

**Índice**

Definiciones …………………………………………………….. 3

Linux Mint ………………………………………………………. 6

MakeFile ………………………………………………………… 10

Código básico de inicio en C …………………………………. 11

Estructuras de control …………………………………………. 12

Funciones y Procedimientos …………………………………. 13

Pasaje de parámetros (**&** y **\***) …………………………………. 14

Bibliotecas (**.h** y **.c**) …………………………………………….. 16

Vectores ………………………………………………………… 18

Referencia automática ……………………………………………. 19

Matrices ………………………………………………………… 21

Matriz como parámetro sin filas definidas ……………………… 21

Strings ………………………………………………………….. 22

Estructuras (**struct**) …………………………………………… 24

Métodos varios en vectores ………………………………….. 27  
 Métodos de ordenamiento ………………………………………. 27  
 Métodos de búsqueda ……………………………………………. 30  
 Métodos de eliminación ………………………………………….. 31

Recursividad ……………………………………………………. 32  
 Búsqueda binaria recursiva ………………………………………. 33

Operaciones en vectores ……………………………………… 34  
 Mezcla ……………………………………………………………… 35  
 Union ………………………………………………………………… 36  
 Diferencia …………………………………………………………… 37  
 Intersección ………………………………………………………… 38

Argumentos …………………………………………………….. 39

Archivos ………………………………………………………… 40  
 config ……………………………………………………………….. 42  
 CSV …………………………………………………………………. 45

Definiciones

**Algoritmo**

Serie ordenada, finita y precisa de acciones que resuelven un problema.

Siempre se debe obtener el mismo resultado si se tienen los mismos datos de entrada (como una receta de cocina).

**Programa**

Traducción de un algoritmo a un lenguaje de programación determinado capaz de ser ejecutado por una computadora.

Un programa es un algoritmo, pero no todo algoritmo es un programa.

* **Datos de entrada:** Toda información que llega al algoritmo.
* **Datos de salida:** Información que sale del algoritmo.

**Pre-condiciones**

Todas las condiciones que el algoritmo asume que cumplen sobre los datos de entrada.

**Pos-condiciones**

Todas las condiciones que va a cumplir el resultado del algoritmo y sólo se pueden cumplir si se cumplen las pre-condiciones.

**Caja negra**

Se observan sólo los datos de entrada y los datos de salida para comprobar el funcionamiento del algoritmo.

Se centra en qué hace el algoritmo y se preocupa en cumplir las pre-condiciones.

**Caja blanca**

Se observa el algoritmo desde dentro para ver todo su funcionamiento.

Se centra en cómo se hace el algoritmo y se preocupa por cumplir las pos-condiciones.

**Variables**

Son como cajas que guardan información que puede variar durante el algoritmo.

**Constantes**

Son variables pero cuya información no varía.

**Tipos de dato**

Conjunto de todos los valores que puede tomar una variable de ese tipo de dato.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Enteros**  (numérico) | short |  | ±32.767 | 2 bytes |
| int | %i | ±2.147.483.647 | 4 bytes |
| long |  | ±2.147.483.647 | 4 bytes |
| unsigned | Establece que sólo se puedan utilizar números positivos, duplicando el alcance máximo del rango | | |
|  |  |  |  |  |
| **Reales**  (numérico) | float | %f | 6 decimales | 4 bytes |
| double | %d | 15 decimales | 8 bytes |
| long double | 19 decimales | 10 bytes |
|  |  |  |  |  |
| **Caracteres** | char | %c | ASCII | 1 byte |
|  |  |  |  |  |
| **Lógicos** | bool |  | true *-* false |  |

* **Datos ordinales**

Tipos de datos ordenados: se conoce el dato que precede o el que le sigue.

**Operadores**

Símbolo que indica que debe ser llevado a cabo una operación especifica sobre cierto número de operandos (tipo de dato).

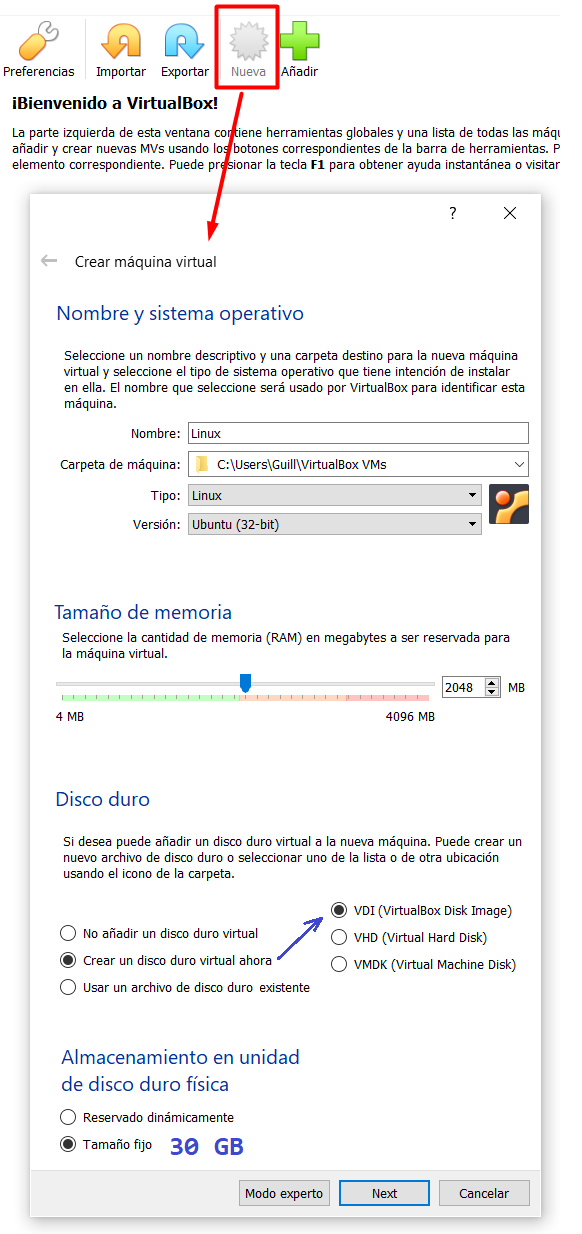
* Asignación (=): asigna un valor a un operando.
* Aritméticos (+ - \* / %) resto de división entera.
* Relacionales (> >= < <= == !=): compara dos valores del mismo tipo y devuelve un valor lógico.
* Lógicos:
  + **And &&** Si todo es verdadero, retorna verdadero.
  + **Or ||** Si al menos uno es verdadero, retorna verdadero.
  + **Not !** Convierte lo verdadero en falso y viceversa.

Scope (ámbito)

Ciclo de vida de las variables o “alcance” de las variables.

Las variables creadas dentro de unas llaves, mueren al cerrarse esas llaves.

* Globales: se declaran fuera del main y al principio (constantes).
* Locales: se declaran dentro del main o entre llaves.

Linux Mint

Lo primero a tener en cuenta es que se debe descargar el VirtualBox Oracle (e instalarlo) y el sistema operativo en formato ISO de 32 bits.

<https://www.oracle.com/virtualization/technologies/vm/downloads/virtualbox-downloads.html>

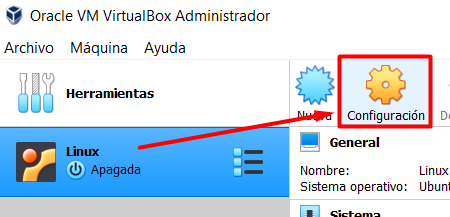
<https://www.linuxmint.com/download.php>

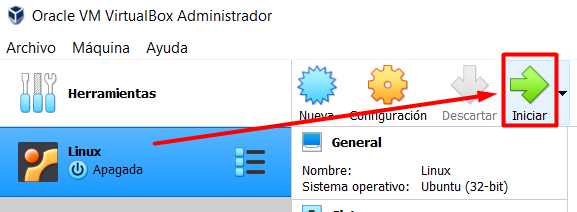
Luego, y antes de iniciar, se debe entrar a la BIOS de la computadora y activar la virtualización por hardware. Ver tutoriales por internet para su computadora.

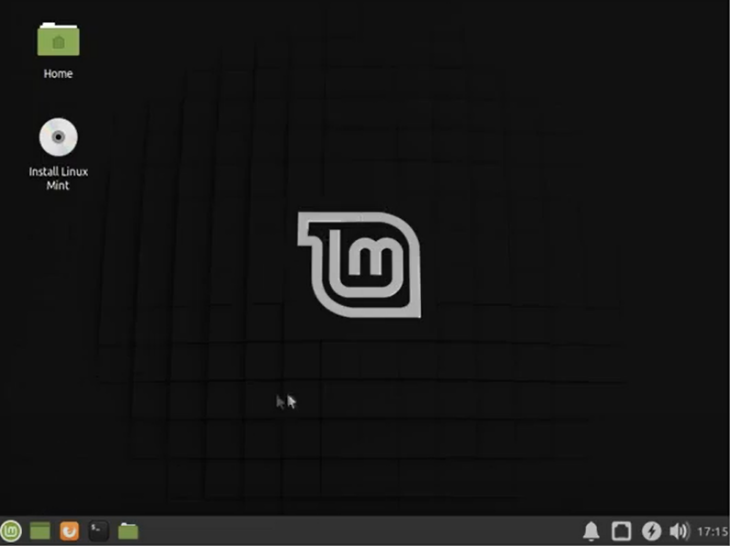
Una vez hecho lo anterior, se procede a crear la máquina virtual desde VirtualBox Oracle.

Haciendo clic en “Nueva” y configurando tal como se ve en la imagen de la derecha.

Una vez creado, se procede a configurarlo de la siguiente manera:





Realizados todos los pasos anteriores, se inicializará Linux Mint.

A continuación, se debe ingresar al “disco” llamado “Install Linux Mint” y configurarlo.

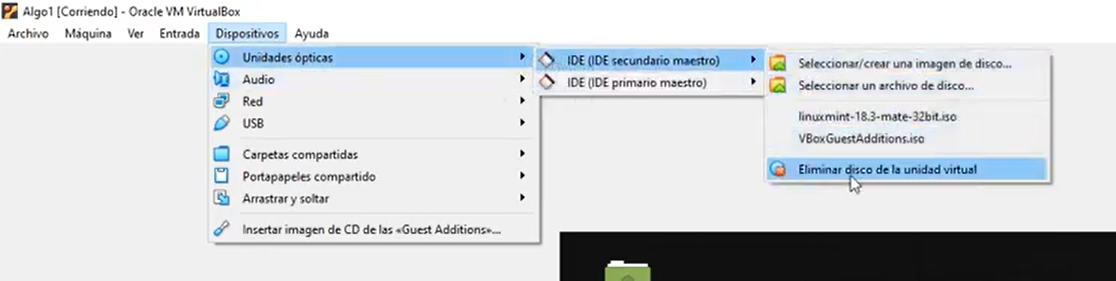
Se recomienda no instalar software de terceros para archivos mp3 y etc.

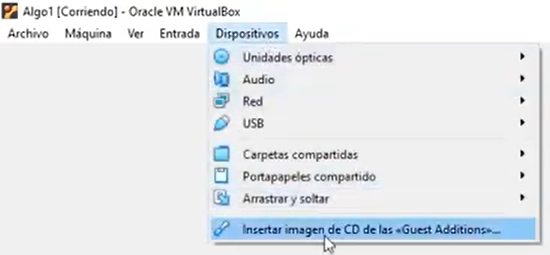
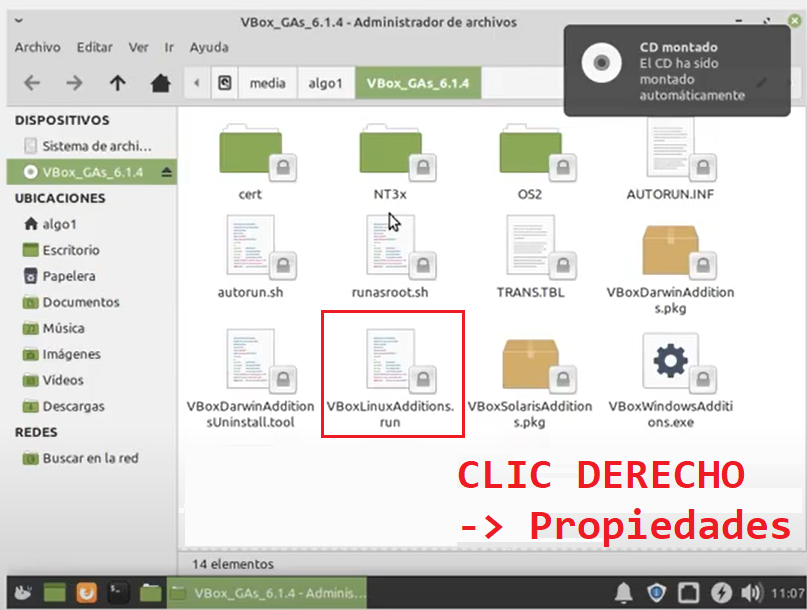
Con el tipo de instalación, seleccionar la primera opción:



Una vez finalizada la instalación y registro del usuario y contraseña (importante no olvidarla). Se procede con la instalación de Guest Additions para tener una mejor resolución y buen funcionamiento.

1. Eliminar discos de las unidades virtuales (si es que los hay)



1. Insertar ISO de las Guest Additions: **VBoxLinuxAdditions.run**

Finalmente, se debe ejecutar el archivo **VBoxLinuxAdditions.run** con la consola de Linux.

Para ello, se abre una terminal y se ejecutan los siguientes códigos, en orden:

* cd /
* cd media
* cd NOMBRE\_USUARIO
* ls
* cd VBox\_Gas\_6.1.4/
* sudo ./VBoxLinuxAdditions.run

Luego, reiniciar la máquina virtual.

Ya se podrán notar los cambios en la resolución de la pantalla.

Comandos de Linux

* ls lista el contenido del directorio
* cd ingresa a un directorio
  + **..** vuelve una carpeta hacia “atrás”.
* clear limpia la terminal (también funciona con Ctrl + L)
* gcc permite compilar un archivo C.
* **./** ejecuta un archivo C compilado.
  + Ctrl + C interrumpe la ejecución del programa.



muestra todos los warnings

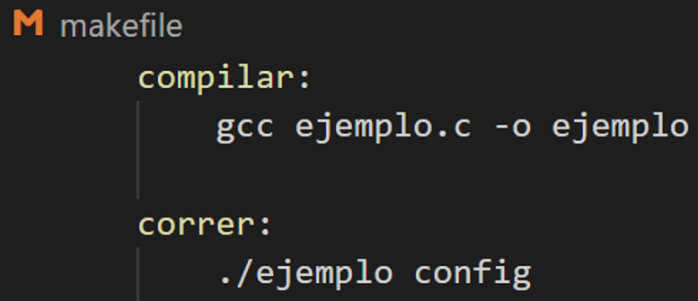
convierte todos los warnings en errores

muestra warnings de conversión de tipos de datos

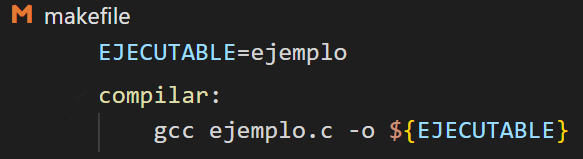
permite elegir el nombre del archivo compilado final

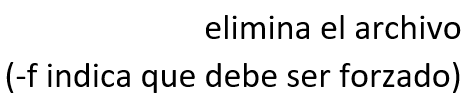
**MakeFile**

Permite crear comandos o atajos dentro de la pantalla de usuario.

Para hacerlo, se crea un archivo llamado “makefile” con las instrucciones necesarias.

Incluso se pueden crear variables.



De esta manera, se pueden agilizar los comandos de compilación de un archivo.

EJECUTABLE=ejemplo

FLAGS=-Wall Werror -std=c99 -Wconversion

CONFIG=config

compile:

    gcc ejemplo.c -o ${EJECUTABLE}

compile-flags:

    gcc ejemplo.c -o ${EJECUTABLE} ${FLAGS}

run:

    ./${EJECUTABLE} ${CONFIG}

compile-run:

    compile run

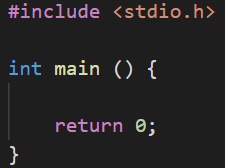
clean:

    rm -f ${EJECUTABLE}

all:

    clean compile-flags run

Código básico de inicio en C



Estructuras de control

Condicionales

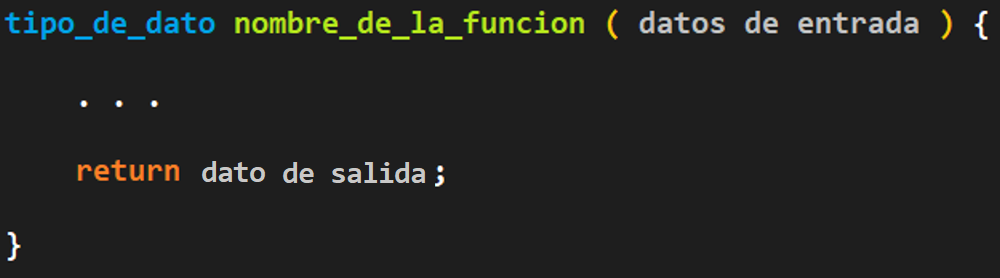
Ejecutan un bloque de código si se cumple una condición (if, if-else, switch).

Iterativas

Ejecutan un bloque de código varias veces (while, for, do-while).

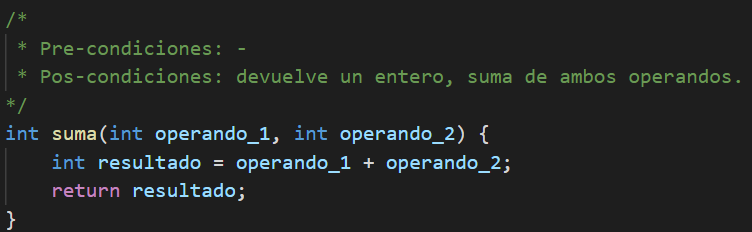
Funciones y procedimientos

Bloque de código el cual se puede utilizar declarándolo fuera del main (modularizar).



* **Funciones**

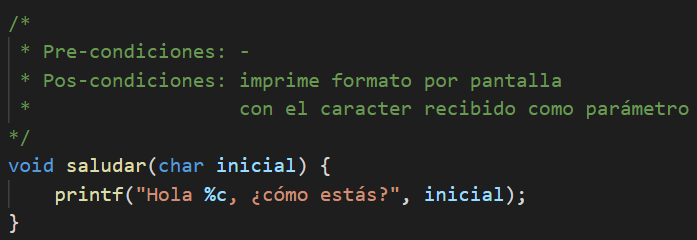
Se utilizan para realizar una acción específica.



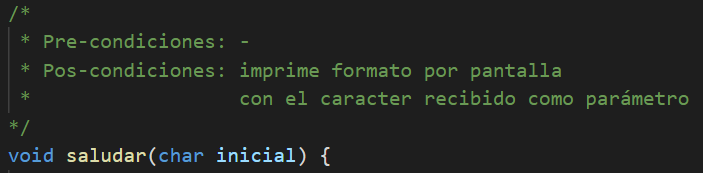
* **Procedimientos (void)**

Realizan acciones sin devolver ningún resultado.

Se nombran con verbos.



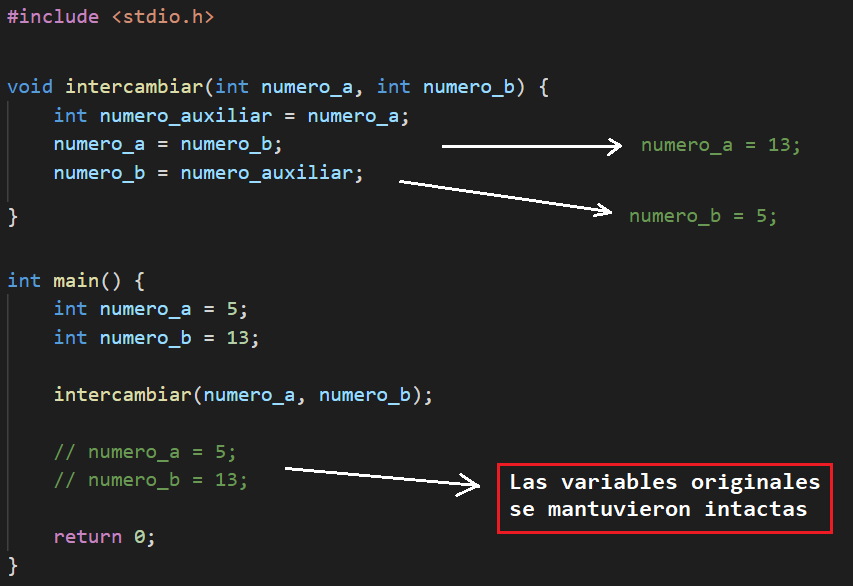
Ninguna función o procedimiento debe recibir más de 5 o 7 parámetros.



Pasaje de parámetros

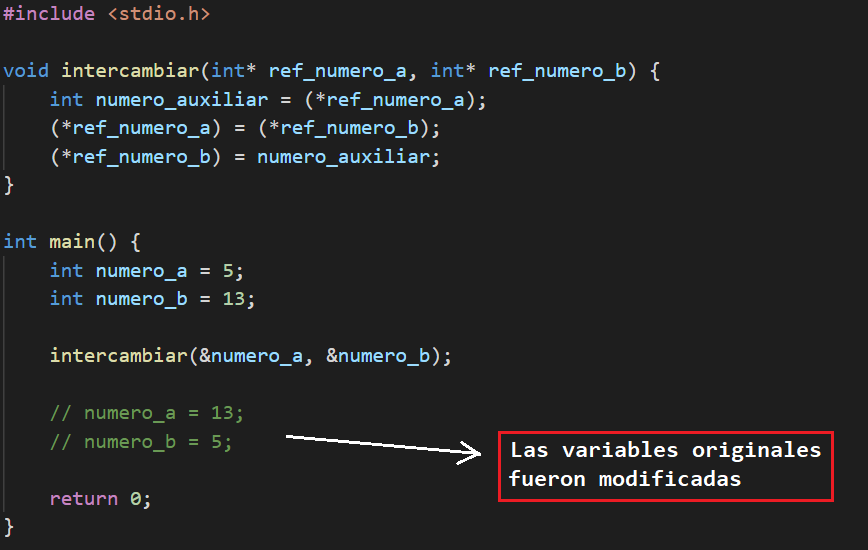
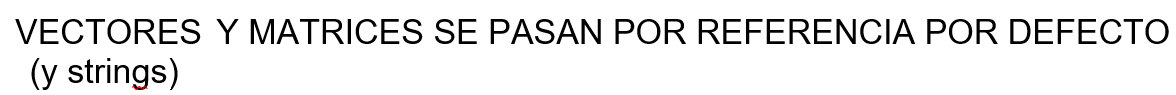
Por valor

No cambia el valor de las variables originales, sino de las copias creadas en la firma de la función / procedimiento.



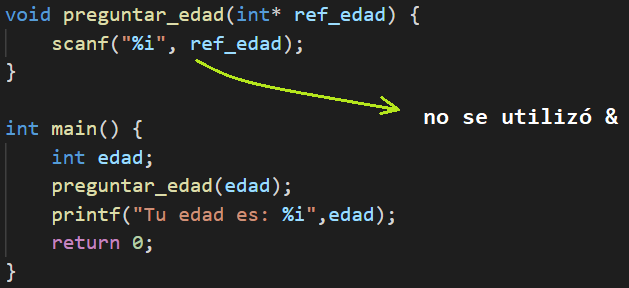
Por referencia

Utilizando los símbolos & y \* se pueden utilizar referencias de variables.

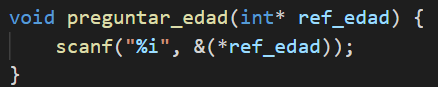


La función **scanf** utiliza pasaje de parámetros por referencia, por eso, se debe utilizar así:

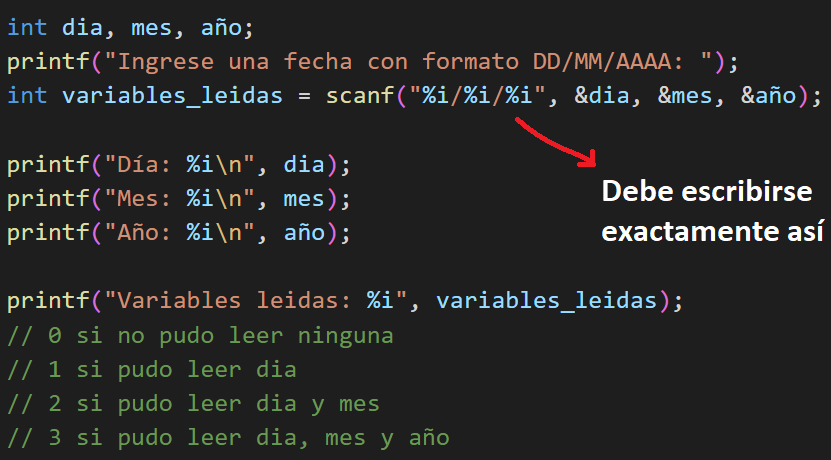
Sin embargo, hay que tener cuidado cuando se utiliza la función **scanf** dentro de un procedimiento donde ya hay un pasaje de parámetros por referencia. En dicho caso, no se vuelve a utilizar la llave &.



También pudo haberse escrito de la siguiente manera:



Mediante la función **scanf** se pueden obtener más de una variable a la vez. Y dicha función devuelve la cantidad de variables que pudo llenar.



Biblioteca

Conjunto de dos archivos que permiten encapsular cierta funcionalidad que resuelve un problema y permite usar esas funciones en un programa.

* **Header**

Archivo de extensión **.h** con las funciones públicas declaradas, pero donde no se muestra cómo funciona. Similar a “caja negra”.

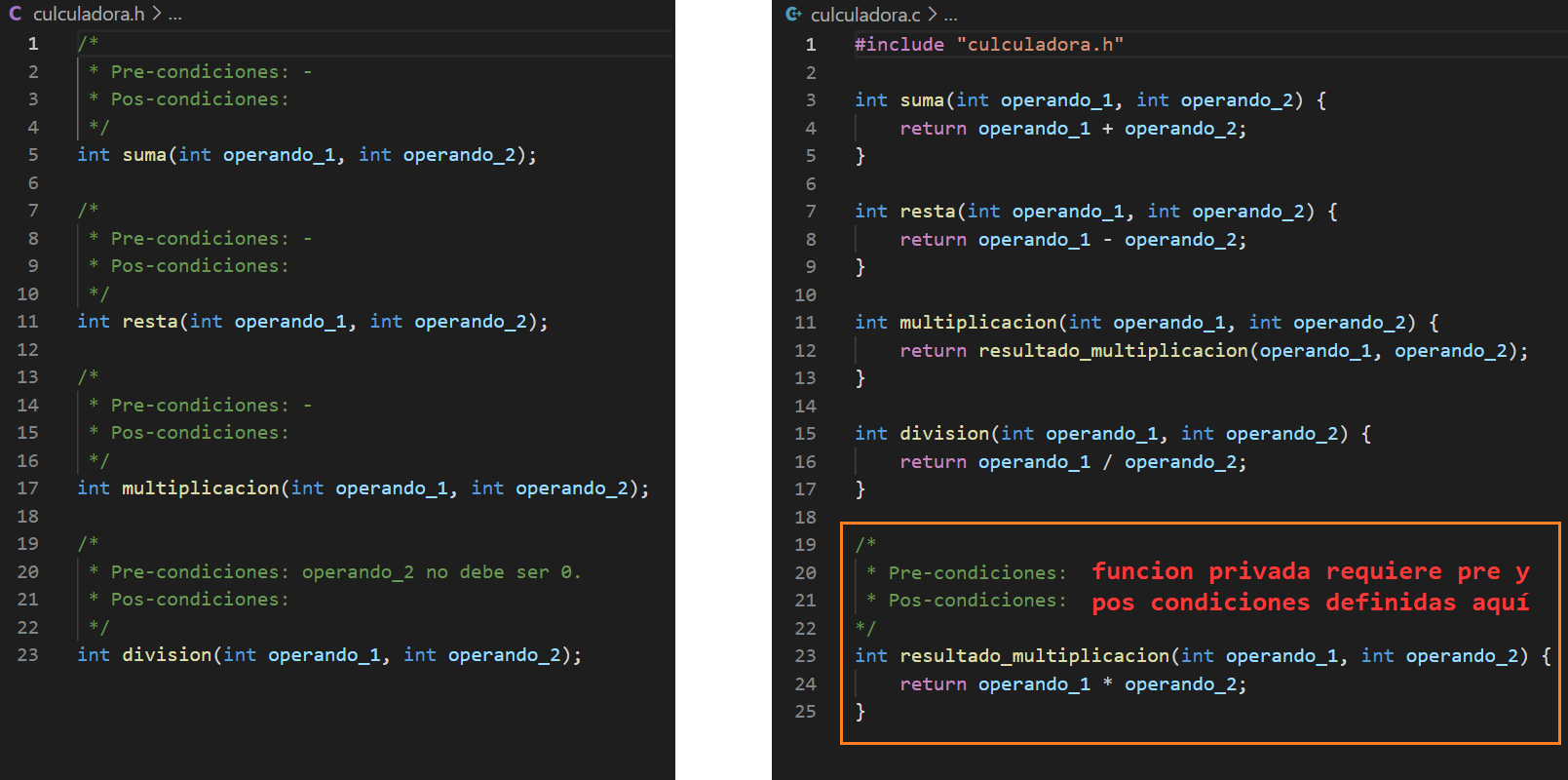
En este archivo no se pueden declarar variables.

* **Body**

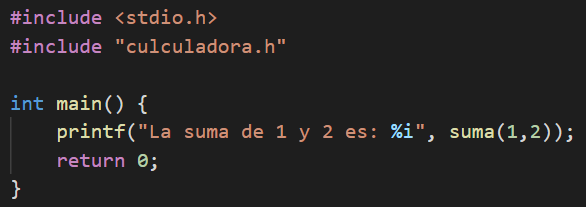
Archivo de extensión **.c** con las funciones previamente declaradas en el header, definiendo cómo funcionan. Similar a “caja blanca”.

Debe contener el #include “*header.h*” (donde header será el nombre de la biblioteca).

Se pueden agregar funciones privadas (funciones declaradas en el body pero no en el header), las cuales no pueden ser utilizadas directamente por el programa donde se implementará la biblioteca.



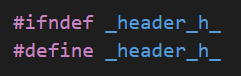
Para poder utilizar una biblioteca en un programa, se debe agregar el header mediante un #include.



El nombre de nuestra biblioteca se escribe entre comillas dobles debido a que es una biblioteca local.

Para compilar un programa junto con las bibliotecas, se debe hacer todo junto:



**Guard Block**

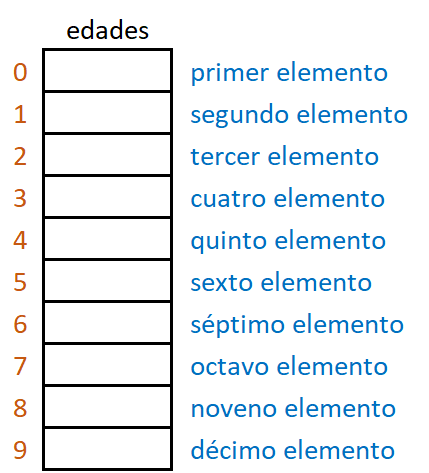
Permite la declaración de variables en el header de una biblioteca.

Para ello, se debe agregar lo siguiente al inicio de nuestro header.

Al final del código, se agrega:

Si se quiere que la variable sea constante, se debe agregar **static** al inicio.

Vectores

Es un conjunto contiguo de datos (variables) de un mismo tipo.

Su nombre debe estar escrito en plural ya que contiene muchas cosas de algo específico.

Se declara con el siguiente formato:





Ejemplo:

En este caso, se crea un vector con 120 elementos.

Para asignar un valor a un determinado elemento, se debe hacer mención al número de su posición.

de esta manera, asigna el valor 3 a la posición 10 del vector.

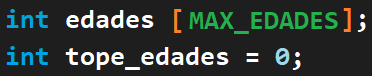
Para asignar valores, la posición del vector no puede ser superior a la cantidad de posiciones declaradas del vector (capacidad del vector) ni un número negativo.

Capacidad del vector

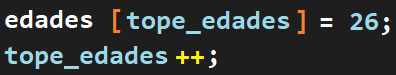
La capacidad del vector, la cual se define entre los **[ ]** al declarar la variable vector, debe establecerse mediante una constante.

Sin embargo, declarar una variable constante para asignarla como capacidad de un vector, es inválido para el lenguaje de programación C. Por lo tanto, se utiliza un #define.

Tope

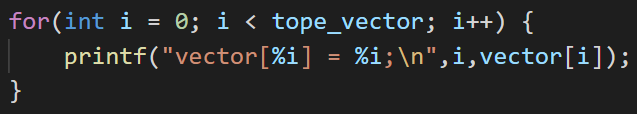
Siempre que se crea un vector, se debe crear una variable que indique la cantidad de elementos asignados en el vector.

Además, la variable *tope* nos indica cuál es la siguiente posición del vector sin asignar.

Por lo tanto, para asignar un valor en la siguiente posición del vector sin asignar, se puede utilizar la variable *tope* y luego actualizarla.

Iteración

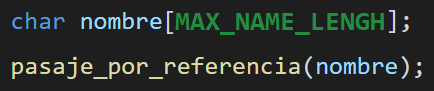
Para recorrer un vector, se utiliza un **for** de la siguiente manera:



Referencia automática

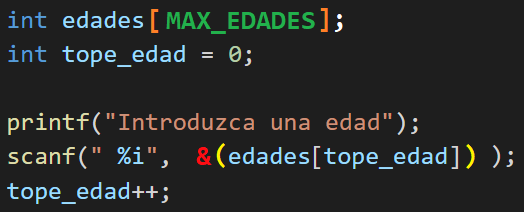
* Pasaje por referencia de un vector

Por defecto, todos los vectores, matrices y strings se pasan por referencia sin necesidad de escribir & ni \*.

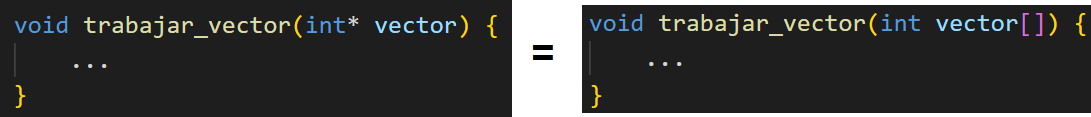


* Pasaje por referencia de un elemento de un vector

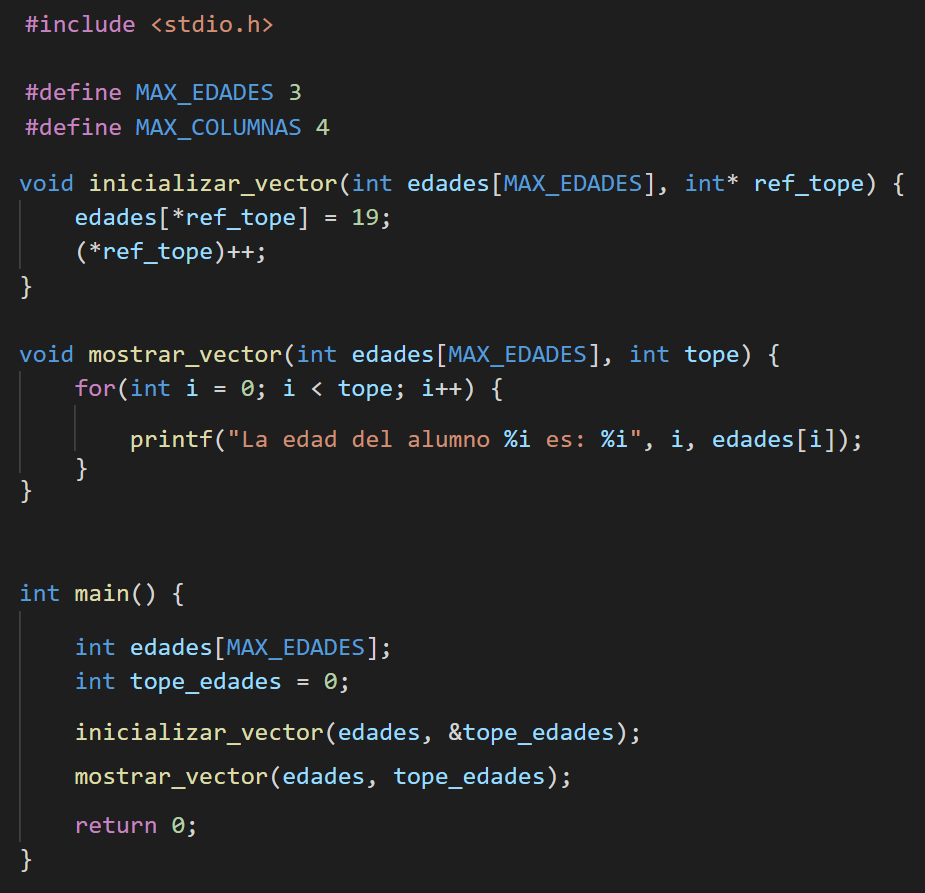
Cuando se utiliza la función **scanf** o cualquier procedimiento que requiera un pasaje de parámetros por referencia de un solo elemento de un vector, se debe escribir de la siguiente manera:

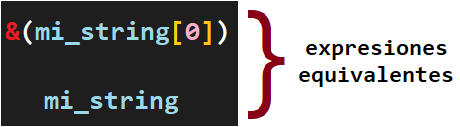


Parámetro sin tamaño “físico” específico

Cuando se desea crear una función o procedimiento que contenga un vector como parámetro, y a dicho vector no se le conoce el tamaño específico que tendrá, se utiliza el \* de la siguiente forma.

**TODOS LOS VECTORES SE ENVÍAN COMO PARAMETROS POR REFERENCIA**



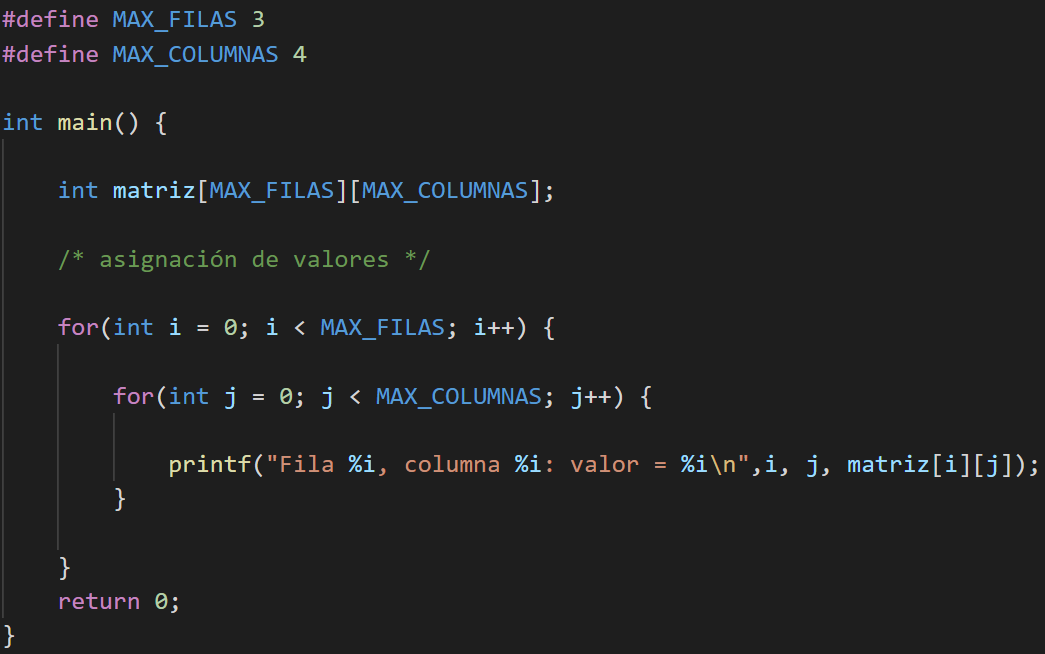


Matriz

Una matriz es un vector de vectores

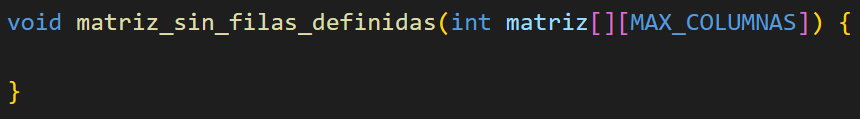
La matríz lleva el nombre de lo que representa (no en plural)

Int matriz [MAX\_FILAS] [MAX\_COLUMNAS] ;



Pasaje de una matriz por parámetros

Cuando se pasa una matriz, no es necesario definir la cantidad de filas (se puede no definir el MAX\_FILAS).



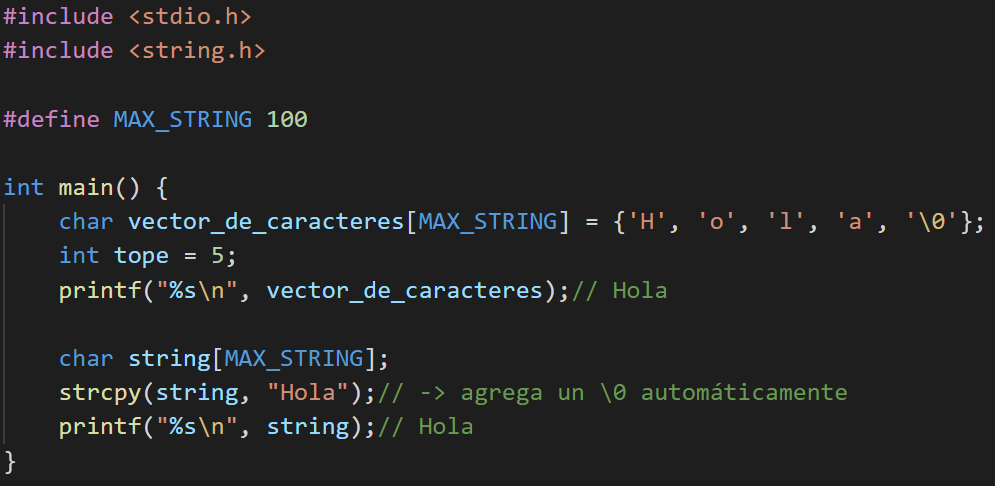
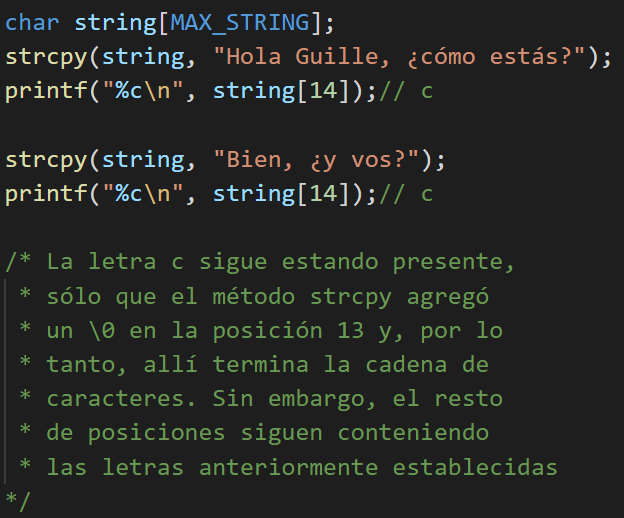
Strings

Un string es una cadena de caracteres.

Se declara, en código, igual que un vector de caracteres (estructuralmente son iguales). Sin embargo, un vector de caracteres no es lo mismo que un string, ya que un string debe tener un \0 en su tope y sus funciones reales son diferentes.

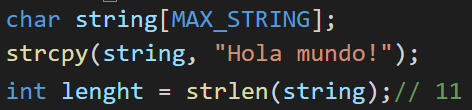
Caracteres especiales:

* \0 Tope de una cadena de caracteres.
* \n Salto de línea.

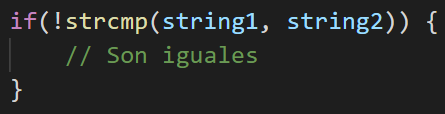


El método **strcpy** se puede usar sobre un string (vector de caracteres cuya funcionalidad es formar una cadena de texto) para asignarle texto.

Si se vuelve a utilizar el método strcpy, sólo trabaja con las posiciones del vector donde necesita trabajar y no “limpia” toda la variable.

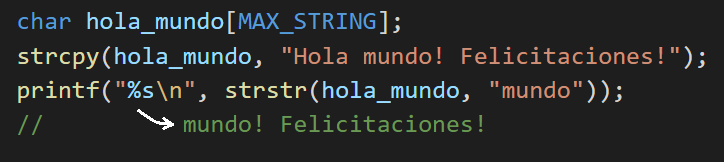
El método **strlen** devuelve la cantidad de carateres de un string.

El método **strcmp** compara dos cadenas de texto y devuelve un **int**.

* 0 Las cadenas de texto son iguales.
* 1 La primera cadena de texto es más larga.
* -1 La segunda cadena de texto es más larga.

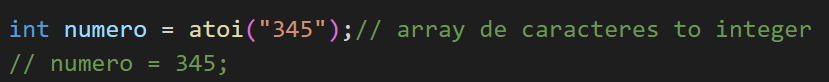
El método **strstr** busca una cadena de caracteres dentro de un string.

* Si se encuentra, devuelve una subcadena de caracteres desde donde empieza la cadena de caracteres dentro del string buscado.



* 0 Si no se encuentra

La funcion **atoi** convierte un array de caracteres (string) en un integer.



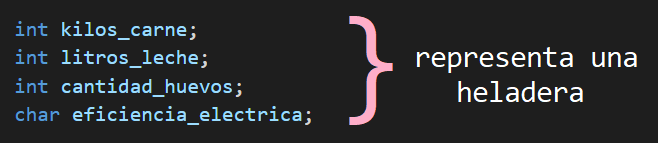
Registros (**struct**)

Recordando que las variables son “cajas” donde se guarda información, y dicha variable representa algo (definido en el nombre de la variable).

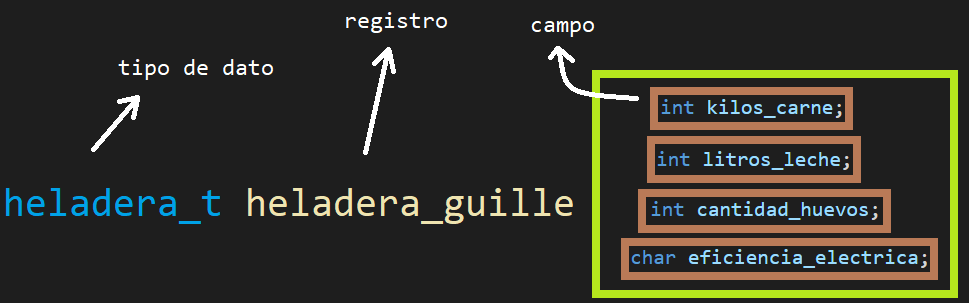
Ejemplo:

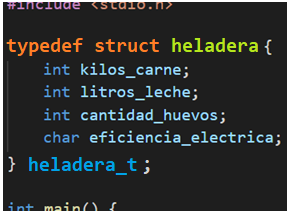
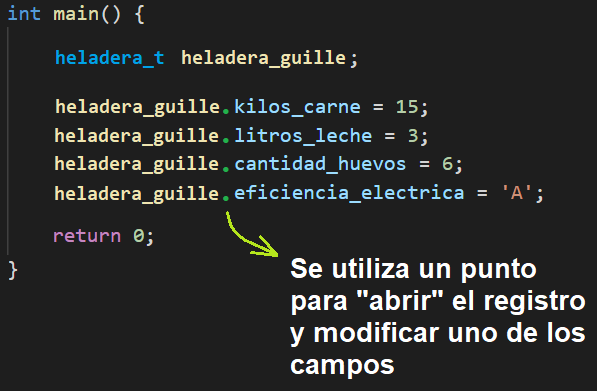
Dicha variable representa la edad de una persona.

Otro ejemplo es la representación de una heladera mediante más de una variable.

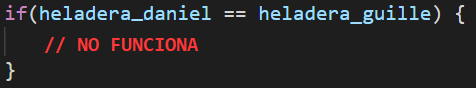


Para estos casos, en que muchas variables representan una sola cosa, se utilizan los registros (**structs**).



Para poder hacer uso de los registros, primero se debe crear el tipo de dato específico para nuestro tipo de registro y luego se pueden inicializar los campos.

Para ello, procedemos de la siguiente manera.

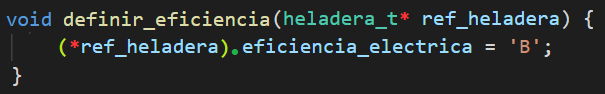
Los registros no pueden compararse.



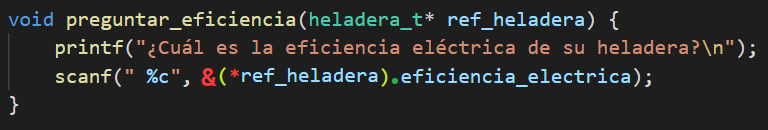
Sin embargo, si se puede usar la asignación.

Pasaje de parámetros por referencia en registros

Se debe aclarar, con paréntesis, cuál es la referencia para que el compilador no se “confunda”.



Para usar **scanf**, se debe aclarar bien con paréntesis, dónde está la llave &.



Estructuras anidadas

Se trata de estructuras dentro de otras.

#include <stdio.h>

const char ROCK = 'R';

const char POP = 'P';

const char CUMBIA = 'C';

typedef struct genero {

    char nombre;

} genero\_t;

typedef struct cancion {

    int id;

    int duracion;

    genero\_t genero;

} cancion\_t;

typedef struct album {

    int id;

    cancion\_t cancion;

    genero\_t genero;

} album\_t;

int main() {

    genero\_t genero\_rock, genero\_pop, genero\_cumbia;

    genero\_rock.nombre = ROCK;

    genero\_pop.nombre = POP;

    genero\_cumbia.nombre = CUMBIA;

    cancion\_t civilizacion;

    civilizacion.id = 1;

    civilizacion.duracion = 140;

    civilizacion.genero = genero\_rock;

    album\_t album;

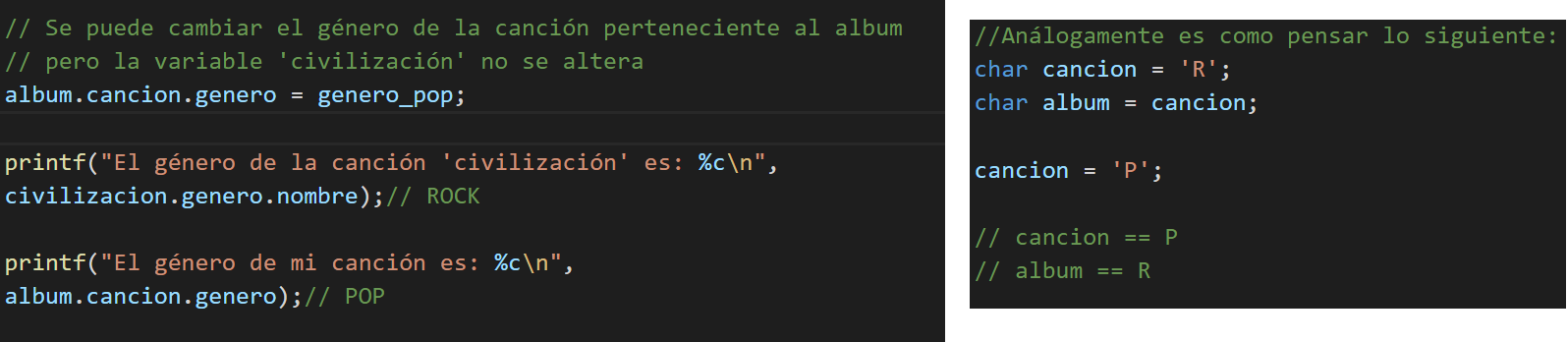
    album.id = 1;

    album.cancion = civilizacion;

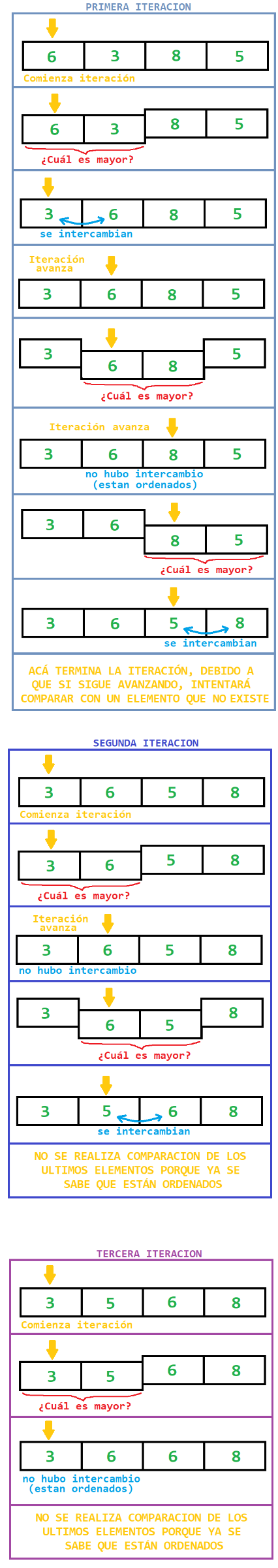
    printf("El género de la canción dentro del álbum es: %c\n",

    album.cancion.genero.nombre);// ROCK

}



Métodos varios

Métodos de ordenamiento

* Burbujeo

Se crea una iteración, la cual selecciona los dos primeros elementos y los compara para ver cuál es mayor. Al mayor lo “empuja” a la posición más alta de entre esos dos elementos (es decir, se intercambian de lugar). Luego se avanza la iteración en +1, seleccionando otros dos elementos y repitiendo el proceso. Así sucesivamente hasta el final del vector, quedando sólo el elemento mayor al final del vector.

Luego, se repite toda la iteración desde el principio (desde los primeros elementos) hasta que quede todo completamente ordenado.

En las siguientes iteraciones a la primera, se puede obviar la última pareja de elementos, debido a que ya se sabe que la del final será el elemento mayor.

La cantidad de iteraciones debe ser un número menor a la cantidad de elementos del vector como máximo.

void ordenar\_por\_burbujeo(int vector[MAX\_ELEMENTOS], int tope) {

    for(int j = 0; j < tope - 1; j++) {

        for(int i = 0; i < tope - 1 - j; i++) {

            if(vector[i] > vector[i+1]) {

                int auxiliar = vector[i];

                vector[i] = vector[i+1];

                vector[i+1] = auxiliar;

            }

        }

    }

}

Este simbolo establece si se ordena de mayor a menor o si se ordena de menor a mayor.

i indica la cantidad de elementos ordenados en todo momento

* Selección

Se realiza una iteración en todo el vector para buscar cuál es el elemento más pequeño (el mínimo) y recordar su posición.

Terminada la iteración y teniendo su posición, se intercambia la posición con el primer elemento del vector (por ser la más pequeña de todas).

A partir de la siguiente iteración, nos ubicamos en el segundo elemento (o el que corresponda) del vector y repetimos todo el proceso. Una vez encontrado el más pequeño (sin tener en cuenta los elementos anteriores al que nos ubicamos), lo volvemos a intercambiar en dicha ubicación.

Es decir, en cada iteración busca cuál es el mínimo para ubicarlo en la posición que le corresponde.

La cantidad de iteraciones debe ser un número menor a la cantidad de elementos del vector.

void ordenar\_por\_seleccion(int vector[MAX\_ELEMENTOS], int tope) {

    for(int i = 0; i < tope - 1; i++) {

        int posicion\_minimo = i;

        for(int j = i + 1; j < tope; j++) {

            if(vector[j] < vector[posicion\_minimo]) {

                posicion\_minimo = j;

            }

        }

        int auxiliar = vector[i];

        vector[i] = vector[posicion\_minimo];

        vector[posicion\_minimo] = auxiliar;

    }

}

* Inserción

Se tiene un vector ordenado y se debe agregar un elemento desordenado para ubicarlo en la posición que corresponda.

Cuando se agrega el elemento desordenado al final del vector, se compara con cada vector anterior, uno a uno, para comprobar si es mayor que el anterior. Si resulta que el elemento es mayor que su elemento anterior, entonces ya se encuentra ordenado.

La cantidad de iteraciones debe ser un número menor a la cantidad de elementos del vector.

void insercion(int vector[MAX\_ELEMENTOS], int tope) {

    for(int i = 1; i < tope; i++) {

        int aux = vector[i];

        int j = i;

        while(j > 0 && aux < vector[j-1]) {

            vector[j] = vector[j-1];

            j--;

        }

        vector[j] = aux;

    }

}

* Inserción ordenada

Agregar un elemento intentando no usar el **for**.

int aux = 3;

int j = tope - 1;

while (j >= 0 && vector[j] > aux) {

    vector[j+1] = vector[j];

    j--;

}

vector[j+1] = aux;

**En todos los procesos de ordenamiento, se requiere n2 operaciones. Donde n es la cantidad de elementos del vector.**

Métodos de búsqueda

* Lineal: Utilizando un **for** sobre todo el vector.

void búsqueda\_lineal(int valor\_buscado, int vector[MAX\_ELEMENTOS], int tope) {

    for(int i = 0; i < tope; i++) {

        if(vector[i] == valor\_buscado) {

            /\* Acciones \*/

        }

    }

}

* Búsqueda binaria *(el vector debe estar ordenado)*

Por mitades, permite ahorrarse buscar en absolutamente todo el vector.

void busqueda\_binaria(int valor\_buscado, int vector[MAX\_ELEMENTOS], int tope) {

    int inicio = 0, fin = tope-1;

    int centro = (inicio + fin) / 2;

    while (vector[centro] != valor\_buscado && inicio <= fin){

        if (vector[centro] > valor\_buscado){

            fin = centro - 1;

        } else if (vector[centro] < valor\_buscado){

            inicio = centro + 1;

        }

        centro = (inicio + fin) / 2;

    }

    if (inicio <= fin) {

        int posicion\_buscada = centro;

        /\* Acciones \*/

    }

}

Métodos de eliminacion

* Eliminación lógica

Se asigna un valor al elemento del vector que se sepa que su significado sea que dicho elemento está eliminado (por ejemplo: -1). El tope no se ve afectado.

void eliminacion\_logica(int valor\_buscado, int vector[MAX\_ELEMENTOS], int\* tope) {

    for(int i = 0; i < \*tope; i++) {

        if(vector[i] == valor\_buscado) {

            vector[i] = -1;

        }

    }

}

* Eliminación física (no ordenada)

El elemento del vector el cual quiere ser eliminado, se le asigna el valor del último elemento del vector, luego se baja el tope en 1 para que el último elemento ya no pertenezca al vector.

void eliminacion\_fisica\_desordenada(int valor\_buscado, int vector[MAX\_ELEMENTOS], int\* tope) {

    for(int i = 0; i < \*tope; i++) {

        if(vector[i] == valor\_buscado) {

            vector[i] = vector[(\*tope)-1];

            (\*tope)--;

        }

    }

}

* Eliminación física (ordenada)

Se busca el elemento del vector que desea eliminar, y a partir de ese elemento, se “mueven” todos los elementos subsiguientes al elemento anterior, para que todos los elementos siguientes se muevan una posición atrás y pueda eliminarse de forma ordenada. Finalmente, se reduce el tope en 1.

void eliminacion\_fisica\_ordenada(int valor\_buscado, int vector[MAX\_ELEMENTOS], int\* tope) {

    for(int posicion\_buscada = 0; posicion\_buscada < \*tope; posicion\_buscada++) {

        if(vector[posicion\_buscada] == valor\_buscado) {

            for(int i = posicion\_buscada; i < \*tope; i++) {

                vector[i] = vector[i+1];

            }

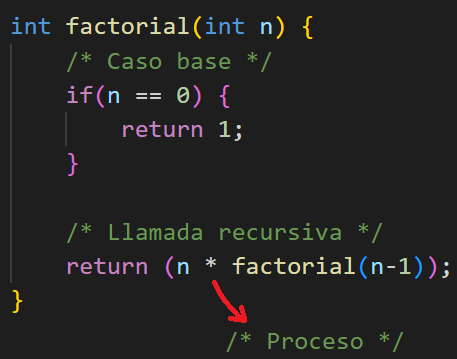
            (\*tope)--;

        }

    }

}

Recursividad

Un procedimiento recursivo está dividido en tres fases principales.

* **Caso base**

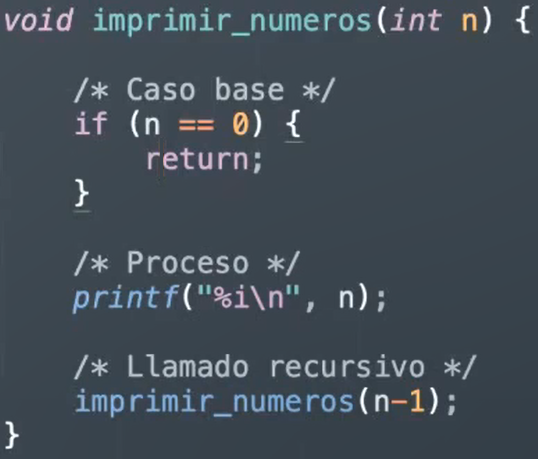
Cuando termina la recursión.

* **Proceso**

Valor agregado o acción de la función.

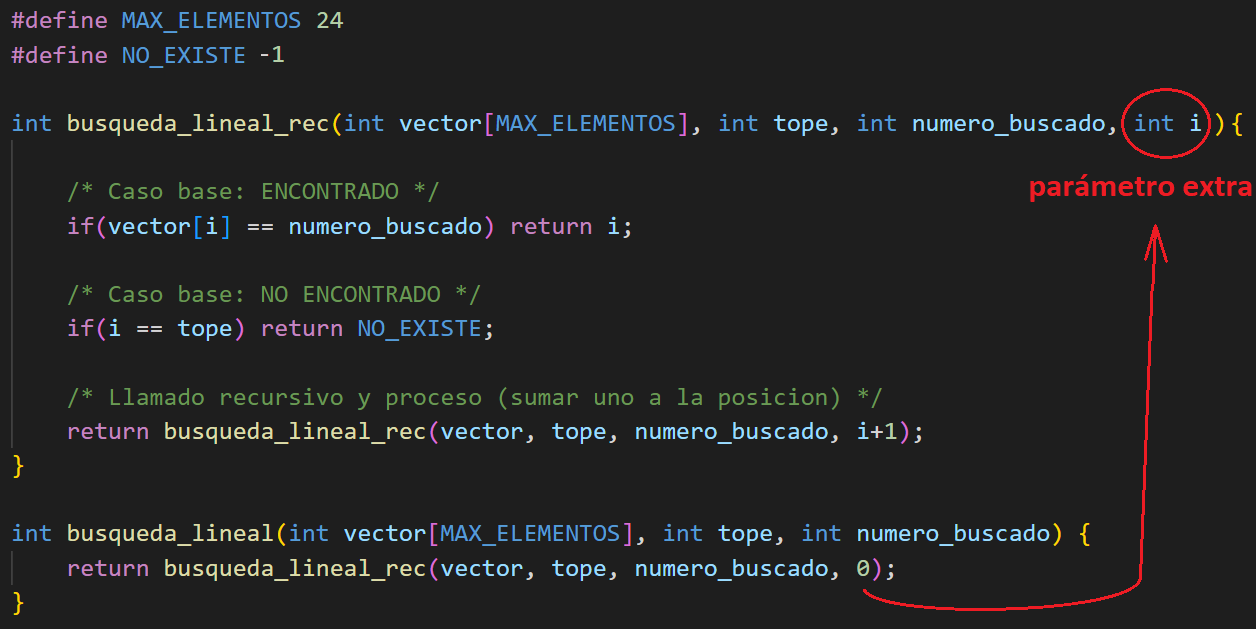
* **Llamada recursiva**

La función/procedimiento se llama a sí mismo



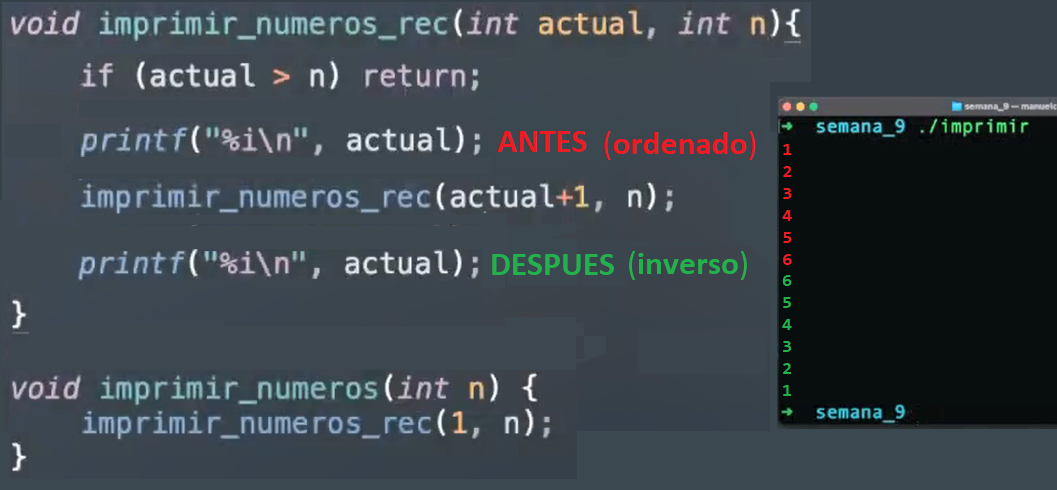
Setup de rutina recursiva

Se utiliza una funcion/procedimiento intermedio con un **parámetro extra** que permita iterar de forma más eficiente.



Posición del Proceso

La fase del proceso en un método recursivo se puede situar antes o después del llamado recursivo. Esto provocará un cambio en el orden de su ejecución.



Búsqueda binaria recursiva

int busqueda\_binaria\_rec(int vector[MAX\_ELEMENTOS], int inicio, int fin, int buscado) {

    /\* Caso base 1 \*/

    if (inicio > fin) return NO\_EXISTE;

    int centro = (inicio + fin) / 2;

    /\* Caso base 2 \*/

    if (vector[centro] == buscado) {

        return centro;

    }

    /\* Llamada(s) recursiva(s) \*/

    if (vector[centro] > buscado) {

        return busqueda\_binaria\_rec(vector, inicio, centro-1, buscado);

    } else {

        return busqueda\_binaria\_rec(vector, centro+1, fin, buscado);

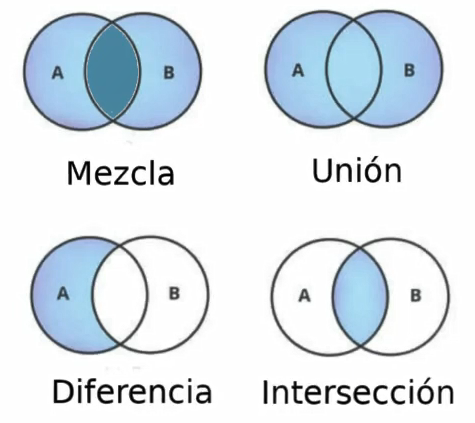
    }

}

int busqueda\_binaria(int vector[MAX\_ELEMENTOS], int tope, int buscado) {

    return busqueda\_binaria\_rec(vector, 0, tope-1, buscado);

}

**Operaciones en vectores**

Los vectores pueden ser interpretados como conjuntos.

* Mezcla

Junta todos los elementos de los vectores, sin importar si hay elementos repetidos o no.

* Union

Junta todos los elementos de los vectores, con excepción de aquellos que se encuentran repetidos.

* Diferencia

Mostrar todos los elementos del vector A que no aparezcan en el vector B

* Intersección

Junta sólo los elementos que se encuentran repetidos en ambos vectores.

Mezcla de vectores

// Pre-condiciones: vector1 y vector2 deben estar ordenados, topes >= 0

// Pos-condiciones: Llena vector\_resultado con la mezcla de vector1 y vector2

void mezclar\_vectores(int vector1[MAX\_ELEMENTOS], int tope1,

                      int vector2[MAX\_ELEMENTOS], int tope2,

                      int vector\_resultado[2\*MAX\_ELEMENTOS], int\* tope\_resultado) {

    int i = 0, j = 0;// contadores

    \*tope\_resultado = 0;

    while(i < tope1 && j < tope2) {

        if(vector1[i] < vector2[j]) {// vector1 es menor y se agrega

            vector\_resultado[\*tope\_resultado] = vector1[i];

            i++;

        } else {// vector2 es menor o igual y se agrega

            vector\_resultado[\*tope\_resultado] = vector2[j];

            j++;

        }

        (\*tope\_resultado)++;

    }

    while(i < tope1) {// Agrega los elementos que quedaron fuera del primer while

        vector\_resultado[\*tope\_resultado] = vector1[i];

        (\*tope\_resultado)++;

        i++;

    }

    while(j < tope2) {// Agrega los elementos que quedaron fuera del primer while

        vector\_resultado[\*tope\_resultado] = vector2[j];

        (\*tope\_resultado)++;

        j++;

    }

}

Union de vectores

// Pre-condiciones: vector1 y vector2 deben estar ordenados y sin elementos repetidos

//                  dentro de un mismo vector, topes >= 0

// Pos-condiciones: Llena vector\_resultado con la union de vector1 y vector2 (sin repetidos)

//                  y establece su tope

void unir\_vectores(int vector1[MAX\_ELEMENTOS], int tope1,

                   int vector2[MAX\_ELEMENTOS], int tope2,

                   int vector\_resultado[2\*MAX\_ELEMENTOS], int\* tope\_resultado) {

    int i = 0, j = 0;// contadores

    \*tope\_resultado = 0;

    while(i < tope1 && j < tope2) {

        if(vector1[i] < vector2[j]) {// vector1 es menor y se agrega

            vector\_resultado[\*tope\_resultado] = vector1[i];

            i++;

        } else if(vector2[j] < vector1[i]) {// vector2 es menor y se agrega

            vector\_resultado[\*tope\_resultado] = vector2[j];

            j++;

        } else {// vector1 es igual a vector2 y se agrega solo uno de ellos

            vector\_resultado[\*tope\_resultado] = vector1[i];

            i++;

            j++;

        }

        (\*tope\_resultado)++;

    }

    while(i < tope1) {// Agrega los elementos que quedaron fuera del primer while

        vector\_resultado[\*tope\_resultado] = vector1[i];

        (\*tope\_resultado)++;

        i++;

    }

    while(j < tope2) {// Agrega los elementos que quedaron fuera del primer while

        vector\_resultado[\*tope\_resultado] = vector2[j];

        (\*tope\_resultado)++;

        j++;

    }

}

Diferencia

// Pre-condiciones: vector1 y vector2 deben estar ordenados, topes >= 0

// Pos-condiciones: Llena vector\_resultado con los elementos del vector1

//                  que NO estén en el vector2 y establece su tope

void restar\_vectores(int vector1[MAX\_ELEMENTOS], int tope1,

                     int vector2[MAX\_ELEMENTOS], int tope2,

                     int vector\_resultado[MAX\_ELEMENTOS], int\* tope\_resultado) {

    int i = 0, j = 0;// contadores

    \*tope\_resultado = 0;

    while(i < tope1 && j < tope2) {

        if(vector1[i] < vector2[j]) {// vector1 es menor y se agrega

            vector\_resultado[\*tope\_resultado] = vector1[i];

            (\*tope\_resultado)++;

            i++;

        } else if(vector1[i] > vector2[j]) {// vector2 es menor y se saltea

            j++;

        } else {// vector1 y vector2 son iguales

            i++;

            j++;

        }

    }

    while(i < tope1) {// Agrega los elementos que quedaron fuera del primer while

        vector\_resultado[\*tope\_resultado] = vector1[i];

        (\*tope\_resultado)++;

        i++;

    }

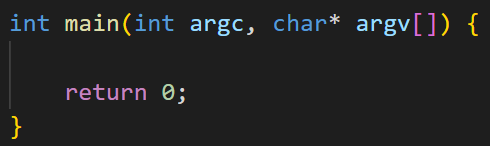
}

Interseccion

**Argumentos**

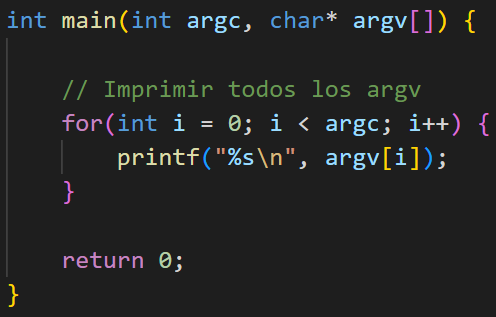
Por línea de comandos

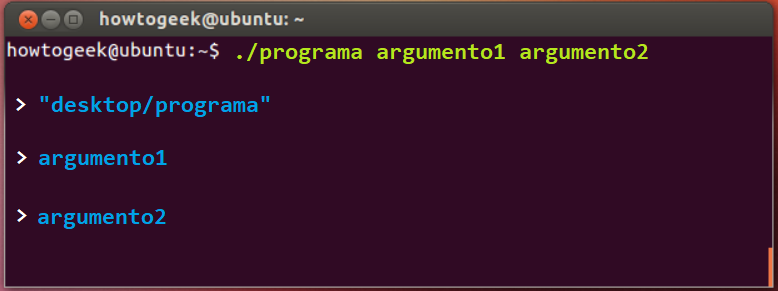
Permite interacción con el programa directamente desde el método main a través de los argumentos ingresados durante la ejecución del programa.



donde argv es un vector de strings y argc es el tope del vector de strings.

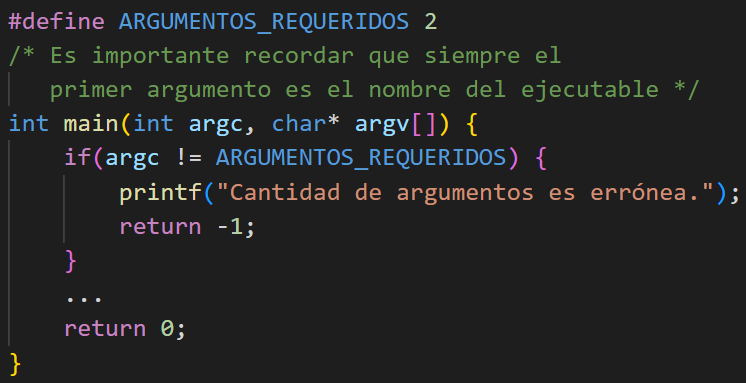
Al ejecutar el programa desde la terminal (con el comando ./), todos los argumentos que se escriban luego, serán recibidos en el main.

**argv[0] siempre será el nombre del archivo compilado o su dirección.**

****

Puede ser necesario crear un control de la cantidad de argumentos que el programa necesita para ejecutarse.

Por ejemplo, si un programa necesita un archivo externo cuya dirección es enviada como argumento al ejecutarse el archivo, entonces es requerido que dicho argumento esté presente para que el programa funcione.

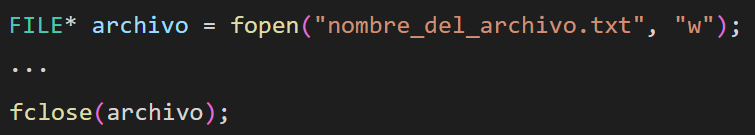


**Archivos**

Permite trabajar con la memoria del disco, la cual es mucho mayor que la memoria RAM. Además, permite persistencia en el tiempo de los datos que se almacenan.

Siempre se inicia abriendo un archivo, mediante la función **fopen** (la cual también crea el archivo), y siempre se debe finalizar cerrando el archivo con **fclose**.

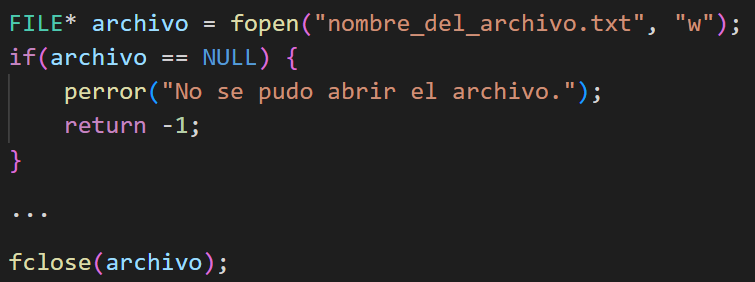
* *w* permite escribir un archivo
* *a* permite añadir texto a un archivo existente
* *r* permite leer un archivo



La variable “archivo” apunta dónde se va a escribir (en este caso) en ese file.

Si el archivo ya existe previamente, lo borra y crea de nuevo.

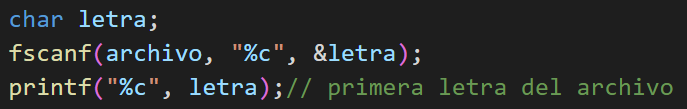
Es importante agregar un controlador de errores en caso que el archivo no haya podido crearse o encontrarse.



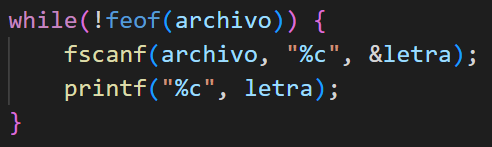
Es importante notar que, **si un programa termina de forma inesperada con un return, es necesario cerrar los archivos previamente** al return.

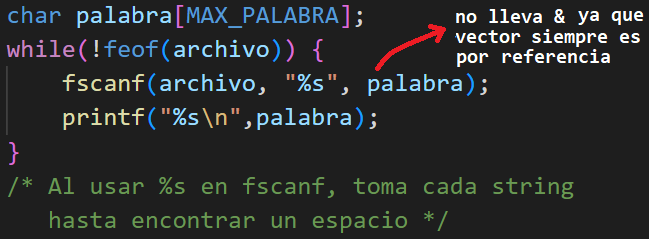
La función **fprintf** permite escribir en un archivo previamente abierto en modo escritura (y sin cerrar aún).

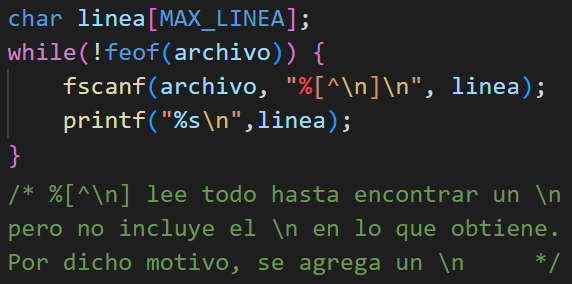
La funcion **fscanf** permite leer los caracteres de un archivo y devuelve la cantidad de variables que pudo llenar (al igual que scanf).

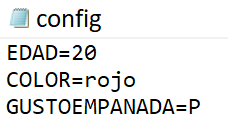


Si se desea leer todo el archivo, se puede utilizar un while con la funcion **feof**, la cual devuelve 0 si el puntero “archivo” se encuentra en el final del archivo.







Archivos tipo config

Son archivos utilizados para guardar configuraciones.

Es común que un archivo “config” sea ejecutado como argumentos al ejecutar el programa desde la terminal. Además, se suele guardar con el nombre “.config” (con un punto al inicio) para que se oculte de la vista del usuario por defecto.

Para leer el archivo config, es necesario utilizar fscanf y se puede utilizar de diferentes maneras.

* + **Lineal**

Se lee línea a línea en orden.

FILE\* archivo\_config = fopen("config", "r");

    if(archivo\_config == NULL) {

        perror("No se pudo abrir el archivo.");

        return -1;

    }

    int edad;

    char color[50];

    char gusto\_empanada;

    fscanf(archivo\_config, "EDAD=%i\n", &edad);

    fscanf(archivo\_config, "COLOR=%s\n", &color);

    fscanf(archivo\_config, "GUSTOEMPANADA=%c\n", &gusto\_empanada);

    printf("La edad leída es %i,", edad);

    printf("el color es %s", color);

    printf("y el gusto de empanadas es %c", gusto\_empanada);

    fclose(archivo\_config);

* + **Dinámico**

Se lee el archivo config sin ser necesario respetar su orden.

    FILE\* archivo\_config = fopen("config", "r");

    if(archivo\_config == NULL) {

        perror("No se pudo abrir el archivo.");

        return -1;

    }

    int edad = 0;

    char color[MAX\_NAME];

    char gusto\_empanada;

    char comando[MAX\_NAME], valor[MAX\_NAME];

    int variables\_leidas = fscanf(archivo\_config,

    "%[^=]=%s\n", &comando, &valor);

**// guarda en la variable matcheada con el % todo lo que hay detrás del =**

    while(variables\_leidas == 2) {

        if(strcmp(comando, "EDAD") == 0) {

            edad = atoi(valor);// convierte string en int

        } else if(strcmp(comando, "COLOR") == 0) {

            strcpy(color, valor);// copia un string

        } else {

            gusto\_empanada = valor[0];

        }

        variables\_leidas = fscanf(archivo\_config,

        "%[^=]=%s\n", comando, valor);

    }

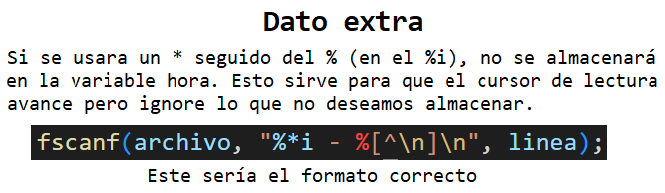
    printf("\nLa edad leída es %i\n", edad);

    printf("El color leído es %s\n", color);

    printf("El gusto de empanada leído es %c\n", gusto\_empanada);

    fclose(archivo\_config);

Es de utilidad saber que, si luego de un % se agrega un \*, dicha variable no se almacenará en ningún lado.

* Ejemplo de lectura de archivos y uso de argumentos

#include <stdio.h>

#include <stdlib.h>

#define ARGUMENTOS\_REQUERIDOS 3

#define MAX\_LINEA 256

int leer\_actividad(char\* nombre\_archivo, int hora\_buscada) {// lectura lineal

    FILE\* archivo = fopen(nombre\_archivo, "r");

    if(archivo == NULL) {

        perror("No se pudo abrir el archivo.");

        return -1;

    }

    char linea[MAX\_LINEA];

    int hora = 0;

    while(!feof(archivo)) {

        fscanf(archivo, "%i - %[^\n]\n", &hora, linea);

        if(hora == hora\_buscada) {

            printf("%s\n", linea);

        }

    }

    fclose(archivo);

    return 0;

}

// argv[0] : nombre del ejecutable

// argv[1] : nombre del archivo txt

// argv[2] : hora

int main(int argc, char\* argv[]) {

    if(argc != ARGUMENTOS\_REQUERIDOS) {

        printf("Cantidad de argumentos es errónea.");

        return -1;

    }

    int hora = atoi(argv[2]);

    return leer\_actividad(argv[1], hora);

}

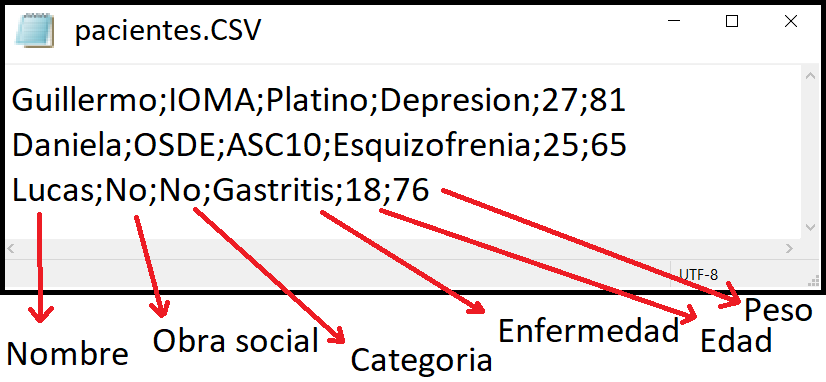
/\* [horarios.txt]

900 - Son las 9 de la mañana, hora de desayunar.

1100 - Son las 11 de la mañana, hora de preparar el almuerzo.

1500 - Son las 3 de la tarde, hora de la siesta.

\*/

**CSV**

Coma S Values es un archivo que se utiliza para guardar datos en gran cantidad, separados por comas.

#include <stdio.h>

#define MAX\_NOMBRE 50

#define MAX\_OBRA\_SOCIAL 30

#define MAX\_CATEGORIA 30

#define MAX\_ENFERMEDAD 40

#define COLUMNAS\_DEL\_CSV 6

#define LECTURA\_PACIENTE "%[^;];%[^;];%[^;];%[^;];%i;%f\n"

typedef struct paciente {

    char nombre[MAX\_NOMBRE];// 50 bytes (cada char ocupa 1 byte)

    char obra\_social[MAX\_OBRA\_SOCIAL];// 30 bytes (cada char ocupa 1 byte)

    char categoria[MAX\_CATEGORIA];// 30 bytes (cada char ocupa 1 byte)

    char enfermedad[MAX\_ENFERMEDAD];// 40 bytes (cada char ocupa 1 byte)

    int edad;// 4 bytes

    float peso;// 4 bytes

} paciente\_t;// TOTAL: 158 bytes por paciente

// Recordando que cada bloque ocupa 64 bytes, entonces cada paciente ocupa 196 bytes

void imprimir\_ficha\_paciente(paciente\_t paciente) {

    printf("------------[ FICHA ]------------\n");

    printf("| Nombre: %s\n", paciente.nombre);

    printf("| Obra Social: %s (%s)\n", paciente.obra\_social, paciente.categoria);

    printf("| Diagnostico: %s\n", paciente.enfermedad);

    printf("| Edad: %i\n", paciente.edad);

    printf("| Peso: %f\n", paciente.peso);

    printf("---------------------------------\n\n\n");

}

int main(int argc, char const \*argv[]) {

    FILE\* f\_pacientes = fopen("pacientes.csv", "r");

    if(f\_pacientes == NULL) {

        perror("No se pudo abrir el archivo 'pacientes.csv'");

        return 1;

    }

    printf("Empezando lectura:\n");

    paciente\_t paciente;

    int leidos = fscanf(f\_pacientes,LECTURA\_PACIENTE,

    paciente.nombre, paciente.obra\_social, paciente.categoria,

    paciente.enfermedad, &(paciente.edad), &(paciente.peso));

    while(leidos == COLUMNAS\_DEL\_CSV) {// Si se encuentra una fila corrupta, se detiene

        imprimir\_ficha\_paciente(paciente);

        leidos = fscanf(f\_pacientes,LECTURA\_PACIENTE,

        paciente.nombre, paciente.obra\_social, paciente.categoria,

        paciente.enfermedad, &(paciente.edad), &(paciente.peso));

    }

    printf("\nFin de lectura\n");

    fclose(f\_pacientes);

    return 0;

}