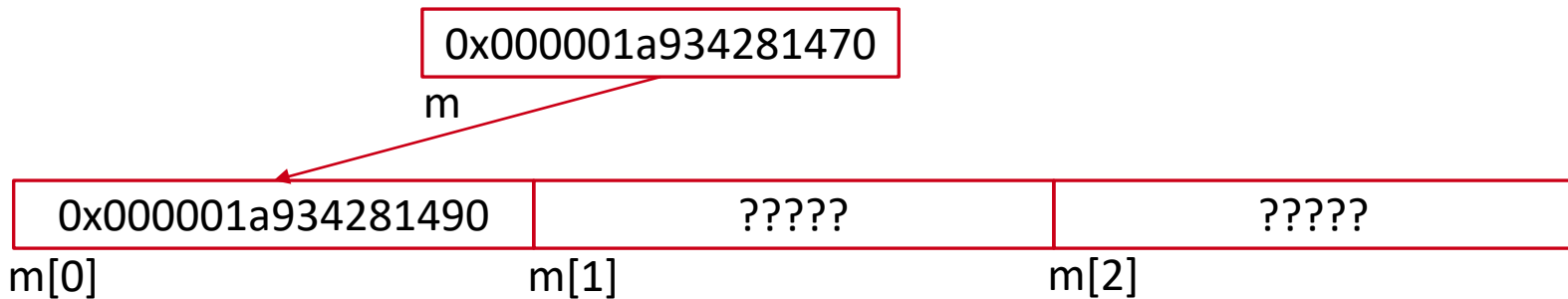
```
int num_cols = 2;
```

```
matriz = (int **) malloc(num_filas * sizeof(int *));
```

```
for (int i = 0; i < num_filas; i++) {
```

```
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));
```

```
}
```



Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;
```

```
int num_filas = 3;
```

```
int num_cols = 2;
```

```
matriz = (int **) malloc(num_filas * sizeof(int *));
```

```
for (int i = 0; i < num_filas; i++) {
```

```
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));
```

```
}
```

0x000001a934281470

m

0x000001a934281490

?????

?????

m[0]

m[1]

m[2]

??

??

m[0][0] m[0][1]

Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;
```

```
int num_filas = 3;
```

```
int num_cols = 2;
```

```
matriz = (int **) malloc(num_filas * sizeof(int *));
```

```
for (int i = 0; i < num_filas; i++) {
```

```
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));
```

```
}
```

0x000001a934281470

m

0x000001a934281490

?????

?????

m[0]

m[1]

m[2]

??

??

m[0][0] m[0][1]

Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;
```

```
int num_filas = 3;
```

```
int num_cols = 2;
```

```
matriz = (int **) malloc(num_filas * sizeof(int *));
```

```
for (int i = 0; i < num_filas; i++) {
```

```
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));
```

```
}
```

0x000001a934281470

m

0x000001a934281490

m[0]

0x000001a9342814b0

m[1]

?????

m[2]

??

??

m[0][0] m[0][1]

Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;
```

```
int num_filas = 3;
```

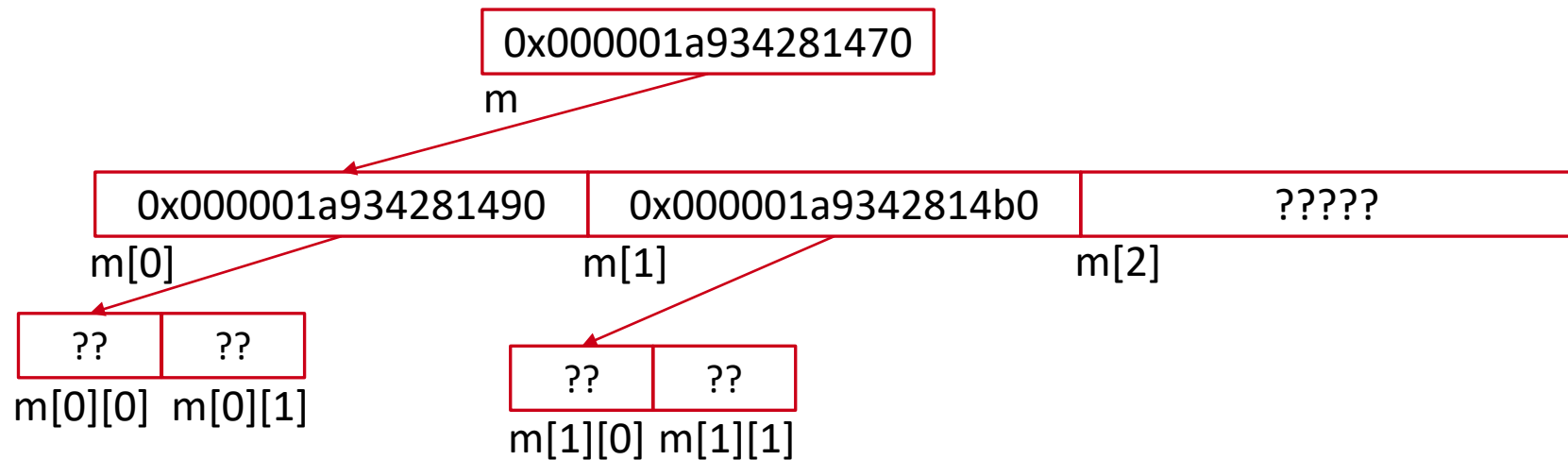
```
int num_cols = 2;
```

```
matriz = (int **) malloc(num_filas * sizeof(int *));
```

```
for (int i = 0; i < num_filas; i++) {
```

```
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));
```

```
}
```



Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;
```

```
int num_filas = 3;
```

```
int num_cols = 2;
```

```
matriz = (int **) malloc(num_filas * sizeof(int *));
```

```
for (int i = 0; i < num_filas; i++) {
```

```
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));
```

```
}
```

0x000001a934281470

m

0x000001a934281490

m[0]

0x000001a9342814b0

m[1]

0x000001a9342814d0

m[2]

?? ??

m[0][0] m[0][1]

?? ??

m[1][0] m[1][1]

Gestión de memoria dinámica

Matrices dinámicas

```
int ** matriz;
```

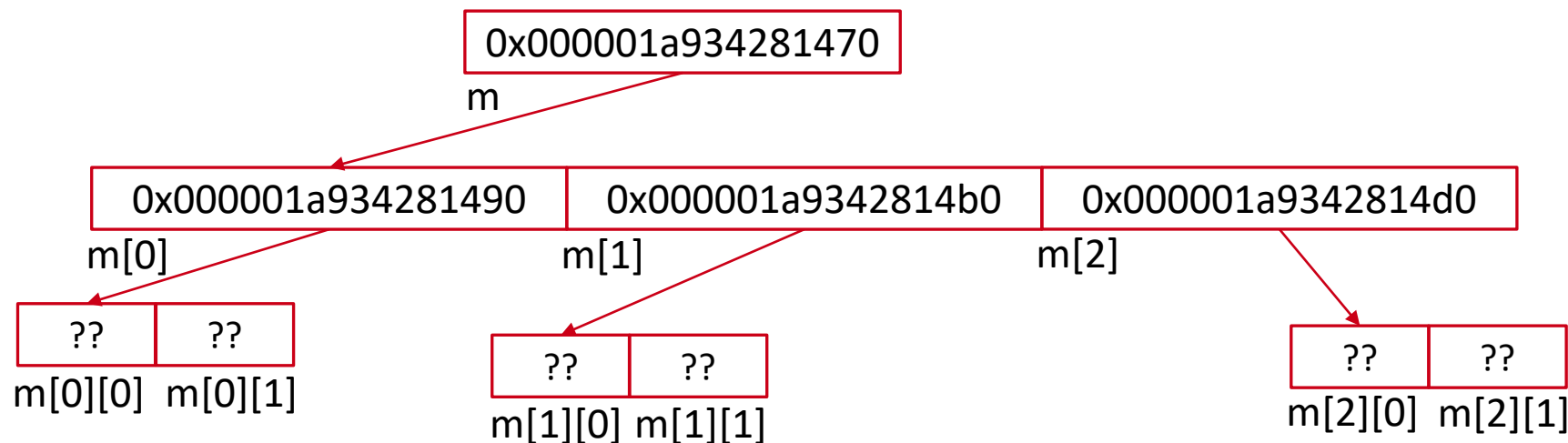
```
int num_filas = 3;
```

```
int num_cols = 2;
```

```
matriz = (int **) malloc(num_filas * sizeof(int *));
```

```
for (int i = 0; i < num_filas; i++) {
```

```
    matriz[i] = (int *) malloc((num_cols) * sizeof(int));  
}
```



Gestión de memoria dinámica

Matrices dinámicas

- Una vez reservada, podemos trabajar con el array bidimensional como hasta ahora:

Gestión de memoria dinámica

Matrices dinámicas

- Una vez reservada, podemos trabajar con el array bidimensional como hasta ahora:

Utilizando indexación:

```
for(int i = 0; i < num_filas; i++)  
{  
    for(int j = 0; j < i + 1; j++)  
    {  
        matriz[i][j] = i * num_filas + j;  
    }  
}
```

Gestión de memoria dinámica

Matrices dinámicas

- Una vez reservada, podemos trabajar con el array bidimensional como hasta ahora:

Utilizando indexación:

```
for(int i = 0; i < num_filas; i++)
{
    for(int j = 0; j < i + 1; j++)
    {
        matriz[i][j] = i * num_filas + j;
    }
}
```

Utilizando aritmética de punteros:

```
for(int i = 0; i < num_filas; i++)
{
    for(int j = 0; j < i + 1; j++)
    {
        (*(matriz + i) + j) = i * num_filas + j;
    }
}
```

Gestión de memoria dinámica

Matrices dinámicas

- Para liberar un array bidimensional en memoria dinámica, debemos liberar primero cada fila (direcciones del array de punteros) y por último la propia matriz (puntero a puntero), en orden inverso a la reserva:

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

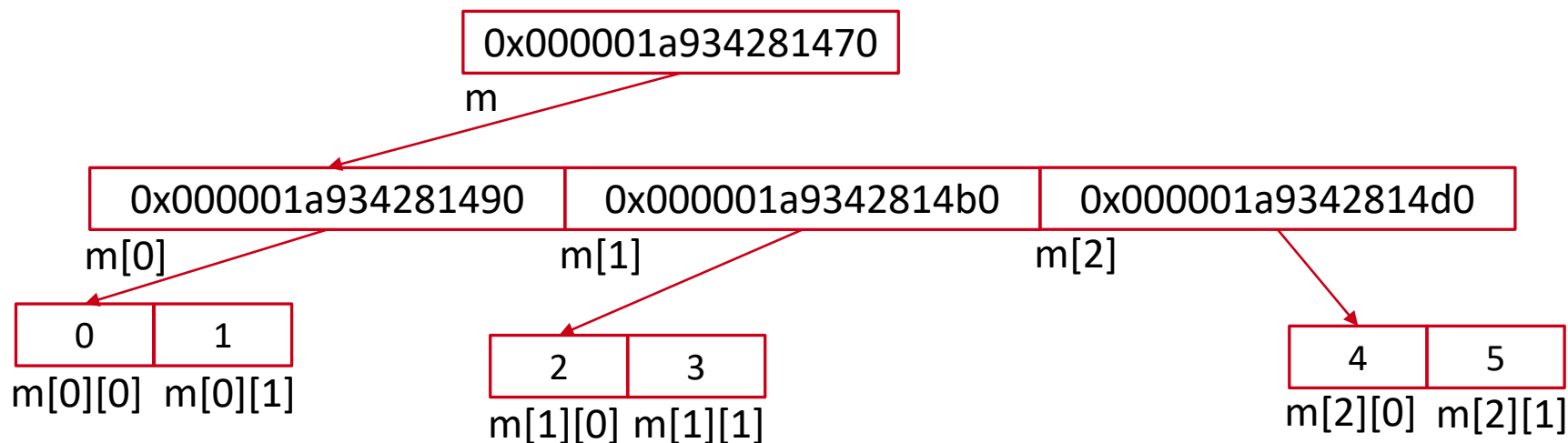
```
free(matriz);
```

Gestión de memoria dinámica

Matrices dinámicas

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

```
free(matriz);  
matriz = NULL;
```

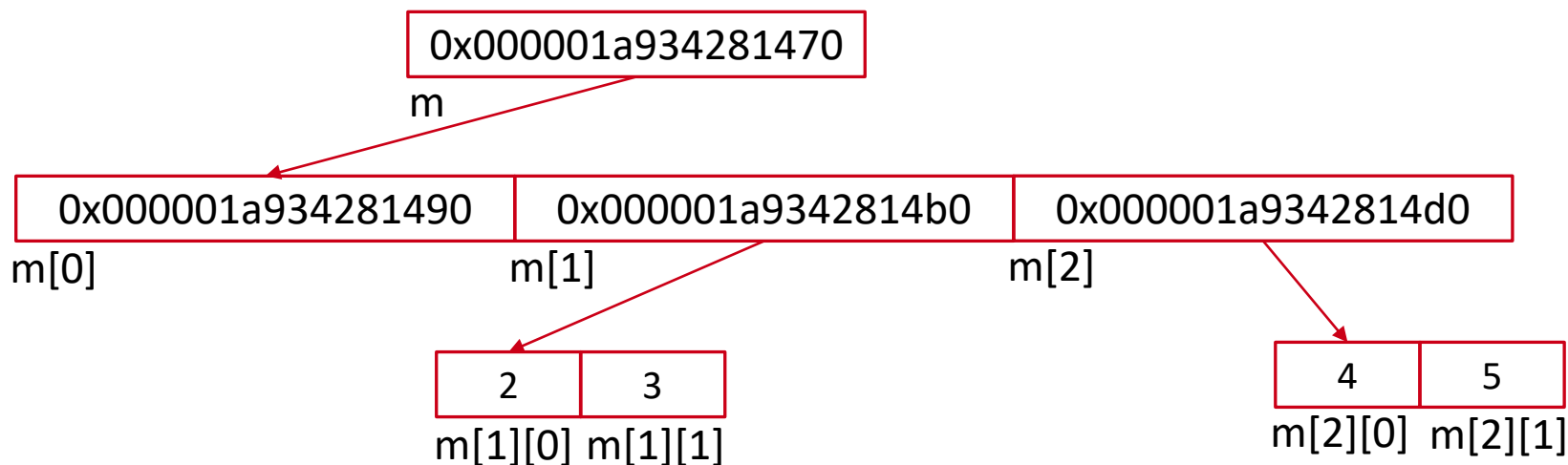


Gestión de memoria dinámica

Matrices dinámicas

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

```
free(matriz);  
matriz = NULL;
```

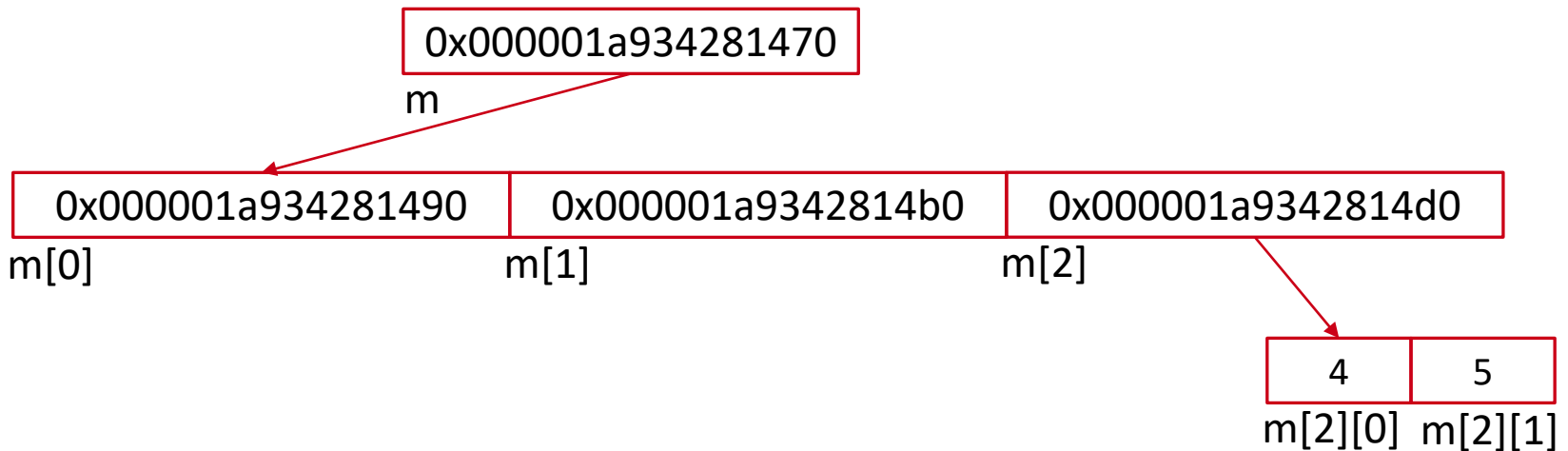


Gestión de memoria dinámica

Matrices dinámicas

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

```
free(matriz);  
matriz = NULL;
```

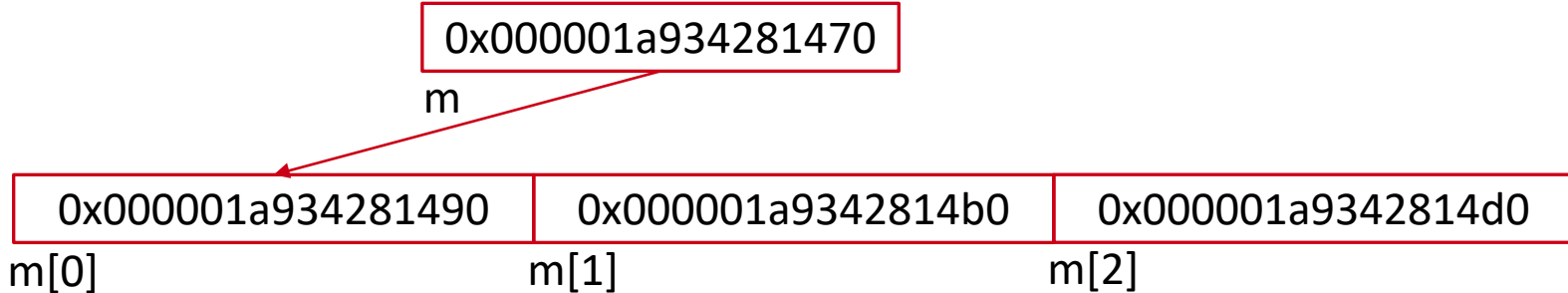


Gestión de memoria dinámica

Matrices dinámicas

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

```
free(matriz);  
matriz = NULL;
```

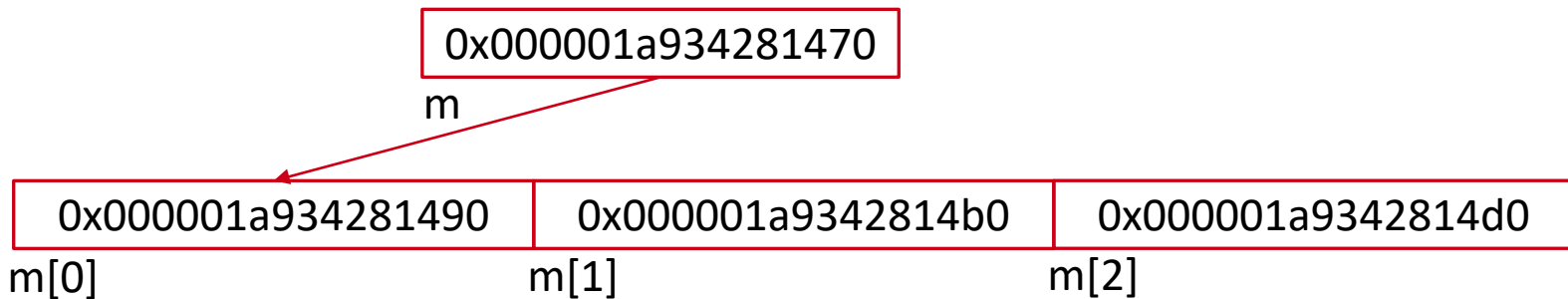


Gestión de memoria dinámica

Matrices dinámicas

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

➔ free(matriz);
matriz = **NULL**;



Gestión de memoria dinámica

Matrices dinámicas

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

➔ free(matriz);
matriz = **NULL**;

0x000001a934281470

m

Gestión de memoria dinámica

Matrices dinámicas

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

```
free(matriz);  
→ matriz = NULL;
```

0x000001a934281470

m

Gestión de memoria dinámica

Matrices dinámicas

```
for (int i=0;i<num_filas;i++) {  
    free(matriz[i]);  
}
```

```
free(matriz);  
→ matriz = NULL;
```



Gestión de memoria dinámica

Matrices dinámicas

- El uso de memoria dinámica permite reservar matrices donde cada fila tiene un número de elementos diferente, por ejemplo, una matriz triangular inferior:

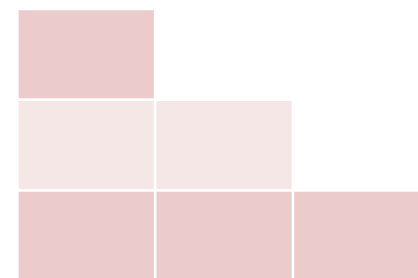
```
int ** matriz;  
int num_filas = 3;  
  
matriz = (int **) malloc(num_filas * sizeof(int *));  
  
for (int i=0; i<num_filas; i++) {  
    matriz[i] = (int *) malloc((i + 1) * sizeof(int));  
}
```

Gestión de memoria dinámica

Matrices dinámicas

- El uso de memoria dinámica permite reservar matrices donde cada fila tiene un número de elementos diferente, por ejemplo, una matriz triangular inferior:

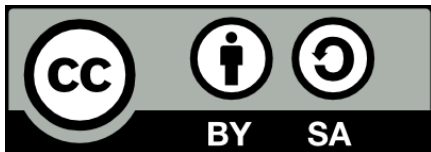
```
int ** matriz;  
int num_filas = 3;  
  
matriz = (int **) malloc(num_filas * sizeof(int *));  
  
for (int i=0; i<num_filas; i++) {  
    matriz[i] = (int *) malloc((i + 1) * sizeof(int));  
}
```



Tema 9: Estructuras y Tipos de Datos Enumerados

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

- Estructuras:
 - ¿Para qué se usan?
 - ¿Qué son?
 - Definición.
 - Sintaxis.
 - Declaración de variables.
- Manejo de estructuras:
 - Acceso a miembros.
 - Operador ->
 - Copia.
 - Struct en funciones.
 - Struct en arrays.
- Typedef:
 - ¿Qué es?
 - Uso.
 - Omisión.
- Tipos enumerados (enum):
 - ¿Qué son?
 - Declaración.

Estructuras

¿Para qué se usan?

- En muchos problemas no es posible representar la información con un solo tipo de datos, sino que necesitamos **combinar varios**.
- En este caso podemos utilizar **estructuras** para representar y organizar datos complejos.
- Las estructuras permiten una organización de datos más lógica y fácil de gestionar.

Estructuras

¿Para qué se usan?

- **Ejemplo 1:** en una aplicación que maneja información de estudiantes, podemos agrupar los atributos de un estudiante, como su *nombre* (`char [50]`), *edad* (`short`) y *calificación* (`float`), en una sola estructura en lugar de manejar variables individuales.
- **Ejemplo 2:** el historial médico de un paciente se compone de atributos de diferentes tipos: *nombre* (`char[100]`), *edad* (`unsigned short`), *peso* (`float`), *NSS* (`unsigned long`) e *historial* (`char[1000]`).

Estructuras

¿Para qué se usan?

- Para los anteriores ejemplos, necesitaríamos las siguientes variables individuales:
 - Ejemplo 1: **tres variables** por cada estudiante (una por cada campo o atributo).
 - Ejemplo 2: **cinco variables** por cada paciente (una por cada campo o atributo).
- Además, **no podemos utilizar un array porque son tipos de datos diferentes.**

Estructuras

¿Qué son?

- Una **estructura** es una colección de variables, llamadas miembros, que pueden ser de diferentes tipos de datos (**datos compuestos**).
- Estas **variables se agrupan bajo un solo nombre**, y cada miembro de la estructura se puede acceder de forma individual.
- Las estructuras permiten trabajar con datos relacionados de una manera unificada, **similar a cómo funcionan los registros en bases de datos**.

Estructuras

¿Qué son?

- Tipo de datos **compuesto**:
 - En C se denomina **struct** pero en otros lenguajes se llama registro (**record**).
- A diferencia de un array, los datos almacenados en una **struct** pueden ser de **tipos diferentes**.
- Se trata de un tipo de datos **definido por el programador**.

Estructuras

Definición

- En la definición de una ***struct*** se establecen los tipos de datos y los nombres de cada miembro que contiene la estructura.

Estructuras

Definición

- En la definición de una `struct` se establecen los tipos de datos y los nombres de cada miembro que contiene la estructura.
- Para declarar una variable `struct`, tenemos primero que definir que se trata de ese tipo de dato compuesto:

```
struct nombre_estructura
```

- Posteriormente, tenemos que definir entre llaves qué datos la va a componer:

```
{ tipo_dato nombre_dato; ... }
```

Estructuras

Sintaxis

- Sólo vamos a definir el tipo de dato, **no declaramos ninguna variable ni reservamos memoria**, por lo que es necesario declarar antes variables de este tipo.
- La sintaxis básica para definir una estructura es la siguiente:
 - Palabra clave `struct` e identificador de la estructura.
 - Cada campo (incluyendo su tipo de dato y su identificador).

```
struct identificador_estructura{  
    tipo_dato campo1;  
    tipo_dato campo2;  
    tipo_dato campo3;  
};
```

Estructuras

Declaración de variables

- Una vez definida la estructura, podremos **declarar** variables de ese tipo.
- Con la declaración, el compilador **reserva espacio** en memoria para almacenar una variable del tipo de la estructura:
 - Reserva memoria para **cada uno de los campos**.
- La declaración puede hacerse en el momento de la definición de la estructura o después de ésta.

Estructuras

Declaración de variables

- Declaración **al definir la estructura:**

```
struct estudiante {  
    char nombre[50];  
    int edad;  
    float calificacion;  
} estudiante1, estudiante2;
```

- En este caso, *estudiante1* y *estudiante2* son variables del tipo *struct Estudiante* que se declaran al mismo tiempo que la definición de la estructura.

Estructuras

Declaración de variables

- Declaración **posterior**:
- Una vez definida la estructura, podemos declarar variables de ese tipo:

```
struct estudiante estudiante1, estudiante2;
```

- Posteriormente, para **acceder** a los campos, utilizaremos el operador punto ".":

```
variable_struct.nombre_campo;
```

- Por ejemplo:

```
estudiante1.nombre;
```

Estructuras

Declaración de variables (ejemplo completo)

```
#include <stdio.h>
#include <string.h>
// Definición de la estructura
struct estudiante {
    char nombre[50];
    int edad;
    float calificacion;
};
int main() {
    // Declaración de una variable de tipo struct estudiante
    struct estudiante est;

    strcpy(est.nombre, "Carlos");
    est.edad = 20;
    est.calificacion = 8.5;

    printf("Estudiante %s. Edad: %d. Calificacion: %f", est.nombre, est.edad, est.calificacion);

    return 0;
}
```

typedef

¿Qué es?

- **typedef** permite crear alias o nombres alternativos para tipos de datos.
- Facilita el manejo de tipos complejos y largos, permitiendo nombrar un tipo de datos con una etiqueta más simple.
- Así, en lugar de repetir una definición compleja, podemos utilizar el alias definido para mejorar la legibilidad y simplicidad del código.

typedef

¿Qué es?

- Utilizando `typedef` podemos **renombrar** un tipo de datos definido en C, ya sea un tipo predefinido (`int`, `float`) o uno definido por el programador (`struct`).
- ¡Importante! `typedef` no define un tipo, sino que proporciona un **sinónimo** para uno ya existente.

typedef

Uso

- `typedef` se utiliza para:
 - **Hacer el código más legible:** al usar alias para tipos complejos, es más fácil entender el propósito de ciertas variables.
 - **Reducir errores:** evitar repetir definiciones complejas, reduce las posibilidades de errores tipográficos y facilita cambios futuros.
 - **Simplificar la declaración de punteros:** en lugar de escribir `int *`, podemos crear un tipo llamado `intpointer` y usarlo en su lugar.

typedef

Uso

- Debe aparecer siempre **antes** de cualquier variable del tipo redefinida.
- Se suele escribir fuera del `main`, justo antes de los prototipos de las funciones, lo que permite utilizarlo en los prototipos.
- Se recomienda que el nombre comience por `type`, `tipo`, `t` o similar para clarificar que se trata de un tipo de datos, o por `st` para indicar que se trata de una estructura.

typedef

Uso - Ejemplo para redefinir una variable

- Redefinir el tipo de las variables que se refieren a una edad.
 - Podemos declarar una edad utilizando el tipo `unsigned short`:

```
//redefinimos y creamos tipo_edad  
typedef unsigned short tipo_edad;
```

- Ahora usamos `tipo_edad`:

```
// usamos el tipo de dato con alias:  
tipo_edad edad;
```


typedef

Uso - Ejemplo para renombrar una estructura

- Usualmente, se utiliza para renombrar una struct:

```
// Definición de la estructura
typedef struct t_paciente{
    char nombre[100];
    unsigned short edad;
    float peso;
} st_paciente;

int main() {
    st_paciente juan;
    juan.peso = 100;
}
```

typedef

Omisión

- Cuando usamos `typedef` al definir una estructura es posible **omitir** el nombre de la estructura.
- Esto simplifica el código, especialmente cuando no necesitamos utilizar el nombre original de la estructura más adelante, y solo requerimos un alias para facilitar su uso.

typedef

Omisión – Ejemplo

Omitimos el nombre de la estructura:

```
#include <stdio.h>
#include <string.h>
typedef struct {
    char nombre[100];
    unsigned short edad;
    float peso;
    unsigned long matricula;
} st_estudiante;
```

Utilización del tipo definido previamente:

```
int main() {
    // Declaración e inicialización de una
    // variable de tipo st_estudiante
    st_estudiante estudiante;
    // Asignación de valores a los
    // campos de la estructura
    strcpy(estudiante.nombre, "Ana X");
    estudiante.edad = 21;
    estudiante.peso = 60.5;
    estudiante.matricula = 987654321;
    //...
    return 0;
}
```

Manejo de estructuras

Acceso a miembros de una estructura

- El operador punto "." se utiliza para acceder a los miembros (campos) de una estructura.
- Este operador permite acceder y modificar los campos de una estructura usando el nombre de la variable estructurada y el nombre del campo.
- Sintaxis:

`variable_struct.nombre_campo`

Manejo de estructuras

Acceso a miembros de una estructura - Ejemplo

```
typedef struct {  
    char nombre[50];  
    int edad;  
    float peso;  
} st_persona;  
  
int main() {  
  
    st_persona persona1;  
    persona1.edad = 30; // Acceso a miembro edad  
    //...  
    return 0;  
}
```

Manejo de estructuras

El operador →

- El operador flecha (→) permite acceder a los campos de una variable estructura a partir de un puntero a la propia estructura.
- Este operador permite simplificar el acceso a los miembros sin necesidad de desreferenciar el puntero manualmente:

```
st_persona persona;  
st_persona * pPersona = &persona;  
(*pPersona).edad;  
pPersona->edad;
```

- **Las últimas dos instrucciones son equivalentes.**

Manejo de estructuras

Copia de estructuras

- Es posible **copiar** el contenido de dos estructuras con el operador **asignación (=)**.

```
st_persona persona1; // Struct de tipo st_persona
st_persona persona2; // Struct de tipo st_persona
persona2 = persona1; // Copiamos todos los
campos de persona1 a persona2
```

Manejo de estructuras

Copia de estructuras – cuidado con campos tipo puntero

- Cada campo de la estructura original se copia directamente en la nueva estructura.
- **¡OJO!** si alguno de los campos de la estructura es un puntero, el resultado puede no ser el deseado.
- Cuando copiamos el contenido de una estructura a otra usando el operador de asignación (=), se realiza una copia superficial (*shallow copy*).

Manejo de estructuras

Copia de estructuras – cuidado con campos tipo puntero

- Si un campo es un puntero, sólo se copia la dirección de memoria, no el contenido al que apunta.
- Ambas estructuras terminarán apuntando al mismo bloque de memoria para el campo puntero.
- Cualquier cambio que se haga en el contenido de ese puntero en una estructura se reflejará en la otra, ya que comparten la misma dirección de memoria.

Manejo de estructuras

Copia de estructuras – cuidado con campos tipo puntero - solución

- Para evitar este problema, necesitamos hacer una copia profunda (**deep copy**) del contenido al que apunta el puntero.
- Esto requiere reservar memoria para el campo puntero de la segunda instancia y entonces copiar el contenido del campo de la primera instancia a esa nueva región de memoria:

//Solución con "copia profunda"

```
st_persona persona2;
```

```
persona2.nombre = (char *)malloc(50 * sizeof(char)); // Reservamos  
memoria para el nuevo nombre
```

```
strcpy(persona2.nombre, persona1.nombre); // Copiar el contenido
```

```
persona2.edad = persona1.edad; // Copiar el resto de los campos
```

- En este caso, *persona1.nombre* y *persona2.nombre* apuntarán a bloques de memoria independientes, lo que evitará el problema de la copia superficial.

Manejo de estructuras

structs en funciones

- Como cualquier otro tipo de datos, las estructuras pueden utilizarse como **argumento** o **tipo de retorno** en funciones.
- Las estructuras pueden ser pasadas a funciones por valor o por referencia.
- Al **pasarlas por valor**, se crea una copia de la estructura.
- Al **pasarlas por referencia** (usando punteros), la función puede modificar los datos originales.

Manejo de estructuras

structs en funciones

- Se recomienda **no pasar** como parámetros copias de estructuras.
- Pueden ser eventualmente **grandes** y desperdiciar **memoria**.

Ejemplo de paso por referencia:

// Definición de la función:

```
void modificar_edad(st_persona *p, int nueva_edad) {  
    p->edad = nueva_edad;  
}
```

// Invocación de la función:

```
modificar_edad(&persona1, 32);
```

Manejo de estructuras

structs en array

- Es posible crear **arrays de estructuras**, igual que de cualquier otro tipo primitivo.

```
st_estudiante clase[70];
```

En este caso, estamos creando un arreglo de 70 elementos del `st_estudiante`.

- También podemos crear **arrays dinámicos**:

```
st_estudiante * clase;
```

```
clase = (st_estudiante *) malloc(70 * sizeof(st_estudiante));
```

Manejo de estructuras

structs en array - ejemplo

- Ejemplo de la creación de un arreglo de 3 elementos de la estructura `st_persona` y la asignación de valores al campo `edad`:

```
st_persona grupo[3];  
grupo[0].edad = 20;  
grupo[1].edad = 25;  
grupo[2].edad = 30;
```

- Al tratarse de un array, también sería posible acceder a sus elementos utilizando aritmética de punteros.

Tipos enumerados (*enum*)

¿Qué son?

- Los tipos enumerados, también conocidos como `enum`, son un tipo de datos definido por el programador.
- Su rango de valores son una serie de constantes simbólicas con valores enteros:
 - Comienzan a numerarse en 0.
- Sintaxis:

```
enum identificador_enum {  
    VALOR1, // toma valor 0  
    VALOR2, // toma valor 1  
    VALOR3, // toma valor 2  
    // ...  
};
```

Tipos enumerados

¿Qué son?

- Por ejemplo, los días de la semana o los meses del año son conjuntos de valores bien definidos que se pueden representar mediante tipos enumerados:

```
enum meses {  
    ENERO,    // toma valor 0  
    FEBRERO,  // toma valor 1  
    MARZO     // toma valor 2  
};
```


Tipos enumerados

¿Qué son?

- También podemos asignar otros valores específicos modificando el valor de cada elemento:

```
enum meses {  
    ENERO = 1,    // toma valor 1  
    FEBRERO,     // toma valor 2  
    MARZO        // toma valor 3  
};
```

En este caso, *ENERO* comienza con el valor 1, y el resto de los meses se incrementa en 1 automáticamente.

Tipos enumerados

Declaración

- Una vez definido el tipo enumerado, podemos declarar una variable enumerada utilizando la siguiente sintaxis:

```
enum nombre_enumeracion nombre_variable;
```

- También podemos utilizar `typedef` con `enum`:

```
typedef enum nombre_enumeracion nuevo_nombre;
```

- Cuando utilicemos las variables de este tipo, estas únicamente podrán tomar el valor del entero asociado.

Tipos enumerados

Declaración – ejemplo

- Una vez declarado un tipo enumerado, se puede usar para declarar variables que solo pueden contener uno de los valores definidos en el `enum`:

```
typedef enum {  
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO  
} e_dias_semana;
```

```
int main() {  
    e_dias_semana hoy = MIERCOLES;  
    if (hoy == MIERCOLES) {  
        printf("Hoy es miércoles.\n");  
    }  
    return 0;  
}
```

- Es posible utilizar `typedef` y omitir el nombre de la enumeración, como con las `struct`.

Tema 10: Ficheros

INTRODUCCIÓN A LA PROGRAMACIÓN

2024/2025



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en <https://creativecommons.org/licenses/by-sa/4.0/deed.es>.



Universidad
Rey Juan Carlos

- Introducción.
- Ficheros.
- Apertura de ficheros.
- Cierre de ficheros.
- Ficheros de texto.
- Ficheros binarios.

Introducción

Persistencia

- Hasta ahora hemos utilizado la entrada y salida estándar (con los flujos `stdin` y `stdout`).
- Los datos **no eran persistentes**, se perdían al finalizar el programa.

Los **ficheros** nos permiten almacenar datos de manera persistente en el sistema, lo cual es crucial para guardar información que se requiere después de que el programa haya terminado su ejecución.

Introducción

Persistencia

- A diferencia de las variables, cuyos valores se pierden al finalizar el programa, los datos en un fichero permanecen hasta que se eliminen o se sobrescriban de manera intencional.
- Los ficheros se utilizan ampliamente para registrar información, como bases de datos, configuraciones y registros de actividad de los programas.
- Los ficheros permiten al software ser más funcional y útil.

Introducción

Problemas de la I/O estándar

- Además, en muchas aplicaciones, los datos no se pueden pedir por teclado, porque:
 - Son **grandes** cantidades de datos.
 - Los datos están **almacenados** en ficheros externos al programa.
- Otra buena razón es poder **guardar** los resultados de nuestro programa para procesarlos en otro momento.

Ficheros

Definición

- **Unidad de almacenamiento** que permite guardar datos de forma persistente en el sistema de archivos de una computadora y que estarán almacenados bajo el mismo nombre en **memoria secundaria**.
- **Ventajas:**
 - Permite el almacenamiento de **grandes volúmenes** de datos.
 - **Persistencia** de los datos.
 - **Identificados** unívocamente.
 - Puede utilizarse como **fFuente de entrada** de otro programa.

Los ficheros se pueden clasificarse de varias maneras:

1. Según el **tipo** de contenido:

- Ficheros de texto.
- Ficheros binarios.

2. Según el **acceso** a los datos:

- Acceso secuencial.
- Acceso aleatorio o directo.

3. Según su **propósito**:

- Ficheros temporales.
- Ficheros permanentes.

Según el **tipo** de contenido:

- **Ficheros de texto:**

- Texto plano (**caracteres ASCII**).
- Pueden ser leídos y escritos por personas o por programas.

- **Ficheros binarios:**

- Datos en formato **binario**.
- Únicamente pueden ser leídos por programas.

Según el **acceso** a los datos

- **Acceso secuencial**

- Para leer una determinada posición de un fichero, es necesario haber leído todas las anteriores.
- En C, el acceso a **ficheros de texto** es secuencial.

- **Acceso directo**

- Es posible posicionarse en cualquier dato del fichero sin pasar por los datos que lo preceden.
- En C los ficheros **binarios** permiten el acceso directo.

Clasificación por su propósito

Según su **propósito**

- **Ficheros temporales:**

- Para almacenar datos de forma temporal durante la ejecución de un programa.
- Se eliminan al finalizar la ejecución del programa y se utilizan para almacenar datos intermedios o resultados parciales.

- **Ficheros permanentes:**

- Guardan información que debe persistir más allá de la ejecución del programa.
- Se suelen utilizar para almacenar configuraciones, registros históricos, y cualquier dato que deba ser conservado entre sesiones.

- Las **operaciones** básicas sobre ficheros y las **funciones** más utilizadas en C son:
 - Apertura: `fopen()`.
 - Lectura: `fscanf()`, `fgets()`, `fread()`, `fgetc()`.
 - Escritura: `fprintf()`, `fputs()`, `fwrite()`, `fputc()`.
 - Cierre: `fclose()`.
 - Manejo de errores: `fclose()`.

Apertura de ficheros

- La apertura de un fichero permite recuperar datos almacenados en un fichero de texto mediante un canal de comunicación entre el programa y el fichero:

```
FILE * p_fich = fopen(const char * n_fich,  
                      const char * modo)
```

- `p_fich`: variable descriptor del fichero.
- `n_fich`: ruta completa donde se leerá/escribirá el fichero. Puede ser absoluta o relativa.
- `modo`: indica si será de texto o binario, y si permitiremos escribir o solo leer.

Apertura de ficheros

Fichero de texto	Fichero binario	Descripción
"r"	"rb"	Abre el archivo para lectura. El archivo debe existir.
"r+"	"r+b"	Abre el archivo para lectura y escritura. El archivo debe existir.
"w"	"wb"	Abre el archivo para escritura. Si existe, se sobrescribe, si no, se crea uno nuevo.
"w+"	"w+b"	Abre el archivo para lectura y escritura. Si existe, se sobrescribe, si no, se crea uno nuevo.
"a"	"ab"	Abre un archivo para escritura en modo anexo. Si existe, los datos se añaden al final, si no, se crea uno nuevo.
"a+"	"a+b"	Abre un archivo para lectura y escritura en modo anexo. Si existe, los datos se añaden al final, si no, se crea uno nuevo.
"x"		Abre un archivo de texto para escritura exclusiva. El archivo no debe existir

Apertura de ficheros

- Es fundamental verificar el éxito de cada operación con el fichero, especialmente al abrir el fichero y al leer o escribir en él.
- Para ello, usamos la función `fopen()` que devuelve un `FILE *`:
 - Puntero a una **estructura** `FILE` definida en `stdio.h` que almacena toda la información del archivo.
- Si no se puede abrir el fichero, devolverá `NULL`.
- De esta forma realizamos la **comprobación** de errores.

Apertura de ficheros

- Algunos casos de fallos de un fichero:
 - Se abre en modo lectura y el fichero no existe.
 - No tiene permiso de escritura y se abre en modo escritura.
 - Se abre un fichero que no existe y la carpeta donde se va a crear no tiene permiso de escritura.
 - Abrimos un fichero de texto para escritura exclusiva y el archivo ya existe.

Apertura de ficheros

FILE * - Tipo de datos

- `FILE *` crea un flujo (**stream**) de datos que comunica con el fichero.
- Es una estructura interna que almacena, entre otros:
 - **Buffer**: almacén intermedio por donde pasan bloques de datos antes de hacer la operación de I/O.
 - **Cursor**: puntero a la posición donde se realizará la siguiente operación.
 - **EOF**: indicador para saber si se ha leído el fichero completo.
 - **Error**: indicador de errores I/O.

Apertura de ficheros de texto

```
FILE * fichero_texto;  
// Apertura de un fichero de texto utilizando una  
ruta relativa en modo lectura y escritura.  
FILE * fichero_texto = fopen("texto.txt", "r+");  
if (fichero_texto == NULL) {  
    printf("No se pudo abrir el archivo de texto.\n");  
    return -1;  
}
```

Apertura de ficheros

```
FILE * fichero_bin;
// Apertura de un fichero binario utilizando una ruta
// absoluta en modo escritura.
fichero_bin = fopen("C:\\Users\\diego.hortelano\\
CLionProjects\\ejemplo\\binario.bin", "wb");
if (fichero_bin == NULL) {
    printf("No se pudo abrir el archivo de texto.\n");
    return -1;
}
```

Cierre de ficheros

- Una vez que se han realizado todas las operaciones, es importante cerrar el fichero para liberar los recursos asociados.
- Esto se hace con `fclose()`, pasando como argumento el puntero al fichero.
- No cerrar el fichero puede provocar pérdida de datos o un consumo innecesario de recursos.

```
int fclose(FILE * p_fichero)
```

- Esta función, devuelve un entero:
 - = 0, si se cerró correctamente.
 - \neq 0, si hubo algún error al cerrar el fichero.

Cierre de ficheros

- Finaliza las operaciones de I/O, libera la memoria apuntada por el puntero a fichero y elimina la comunicación entre programa y fichero.
- Una vez cerrado, no podemos volver a escribir o leer de él hasta que no lo abramos de nuevo.
- Ejemplo de cierre de fichero:

```
if(fclose(fichero_texto) != 0)
{
    printf("Error al cerrar el fichero de texto");
}
```

Final de fichero

Función `feof()`

- La función `feof()` permite comprobar si se ha alcanzado el final de un *stream*.

```
int feof(FILE * fichero)
```

- Devuelve un valor distinto de cero cuando se ha alcanzado el fin de fichero o cero en caso contrario.

```
if(feof(fichero_texto)) {  
    printf("Se alcanzó el final del fichero de texto.\n");  
}
```


Ficheros de texto

Operaciones de lectura

- La lectura permite recuperar datos almacenados en un fichero de texto y se puede realizar usando varias funciones en C:
 - `fgetc()`: lee un carácter a la vez del fichero.
 - `fgets()`: lee una línea completa o hasta un máximo de caracteres. Es útil para leer líneas de texto de tamaño conocido.
 - `fscanf()`: lee datos formateados, similar a `scanf`, pero desde el fichero. Es útil para leer distintos tipos de datos, como enteros o cadenas, en un formato específico.

Ficheros de texto

Operaciones – Lectura de ficheros - Ejemplo

```
char buffer[100];  
printf("Contenido del archivo:\n");  
while (fgets(buffer, 100, file) != NULL) {  
    printf("%s", buffer); // Imprime cada línea leída  
}
```

Ficheros de texto

Operaciones de lectura

```
char nombre[20];
int edad;
float altura;
while (fscanf(file, "%s %d %f", nombre, &edad, &altura) == 3) {
    printf("Nombre: %s, Edad: %d, Altura: %.2f\n", nombre,
        edad, altura);
}
```

Ficheros de texto

Operaciones de escritura

- Para escribir datos en un fichero de texto, las funciones más comunes son:
 - `fputc()`: escribe un solo carácter en el fichero.
 - `fputs()`: escribe una cadena de caracteres.
 - `fprintf()`: escribe datos formateados, como una combinación de números y texto. Es similar a *printf*, pero redirige la salida al fichero.
- Ejemplo de escritura de una línea con `fputs()`:

```
fputs("Primera línea de texto.\n", file);
```

Ficheros de texto

Operaciones de escritura

```
const char * nombres[3] = {"Ana", "Luis", "Carlos"};
int edades[3] = {20, 22, 19};
float notas[3] = {8.5, 9.0, 7.8};

for (int i = 0; i < 3; i++) {
    // Comprobamos que fprintf() no retorne un valor negativo
    if (fprintf(file, "Nombre: %s, Edad: %d, Nota: %.1f\n", nombres[i],
                                                         edades[i], notas[i]) < 0) {
        printf("Error al escribir en el fichero\n");
        fclose(file);
        return 1; // Salimos del programa si ocurre un error
    }
}
```

Ficheros binarios

¿Qué son?

- Un fichero binario es un tipo de archivo en el que los datos se almacenan en formato binario (1s y 0s) en lugar de en formato de texto legible.
- Los datos se guardan tal cual se representan en la memoria, sin conversión a texto, lo cual hace que ocupen menos espacio y se lean/escriban más rápido.
- Los ficheros binarios son ideales para guardar estructuras complejas, como matrices o estructuras de datos, ya que preservan el formato exacto de la memoria.

Ficheros binarios

¿Para qué se utilizan?

- Los ficheros binarios se utilizan para almacenar grandes cantidades de datos, mantener la precisión de los datos y optimizar el espacio de almacenamiento.
- Algunos ejemplos comunes son:
 - Bases de datos y archivos de configuración.
 - Almacenamiento de datos científicos (con precisión en decimales).
 - Imágenes, videos, y otros datos multimedia, que requieren mucha capacidad de almacenamiento y una rápida recuperación.
 - Juegos y programas, que necesitan leer y escribir rápidamente estructuras de datos complejas, como tablas de puntuación, configuraciones, o estados de guardado.

Ficheros binarios

Operaciones de lectura

- La lectura de datos en ficheros binarios se realiza utilizando la función `fread()` en C. Esta función permite leer datos en el formato binario original y almacenarlos en memoria tal como fueron guardados.
- La sintaxis básica es:

```
size_t fread(void *ptr, size_t size,  
             size_t count, FILE *stream);
```

donde:

- `ptr`: puntero al bloque de memoria donde se guardarán los datos leídos.
- `size`: tamaño en bytes de cada elemento a leer.
- `count`: número de elementos a leer.
- `stream`: puntero al fichero.

Ficheros binarios

Operaciones de lectura

- Ejemplo de lectura de una estructura desde un fichero binario:

```
FILE * file = fopen("datos.bin", "rb");  
if (file != NULL) {  
    tipoAlumno alumno;  
    fread(&alumno, sizeof(tipoAlumno), 1, file);  
    fclose(file);  
}
```

Ficheros binarios

Operaciones de escritura

- Para escribir datos en un fichero binario se usa `fwrite()`, que permite almacenar datos en su formato binario directo, manteniendo su estructura en memoria sin conversión.
- La sintaxis básica es:

```
size_t fwrite(const void *ptr, size_t  
             size, size_t count, FILE *stream);
```

donde:

- `ptr`: puntero al bloque de memoria que contiene los datos a escribir.
- `size`: tamaño en bytes de cada elemento.
- `count`: número de elementos a escribir.
- `stream`: puntero al fichero.

Ficheros binarios

Operaciones de escritura

- Ejemplo de escritura de una estructura en un fichero binario:

```
FILE * file = fopen("datos.bin", "wb");  
if (file != NULL) {  
    tipoAlumno alumno = {"Juan Perez", 20, 68.5, 123456};  
    fwrite(&alumno, sizeof(tipoAlumno), 1, file);  
    fclose(file);  
}
```

Ficheros binarios

Desplazamiento del cursor

- En un fichero binario es posible desplazar el cursor para acceder la posición deseada directamente utilizando la función `fseek()`.
- La sintaxis básica es:

```
int fseek(File * fichero, long desplazamiento,  
          int origen)
```

donde:

- `fichero`: puntero a la estructura del fichero.
- `desplazamiento`: número de bytes a desplazarnos. Puede ser positivo o negativo.
- `origen`: punto de referencia. Admite tres valores: `SEEK_SET` (inicio), `SEEK_CUR` (actual) y `SEEK_END` (final).

Ficheros binarios

Desplazamiento del cursor

```
int main() {
    FILE * file = fopen("fichero_bin.bin", "wb");
    if (file == NULL) {
        printf("Error al abrir el fichero\n");
        return 1;
    }

    int dato;
    // Mover el puntero de posición 2 elementos (2 * sizeof(int) bytes) desde el inicio
    if (fseek(file, 2 * sizeof(int), SEEK_SET) != 0) {
        printf("Error al mover el puntero en el fichero\n");
        fclose(file);
        return 1;
    }
    // Leer el dato en la posición deseada
    fread(&dato, sizeof(int), 1, file);
    printf("Dato en la tercera posición: %d\n", dato);

    fclose(file);
    return 0;
}
```