

SESION 0 DE PRÁCTICAS

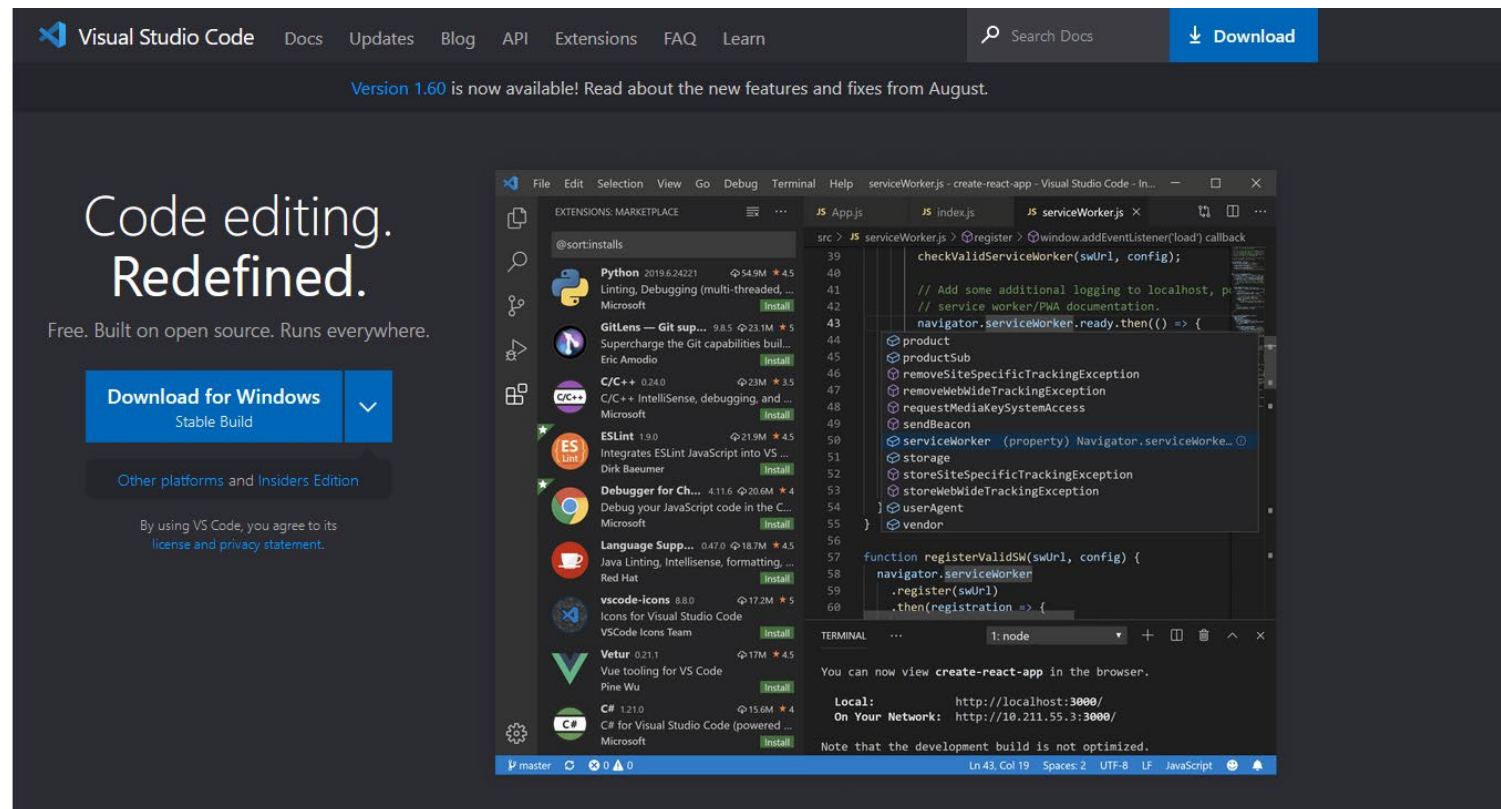
- ❑ **VISUAL STUDIO CODE**
- ❑ **LENGUAJE C**
 - ❑ **ENTRADA Y SALIDA**
 - ❑ **VECTORES Y MATRICES**
 - ❑ **PUNTEROS**



VISUAL STUDIO CODE

2

- ❑ Se trata de un editor de código fuente desarrollado por Microsoft para Windows, Linux y macOS.
- ❑ Para descarga, obtener documentación, ..., ir a <https://code.visualstudio.com/>



VISUAL STUDIO CODE en el aula de prácticas de laboratorio

3

Pasos a seguir:

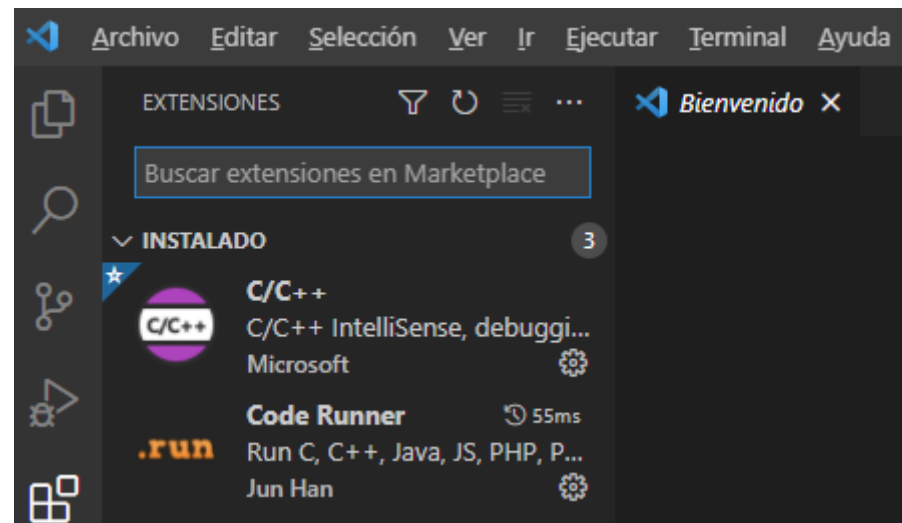
1.- Descargar e instalar el editor de código Visual Studio Code: <https://code.visualstudio.com/> ✓

2.- Instalar MinGW (Compilador C/C++) ✓

Una vez instalado MinGW, tendremos que agregar la carpeta bin de MinGW a las variables del entorno del sistema ✓

3.- Instalar extensión para Visual Studio Code: C/C++

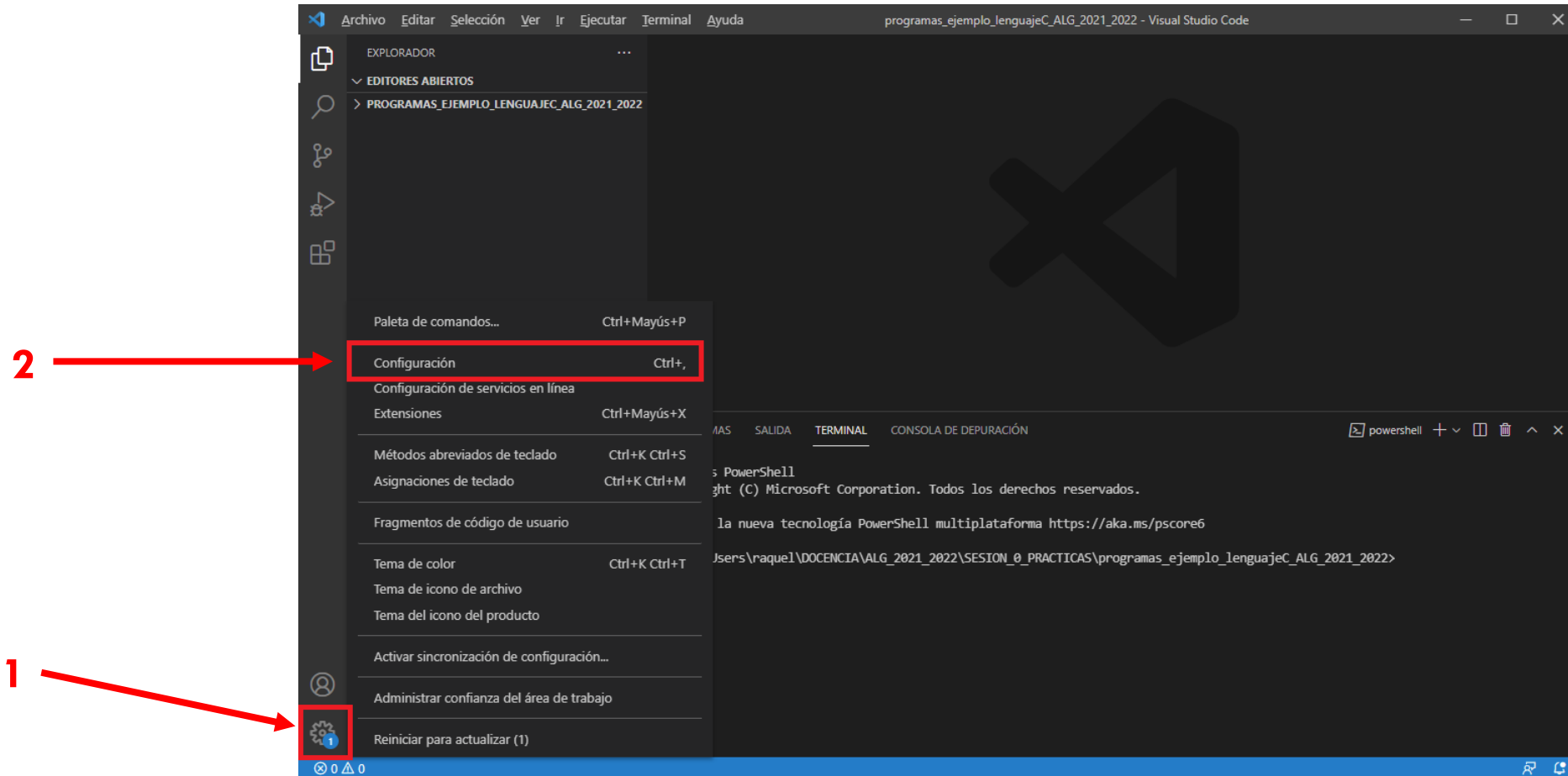
4.- Instalar extensión para Visual Studio Code: Code Runner



VISUAL STUDIO CODE en el aula de prácticas de laboratorio

4

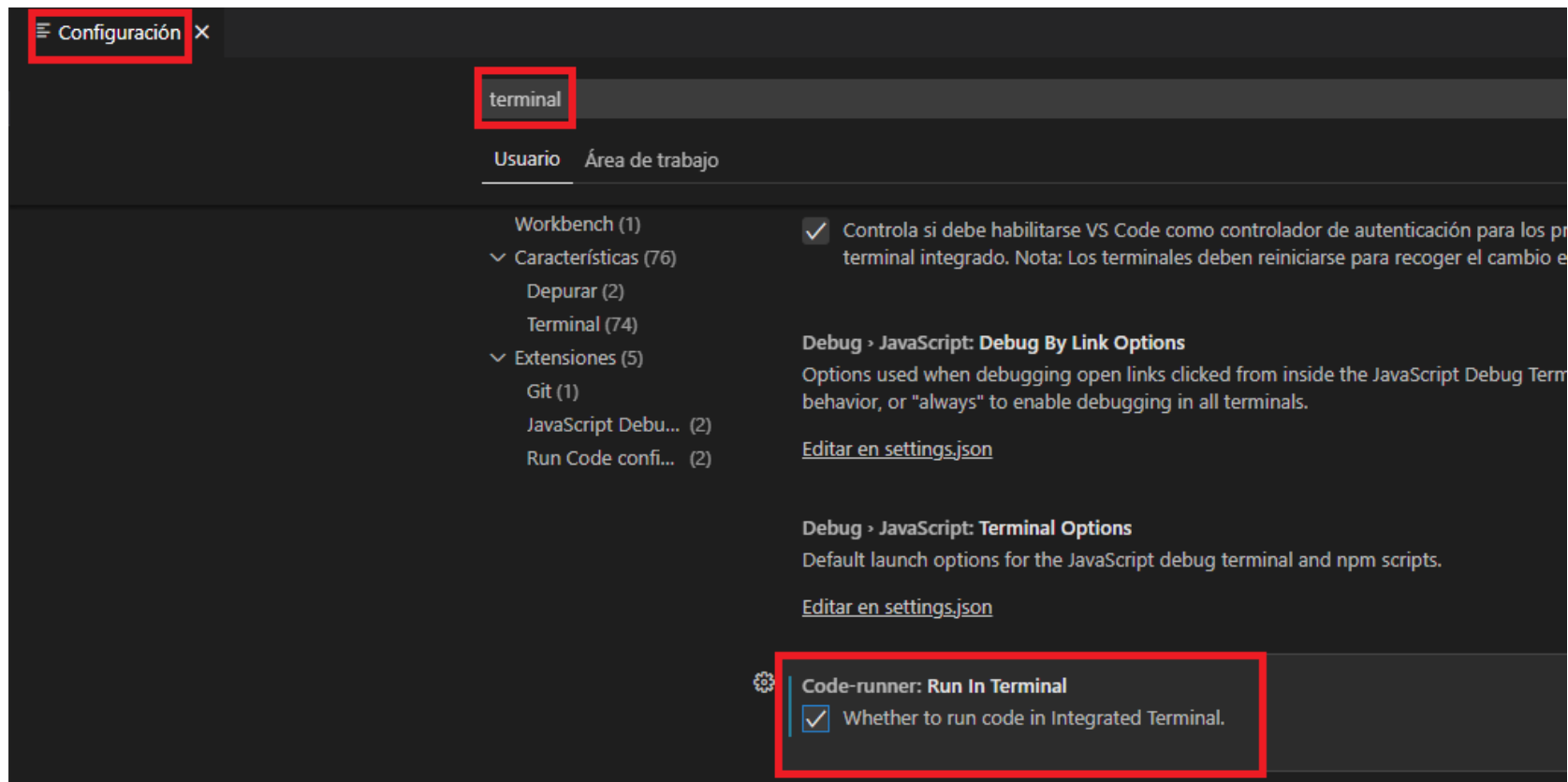
5.- Por último, pinchar en la rueda dentada (1), seguidamente seleccionar Configuración (2),



VISUAL STUDIO CODE en el aula de prácticas de laboratorio

5

teclear “terminal” en la parte superior de la pantalla y marcar la opción “Code-runner: Run in Terminal”



VISUAL STUDIO CODE en casa

6

- ❑ A lo largo de las diapositivas 3, 4 y 5 se indicaron los 5 pasos necesarios para la instalación de Visual Studio Code para programar en C.
- ❑ Los pasos 1 y 2 ya los había realizado el servicio de informática en los equipos del aula de prácticas de laboratorio, por lo que nosotros (alumnos y profesores) sólo tuvimos que realizar los pasos 3-5 para poder comenzar las prácticas.
- ❑ El alumno debe realizar todos los pasos para poder trabajar en su equipo particular.
- ❑ A continuación mostramos más detalles acerca de los pasos 1 y 2 (fases que el alumno no tuvo que realizar en el aula de prácticas, pero sí debe realizar en su equipo)
- ❑

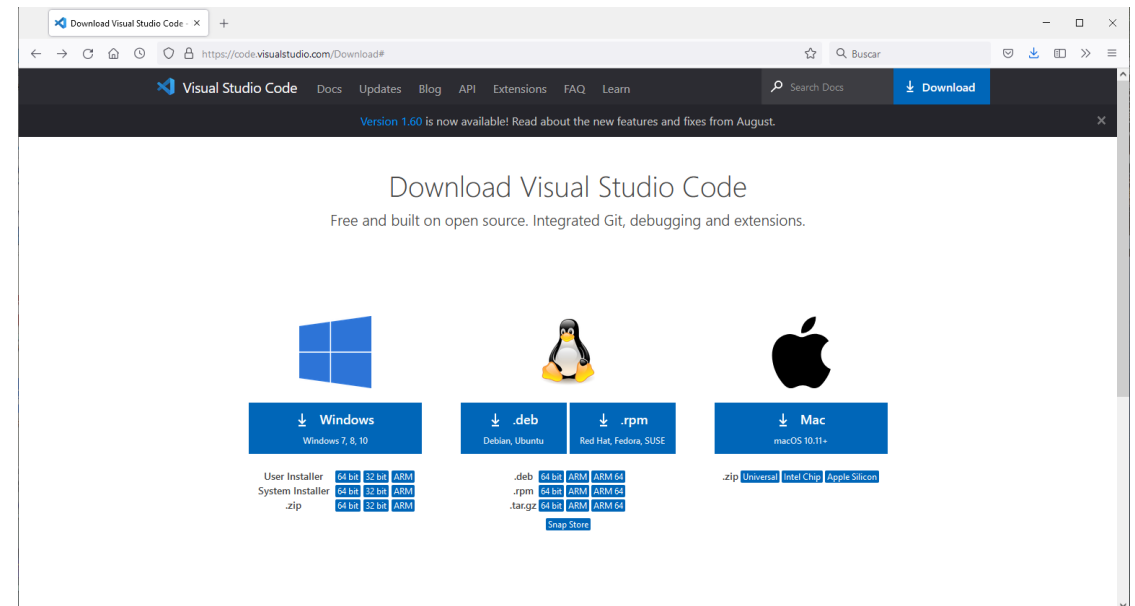
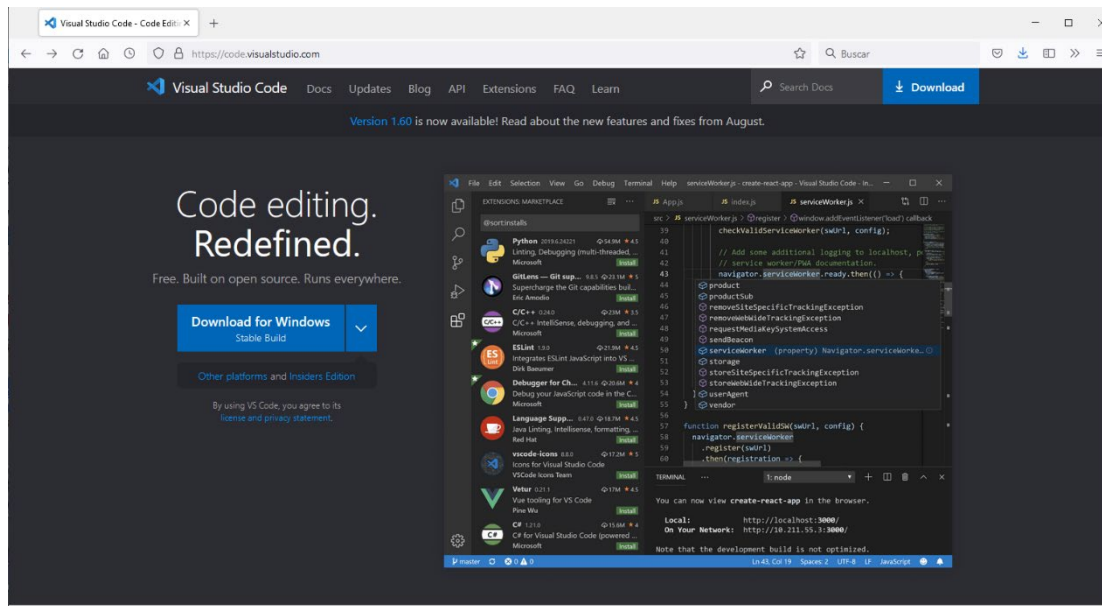


VISUAL STUDIO CODE en casa

7

Paso 1: Descargar e instalar Visual Studio Code según plataforma (Windows, Linux y macOS) en: <https://code.visualstudio.com/>

Se pueden consultar diferentes videos explicativos en <https://code.visualstudio.com/docs/getstarted/introvideos>



VISUAL STUDIO CODE en casa

8

Paso 2: Descargar e instalar MinGW (Compilador C/C++), según plataforma.

Visual Studio Code Docs Updates Blog API Extensions FAQ Learn

Version 1.60 is now available! Read about the new features and fixes from August.

Using GCC with MinGW

In this tutorial, you configure Visual Studio Code to use the GCC C++ compiler (g++) and GDB debugger from [mingw-w64](#) to create programs that run on Windows.

After configuring VS Code, you will compile and debug a simple Hello World program in VS Code. This tutorial does not teach you about GCC, GDB, Mingw-w64, or the C++ language. For those subjects, there are many good resources available on the Web.

If you have any problems, feel free to file an issue for this tutorial in the [VS Code documentation repository](#).

Prerequisites

To successfully complete this tutorial, you must do the following steps:

1. Install [Visual Studio Code](#).
2. Install the [C/C++ extension for VS Code](#). You can install the C/C++ extension by searching for 'c++' in the Extensions view ([Ctrl+Shift+X](#)).
3. Get the latest version of Mingw-w64 via [MSYS2](#), which provides up-to-date native builds of GCC, Mingw-w64, and other helpful C++ tools and libraries. [Click here](#) to download the MSYS2 installer. Then follow the instructions on the [MSYS2 website](#) to install Mingw-w64.
4. Add the path to your Mingw-w64 [bin](#) folder to the Windows [PATH](#) environment variable by using the following steps:
 1. In the Windows search bar, type 'settings' to open your Windows Settings.
 2. Search for [Edit environment variables for your account](#).

IN THIS ARTICLE

- Prerequisites
- Create Hello World
- Explore IntelliSense
- Build helloworld.cpp
- Debug helloworld.cpp
- Step through the code
- Set a watch
- C/C++ configurations
- Troubleshooting
- Next steps

[Tweet this link](#)
[Subscribe](#)
[Ask questions](#)
[Follow @code](#)
[Request features](#)
[Report issues](#)
[Watch videos](#)

Overview

- SETUP
- GET STARTED
- USER GUIDE
- LANGUAGES
- NODE.JS / JAVASCRIPT
- TYPESCRIPT
- PYTHON
- JAVA
- C++**
 - Intro Videos
 - GCC on Linux
 - [GCC on Windows](#)
 - GCC on Windows Subsystem for Linux
- Clang on macOS
- Microsoft C++ on Windows
- CMake Tools on Linux
- Debugging
- Editing
- Settings
- Configure IntelliSense for cross-compiling
- FAQ
- CONTAINERS
- DATA SCIENCE
- AZURE
- REMOTE

VISUAL STUDIO CODE en casa

9

En el supuesto que la plataforma fuese Windows, tras la descarga y la instalación, se debe añadir el path de la carpeta bin de Mingw-w64 a la variable de entorno path de Windows:

- En la barra de búsqueda de Windows, buscamos “variables de entorno”
- Elegimos “Editar variables de entorno para nuestra cuenta”
- Escogemos la variable path y seleccionamos editar
- Añadimos el path de la carpeta bin de Mingw-w64. El path exacto dependerá de la versión que se haya instalado y donde esté instalada (C:\...\mingw64\bin)
- Seleccionar ok para guardar el path modificado.

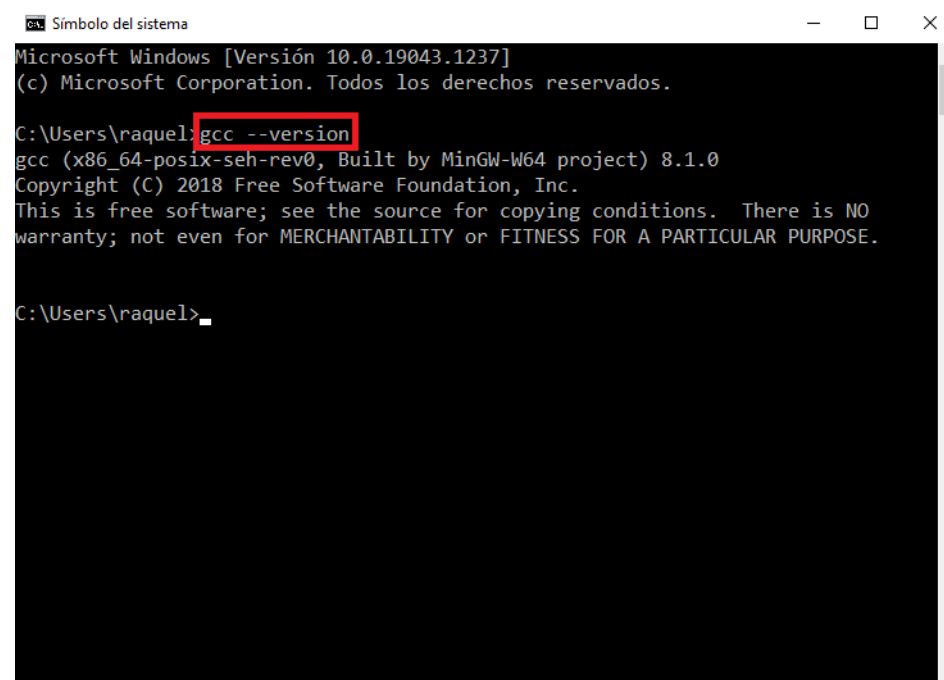
Estas indicaciones se encuentran en <https://code.visualstudio.com/docs/cpp/config-mingw>



VISUAL STUDIO CODE en casa

10

Para asegurarse que la instalación en Windows ha sido correcta, abrimos la consola de comandos y tecleamos “gcc --version”. Si todo es correcto podremos ver la salida correspondiente, en la que se nos muestra la versión instalada, en este caso, la 8.1.0



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19043.1237]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\raquel>gcc --version
gcc (x86_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

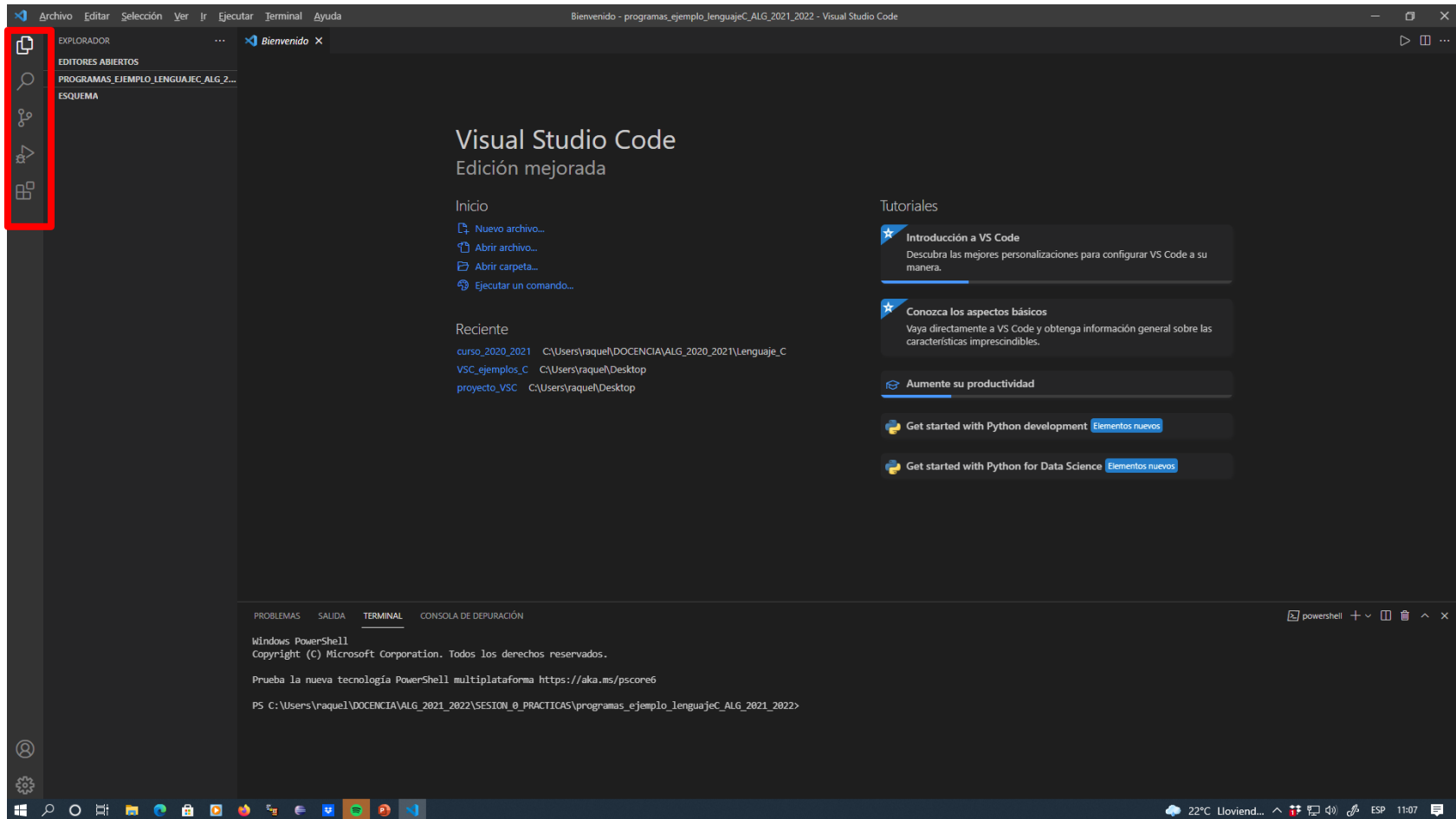
C:\Users\raquel>
```



ASPECTO GENERAL DE VISUAL STUDIO CODE

11

BARRA ACTIVIDADES:
EXPLORADOR,
BUSCAR,
CONTROL CÓDIGO FUENTE,
DEPURADOR
EXTENSIONES



ABRIR, COMPILAR Y EJECUTAR “HOLA MUNDO”

12

- Este es el clásico programa “Hola mundo” en C

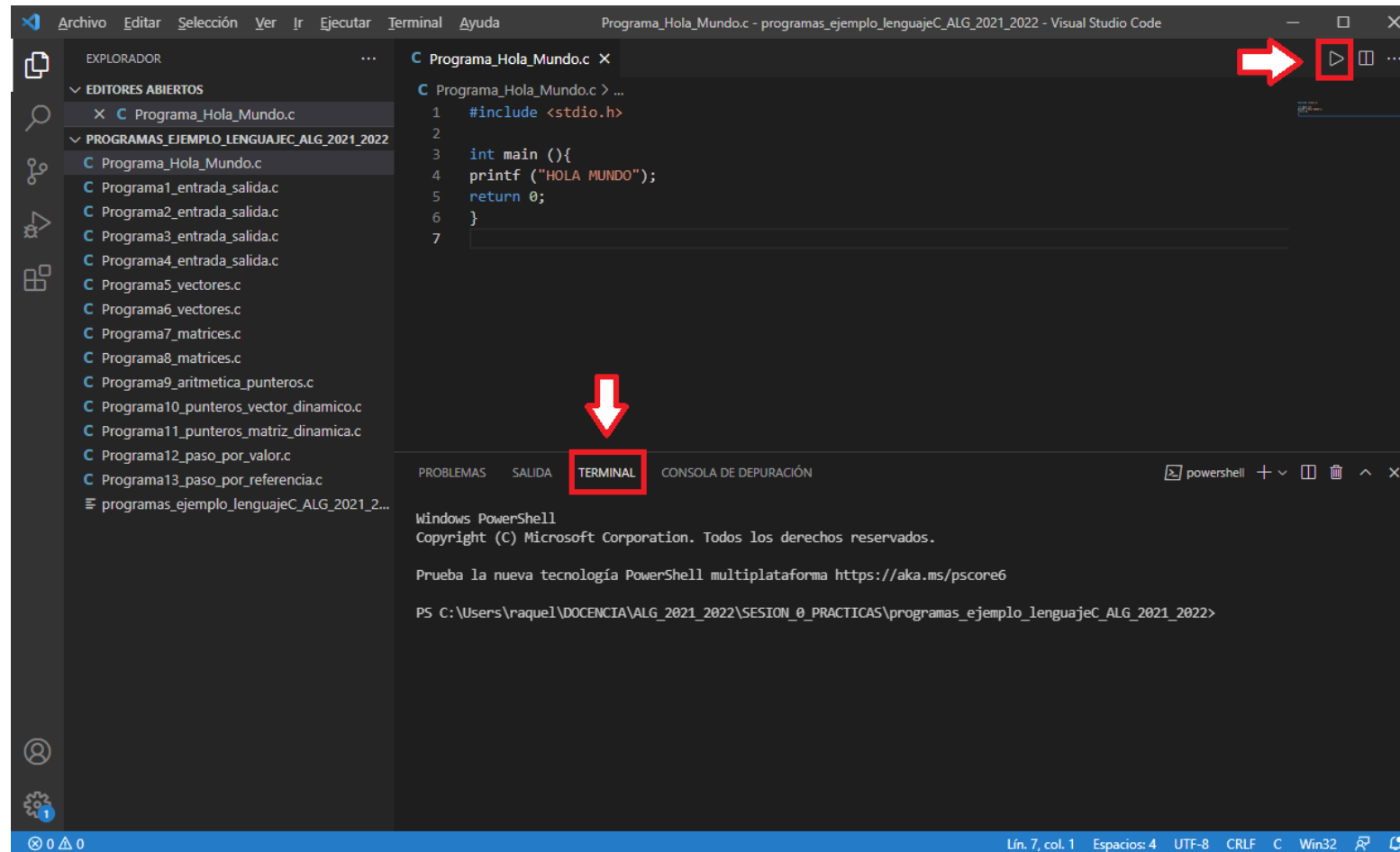
```
#include <stdio.h>
int main() {
    printf("Hola mundo");
    return 0;
}
```

- Abrimos el archivo Programa_Hola_Mundo.c, compilamos y ejecutamos en Visual Studio Code



ABRIR, COMPILAR Y EJECUTAR “HOLA MUNDO”

13



ERRORES Y AVISOS MÁS HABITUALES

14

Al compilar

- ❑ Instrucción no escrita correctamente
 - ❑ Por ejemplo, wihle en vez de while
- ❑ Falta un signo o una instrucción
 - ❑ Típicamente, olvidarse del punto y coma al final de cada instrucción o la falta alguna llave o paréntesis
- ❑ La instrucción no está ubicada en el lugar correcto
- ❑ Una instrucción es sospechosa (warning)
 - ❑ El compilador requiere la atención del programador

Al ejecutar

- ❑ El programa no produce el resultado esperado
 - ❖ Ejemplo: Teclear * en vez de +
- ❑ Se produce algún acceso incorrecto a la memoria



LAS LIBRERÍAS EN C

15

- C es un lenguaje de programación funcional. Incluso el cuerpo del programa es una función llamada `main` que devuelve un dato de tipo `int`, (0 en este ejemplo).
- `#include <stdio.h>` es una sentencia que añade al programa (incluye en él) un conjunto de funciones que facilitan la entrada y salida de datos. Por ejemplo, incluye la función `printf` que utilizamos en el ejemplo.



LENGUAJE C

ENTRADA/SALIDA



ENTRADA/SALIDA

17

En general, en el mundo de la informática la entrada y salida de datos es un tema complejo. Desafortunadamente la entrada y salida en C no es una excepción.

En estas transparencias explicaremos algunos conceptos básicos para que los alumnos puedan leer y escribir datos de tipo numérico y caracteres en C.

Para leer datos utilizaremos la función `scanf` y para escribir datos la función `printf`, de la que ya hemos hablado.

Ambas funciones son parte de una biblioteca estándar de C, y para utilizarlas es preciso incluir el fichero de cabecera `stdio.h`



LECTURA DE DATOS

18

Para leer un dato utilizaremos la función `scanf` de la siguiente forma:

```
scanf ("%X", &var);
```

La `X` representa el tipo de los datos que queremos leer y debe ser sustituida por alguna de las siguientes opciones (hay más posibilidades que las enumeradas aquí):

- `d`, para leer un entero (int).

- `f`, para leer un valor de punto flotante (float).

- `lg`, para leer un valor de punto flotante en doble precisión (double).

- `c`, para leer un carácter (char).

`var` es el nombre de la variable donde se almacenará el valor leído. Observe que el nombre de la variable va precedido del operador `&`. Más adelante se verá por qué es necesario poner este operador en la función `scanf`.



ESCRITURA DE DATOS

19

La función `printf` nos permite escribir una secuencia de caracteres en la salida, su formato es:

```
printf ("Secuencia a escribir" [, expresión1, expresión2, ...]);
```

La serie de expresiones es opcional. Ya hemos escrito `Hola mundo` con la sentencia

```
printf ("Hola mundo");
```

Para producir un salto de línea en la salida hemos utilizado la secuencia `\n`.

Escribe `Hola`, dos líneas vacías y después `a todos` seguido de un salto de línea.



EL BUFFER DE LECTURA

20

En esta sección se describen algunas peculiaridades de la lectura de datos con `scanf`. En concreto queremos destacar los siguientes aspectos:

- Cuando el usuario introduce un dato desde el teclado, los datos tecleados no están disponibles para el programa hasta que no pulsa la tecla `Enter`. Por tanto, el programa no leerá nada hasta ese momento.
- Los datos tecleados se almacenan sucesivamente en un `buffer` de lectura. Al pulsar la tecla `Enter` se incluye también un carácter en el buffer (lo representaremos por `#`).
- De este buffer (o memoria intermedia) es de donde el programa leerá los datos.



EL BUFFER DE LECTURA

21

- Cuando se lee un dato de tipo **numérico** con **scanf** ocurre lo siguiente.

Primero se descartan todos los caracteres blancos que haya en el buffer (por caracteres blancos se entiende el espacio en blanco, el tabulador y el salto de línea), después se lee del buffer el número introducido hasta que se encuentre un carácter que no sea de tipo numérico (normalmente, un espacio blanco). La siguiente lectura con **scanf** comenzará con este último carácter.

- Cuando se lee un dato de tipo **carácter** con **scanf** se lee el primer carácter que haya en el buffer (sea un carácter blanco o no). Es decir, no hay descarte de caracteres blancos.



EL PROBLEMA DE LOS CARACTERES NO PROCESADOS EN EL BUFFER DE LECTURA

22

- A continuación exponemos dos problemas que surgen al leer datos con `scanf` y una posible solución a ellos. Para ejemplificar el primer problema supongamos el siguiente programa:

```
#include <stdio.h>
int main () {
    int x, y;
    printf ("Introduzca un entero: ");
    scanf ("%d", &x);
    printf ("Introduzca un segundo entero: ");
    scanf ("%d", &y);
    printf ("Los valores introducidos son: %d y %d\n", x, y);
}
```

Programa1_entrada_salida.c

- Si el usuario teclea un número, pulsa **Enter**, después otro número y otra vez **Enter** entonces el programa tendrá el comportamiento “esperado”.



EL PROBLEMA DE LOS CARACTERES NO PROCESADOS EN EL BUFFER DE LECTURA

23

- Pero si cuando pida el primer entero escribimos '2 3' y pulsamos **Enter**, observaremos que el resultado de la ejecución es otro.
- Lo que ocurre es lo siguiente. Cuando se ejecute el primer **scanf**, el buffer está vacío y el programa no comenzará a leer datos hasta que el usuario pulse Enter, en cuyo momento ya tendrá el siguiente contenido: '2 3#'.
- El primer **scanf** leerá el 2 y se detendrá al encontrar el carácter blanco. El buffer no queda vacío por lo que se ejecuta el segundo **scanf** que saltará el espacio en blanco y leerá el 3, ya que va seguido de un carácter no numérico. En el programa ya no hay más sentencias de lectura y finaliza su ejecución.



EL PROBLEMA DE LOS CARACTERES NO PROCESADOS EN EL BUFFER DE LECTURA

24

- Si queremos leer un único dato por línea, independientemente de que se teclee más de un dato en una misma línea, podemos escribir lo siguiente.

```
#include <stdio.h>

int main () {
    int x, y;
    printf ("Introduce un entero: ");
    scanf ("%d", &x);
    while (getchar () != '\n');
    printf ("Introduce un segundo entero: ");
    scanf ("%d", &y);
    printf ("Los valores introducidos son: %d y %d\n", x, y);
}
```

Programa2_entrada_salida.c



EL PROBLEMA DE LOS CARACTERES NO PROCESADOS EN EL BUFFER DE LECTURA

25

- En el ejemplo anterior, la instrucción situada tras el `scanf` elimina (realmente lee, aunque no haga nada con ellos) todos los caracteres pendientes de procesamiento almacenados en el buffer, incluido el `Enter`. Es decir, lo vacía.
- Por lo tanto, tras leerse el primer número, se eliminará todo lo que el usuario haya tecleado en el resto de la línea.
- Al ejecutarse el segundo `scanf`, el buffer está vacío y el programa se detiene hasta que se introduzcan nuevos datos y se pulse `Enter`.
- A continuación describimos un problema muy parecido al anterior y que tiene la misma solución.



EL PROBLEMA DE LOS CARACTERES NO PROCESADOS EN EL BUFFER DE LECTURA

26

- ❑ Este programa lee un entero y después un carácter.
- ❑ Ejecútalo, teclea un número entero y luego pulsa **Enter**. Observa la salida.

```
#include <stdio.h>
int main () {
    int x;
    char c;
    printf ("Introduce un entero: ");
    scanf ("%d", &x);
    printf ("Introduce un caracter: ");
    scanf ("%c", &c);
    printf ("Los valores introducidos son: %d y %c\n", x, c);
}
```

Programa3_entrada_salida.c



EL PROBLEMA DE LOS CARACTERES NO PROCESADOS EN EL BUFFER DE LECTURA

27

¿Qué ha ocurrido?

- Mientras el usuario no pulse **Enter**, el programa no dispone de los datos. Cuando lo pulsa, **scanf** lee el número y detiene la lectura al encontrar el **Enter**, que recordemos es un carácter más del buffer. Ese **Enter** queda en el buffer pendiente de procesamiento.
- A continuación se ejecuta el segundo **scanf**. Se empieza leyendo por lo que queda en el buffer (o sea, el **Enter**). Como estamos leyendo un carácter, el **Enter** no se descarta y se almacena en la variable **c**.
- Para evitar esto se recomienda la misma solución que para el problema anterior. Es decir, utilizar

```
while (getchar () != '\n');
```

para eliminar los caracteres que quedan por procesar de la línea en el **buffer** (en este caso el **Enter**).



EL PROBLEMA DE LOS CARACTERES NO PROCESADOS EN EL BUFFER DE LECTURA

28

La solución sería:

```
#include <stdio.h>
int main () {
    int x;
    char c;
    printf ("Introduce un entero: ");
    scanf ("%d", &x);
    while (getchar () != '\n');
    printf ("Introduce un carácter: ");
    scanf ("%c", &c);
    printf ("Los valores introducidos son: %d y %c\n", x, c);
}
```

Programa4_entrada_salida.c



LENGUAJE C

VECTORES Y MATRICES



VECTOR UNIDIMENSIONAL

30

- Un vector es un modo de manejar una gran cantidad de datos del mismo tipo bajo un mismo nombre o identificador.
- La forma general de la declaración de un vector es la siguiente:

tipo_dato nombre[numero_elementos];

- Cada uno de los valores contenidos en el vector tiene una posición asociada que se usará para acceder a él. Esta posición o índice será siempre un número entero positivo.
- Los elementos se numeran desde **0** hasta **numero_elementos-1**.



VECTOR UNIDIMENSIONAL

31

- Por ejemplo, mediante la sentencia:

```
int mi_vector[10];
```

Se reserva espacio para las 10 variables de tipo int.

Las 10 variables se llaman `mi_vector` y se accede a una u otra por medio del índice, que es una expresión entera escrita a continuación del nombre entre corchetes.

- De tal modo que para acceder al primer elemento del ejemplo haríamos `mi_vector[0]`, mientras que para acceder al último `mi_vector[9]`.



VECTOR UNIDIMENSIONAL

32

- En **C** no se puede operar con todo un vector como una única entidad, sino que hay que tratar sus elementos uno a uno por medio de bucles **for**, **while** o **do..while**.
- Los elementos de un vector se utilizan en las expresiones de **C** como cualquier otra variable.
- Ejemplos de uso:

```
mi_vector[5] = 8;
```

```
mi_vector[9] = 2 * mi_vector [5];
```

```
mi_vector[0] = 4;
```

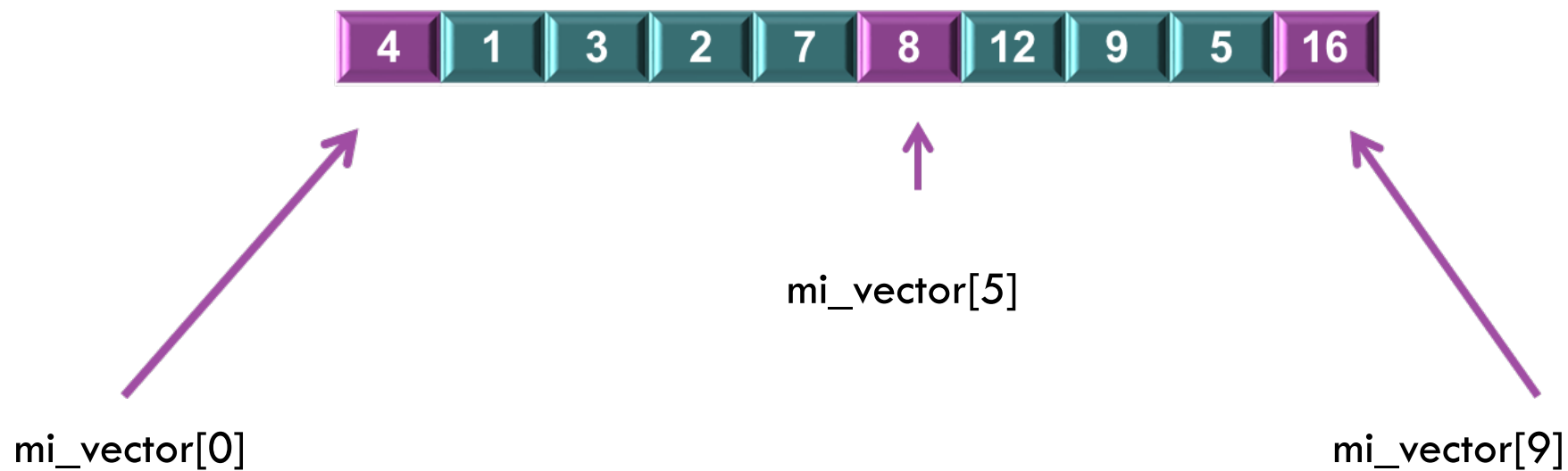
```
mi_vector[4] = mi_vector [0] + mi_vector[2]
```



VECTOR UNIDIMENSIONAL

33

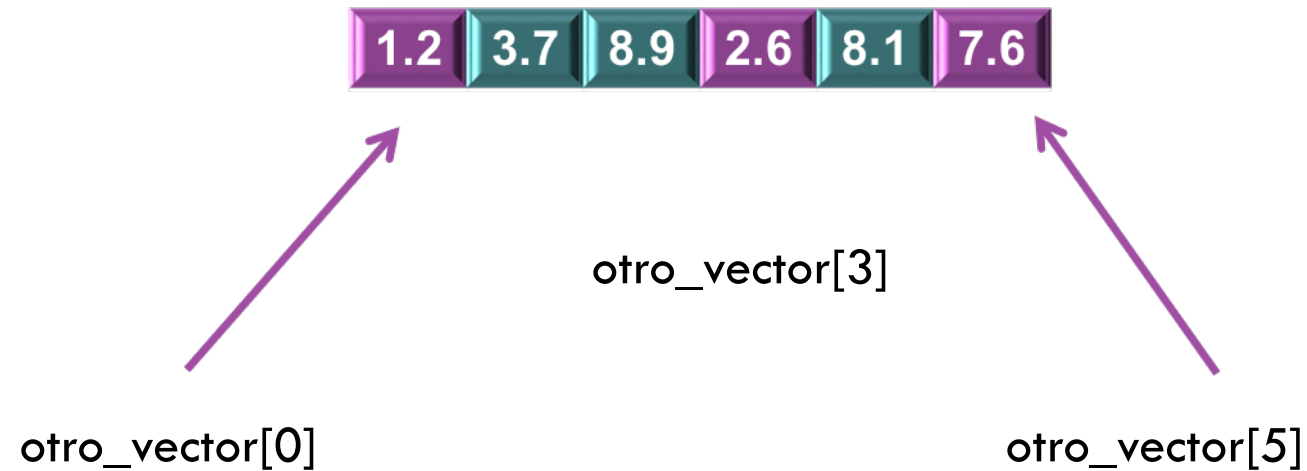
- Ejemplo: `int mi_vector[10]`



VECTOR UNIDIMENSIONAL

34

- Ejemplo: `float otro_vector[6] = { 1.2, 3.7, 8.9, 2.6, 8.1, 7.6 }`



VECTOR UNIDIMENSIONAL Y FUNCIONES

35

```
#include <stdio.h>

// la directiva #define indica al preprocesador que debe sustituir, en el código fuente del programa, todas las apariciones de ELEMENTOS por 6
#define ELEMENTOS 6

// Al pasar como parámetro un vector unidimensional, no tiene por qué tener la dimensión acotada
int suma_vector(int []); /* declaración de la función suma_vector */

int main () {
    //Declaramos un vector de tamaño fijo
    int mi_vector[ELEMENTOS]={5, 8, 9, 4, 3, 2};
    printf("La suma es: %d\n", suma_vector(mi_vector));
    return 0;
}

int suma_vector(int V[]) {
    /* definición de la función suma_vector */
    int i,suma=0;
    for (i=0;i<ELEMENTOS;i++)
        suma+=V[i];
    return suma;
}
```



VECTOR MULTIDIMENSIONAL

36

- ❑ Se pueden definir matrices con dos o más dimensiones.

`tipo_dato nombre [dimensiónA] [dimensiónB] ... ;`

- ❑ Ejemplo de una matriz de dos dimensiones con 3 filas y 4 columnas

`int mi_matriz [3][4];`

- ❑ Las **filas** se numeran de **0** a **dimensionA-1** y las columnas de **0** a **dimensionB-1**
- ❑ Se accede a los elementos de la matriz con esta expresión:

`mi_matriz [i][j];`

- ❑ El primer índice indica la fila y el segundo índice, la columna



VECTOR BIDIMENSIONAL

37

- Las matrices en C se **almacenan por filas**, en posiciones consecutivas de memoria. En cierto modo, una matriz se puede ver como un vector de vectores-fila.

- Ejemplo, si inicializamos una matriz de 3 filas y 4 columnas de la forma

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

- Los valores quedan dispuestos de la siguiente manera

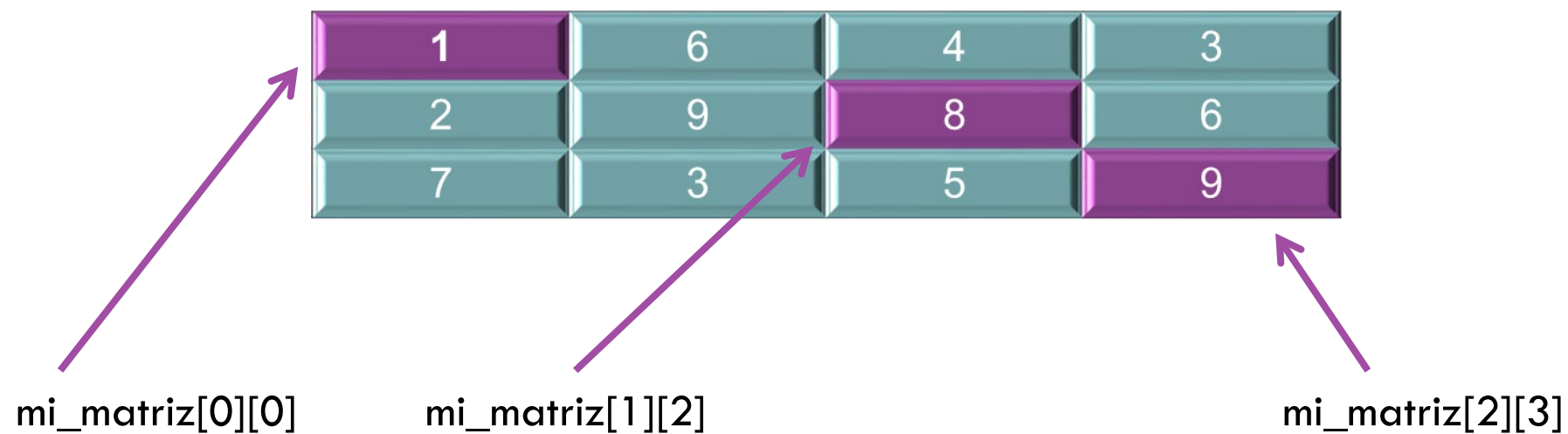
	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	1	6	4	3
Fila 1	2	9	8	6
Fila 2	7	3	5	9



VECTOR BIDIMENSIONAL (MATRIZ)

38

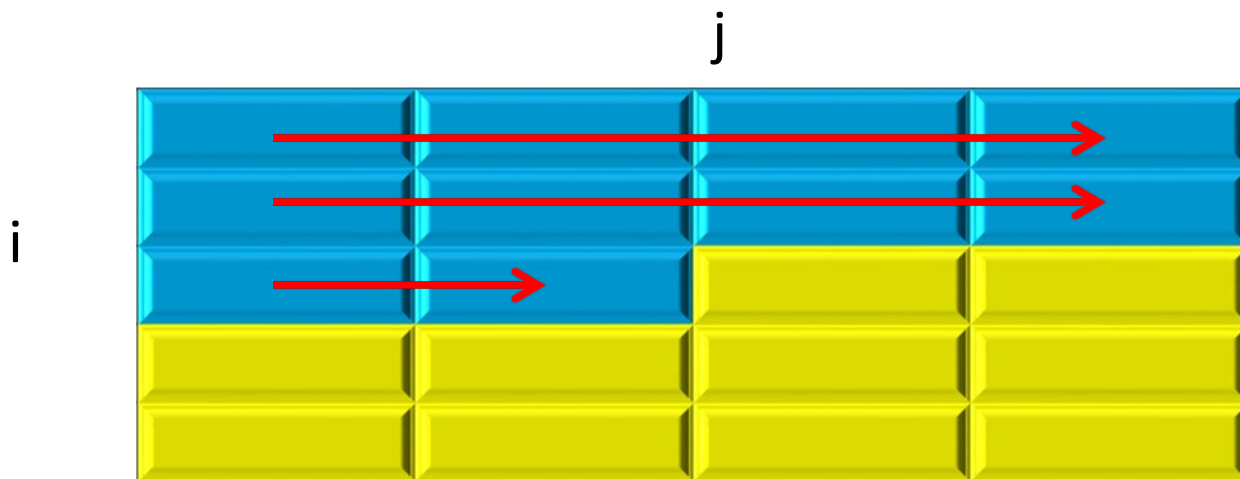
- Ejemplo: `int mi_matriz[3][4]`



VECTOR BIDIMENSIONAL (MATRIZ)

39

- Recorrer los elementos de la matriz M por filas



Para cada fila i

Para cada columna j

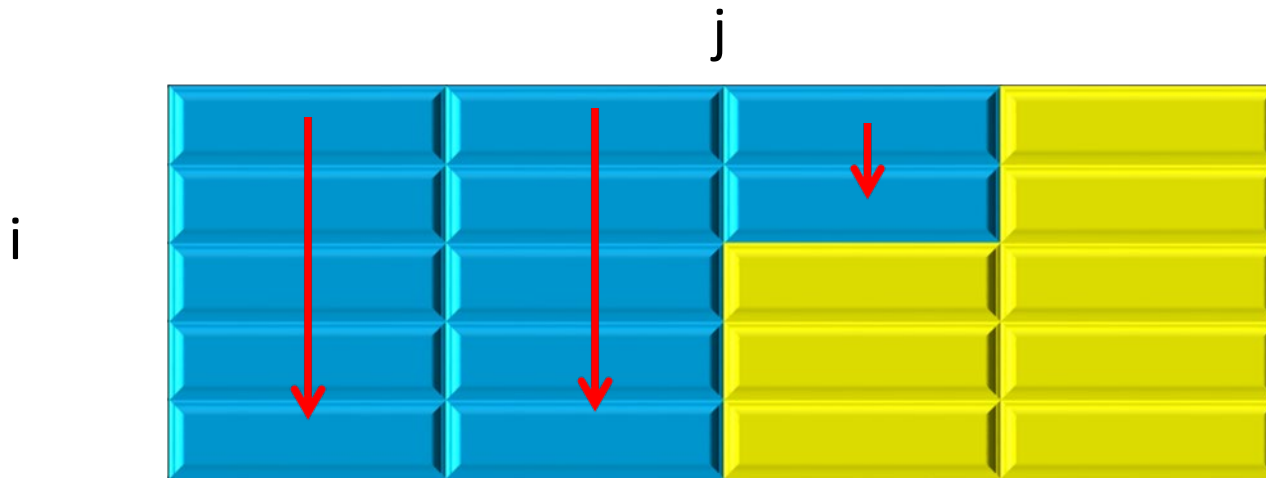
visitar $M[i][j]$



VECTOR BIDIMENSIONAL (MATRIZ)

40

- Recorrer los elementos de la matriz M por columnas



Para cada columna j

Para cada fila i

visitar $M[i][j]$



VECTOR BIDIMENSIONAL Y FUNCIONES

41

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

```
#define FILAS 2
```

```
#define COLUMNAS 3
```

```
// Al pasar como parámetro un vector multidimensional, éste debe tener todas las dimensiones acotadas, salvo la primera
```

```
int suma_matriz (int matriz[][COLUMNAS]);
```

```
int main () {
```

```
    //Declaramos una matriz y la inicializamos en la misma declaración
```

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

```
    //Invocamos a la función que suma las componentes de la matriz
```

```
    printf("\n\nLa suma de los elementos de la matriz es: %d\n\n",suma_matriz (mi_matriz));
```

```
    return 0;
```

```
}
```



VECTOR BIDIMENSIONAL Y FUNCIONES

42

```
int suma_matriz (int matriz[][COLUMNAS]) {  
    int i, j, suma = 0;  
    printf("\n");  
    for (i=0;i<FILAS;i++) {  
        for (j=0;j<COLUMNAS;j++)  
            suma+=matriz[i][j];  
        printf("\n");  
    }  
    printf("\n");  
    return suma;  
}
```



PROGRAMAS EJEMPLO

43

Los programas

- ❖ Programa5_vectores.c
- ❖ Programa6_vectores.c
- ❖ Programa7_matrices.c
- ❖ Programa8_matrices.c

que puedes descargar en el Campus Virtual, contienen ejemplos sobre la utilización de vectores unidimensionales y bidimensionales



LENGUAJE C

❑ PUNTEROS

- ❑ Definición de variables puntero
- ❑ Operadores dirección e indirección
- ❑ Aritmética de punteros
- ❑ Relación entre punteros y vectores
- ❑ Relación entre punteros y matrices
- ❑ Paso por valor y paso por referencia



DEFINICIÓN VARIABLES PUNTERO

45

- El valor de cada variable está almacenado en un lugar determinado de la memoria, caracterizado por una dirección (que se suele expresar con un número hexadecimal).
- El ordenador mantiene una tabla de direcciones que relaciona el nombre de cada variable con su dirección en la memoria.
- Gracias a los identificadores, no es necesario que el programador se preocupe de la dirección de memoria en la que están almacenados sus datos.
- Aunque, en ocasiones es más útil trabajar con las direcciones que con los propios nombres de las variables.



DEFINICIÓN VARIABLES PUNTERO

46

- El lenguaje C dispone de un tipo especial de variables destinadas a contener direcciones de variables.
- Estas variables se denominan **punteros**.
- Así, un **puntero es una variable que puede contener la dirección de otra variable**.
- Los punteros lógicamente están almacenados en algún lugar de la memoria y tienen su propia dirección (punteros a punteros).



DEFINICIÓN VARIABLES PUNTERO

47

- Se dice que **un puntero apunta a una variable** si su contenido es la dirección de esa variable.
- Un puntero ocupa 4 bytes de memoria y se debe declarar o definir de acuerdo al tipo de dato al que apunta.
- Ejemplo: un puntero a una variable de tipo int se declara así:

```
int *p;
```

- Lo que quiere decir que la variable **p** podrá contener la dirección de cualquier variable entera.
- Por tanto, el valor al que apunta **p** es de tipo **int**.
- Los punteros a char, float y double se definen de manera análoga.



OPERADORES DIRECCIÓN (&) E INDIRECCIÓN (*)

48

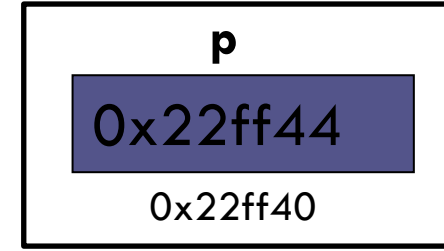
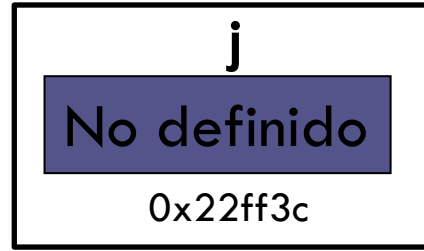
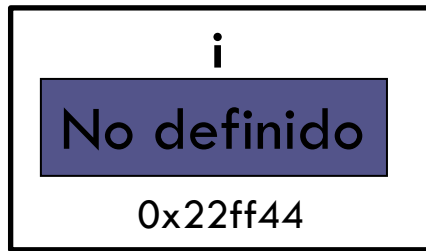
- El operador de dirección (&) permite hallar la dirección de la variable a la que se aplica.
- Un puntero es una verdadera variable, por tanto puede cambiar de valor, esto es, puede cambiar la variable a la que apunta.
- Para acceder al valor depositado en la zona de memoria a la que apunta un puntero se debe de utilizar el operador indirección (*).



EJEMPLO

49

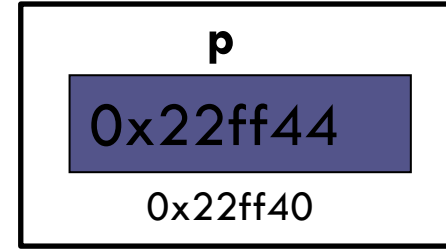
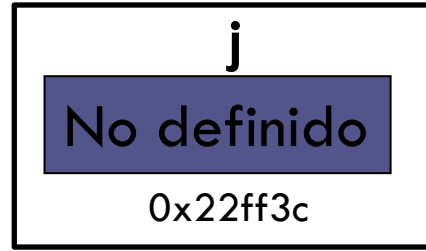
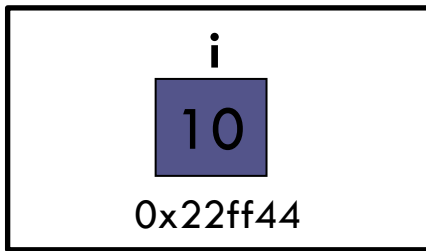
```
int i, j, *p;    // p es un puntero a int  
p=&i;           // p contiene la dirección de i
```



EJEMPLO

50

***p=10;** // i toma el valor 10;

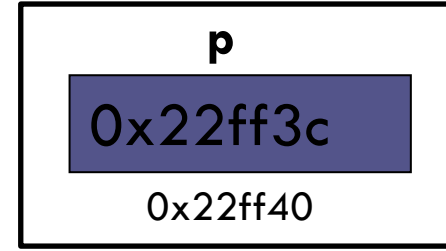
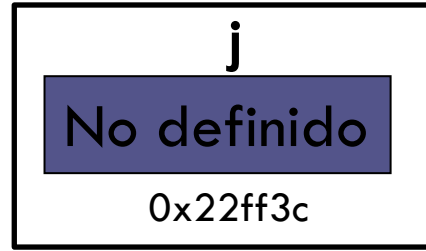
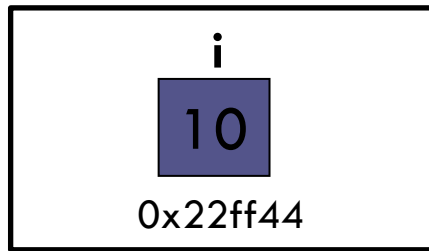


EJEMPLO

51

`p=&j;`

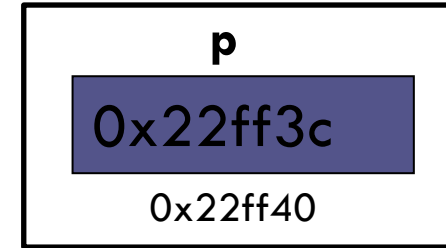
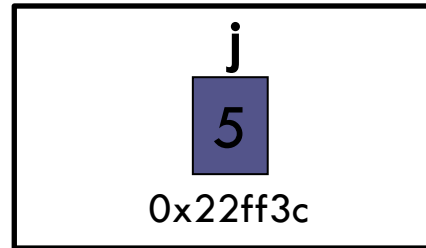
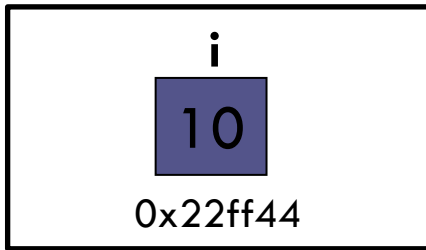
// p contiene ahora la dirección de j



EJEMPLO

52

***p=5;** // j toma el valor 5



OPERADORES DIRECCIÓN (&) E INDIRECCIÓN (*)

53

- ❑ Las constantes y las expresiones no tienen dirección, por lo que no se les puede aplicar el operador (&).
- ❑ Tampoco puede cambiarse la dirección de una variable.
- ❑ Para imprimir punteros con la función `printf()` se debe utilizar el formato `%p`



ARITMÉTICA DE PUNTEROS

54

- Hemos visto que un puntero es una variable especial ya que guarda información tanto de la dirección a la que apunta así como del tipo de variable almacenado en esa dirección.
- Por ello, no van a estar permitidas las operaciones que no tienen sentido con direcciones de variables, como sumar, multiplicar y dividir, pero si otras como incrementar, decrementar o restar.

- Ejemplo:

```
int *p;
```

```
p = p + 1;
```

hace que p apunte a la siguiente dirección a la que apuntaba, teniendo en cuenta el tipo de dato.



ARITMÉTICA DE PUNTEROS

55

- Esto quiere decir que si el valor apuntado por `p` es `int` y por tanto, ocupa 4 bytes, el sumar 1 a `p` implica añadir 4 bytes a la dirección que contiene.
- Mientras que si `p` apuntase a un tipo `double`, implicaría añadirle 8 bytes.
- Tiene sentido también la diferencia de punteros al mismo tipo de variable. El resultado es la distancia entre las direcciones de las variables apuntadas por ellos, no en bytes, sino en datos del mismo tipo.



RELACIÓN ENTRE PUNTEROS Y VECTORES

56

- Existe una relación muy estrecha entre los vectores y los punteros.
- De hecho el nombre de un vector es un puntero a la dirección de memoria que contiene el primer elemento del vector.

```
double V[10];
```

```
double *p;
```

```
...
```

```
p=&V[0];
```

```
...
```

- El identificador **V** es un puntero al primer elemento del vector.
- Por tanto, es lo mismo **V** que **&V[0]**



RELACIÓN ENTRE PUNTEROS Y VECTORES

57

- Dado que el nombre de un vector es un puntero, se regirá por las leyes de la aritmética de punteros, esto es,

V apunta a V[0]

(V+1) apunta a V[1]

(V+i) apunta a V[i]

- Y al revés

***V es igual a V[0]**

***(V+1) es igual a V[1]**

***(V+i) es igual a V[i]**



DEFINICIÓN DINÁMICA DE VECTORES

58

- ❑ Definimos una variable de tipo puntero a int

```
int *mi_vector;
```

- ❑ Reservamos espacio para los n elementos

```
mi_vector = ( int * ) malloc ( n * sizeof(int) );
```

(esto crea el espacio para las variables mi_vector[0] ... mi_vector[n-1],
que son enteros)

- ❑ Una vez concluidas las operaciones con el vector, debe liberarse la memoria con la sentencia:

```
free (mi_vector);
```

- ❑ Un buen programador nunca deja la memoria ocupada con variables inútiles



RELACIÓN ENTRE PUNTEROS Y MATRICES

59

- En el caso de las matrices la relación es más compleja.

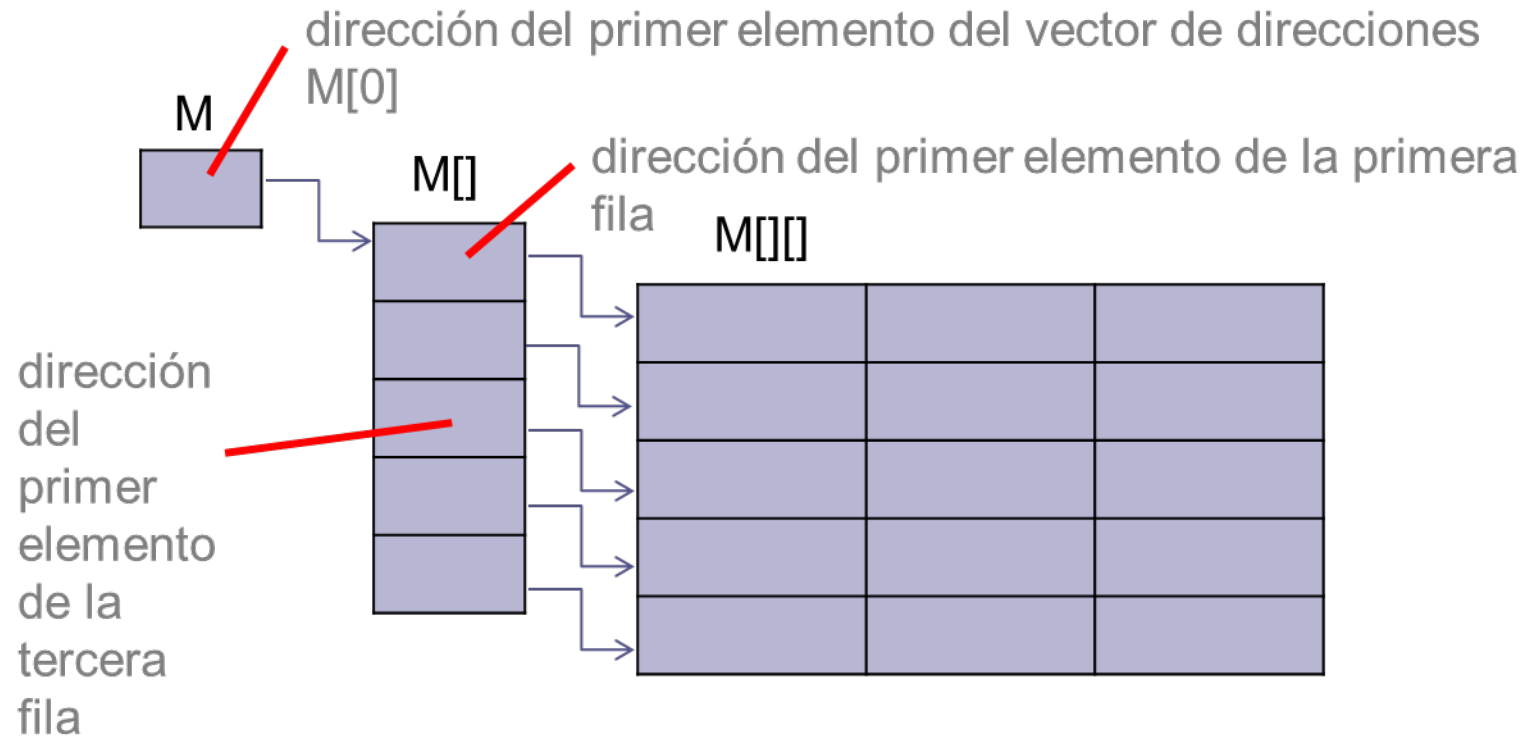
```
int M[5][3];
```

- El nombre de la matriz (**M**) es un puntero al primer elemento de un vector de punteros **M[]**, cuyos elementos contienen las direcciones del primer elemento de cada fila de la matriz.
- El nombre **M** es por tanto un puntero a puntero.
- Veamos un dibujo explicativo:



RELACIÓN ENTRE PUNTEROS Y MATRICES

60



EJEMPLOS

61

Diferentes formas de acceder al elemento $M[i][j]$

$M[i][j]$

$*(M[i] + j)$

$*(*(M + i) + j)$



DEFINICIÓN DINÁMICA DE MATRICES

62

- ❑ Definimos una variable de tipo puntero a puntero a int

```
int **mi_matriz
```

- ❑ Reservamos espacio para las n filas (espacio para n punteros a entero)

```
mi_matriz = ( int ** ) malloc ( n * sizeof( int * ) );
```

(esto crea el espacio para las variables `mi_matriz[0]` ... `mi_matriz[n-1]`, que son punteros a entero)

- ❑ A continuación, para cada fila, reservamos espacio para las m columnas

```
for (int i=0;i<n;i++) mi_matriz[i] = ( int * ) malloc ( m * sizeof( int ) );
```

(esto crea el espacio para las componentes `mi_matriz[i][j]` que son enteros)

- ❑ Una vez concluidas las operaciones con la matriz, liberamos memoria del siguiente modo

```
for (int i=0;i<n;i++) free(mi_matriz[i]);
```

```
free(mi_matriz);
```



PASO DE PARÁMETROS: POR VALOR

63

- ❑ **Paso por valor:** hace una copia del argumento en el inicio de la función. Es equivalente a tener, una declaración que asigna el valor de cada argumento a el parámetro correspondiente.

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

```
void CAMBIA (int , int );
```

```
int main() {
```

```
int n=10,m=5;
```

```
CAMBIA (n,m);
```

```
printf("\n\n En el main: n=%d y m=%d \n\n",n,m) ;
```

```
return 0;
```

```
}
```



PASO DE PARÁMETROS: POR VALOR

64

```
void CAMBIA (int n, int m) {  
    int aux;  
    aux=m;  
    m=n;  
    n=aux;  
    printf("\n\n Dentro de la funcion: n=%d y m=%d \n\n",n,m) ;  
    return 0;  
}
```

Resultado: Los valores NO se intercambian fuera de la función CAMBIA



PASO DE PARÁMETROS: POR REFERENCIA

65

- ❑ **Paso por referencia:** el parámetro es el mismo objeto (variable), es decir, el parámetro se convierte en un alias.

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

```
void CAMBIA (int *, int *);
```

```
int main() {
```

```
int n=10,m=5;
```

```
CAMBIA (&n, &m);
```

```
printf("\n\n En el main: n=%d y m=%d \n\n",n,m) ;
```

```
return 0;
```

```
}
```



PASO DE PARÁMETROS: POR REFERENCIA

66

```
int CAMBIA (int *n, int *m) {  
    int aux;  
    aux=*m;  
    *m=*n;  
    *n=aux;  
    printf("\n\n Dentro de la funcion: n=%d y m=%d \n\n",*n,*m) ;  
    return 0;  
}
```

Resultado: Los valores SÍ se intercambian fuera de la función CAMBIA



PROGRAMAS DE EJEMPLO

67

Los programas

- ❖ Programa9_aritmetica_punteros.c
- ❖ Programa10_punteros_vector_dinamico.c
- ❖ Programa11_punteros_matriz_dinamica.c
- ❖ Programa12_paso_por_valor.c
- ❖ Programa13_paso_por_referencia.c

que puedes descargar en el Campus Virtual, contienen ejemplos sobre la utilización de punteros.

