



Universidad Provincial de Ezeiza

Tecnicatura Universitaria en Desarrollo de Software

INTRODUCCIÓN AL DESARROLLO DE SOFTWARE ESTRUCTURA DE DATOS Y ALGORITMOS I

Ing. Gustavo Pascual



UNIVERSIDAD
NACIONAL
DE MORENO

Universidad Nacional de Moreno
Ingeniería en Electrónica

**TEMAS DE
INFORMÁTICA I
E
INFORMÁTICA II**

Ing. Gustavo Pascual

Bienvenidos...



Indice general (...solo para facilitar la búsqueda de los temas, no es completo...) (puede estar desactualizado... o faltar referencias.)

¿	
¿Cómo puedo ingresar un texto en español e imprimirllo en pantalla?	287
¿Cómo trabajar con char numéricos bajo entorno Windows 64 bits?	289
¿Cómo usar las constantes matemática de math.h en Windows 64 con el compilador mingw64?	294
¿Como usar long long en entornos Windows 64 bits con compiladores de 64 bits?	293
¿Con que función ingreso una cadena de caracteres (string) por teclado?	273
¿De que tipo pueden ser los vectores(los más comunes...)?	232
¿Por que aprender C?	56
¿Por qué estudiar C (en 2021)?	56
¿Por que y cuando usar estructuras?.....	124
¿Qué es Léxico?	36
¿Qué es Semántica?.....	36
¿Qué es Sintaxis?	36
¿Que es un "header"?.....	70
¿Qué es un puntero?.....	296
¿Qué es y como funciona un compilador?.....	35
¿Qué software voy a necesitar?	10
A	
Ahora la pasamos la matriz con un puntero a un vector.....	334
algoritmo	15, 17
Algoritmo de búsqueda con inserción	263
Algoritmos básicos.....	241, 401
Algunas ideas de como pensar un programa en C	21
Algunos ejemplos de declaración e inicialización de variables en C	95
algunos ejemplos matemáticos	210
Algunos ejemplos prácticos de uso de la estructura condicional "IF"	150
Algunos errores típicos utilizando if – else	148
Algunos sitios web recomendados	9
AND (&&) y OR ()	143
Anexo 1	401

Anexo 2	409
Archivo binario	366
Archivo: Definición	362
Archivos de texto.....	364
Archivos en C.....	361
Asi se almacena el string en el vector de char	270
B	
Buffer circular.....	358
Búsqueda binaria o dicotómica.....	259, 261
Busqueda secuencial en vectores.....	247
Busquedas y Ordenamientos	246
C	
C tiene las siguientes características	56
Cálculo de factorial de un número en forma iterativa	163
Cálculo de un promedio con datos almacenados en un vector numérico.....	403
Cálculo de un promedio con datos numéricos.....	402
Caracteres y strings.....	272
castear de float a long	103
Casteo	101
código con if (condicional) y switch – case (incondicional)	178
Como crear y escribir en un archivo de texto	374
como devolver mas de un valor desde una función.....	201
COMO ESCRIBIR MIS PROPIAS FUNCIONES	193
Como leer un archivo de texto	373
Comparación de las tres únicas iterativas que tiene el lenguaje C	172
Comparando las estructuras de control desde Assembler	388
Con scanf tambien podemos impedir el desborde de nuestro vector	275
Concepto general de funciones desde la programación en C	190
Constantes en C	105
Contenido de los elementos de un vector	311
Creación de variables de tipo estructura.....	126

Cuatro ejemplos simples de uso de la estructura de control if 139

D

Definición de string (Cadena de caracteres)	269
Definiciones básicas sobre programación.....	14
DETALLE LOS VECTORES	235
Diagramación lógica.....	18
Diferencia entre scanf, gets, gets_s y fgets	276
Dirección de los elementos de un vector	309
Distintos tipos de datos standard en C	78
Distintos tipos de datos standard en C y rangos de valores.....	78
do-while	166

E

Ejemplo completo del uso de un vector de estructuras en C.....	266
Ejemplo completo ordenamiento de un vector de estructuras, con función, con un miembro string.....	353
Ejemplo completo ordenamiento de un vector de estructuras, sin función	280
Ejemplos de uso de vectores (primeros pasos...)	238
El "ciclismo" de las variables enteras y los tipos de datos.....	94
El algoritmo de búsqueda con inserción aplicado a un vector de estructuras y a un string para contar letras de un texto.....	279
El intercambio de datos y la variable auxiliar.....	251
El lenguaje C	50
El lenguaje C posee 32 palabras reservadas.....	55
EL NOMBRE DEL VECTOR	235
Enteros (integers) almacenados bajo norma IEEE 754	84
Entrada: scanf	115
Entradas y salidas	111
Especificador para printf	117
Especificador para scanf.....	118
Especificadores de almacenamiento de los tipos de datos	96
Estructura <u>recomendada</u> de un programa en C.....	73
Estructuras.....	122
Estructuras anidadas.....	129
Estructuras de control	132
Explicación lazo for	157
Expresiones relacionales	137

F

fclose.....	369
fopen.....	367
For.....	156
fseek	369
ftell.....	371
Función de búsqueda binaria	329
Funciones básicas para el manejo de archivos en C (texto y binarios).....	367
Funciones de entradas y salidas básicas	113
Funciones de ordenamiento	326
Funciones en C	185
Funciones en matemática	186
Funciones matemáticas en C	207
Funciones matemáticas en C (tema optativo)	207
Funciones para el ingreso de caracteres por teclado	215
Funciones y estructuras	201
Funciones y matrices	332
Funciones y string	335
Funciones y vectores	325

G

Gráfico de tipos de datos en C	77
--------------------------------------	----

I

Imprimir en pantalla la ñ y vocales acentuadas (tema optativo).....	121
Iterativas(bucles o lazos).....	153

L

Lazos infinitos	173
Lectura y escritura de una archivo binario	377
Lee el archivo de texto creado en el ejemplo anterior	376
Los modificadores más comunes para printf y scanf	119
Los números reales en programación C (IEEE 754).....	80

M	
Más sobre punteros y funciones	355
Máximos y mínimos con vector numérico	407
Máximos y mínimos: forma 1 de calcularlos	404
Máximos y mínimos: forma 2 de calcularlos	405
Máximos y mínimos: forma 3 de calcularlos	406
Mezclando tipos de datos	100
mi_gets.....	336
mi_puts	337
mi_strcmp	341
mi_strlen	338
Modificadores usuales de formato para printf y scanf	117
Monumento a al-khwarizmi, Khiva, Uzbekistán	16
N	
Nuestro primer código	68
O	
Operadores pre y post	158
Ordena un vector de estructuras por miembro string	354
Ordenamiento por burbuja	254
Ordenamiento por pivot	249
Ordenamiento utilizando un vector de índices sin mover la información	258
Ordenamiento utilizando vector de direcciones sin mover físicamente los datos	319
Ordenamientos de datos en vectores	249
Otra forma de escribir en un archivo de texto	375
Otro ejemplo que nos ayudará a razonar los códigos	25
Otros ejemplos de funciones.....	198
P	
Pasaje de binario a decimal.....	44
Pasaje de binario a octal y hexadecimal agrupando de a 3bits o de a 4bits.....	49
Pasaje de decimal a binario.....	43
Pasaje de decimal a hexadecimal	47
Pasaje de decimal a octal	45
Pasaje de hexadecimal a decimal	48
Q	
Que elementos, en general, forman parte de un código en C	33
R	
Raíz cuadrada sin usar la función raíz cuadrada	32
Rango de valores de tipos enteros y fórmulas para obtenerlos	79
Recursividad	220
Resolución de problemas utilizando la computadora	15
rewind.....	371
S	
Salida: printf.....	113
scanf.....	115
Secuencia (de instrucciones)	135
Secuencia de construcción de un programa en C.....	34
Selección (condicional)	136
Serie de Fibonacci.....	412
Set de caracteres para códigos mayores a 80h (128 en decimal)	284
Si queremos unir en una expresión una o más expresiones tenemos dos operadores	143
Sistemas de numeración	38
sizeof.....	90
Strings	268
Switch - case	176
T	
Tabla ASCII.....	410

Tabla de factoriales	411
Tabla de modos de archivo.....	382
Tablas,diagramas	409
tipo de dato struct	122
Tipos de datos en C	75

U

Un código que nos puede dar problemas...	323
Un puntero puede apuntar a un puntero (en mucha bibliografía se denominan punteros dobles)	301
Un simple ejemplo de una suma de dos números: uso de printf y scanf.	120
Un simple ejemplo desde la matemática finaciera: el interés compuesto	30
Un simple programa con funciones y pasaje por referencia.....	322
Una alternativa a las variables tipo static	324
Una forma de comparar el contenido de dos strings usando switch - case.....	408
Una función que no recibe nada y devuelve un entero	199
Una función que recibe un valor entero y no devuelve nada	198

ungetc	372
--------------	-----

V

Vamos a ver algunos ejemplos de como usar archivos en C.....	373
Variables de tipo static	213
Variables en C.....	86
Veamos un primer ejemplo del uso de punteros	298
Vector de estructuras.....	264
Vector de punteros a cadena de caracteres (ejemplo)	347, 349
Vector de strings (vector de vectores)	344
Vectores (Arrays)	230
Visibilidad y alcance de variables.....	93
Visión general de lo que vimos y vamos a ver (que debo saber)	182

W

While.....	155
------------	-----

Estimado alumno:

Este texto está en continua actualización que puede ser publicada en cualquier momento. Es una guía para los temas más importantes. No está necesariamente completo. Puede contener errores involuntarios. Complementelo usted con libros, sitios web y apuntes de clase.

Muchos temas están resumidos, otros se explica lo mínimo necesario. No lo haga su única fuente de estudio.

Algunos sitios web recomendados:

- <https://www.cplusplus.com/> (en inglés)
- <http://www.conclase.net/> (en español)
- <https://es.cppreference.com/w/> (en español)
- <https://www.tutorialspoint.com/cprogramming/index.htm> (en inglés)
- <https://es.stackoverflow.com/questions/tagged/c> (en español)
- <https://www.codingunit.com/the-history-of-the-c-language> (en inglés)
- <https://scc-forge.lancaster.ac.uk/open/char/> (en inglés)

Ojo!! Los sitios se recomiendan bajo su responsabilidad. Es imposible chequear toda la información publicada por lo tanto si duda de ella, pregunte a sus docentes.

Ademas busque usted información en universidades de prestigio conocido: Universidad de Valencia, Universidad Autonoma de Mexico, UBA, UTN, Politécnico de Madrid, entre otras. Muchas tienen apuntes públicos de lenguaje C. Es recomendable, particularmente cuando se tiene un minimo de conocimiento, ir a la NORMA: ISO/IEC 9899:2011, ISO/IEC 9899:2018 son las dos más nuevas. Busquelas en PDF en internet. Son las normas la referencia oficial del lenguaje C. Son como el diccionario de castellano publicado por la Real Academia Española.

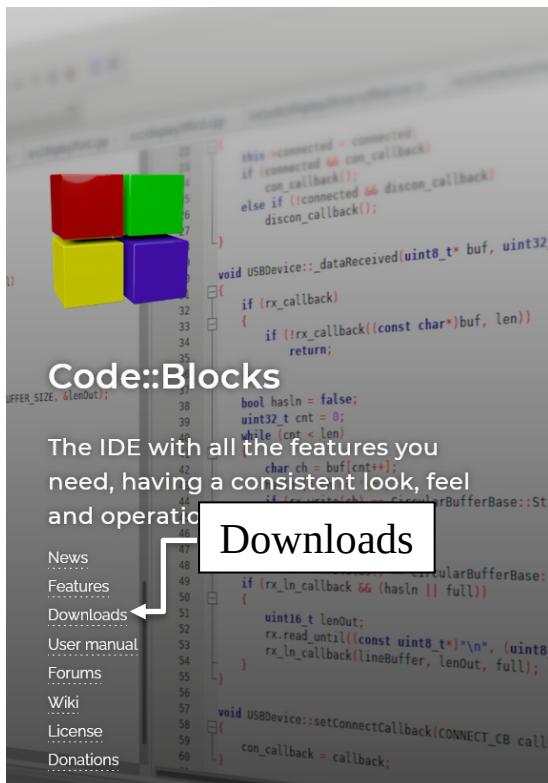
¿Qué software voy a necesitar?

Basicamente lo único que necesitaríamos es un editor texto al estilo del notepad de Windows, para lo que se denomina texto plano (ASCII) y un compilador (gcc, mingw, visual studio, etc)

Lo que recomendamos es que uses codeblocks que viene con todo lo necesario y es posible usarlo en Windows, desde el xp hasta el Win10 y en varias distribuciones de Linux, como Debian.

Lo descargas desde (link directo): <http://www.codeblocks.org/downloads/binaries/>

La página de codeblocks es: <https://www.codeblocks.org/>



Code::Blocks

Code::Blocks

The free C/C++ and Fortran IDE.

Code::Blocks is a free C/C++ and Fortran IDE built to meet the most demanding needs of its users. It is designed to be very extensible and fully configurable.

Built around a plugin framework, Code::Blocks can be extended with plugins. Any kind of functionality can be added by installing/coding a plugin. For instance, event compiling and debugging functionality is provided by plugins!

If you're new here, you can read the [user manual](#) or visit the [Wiki](#) for documentation. And don't forget to visit and join our [forums](#) to find help or general discussion about Code::Blocks.

We hope you enjoy using Code::Blocks!

The Code::Blocks Team

Latest news

Migration successful

We are very happy to announce that the process of migrating to the new infrastructure has completed successfully!

[Read more](#)

Forums and Wiki migration



Binaries

Code::Blocks / Downloads

Downloads

There are different ways to download and install Code::Blocks on your computer:

- **Download the binary release**

This is the easy way for installing Code::Blocks. Download the setup file, run it on your computer and Code::Blocks will be installed, ready for you to work with it. Can't get any easier than that!

- **Download a nightly build**

There are also more recent so-called nightly builds available in the [forums](#). Please note that we consider nightly builds to be stable, usually, unless stated otherwise.

- Other distributions usually follow provided by the community (big "Thank you!" for that!). If you want to provide some, make sure to announce in the forums such that we can put it on the official C::B homepage.

- **Download the source code**

If you feel comfortable building applications from source, then this is the recommend way to download Code::Blocks. Downloading the source code and building it yourself puts you in great control and also makes it easier for you to update to newer versions or, even better, create patches for bugs you may find and contributing them back to the community so everyone benefits.

- **Retrieve source code from SVN**

This option is the most flexible of all but requires a little bit more work to setup. It gives you that much more flexibility though because you get access to any bug-fixing we do at the time we do it. No need to wait for the next stable release to benefit from bug-fixes!

Besides Code::Blocks itself, you can compile extra plugins from contributors to extend its functionality.

Thank you for your interest in downloading Code::Blocks!

See also

Code::Blocks

The IDE with all the features you need, having a consistent look, feel and operation.

Cualquiera de estos dos para Windows 64 bits

Cualquiera de estos dos para Windows 32 bits o Windows 64 bits (funciona en ambos).

News
Features
Downloads
User manual
Forums
Wiki
License
Donations

GPLV W3C CSS GetFirefox SOURCEFORGE Support this project

Code::Blocks / Downloads / Binary releases

Binary releases

Please select a setup package depending on your platform:

- Windows XP / Vista / 7 / 8.x / 10
- Linux 32 and 64-bit
- Mac OS X

NOTE: For older OSes use older releases. There are releases for many OS version and platforms on the [Sourceforge.net](#) page.

NOTE: There are also more recent nightly builds available in the [forums](#) or (for Ubuntu users) in the [Ubuntu PPA repository](#). Please note that we consider nightly builds to be stable, usually.

NOTE: We have a [Changelog for 20.03](#), that gives you an overview over the enhancements and fixes we have put in the new release.

NOTE: The default builds are 64 bit (starting with release 20.03). We also provide 32bit builds for convenience.

Microsoft Windows

File	Download from
codeblocks-20.03-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03-setup-nonadmin.exe	FossHUB or Sourceforge.net
codeblocks-20.03-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03mingw-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03mingw-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup-nonadmin.exe	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-nosetup.zip	FossHUB or Sourceforge.net

NOTE: The codeblocks-20.03-setup.exe file includes Code::Blocks with all plugins. The codeblocks-20.03-

Siempre el codeblocks que nos bajemos debe contener la palabra “**mingw**” que es el compilador. Luego lo mas recomendable es si tenes un Windows 64 bits es que te bajes el de 64 bits para poder aprovechar todas las nuevas características del lenguajes C. De todas formas todo lo que hagamos en este curso va a ser para cualquier compilador de C que se tenga, incluso para versiones viejas de codeblocks. Si no sabes si tu Windows es 32 o 64 bits, bajate el codeblocks de 32 bits, que para este curso te sobra...

El archivo que termina en **.zip** tiene como ventaja que se descomprime, por ejemplo en el escritorio y listo. El **.exe** se instala y muchas veces no tenemos los permisos de usuario para poder hacer esto. **Si dudas, bajate el .zip.**

No es el codeblocks el único ide que se puede utilizar, pero si es el que te recomendamos para dar tus primeros pasos en programación. Es muy completo y potente... Tambien existen programas que te permiten programar en C bajo Android. Uno de los mejores que he probado para Android que funciona muy bien (marzo 2022) es Movil C (muy recomendable).

Tambien podes programar online por medio de <https://replit.com/> , seguramente tendras que crear una cuenta pero es gratuita.

Otro es el IDE online <https://www.codechef.com/ide> que es muy facil y directo.

Otro fácil online es <https://ide.geeksforgeeks.org/>

Otro también fácil para muchos lenguajes es <https://www.jdoodle.com/>

Otro más online: <https://www.programiz.com/c-programming/online-compiler/>

Otro más: <https://www.onlinegdb.com/> (muy recomendable!!)

Uno avanzado, no para el que comienza, es: <https://godbolt.org/>

Use el que quiera, no habrá grandes diferencias si lo usa bien. Todo se puede resolver.

Definiciones básicas sobre programación

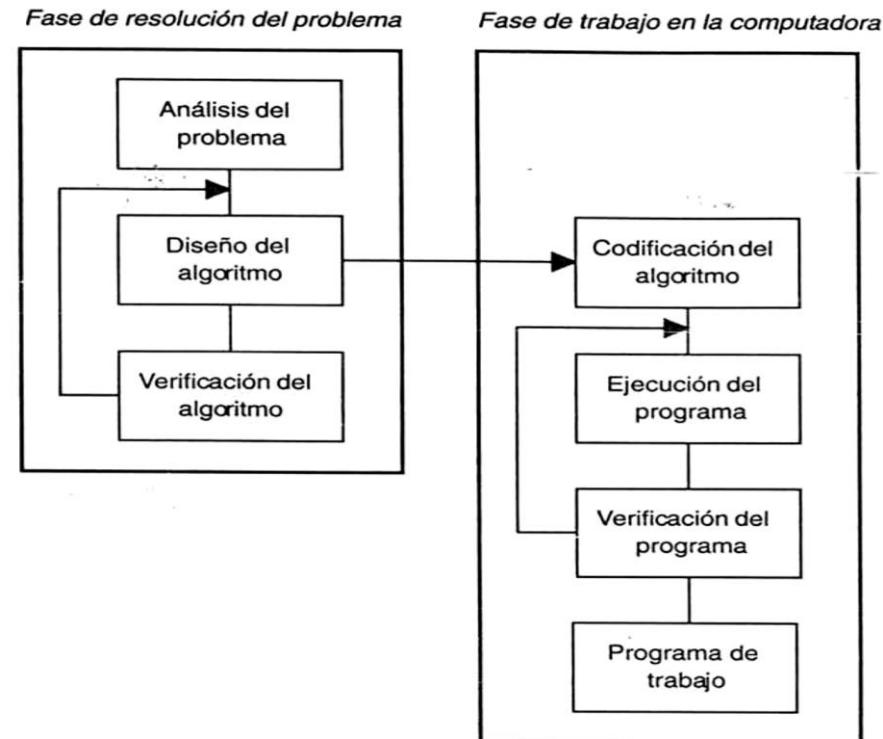
Resolución de problemas utilizando la computadora

El proceso de diseñar programas es un proceso creativo, en el que se pueden considerar tres pasos que ayudan al programador en este proceso:

- **Análisis del problema**
- **Diseño del algoritmo**
- **Resolución del algoritmo en la computadora**

Las fases de análisis y diseño del algoritmo requieren la descripción del problema en subproblemas y una herramienta de programación: diagrama de flujo, pseudocódigo o diagrama N-S (diagramas de Nassi-Schneider).

En la tercera fase se implementa este algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas obtenidas en las fases de análisis y diseño. Antes de conocer las tareas a realizar en cada fase, definiremos el concepto de algoritmo.





Monumento a al-khwarizmi, Khiva, Uzbekistán

La palabra **algoritmo** deriva del nombre del famoso matemático y astrónomo árabe (Persa) **Abu Abdallah Muḥammad ibn Mūsā al-Jwārizmī**, que vivió aproximadamente entre 780 y 850 de la era Cristiana, que escribió un conocido tratado sobre la manipulación de números y ecuaciones titulado ***al-Kitāb al-mukhtaṣar fī ḥisāb al-ŷabṛ wa-l-muqābala***, de donde se deriva la palabra **álgebra**.

Un algoritmo puede ser definido como la secuencia ordenada de pasos, sin ambigüedad, que conducen a la solución de un problema dado y expresado en lenguaje natural, por ejemplo en castellano.

Todo algoritmo debe ser:

- **Preciso:** Indicando el orden de realización de cada uno de los pasos.
- **Definido:** Si se sigue el algoritmo varias veces proporcionandole los mismos datos, se deben obtener los mismos resultados.
- **Finito:** Al seguir el algoritmo, se debe terminar en algún momento, es decir tener un número finito de pasos.

Para diseñar un algoritmo se debe comenzar por identificar las tareas mas importantes para resolver el problema y disponerlas en el orden en el que han de ser ejecutadas. Los pasos en esta primera descripción de actividades deberán ser refinados, añadiendo mas detalles a los mismos e incluso, algunos de ellos, pueden requerir un refinamiento adicional antes de que podamos obtener un algoritmo claro, preciso y completo.

Este método de diseño de los algoritmos en etapas, yendo de los conceptos generales a los detalles a través de refinamientos sucesivos, se conoce como método descendente o **top – down**.

En un algoritmo se deben de considerar tres partes:

Entrada: Informacion dada al algoritmo.

Proceso: Operaciones o cálculos necesarios para encontrar la solución del problema

Salida: Respuestas dadas por el algoritmo o resultados finales de los cálculos.

Lo primero que deberá hacer es plantearse y contestar a las siguientes preguntas:

Especificaciones de entrada	Especificaciones de salida
<ul style="list-style-type: none">• ¿Qué datos son de entrada?• ¿Cuántos datos se introducirán?• ¿Cuántos son datos de entrada válidos?	<ul style="list-style-type: none">• ¿Cuáles son los datos de salida?• ¿Cuántos datos de salida se producirán?• ¿Qué precisión tendrán los resultados?

Diagramación lógica

(Nos ayudará PSEint (<http://pseint.sourceforge.net/>)... que no es tema de nuestras materias, vemos C, pero si sirve para aprender a pensar un algoritmo, usalo, además esta en castellano.)

Antes de empezar a codificar en alguno de los tantos lenguajes de programación que hoy existen, nosotros usaremos C, debemos entender como pensar el problema a resolver. Para esto tenemos un gran auxiliar: los diagramas. Nosotros vamos a proponer los llamados diagramas de Nassi-Shneiderman que normalmente se los suele llamar diagramas de Chapin.

Los diagramas son grandes auxiliares a la hora de programar y buscar errores en la secuencia ejecución de las instrucciones:

Veamos un ejemplo de la vida cotidiana llevado a diagrama: la recarga de la tarjeta SUBE. Veamoslo desde el pseudocódigo y desde su diagramación:

Pseudocódigo:

```
Algoritmo tarjeta_sube
    Escribir 'Compramos una tarjeta SUBE'
    Repetir
        Si saldo_tarjeta<saldo_permitido Entonces
            Escribir 'Recargar tarjeta'
        SiNo
            Escribir 'La uso normalmente'
        FinSi
    Hasta Que (se_rompa O no_se_use_mas)
FinAlgoritmo
```

Diagrama de Chapin:

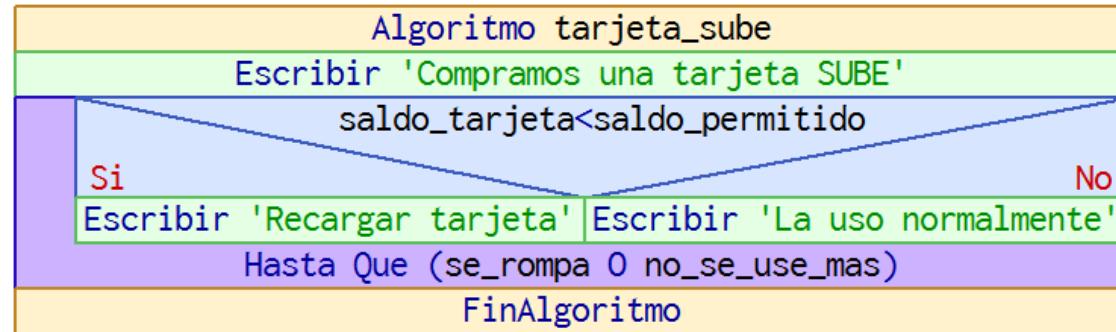
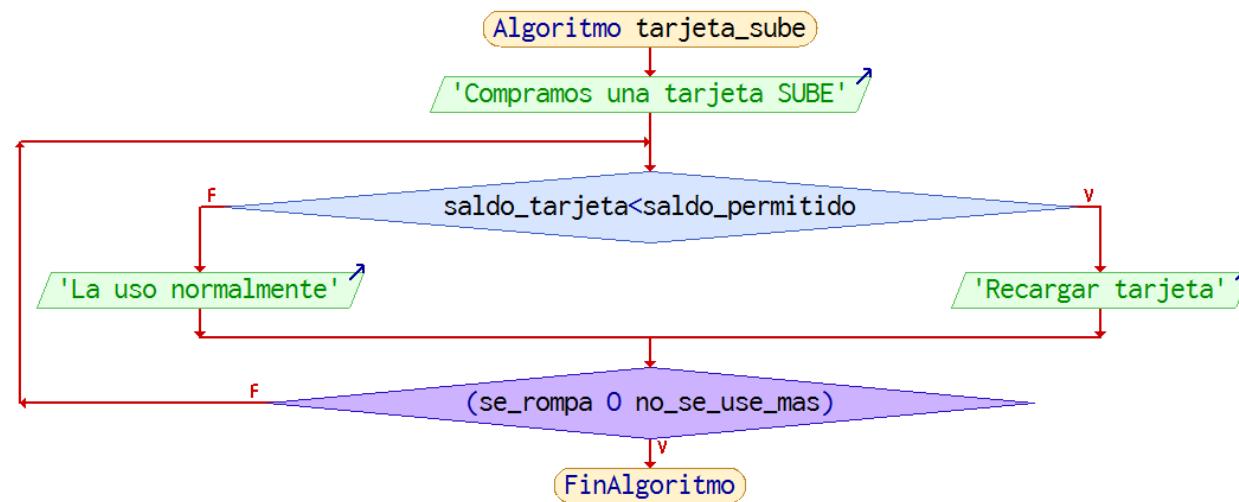


Diagrama Ansi: Existe otro tipo de diagrama muy usado en la práctica, al que solo se recomienda para documentar programas ya que usado a la hora de diseñar, genera muchas veces códigos poco optimos. Lo usaremos en este texto para explicar cual es la secuencia de ejecución de las distintas instrucciones, pero lo evitamos a la hora de diseñar dicho código.



Te recomendamos que uses los diagramas de Chapin a la hora de aprender programación.

Algunas ideas de como pensar un programa en C (y en cualquier lenguaje similar)

Lo primero a lo que se enfrenta un alumno al aprender a programar es como obtener la secuencia de instrucciones.

Supongamos el siguiente enunciado: “**Se pide obtener el promedio de 100 numeros enteros ingresados por teclado**”.

Vamos a analizar el texto:

- **Son 100 datos, no mas no menos.**
- **Son datos enteros.**
- **Se ingresan por teclado.**
- **Se quiere obtener el promedio.**
- **No se pide almacenar cada valor individualmente (importante!!).**
- **No lo dice explícitamente, pero el resultado del promedio lo tengo que presentar en pantalla.**

Que tengo que saber: tengo que conocer como se calcula un promedio, esto es:

- **Sumar cada uno de los valores ingresados (sumatoria).**
- **En este caso, dividir por 100 dicha suma para obtener el promedio.**

Ahora, ¿como hago la suma?: ¡debo encontrar un algoritmo básico para esto!:

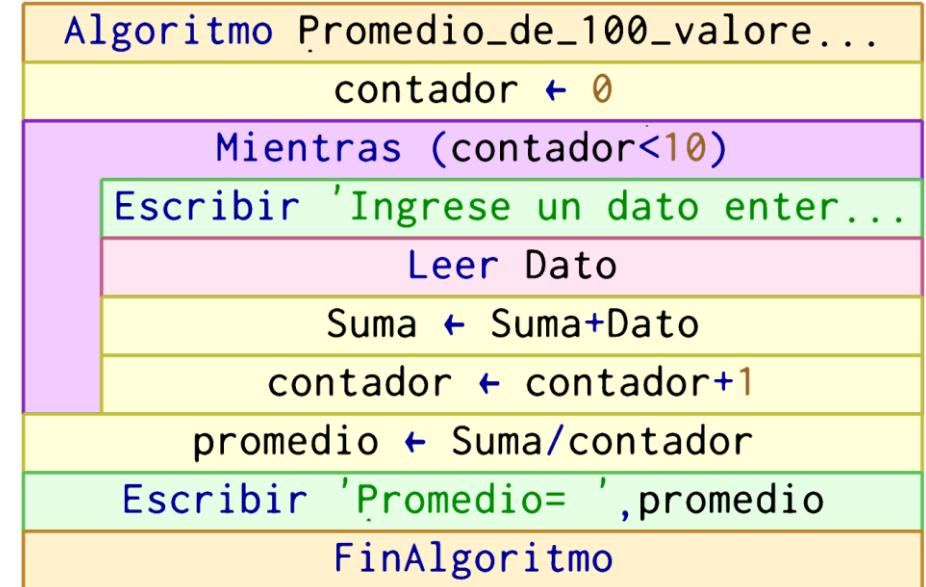
- Necesito una variable para almacenar la sumatoria: ej. **Suma**
- Debo obtener el algoritmo de la sumatoria: **Suma=Suma + Dato**
- **Dato** es la variable que sobreescribo 100 veces donde ingreso los valores a promediar.
- Debo pensar la manera de repetir 100 veces el ingreso por teclado y la sumatoria: **lazo o iterativa**.
- **Una vez que tenga la sumatoria de todos los valores a promediar, recién ahí calculo el promedio como:**

$$\text{Promedio} = \text{Suma} / 100$$

Promedio, en general, deberá ser una variable tal que pueda almacenar datos con decimales.

Algoritmo en pseudocódigo y en diagrama de Chapin

```
1 Algoritmo Promedio_de_100_valores_enteros
2     contador=0;
3     Mientras (contador<10) Hacer
4         Imprimir "Ingrese un dato entero= "
5         leer Dato
6         Suma=Suma+Dato;
7         contador=contador+1;
8     FinMientras
9     promedio=Suma/contador;
10    Imprimir "Promedio= ",promedio;
11 FinAlgoritmo
```

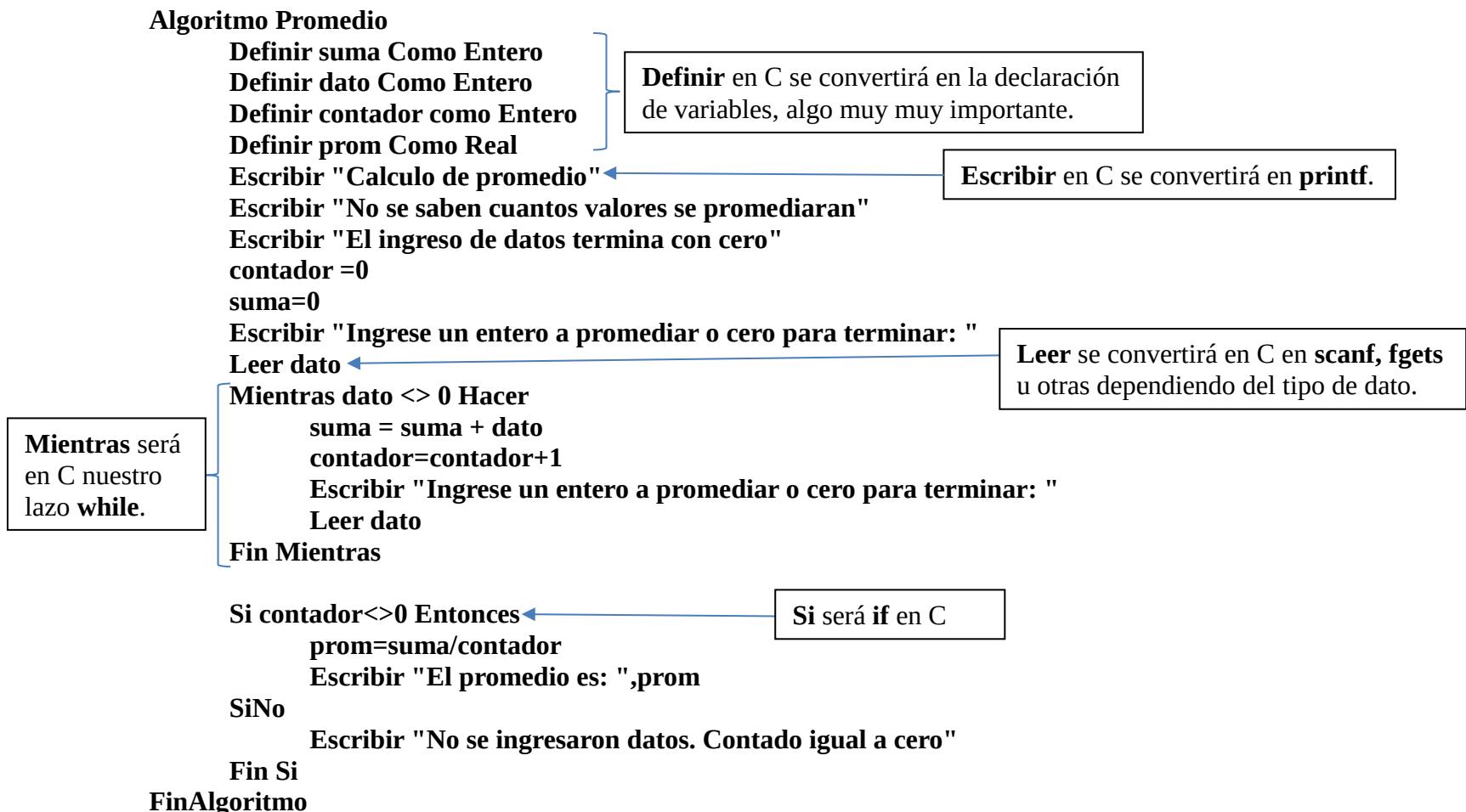


Es muy importante diagramar para poder realizar nuestros códigos. Debe primero ver cada detalle de lo que debe obtener. Sin entender claramente el problema es imposible solucionarlo correctamente. Debe evaluar, de haberlas, distintas estrategias de solución. Papel y lápiz es un buen comienzo para cualquier código.

Todo esto lo aprenderá en este curso y en cursos posteriores. Aprender C en particular es como aprender Ingles conociendo solo Castellano. Es progresivo, no se puede aprender todo junto. Sea metódico. No aprenda de memoria, no sirve. Razone cada instrucción, busque porque se necesita. Divida el código buscando “bloques de instrucciones”. Nosotros aprenderemos programación estructurada. El problema en general siempre se puede subdividir en “pequeñas tareas”. No busque resolver todo junto, en general no sirve. Busque esos bloques que pueden formar nuestro código.

Otro ejemplo

Vamos a calcular el promedio de una serie de números ingresados por teclado cuando se desconoce su cantidad. Calculamos al igual que el ejemplo anterior el promedio pero ahora varia el ingreso de los datos.



Algoritmo visto con diagrama de Chapin	Algoritmo visto con diagrama ANSI
<pre> Algoritmo Promedio Definir suma Como Entero Definir dato Como Entero Definir contador Como Entero Definir prom Como Real Escribir 'Calculo de promedio' Escribir 'No se saben cuantos valores se promediaran' Escribir 'El ingreso de datos termina con cero' contador ← 0 suma ← 0 Escribir 'Ingrese un entero a promediar o cero para terminar: ' Leer dato Mientras dato<>0 suma ← suma+dato contador ← contador+1 Escribir 'Ingrese un entero a promediar o cero para terminar: ' Leer dato contador<>0 Si prom ← suma/contador Escribir 'No se ingresaron datos. Contado igual a cero' FinAlgoritmo No Escribir 'El promedio es: ',prom FinAlgoritmo </pre>	<pre> graph TD Start([Algoritmo Promedio]) --> DefSuma[Definir suma Como Entero] DefSuma --> DefDato[Definir dato Como Entero] DefDato --> DefContador[Definir contador Como Entero] DefContador --> DefProm[Definir prom Como Real] DefProm --> Calculo[('Calculo de promedio')] Calculo --> Cond1{dato <> 0} Cond1 -- F --> FinAlgoritmo([FinAlgoritmo]) Cond1 -- V --> Suma[suma ← suma+dato] Suma --> Contador[contador ← contador+1] Contador --> Input[('Ingrese un entero a promediar o cero para terminar: ')] Input --> LeerDato1[/Leer dato/] LeerDato1 --> Cond1 LeerDato1 --> Cond2{dato <> 0} Cond2 -- F --> FinAlgoritmo Cond2 -- V --> Suma Suma --> Contador Contador --> Input Contador --> Cond2 Contador --> Promedio[prom ← suma/contador] Promedio --> Resultado[('El promedio es: ', prom)] Resultado --> FinAlgoritmo Resultado --> NoDatos[('No se ingresaron datos. Contado igual a cero')] </pre>

Otro ejemplo que nos ayudará a razonar los códigos

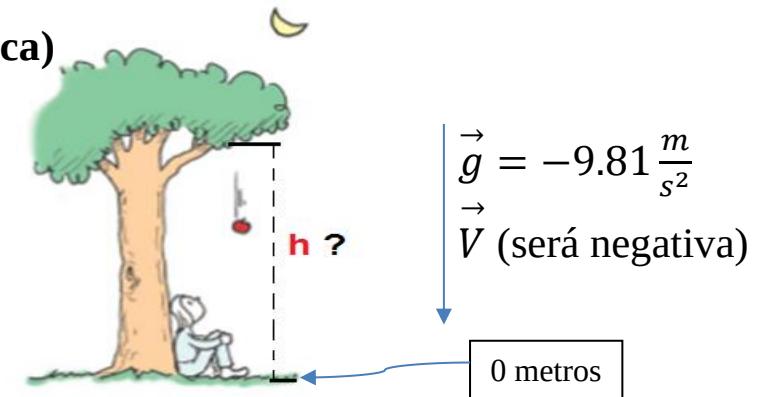
Se desea escribir un código que genere una tabla en pantalla que permita analizar la caída de un objeto desde una altura determinada. La tabla será de velocidad y posición en función del tiempo y contendrá 10 valores.

Separaremos la información:

- Solo se ingresa la altura desde donde cae el objeto.
- Son 10 valores los que debo imprimir en pantalla.
- Debo conocer / buscar las ecuaciones del movimiento uniformemente acelerado.
- Debo conocer la aceleración de la gravedad en el lugar donde caerá el objeto.

Ecuaciones del movimiento uniformemente acelerado (tema de física)

$$X = X_0 - V_0 \cdot t - 0.5 \cdot g \cdot t^2$$
$$V = -V_0 - g \cdot t$$



Desde la física sabemos que:

$V_0 = 0 \frac{m}{s}$ Velocidad inicial cero ya que se cae y no es impulsado hacia abajo

$X_0 = h$ La posición cero está desde donde cae el objeto, donde $t = t_0 = 0s$

$g = 9.81 \frac{m}{s^2}$ Si no tenemos otro dato de g usaremos este. Varía con la altura.

Las ecuaciones quedarán como:

$$X = h - 0.5 \cdot g \cdot t^2 \quad (\text{ecuación 1})$$

$$V = -g \cdot t \quad (\text{ecuación 2})$$

Luego:

$t = \sqrt{\left(\frac{(2 \cdot h)}{g}\right)}$ obtenida desde la ecuación 1 nos dará cuanto tarda el objeto en llegar al suelo, es t_{\max} .

Este será el segundo límite de nuestra tabla ya que el primero es cero segundos.

Si por ejemplo cae desde 50 metros, el t_{\max} será de 3,192754 segundos.

En este tiempo t_{\max} , $X= 0\text{m}$ esto es, el objeto toca el suelo y con ese tiempo y la ecuación 2 podríamos saber a que velocidad lo ha hecho, esto es: 31,32 m/s (unos 112,75 km/h).

Para obtener la separación (paso) de cada valor de t en nuestra tabla dividimos por 9 (¿pensó en dividir por 10?).

No se olvide que el cero forma parte de nuestra tabla. Si el primer valor no fuese cero, otra será la cuenta, búsquela.

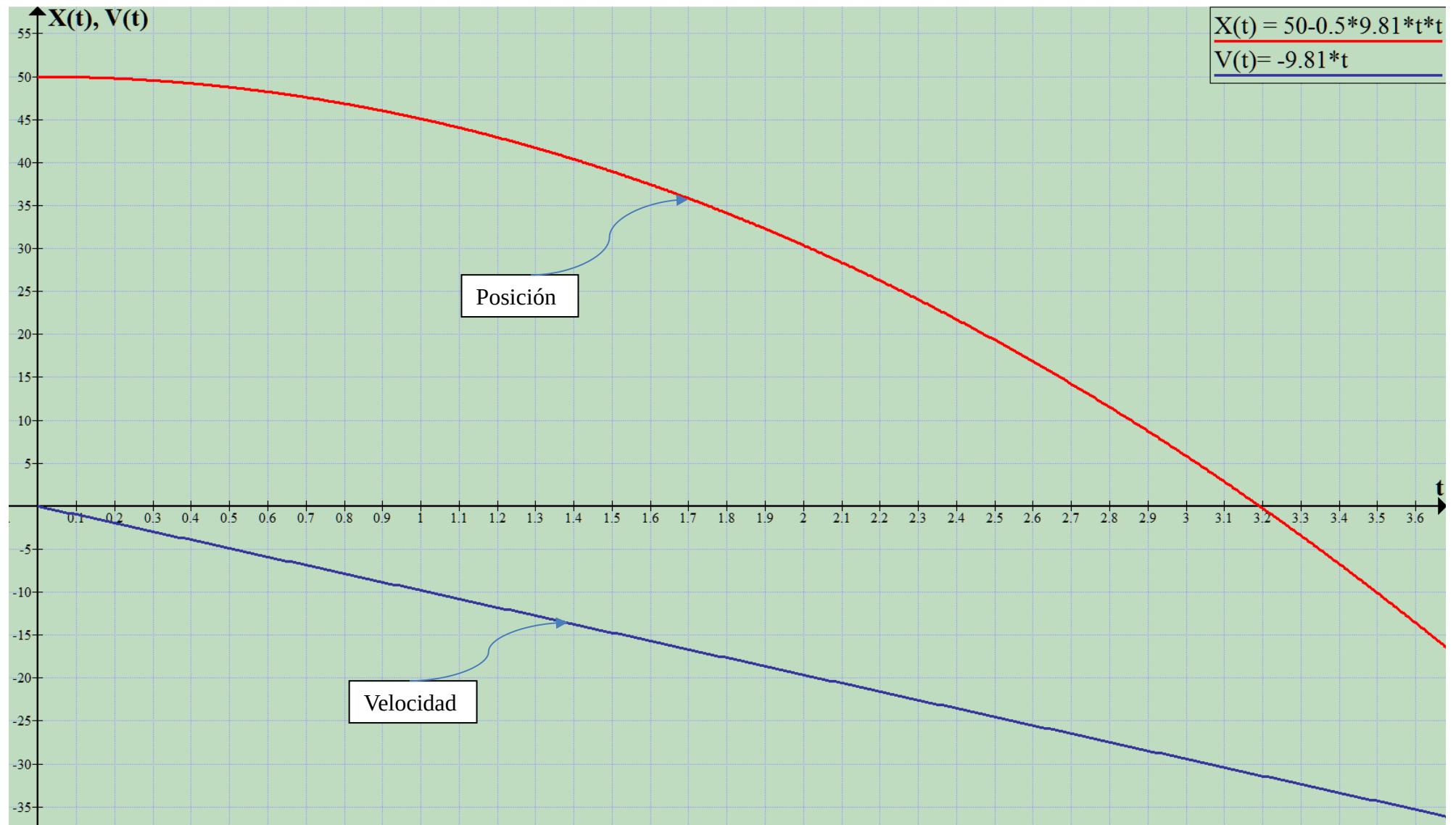
El paso será para este caso particular de 0,354750 segundos. Un ejemplo de lo que deberíamos obtener se muestra a continuación.

Esta es la tabla que deberíamos ver por pantalla para $h=50$ metros y $g=9.81$ metros sobre segundo cuadrado.

	t en segundos	X en metros	V en metros por segundo
1	0,000000	50,000000	0,000000
2	0,354750	49,382718	-3,480098
3	0,709500	47,530871	-6,960195
4	1,064250	44,444459	-10,440293
5	1,419000	40,123483	-13,920390
6	1,773750	34,567943	-17,400488
7	2,128500	27,777837	-20,880585
8	2,483250	19,753168	-24,360683
9	2,838000	10,493933	-27,840780
10	3,192750	0,000134	-31,320878

Vea que X en la posición 10 de la tabla no es exactamente cero metros. En este caso, no es importante.

En el siguiente gráfico se refleja esta tabla. Vea cuantas herramientas tiene para poder ver que debe obtener.



Como verá, generar este código tiene mucho de análisis previo antes de sentarnos frente a la computadora a codificar. Vea que la velocidad es negativa, y eso está bien ya que tiene que ver donde hemos colocado nuestras referencias (física).

Ahora haremos un código que resuelva esto, y lo haremos en pseudocódigo.

Algoritmo MRUA

```

Escribir 'Ingrese la altura: ', Sin
Saltar;
    Leer altura;
    g = 9.81;
    tmax = raiz((2*altura) / g);
    n = 10;
    Escribir 'Tiempo maximo= ',tmax;
    PASO = tmax / (n-1);
    Escribir 'Paso= ',PASO;
    t = 0;
    Mientras (t<=tmax) Hacer
        X = altura - 0.5*g*t*t
        V = -g*t
        Escribir t,'      ',V,'      ',X;
        t = t+PASO;
    FinMientras
FinAlgoritmo

```

Algoritmo MRUA
Escribir 'Ingrese la altura: ',
Leer altura
g \leftarrow 9.81
tmax \leftarrow raiz((2*altura)/g)
n \leftarrow 10
Escribir 'Tiempo maximo= ',tmax
PASO \leftarrow tmax/(n-1)
Escribir 'Paso= ',PASO
t \leftarrow 0
Mientras (t<=tmax)
X \leftarrow altura-0.5*g*t*t
V \leftarrow -g*t
Escribir t,' ',V,' ...
t \leftarrow t+PASO
FinAlgoritmo

Este algoritmo está conformado por constantes, variables, lazos, funciones, etc, temas que veremos a continuación. En pocas clases podrá codificarlo en C. Tambien es posible hacerlo en otros lenguajes.

Un simple ejemplo desde la matemática financiera: el interés compuesto

Supongamos que se deposita 10000\$ en una cuenta de ahorros con una tasa de interés anual del 8%, que se capitaliza mensualmente. Queremos saber si no extraemos ni agregamos dinero a dicha cuenta, cuanto obtenemos en 10 años. Dicho depósito generaría unos 12196,40\$ de interés al finalizar un periodo de 10 años.

Al interés compuesto se lo define matemáticamente como (por dudas respecto a la formula de calculo, busque un libro de matemática financiera):

$$I_c = C_i \cdot \left(1 + \frac{t}{n}\right)^{(n \cdot u)} - C_i$$

I_c = Interés compuesto obtenido o pagado

C_i = Capital inicial (el monto del depósito inicial o del préstamo) = 10000\$

t = tasa de interés anual (dividida por 100). Se obtiene como interés anual dividido por 100 = 0.08

n = cantidad de periodos de capitalización por unidad de tiempo = 12 meses

u = cantidad de unidades de tiempo en que el dinero se invierte o se solicita en préstamo = 10 años

Si hacemos la cuenta dicho depósito generaría unos \$12196,40 de interés al finalizar un periodo de 10 años.

Implementaremos el algoritmo en pseudocódigo.

Algoritmo Interes_comuesto

Imprimir "Capital inicial Ci= ";

leer Ci;

Imprimir "tasa de interés anual t= ";

leer t;

Imprimir "cantidad de periodos de capitalización por unidad de tiempo n= ";

leer n;

tp=t/(100 * n)

Imprimir "Tasa por periodo= ",tp;

Imprimir "cantidad de unidades de tiempo u= ";

leer u;

Ic=Ci*(1+tp)^(n*u) - Ci ;

Imprimir "Interes compuesto calculado= ",Ic;

FinAlgoritmo

Algoritmo Interes_comuesto

Escribir 'Capital inicial Ci= '

Leer Ci

Escribir 'tasa de interés anual...

Leer t

Escribir 'cantidad de periodos ...

Leer n

tp ← t/(100*n)

Escribir 'Tasa por periodo= ',tp

Escribir 'cantidad de unidades ...

Leer u

Ic ← Ci*(1+tp)^(n*u)-Ci

Escribir 'Interes compuesto cal...

FinAlgoritmo

Como ejercicio, piense como haría para saber cual es el interés de cada uno de esos 120 meses, esto es 10 años. Además calcule, cual es el interés real que ha obtenido en esos 120 meses.

Como vemos, nuevamente antes de escribir nuestro código es fundamental tener muy en claro los pasos a seguir para la obtención de nuestro algoritmo, en este caso comprendiendo la fórmula de cálculo del interés compuesto.

Raiz cuadrada sin usar la función raíz cuadrada... bueno en verdad nosotros crearemos nuestra función raíz cuadrada....

Veamos un algoritmo para calcular la raíz cuadrada. Bajo ningún concepto vamos a demostrar la validez de dicho algoritmo. El uso de la función lo validará en la práctica. La teoría queda para los matemáticos y a quien le interese. Solo diremos que lo que usaremos se conoce como el algoritmo de Newton para la aproximación de la raíz cuadrada. Lo que se propone es un método numérico para aproximar dicha raíz cuadrada.

```
funcion estimacion<-raiz_cuadrada(X)
    estimacion=X/2.0;
    margen=0.000001;
    Mientras (ABS((estimacion * estimacion)-X)) >=margen Hacer
        cociente=X / estimacion;
        promedio=(cociente + estimacion)/2.0;
        estimacion=promedio;
    FinMientras
FinFuncion

Algoritmo main
    Escribir "Ingrese X= ",Sin Saltar;
    leer X;
    y=raiz_cuadrada(X);
    Imprimir "Raiz cuadrada de ",X," = ",y;
FinAlgoritmo
```

```
Funcion estimacion ← raiz_cuadrada(X)
    estimacion ← X/2.0
    margen ← 0.000001
    Mientras (ABS((estimacion*estimacion)-X))>=margen
        cociente ← X/estimacion
        promedio ← (cociente+estimacion)/2.0
        estimacion ← promedio
    FinFuncion

Algoritmo main
    Escribir 'Ingrese X= ',
    Leer X
    y ← raiz_cuadrada(X)
    Escribir 'Raiz cuadrada de ',X,' = ',y
FinAlgoritmo
```

Los métodos numéricos para la solución de ecuaciones, integrales, derivadas, etc son una parte fundamental de los algoritmos computacionales. Es una rama de la informática muy importante. Los métodos numéricos son fundamentales en simulación de sistemas físicos, biológicos, químicos, económicos, estadísticos entre decenas de ciencias que los usan.

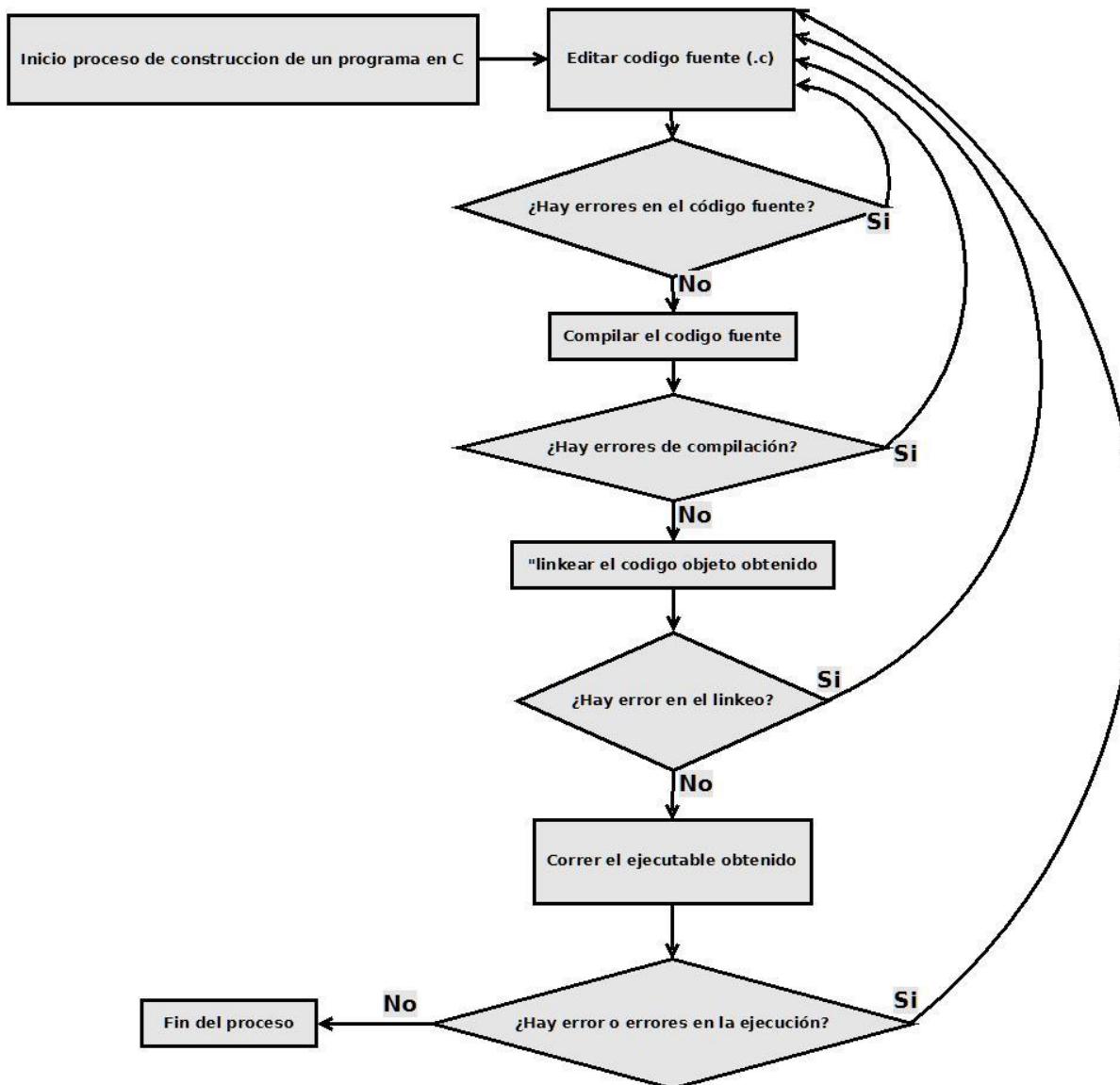
Que elementos, en general, forman parte de un código en C

Un código, en general, tiene varios de estos “elementos” (veremos los detalles de todo esto a lo largo del curso):

“elemento”	Ejemplo	Observaciones
Constantes	<code>const float PI=3.1415</code>	Ejemplo de constantes son PI, e, el IVA, la velocidad de la luz en el vacío. Se usa la palabra <code>const</code> y un tipo de dato (ya se verá esto en detalle), en este caso <code>float</code> que significa que se aceptan valores con decimales. Tanto <code>const</code> como <code>float</code> son palabras reservadas de C.
Constante simbólica	<code>#define PI 3.1415</code>	Usamos la instrucción de precompilador <code>#define</code> para hacer “lo mismo” que el ejemplo anterior. Preferiremos esta forma a la anterior en este curso. <code>Define</code> es palabra reservada de C.
Variables	<code>short A</code>	Esta expresión lo que hace es reservar espacio en memoria para una variable que llamaremos A dentro de nuestro código. Una variable es un objeto al que se le puede modificar su contenido en cualquier momento. A una constante no se le puede modificar su contenido. Al “crearse” la variable se da: un tipo de dato y un identificador. <code>Short</code> es el tipo de dato y <code>A</code> es el identificador en nuestro ejemplo.
Funciones de entrada de información	<code>scanf, fgets, gets, getche, getch, getchar, entre otras.</code>	Permiten ingresar “valores” a nuestro programa ya sea desde teclado como desde otras fuentes como lo son los archivos.
Funciones de salida de información	<code>printf, puts, putchar, fprintf, entre otras</code>	Permiten, en general, almacenar o escribir información en algún dispositivo, ya sea un archivo, ya sea la placa de video y de esa forma ver datos en el monitor, escribir al USB, entre otras.
Otras funciones	De comparación de datos, de copiado de datos, de conversión entre tipos de datos, etc	C tiene cientos de funciones, algunas que funcionan en cualquier plataforma y otras que solo funcionan en “alguna plataforma”. Por suerte hay un gran número de funciones que pueden ser usadas tanto en Linux, Windows, Mac, Android, etc. De otras habrá que buscar su equivalente que en general seguro existe o llegado el caso escribirla uno mismo.
Instrucciones para el control del flujo de información.	<code>For, while, do-while, switch-case, if-else.</code>	Permiten variar el orden de como se ejecutan las instrucciones o cuales instrucciones se ejecutarán en función de algo que suceda o no suceda.
Funciones y tipos de datos creados por el usuario.	Estructura, unión, vectores, matrices, cadenas de caracteres, listas, otros tipos de datos creados por el usuario. Funciones propias.	La mayor potencia de este lenguaje es la capacidad que tiene para aceptar funciones creadas por el usuario para poder resolver problemas concretos. Además puedo crear tipos de datos que me van a ayudar a resolver más fácilmente un determinado problema.
Operadores	Operadores matemáticos, lógicos, relacionales, etc	Permiten realizar las operaciones básicas de la matemática, realizar operaciones lógicas, generar expresiones sumamente complejas para resolver situaciones particulares.

La tabla anterior es para tener una orientación de lo que nos vamos a encontrar a la hora de realizar un código. Con estos elementos se “crean” los distintos algoritmos que nos van a ayudar a resolver los problemas informáticos. Para encontrar estos algoritmos muchas veces debemos recurrir a conocimientos de matemática, física, economía, estadística, biología, etc. Hay libros que tienen en específico el tema “algoritmos”. Es recomendable que los busque en la web y los consulte a medida que sus conocimientos en programación avanzan.

Secuencia de construcción de un programa en C

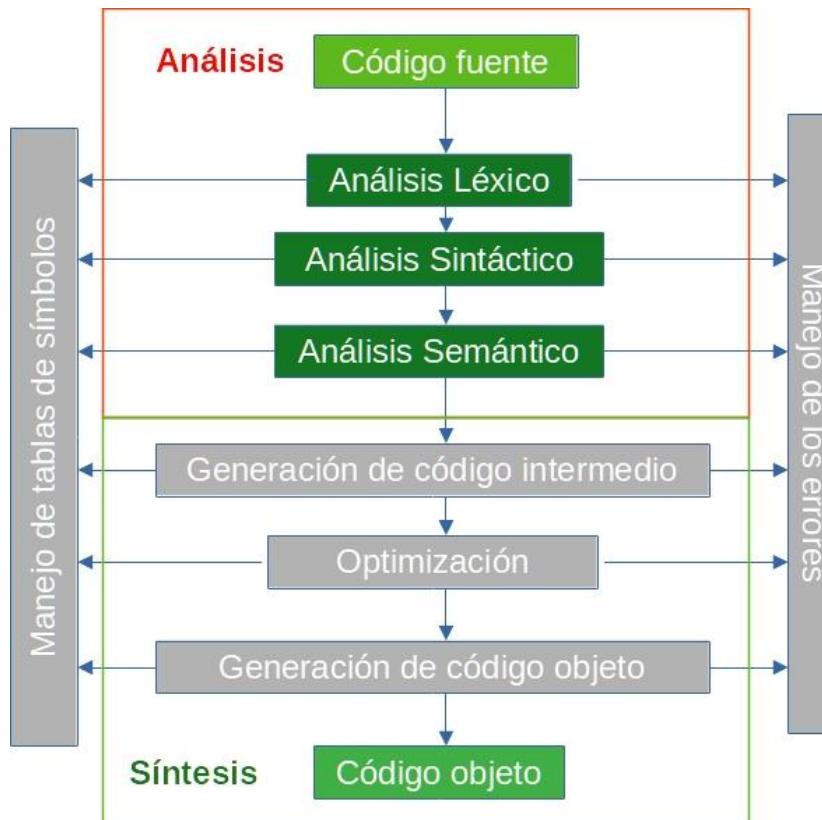


¿Qué es y como funciona un compilador?

¿A que se llama compilador?

Un **compilador** es un programa que permite traducir (convertir) el código fuente en lenguaje C, en nuestro caso será un archivo con extensión .c, a otro lenguaje de nivel inferior (típicamente lenguaje de máquina), permitiendo tener un código muchas veces directamente ejecutable. Lo que obtiene se denomina código objeto

¿Cómo se estructura un compilador?



¿Qué es Léxico?:

Como léxico se denomina al **conjunto de palabras que constituyen una lengua** y también se designa al **diccionario de un determinado idioma**.

También se conoce al **conjunto de palabras que son de uso particular en una región**: léxico mexicano, argentino, colombiano; **en una actividad o disciplina particular**: léxico legal, científico, informático; **o de un campo semántico en especial**: léxico del amor.

De allí que **léxico** y **vocabulario** sean términos **sinónimos**.

¿Qué es Sintaxis?:

Sintaxis es la **parte de la gramática que estudia la forma en que se combinan y se relacionan las palabras** para formar secuencias mayores como las oraciones, así como la función que desempeñan dentro de éstas.

A través de la sintaxis se estudia cómo están construidos los tipos de oraciones según el orden y el modo en que se relacionan las palabras dentro de una oración o las oraciones a fin de expresar el contenido de un discurso o concepto de manera clara y coherente.

La sintaxis tiene como principal función analizar el orden correcto de las palabras a fin de que las frases, oraciones, textos e ideas sean expresados de manera correcta para que pueda llegar el mensaje que se desea transmitir.

¿Qué es Semántica?:

Se denomina como semántica a la **ciencia lingüística que estudia el significado de las palabras y expresiones**, es decir, lo que las palabras quieren decir cuando hablamos o escribimos.

En **informática**, la semántica se encarga de estudiar desde un punto de vista matemático, el significado de los programas o funciones.

En resumen, el compilador debe determinar si esa palabra existe, está bien escrita, si cumple con el orden en el que se la debe encontrar en una determinada frase y además busca su significado o el significado de la expresión en la que se encuentra para obtener el código de dicha palabra o el código de dicha expresión.

No será lo mismo escribir **A = 5** que **5 =A** en informática. **A = 5 es una expresión válida** mientras que **5 = A no lo es**, a pesar de estar formadas ambas expresiones por los mismo símbolos. **El orden es fundamental y no puede ni debe ser cambiado.**

Esto es muy importante tenerlo siempre en cuenta.

Los compiladores son inflexibles, por eso uno tiene que conocer los detalles de cada lenguaje, sus palabras claves, sus signos de puntuación, cuando se consideran expresiones válidas y cuando no, respetar la sintaxis, esto es si una palabra va en minúscula no escribirla en mayúscula, etc. Todas estas violaciones a lo establecidos redundaran en “**warnings**” o en “**errores**”. A los **warnings** o a los **errores** nunca habría que ignorarlos. De hecho es imposible ignorar los errores ya que sin resolverlos no podremos llegar a nuestro ejecutable desde nuestro código fuente.

Con warnings no resueltos en general podremos llegar a nuestro ejecutable, pero siempre es muy importante evaluarlo, determinar su origen y de ser posibles, solucionar el problema que los genera.

NUNCA HAY QUE IGNORAR LOS WARNINGS SIN ANTES EVALUAR QUE ES LO QUE LOS CAUSA. EN MUCHAS INDUSTRIAS COMO LA NUCLEAR, LA AEROESPACIAL, LA AUTOMOTRIZ, ENTRE OTRAS NO ES PERMITEN EJECUTABLES QUE EN EL MOMENTO DE COMPILAR TENGAN WARNINGS IRRESUELTOS.

O SEA DEBEMOS LOGRAR CERO WARNINGS CERO ERRORS. INTENTE EN TODO LO POSIBLE LOGRAR ESTO.

Sistemas de numeración

Un sistema de numeración puede definirse como un conjunto de signos, relaciones, convenios y normas destinados a expresar de modo gráfico y verbal el valor de los números y las cantidades numéricas.

En la actualidad, se usan predominantemente sistemas de numeración de carácter posicional, donde cada numeral o guarismo representa un valor distinto según la posición que ocupa en la cadena numérica (por ejemplo, el numeral 1 significa unidad en la cantidad 1, pero es decena en 13, centena en 148, etcétera).

En un sistema de numeración se contemplan varios elementos fundamentales:

La base del sistema, que se define como un convenio de agrupación de sus unidades. Por ejemplo, la base 10 o decimal agrupa diez unidades, mientras que la binaria únicamente agrupa dos.

Los numerales del sistema, o cifras elementales que se utilizan, según la base. En el sistema decimal, se usan los numerales 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. En cambio, en el sistema binario tan sólo se emplean el 0 y el 1.

Las normas de combinación de los numerales para formar los números. Según ello, a cada cifra se le asocian dos propiedades: su valor absoluto intrínseco y su valor posicional o relativo, que depende de la posición que ocupa en la cantidad numérica.

Existen muchas bases numéricas pero en la práctica informática solo se usan cuatro.

Base 10 o decimal. Posee 10 símbolos que van del 0 al 9.

Base 2 o binaria. Posee 2 símbolos que son el 0 y el 1.

Base 8 u octal. Posee 8 símbolos que van del 0 al 7.

Base 16 o hexadecimal. Posee 16 símbolos que van del 0 al 9 y de A a F.

El C para PC (no para microcontroladores) utiliza todas menos la binaria.

Si yo escribo **23** será interpretado como **base 10** siempre.

Si yo escribo **023** será interpretado como **base 8** siempre.

Si yo escribo **0x23** será interpretado como **base 16** siempre.

Decimal	Binario	Octal	Hexadecimal	Decimal	Binario	Octal	Hexadecimal
0	000000	00	00	18	010010	22	12
1	000001	01	01	19	010011	23	13
2	000010	02	02	20	010100	24	14
3	000011	03	03	21	010101	25	15
4	000100	04	04	22	010110	26	16
5	000101	05	05	23	010111	27	17
6	000110	06	06	24	011000	30	18
7	000111	07	07	25	011001	31	19
8	001000	10	08	26	011010	32	1A
9	001001	11	09	27	011011	33	1B
10	001010	12	0A	28	011100	34	1C
11	001011	13	0B	29	011101	35	1D
12	001100	14	0C	30	011110	36	1E
13	001101	15	0D	31	011111	37	1F
14	001110	16	0E	32	100000	40	20
15	001111	17	0F	33	100001	41	21
16	010000	20	10	34	100010	42	22
17	010001	21	11	35	100011	43	23

Pasaje entre bases enteras y positivas

Pasaje de decimal a binario

Convertir 217_{10} a binario

$$\begin{array}{r} 217 \mid 2 \\ \textcircled{1} \quad 108 \mid 2 \\ \textcircled{0} \quad 54 \mid 2 \\ \textcircled{0} \quad 27 \mid 2 \\ \textcircled{1} \quad 13 \mid 2 \\ \textcircled{1} \quad 6 \mid 2 \\ \textcircled{0} \quad 3 \mid 2 \\ \textcircled{1} \quad \textcircled{1} \end{array}$$

↓o ↓eo ↓sc

$$217_{10} = 11011001_2$$

Pasaje de binario a decimal

Convertir 11011001_2 a base 10

$$\begin{array}{r} 11011001 \\ | \quad | \\ 1 \times 2^0 = 1 \\ 0 \times 2^1 = 0 \\ 0 \times 2^2 = 0 \\ 1 \times 2^3 = 8 \\ 1 \times 2^4 = 16 \\ 0 \times 2^5 = 0 \\ 1 \times 2^6 = 64 \\ 1 \times 2^7 = \underline{128} \\ \hline 217_{10} \end{array}$$

Pasaje de decimal a octal

Convertir 217_{10} a octal

$$\begin{array}{r} 217_{10} \text{ L } 8 \\ \text{①} \quad \quad \quad 27 \text{ L } 8 \\ \text{③} \quad \quad \quad \text{③} \\ \text{Le o} \quad \quad \quad \text{asc} \end{array}$$

$$217_{10} = 331_8$$

Pasaje de octal a decimal

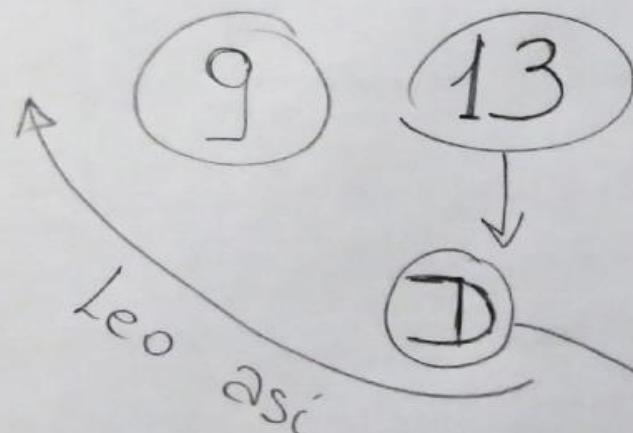
Convertir 331_8 a base 10

$$\begin{array}{r} 3 \quad 3 \quad 1 \\ | \quad | \quad | \\ 1 \times 8^0 = 1 \\ 3 \times 8^1 = 24 \\ 3 \times 8^2 = \frac{192}{217_{10}} \end{array}$$

Pasaje de decimal a hexadecimal

Convertir 217_{10} a base 16

$$217_{10} \mid 16$$



$$217_{10} = D9_{16}$$

Decimal	Hexa
0	0
1	1
2	2
:	:
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Pasaje de hexadecimal a decimal

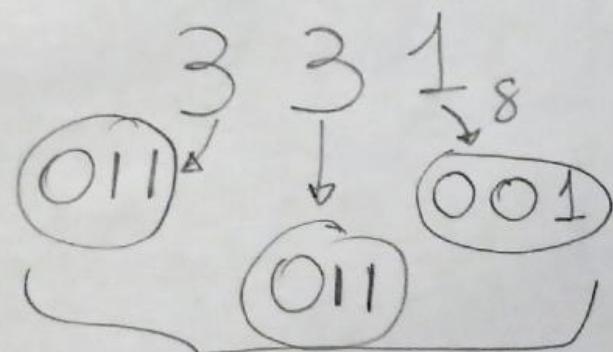
Convertir $D9_{16}$ a decimal

$$\begin{array}{r} D \quad 9 \\ | \quad \boxed{16} \\ \xrightarrow{\hspace{1cm}} 9 \times 16^0 = 9 \\ \xrightarrow{\hspace{1cm}} 13 \times 16 = \frac{208}{217_{10}} \end{array}$$

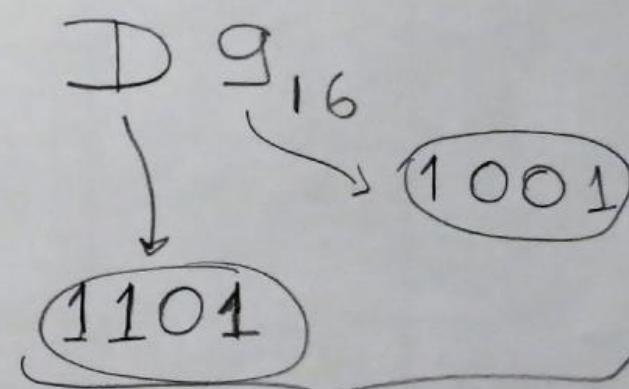
$$D9_{16} = 217_{10}$$

Pasaje de binario a octal y hexadecimal agrupando de a 3bits o de a 4bits

Convertir octal y hexa a base2



011011001₂



11011001₂

El lenguaje C

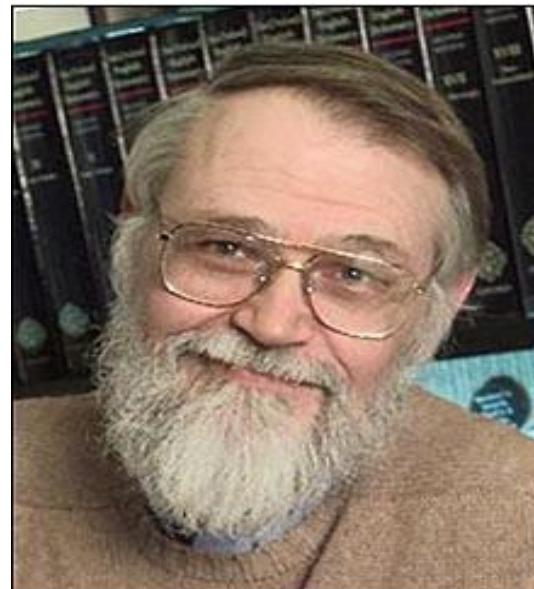
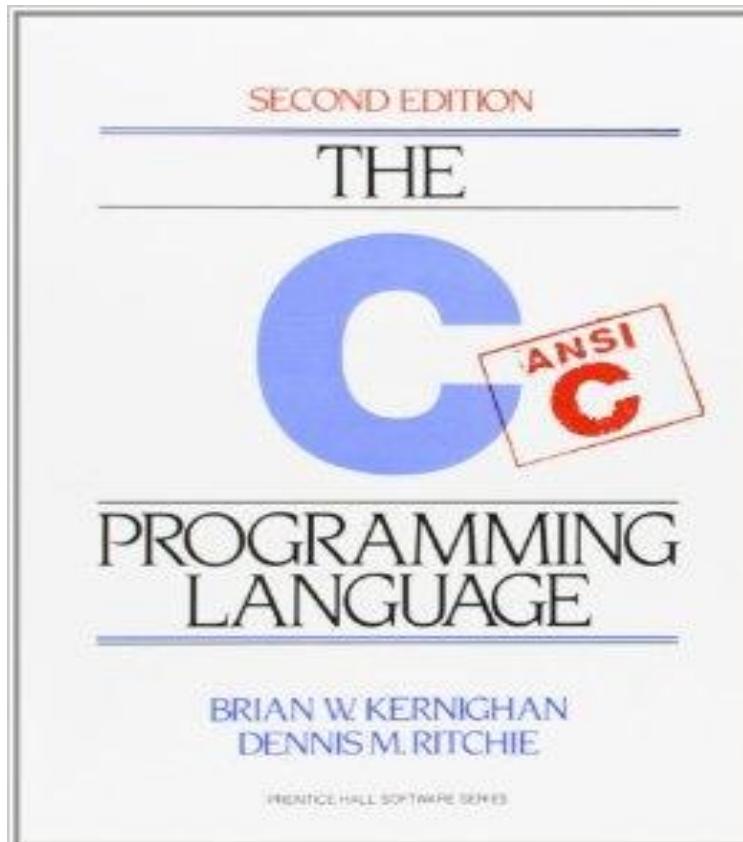
Vamos a aprender en este curso
a programar en lenguaje C



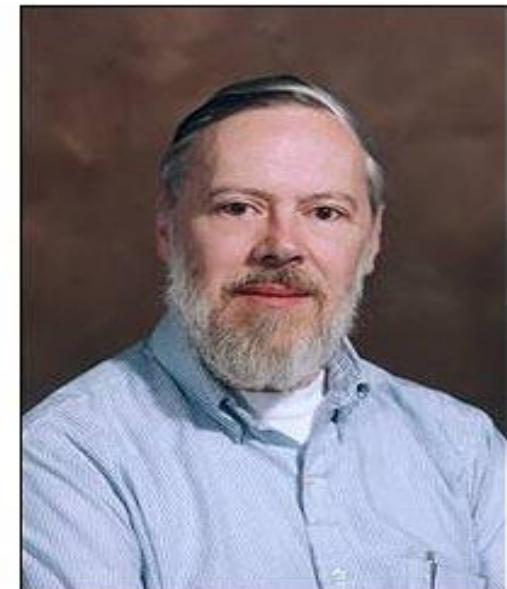
... Y en ANSI C !!!

... en todo lo que sea posible. Como se verá el C es un lenguaje vivo y en constante revisión. Hoy la norma C17 es la que se encuentra en vigencia y la que se propone para nuevos proyectos. Esto dependerá de que el compilador este actualizado a esa norma ISO. El mingw64 de la versión de Codeblocks 20.03 para 64 bits lo está. Por eso se recomienda, siempre que se pueda, actualizar a la ultima versión.

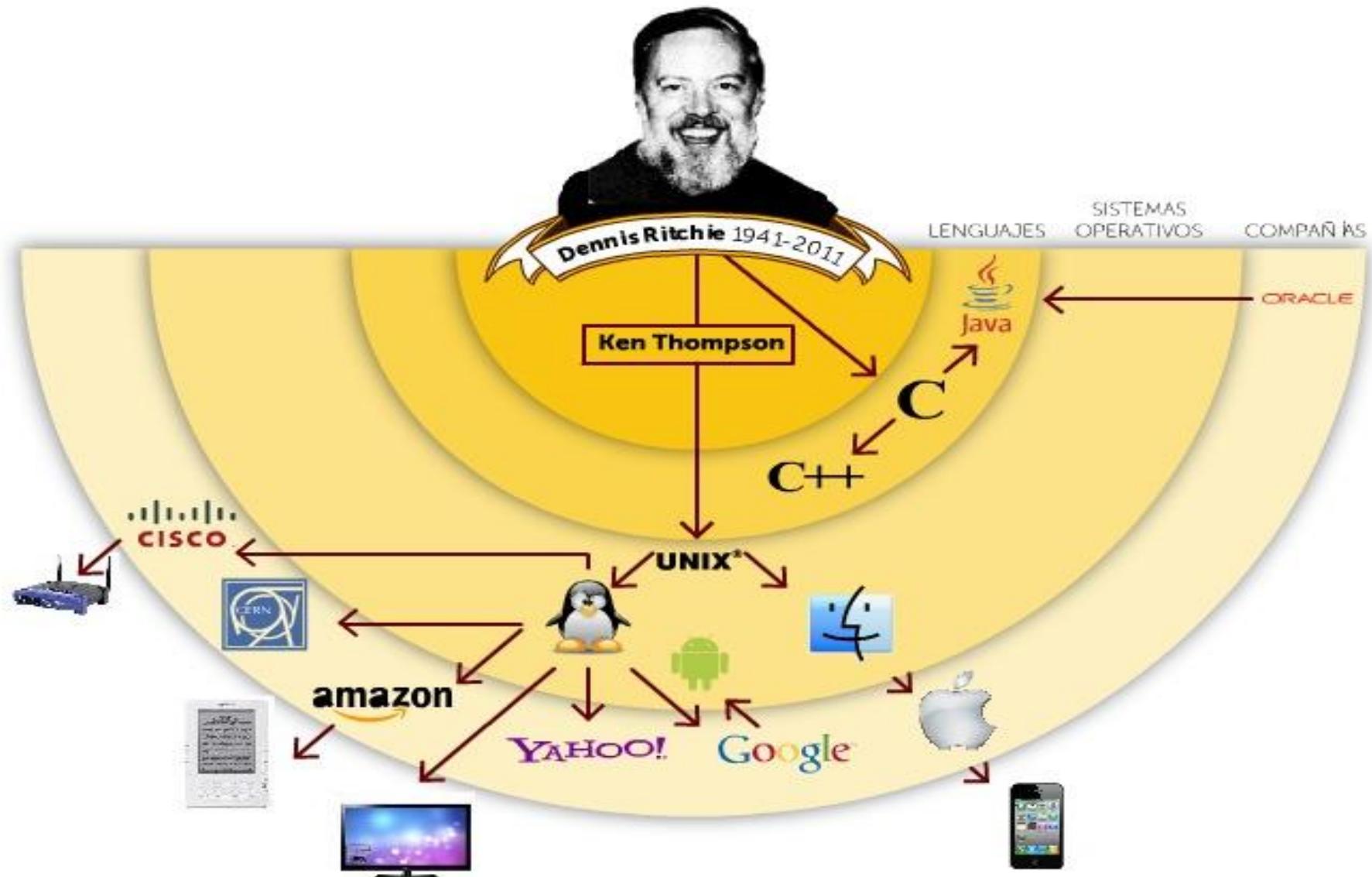
Todo antes de la estandarización es generalmente llamado "K & R C" por Brian W. Kernighan y Dennis M. Ritchie, sus autores.



Brian Kernighan



Dennis Ritchie



El lenguaje C posee 32 palabras reservadas que se escribirán siempre en minúscula.

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

- Ninguna es para entrada ni salida de datos. Se usan funciones de librería.
- Son de control de flujo de programa (if, while, do, else, for, etc).
- Son tipos de datos o modificadores (short, long, int, float, char, unsigned, struct, etc).
- Definen la forma de almacenamiento de la información (auto, static, etc).
- Siempre se escriben en minúscula. Tengalo muy en cuenta.
- No se puede usar el identificador de cada una de ellas para otra cosa diferente para lo que fueron creadas.
- Sizeof es un operador, no una función, que permite conocer la cantidad de bytes reservados para almacenar una determinada información o el tamaño de un determinado tipo de dato.

¿Por qué estudiar C (en 2022)?

C tiene las siguientes características:

- Es un lenguaje estructurado.
- Puedo hacer código de bajo nivel y de alto nivel. Puedo manejar el hardware “directamente”.
- No depende del hardware, por lo que se puede migrar a otros sistemas. Corre sobre el hardware no necesitando “máquinas virtuales” o interpretes.
- No es un código para aplicaciones específicas. Puedo hacer “de todo”.
- Como lenguaje de bajo nivel en general reemplaza al Assembler. De hecho para esto fue creado.
- Los programas son producidos de forma rápida y son sumamente potentes. Son en general muy veloces y eficientes. Se pueden optimizar.
- Tiene muchos tipos de datos para definir constantes y variables. Puede de esta forma optimizar el uso de la memoria y el acceso a la información
- Una enorme cantidad de documentación, sitios web de referencia, libros, etc.
- Con C se pueden escribir sistemas operativos, (como el núcleo de Linux que el el 96% de las líneas de código son en C!!), juegos, aplicaciones matemáticas, científicas, comerciales, etc.

¿Por que aprender C?:

- Porque el compilador es gratuito y existen muchas versiones del mismo. No pago nada por usarlo.
- Porque existen IDE's gratuitos y muy potentes en todos los sistemas operativos, incluso podremos programar desde páginas web.
- Existe una enorme cantidad de documentación, sitios web de referencia, libros, apuntes, etc.
- Con C se pueden escribir sistemas operativos, (como el núcleo de Linux que el el 96% de las líneas de código son en C!!), juegos, aplicaciones matemáticas, científicas, comerciales, etc.
- Porque aprendiendo C podrás aprender relativamente fácil otros lenguajes como Python, Java, R, etc.
- **PORQUÉ SE USA!!** La programación de microcontroladores es en general en C (AVR, PIC, ARM, etc).
- Porque según el índice TIOBE (<https://www.tiobe.com/tiobe-index/c/>) fue el lenguaje del año en 2008, 2017 y 2019 y es según el mismo índice el que ocupa el puesto Nro 1 (<https://www.tiobe.com/tiobe-index/>).
- Porque existen continuas revisiones del mismo y es norma siendo la última la **ISO/IEC 9899:2018**.
- Es relativamente fácil de aprender.
- Puedo mezclar código C con Assembler dentro del mismo programa.
- Posee librerías para “casi” todo. Hay miles de programadores en el mundo escribiendo librerías para gráficos, juegos, bases de datos, acceso al hardware, etc.

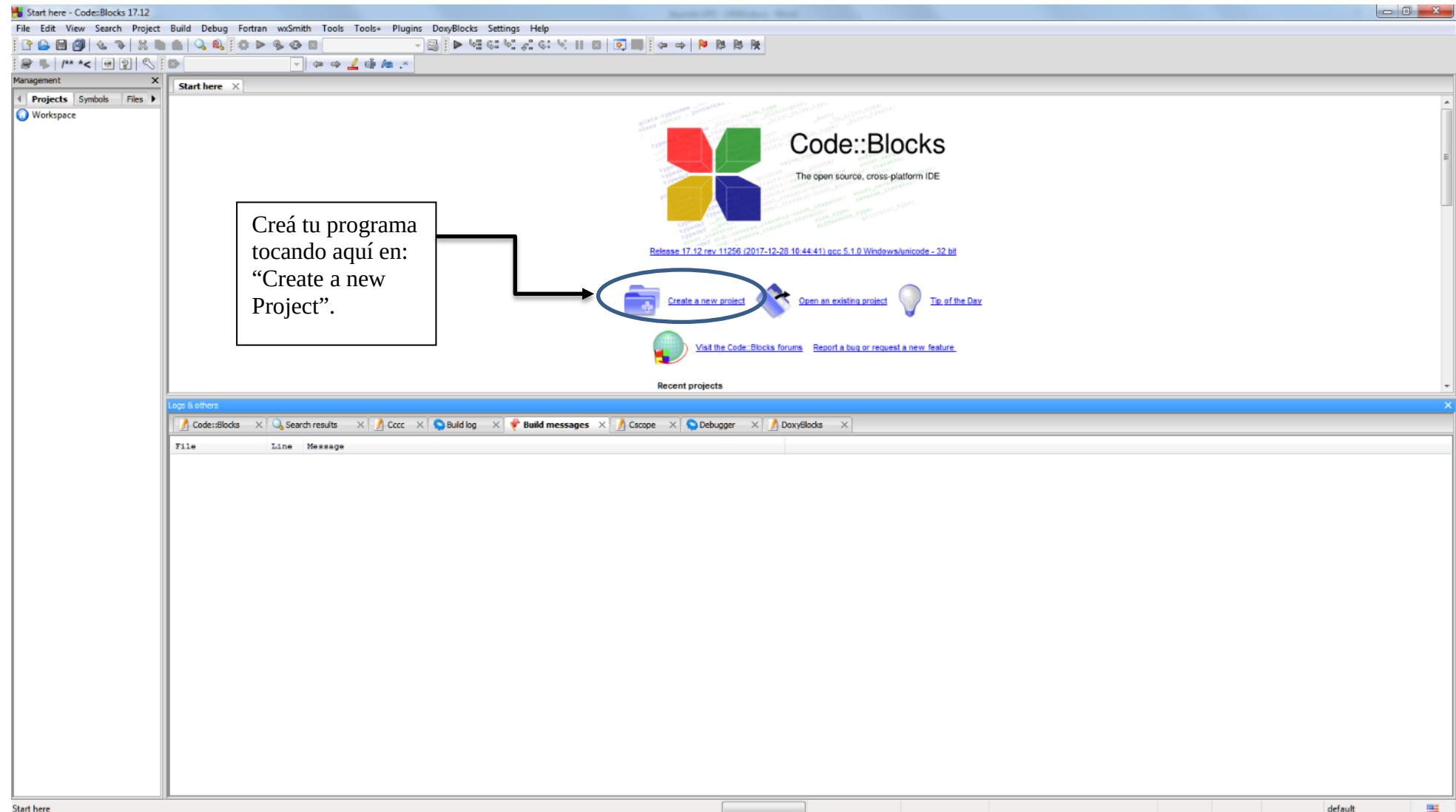
Primeros pasos en C

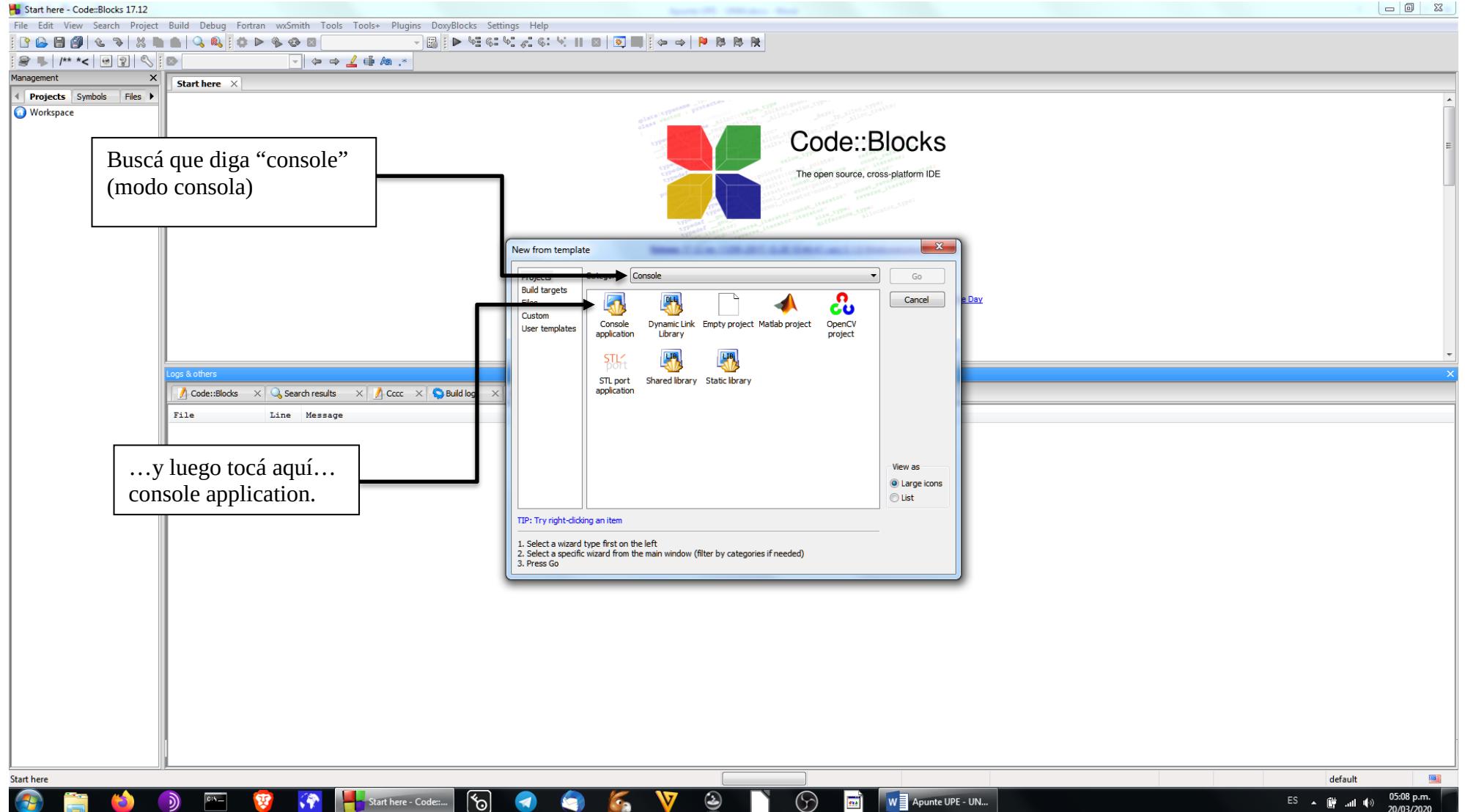
Vamos a ver los pasos necesarios para poder probar nuestro compilador y saber si esta bien instalado o no.

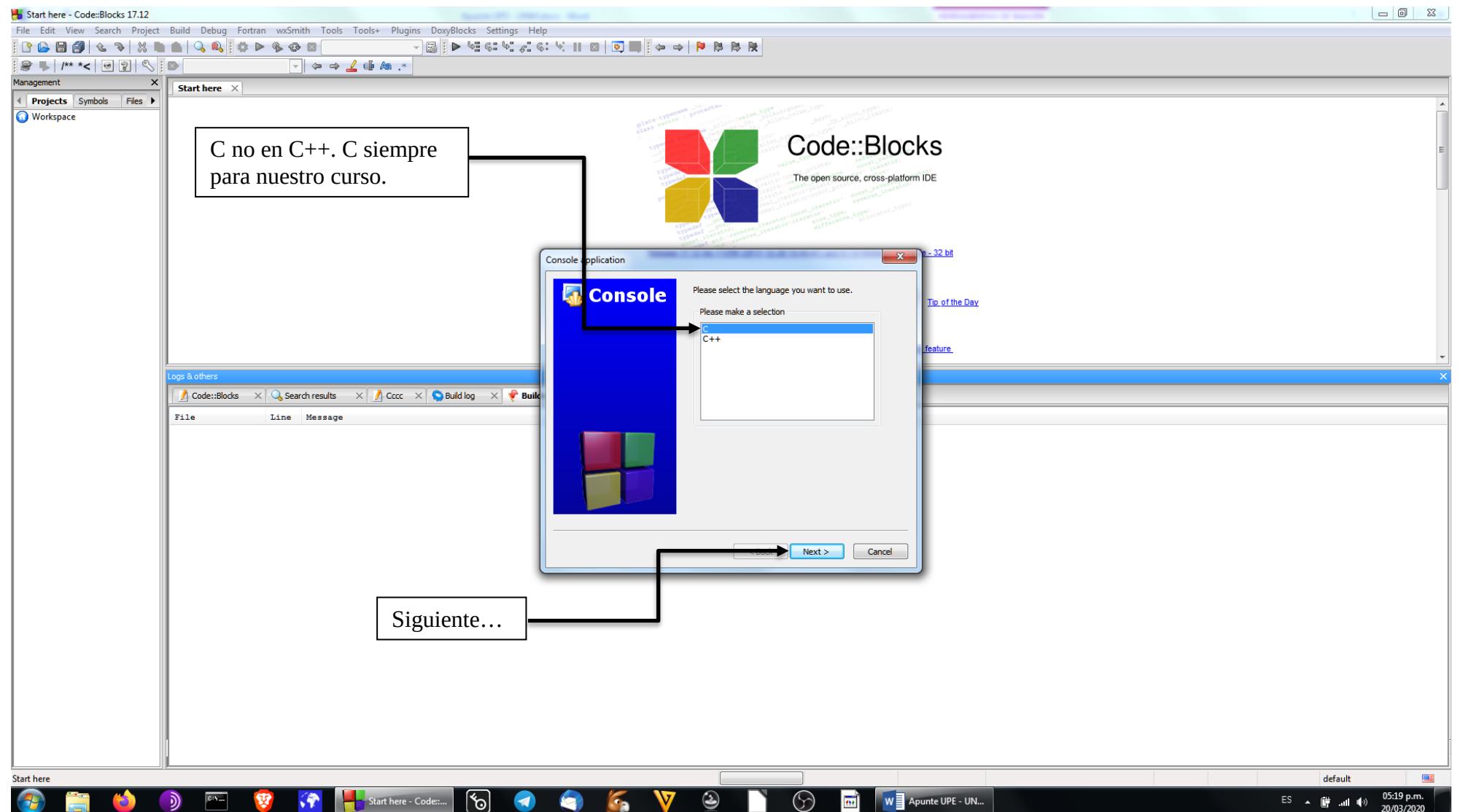
Para esto, no importa si se tiene la version de codeblocks 17.12 o 20.03 de codeblocks, los pasos son exactamente los mismos. (a Marzo de 2022).

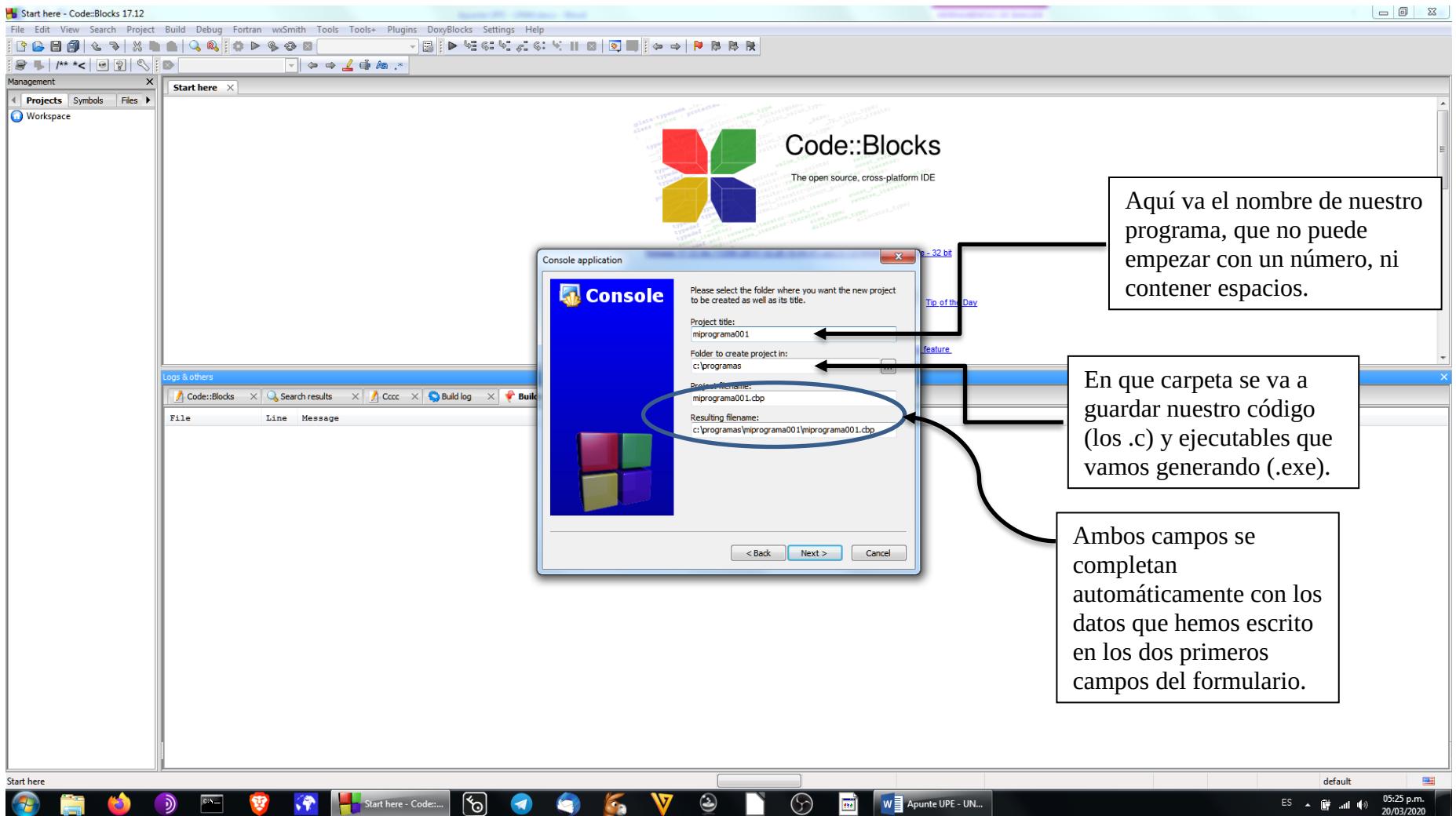
Debemos siempre tener en cuenta que vamos a programar en C no en C++.

A continuacion, las siguientes pantallas nos guiaran en nuestro primer “hola mundo”.

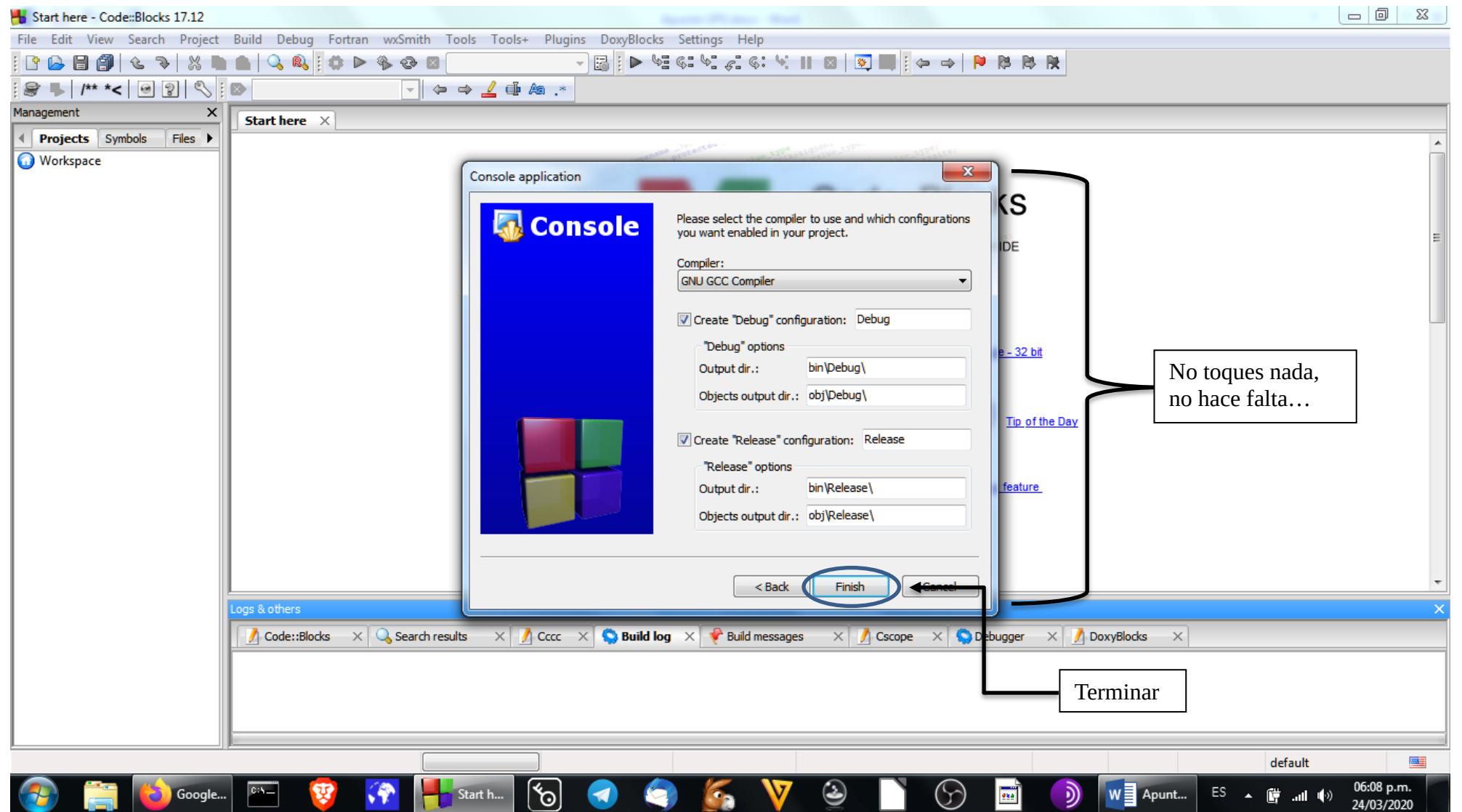




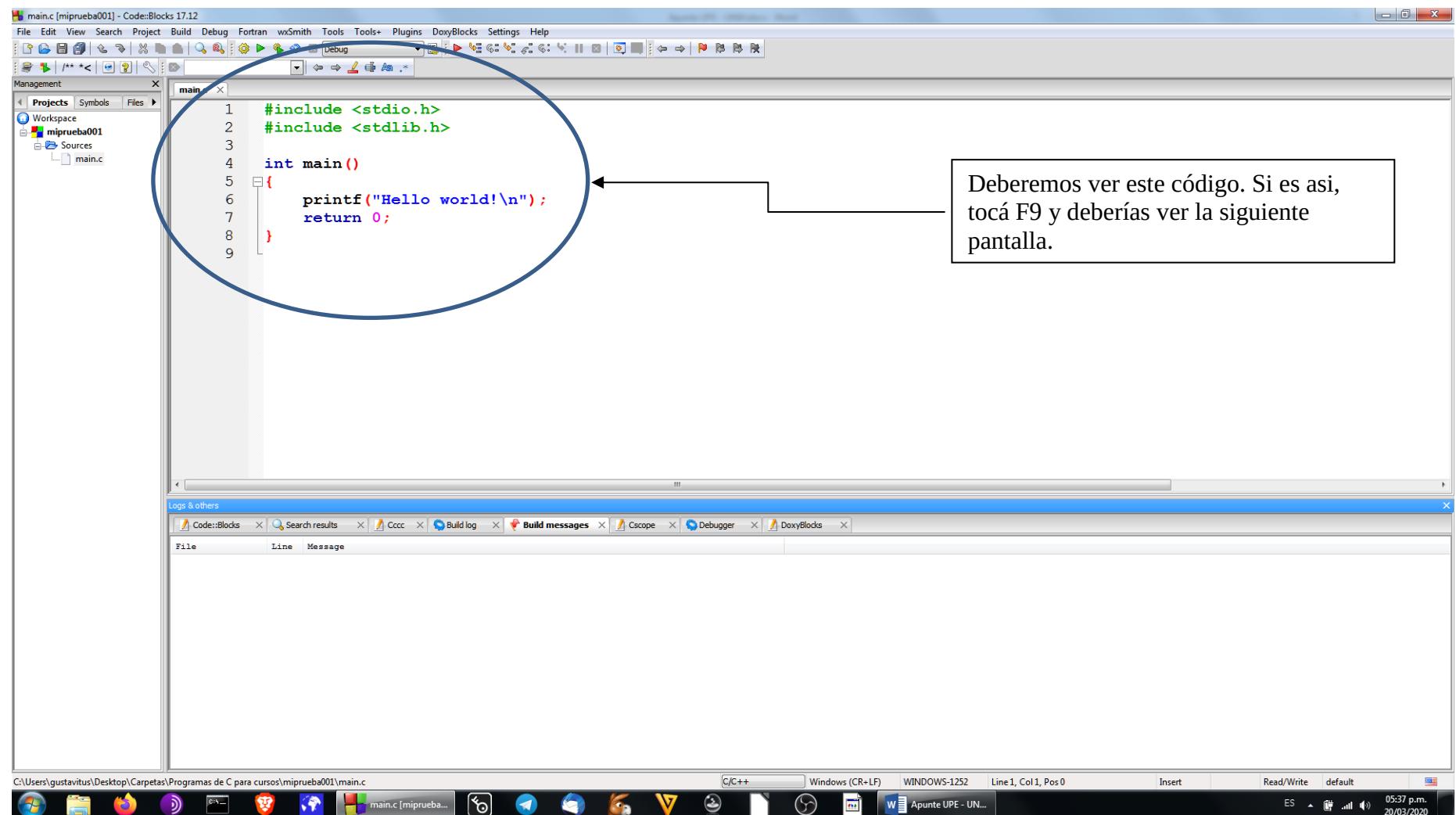




La carpeta donde vamos a guardar los códigos .c y los ejecutables .exe debe ser creada en un lugar de nuestro disco que sea de lectura y escritura. Si no es su maquina deberá informarse con el responsable de la misma. Puede crearla por ejemplo en el escritorio que en general no hay problemas.



...Y si todo fue bien...



main.c [miprueba001] - Code::Blocks 17.12

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

Management X

Projects Symbols Files >

Workspace

miprueba001

Sources main.c

main

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     printf("Hello world!\n");
7     return 0;
8 }
```

Logs & others

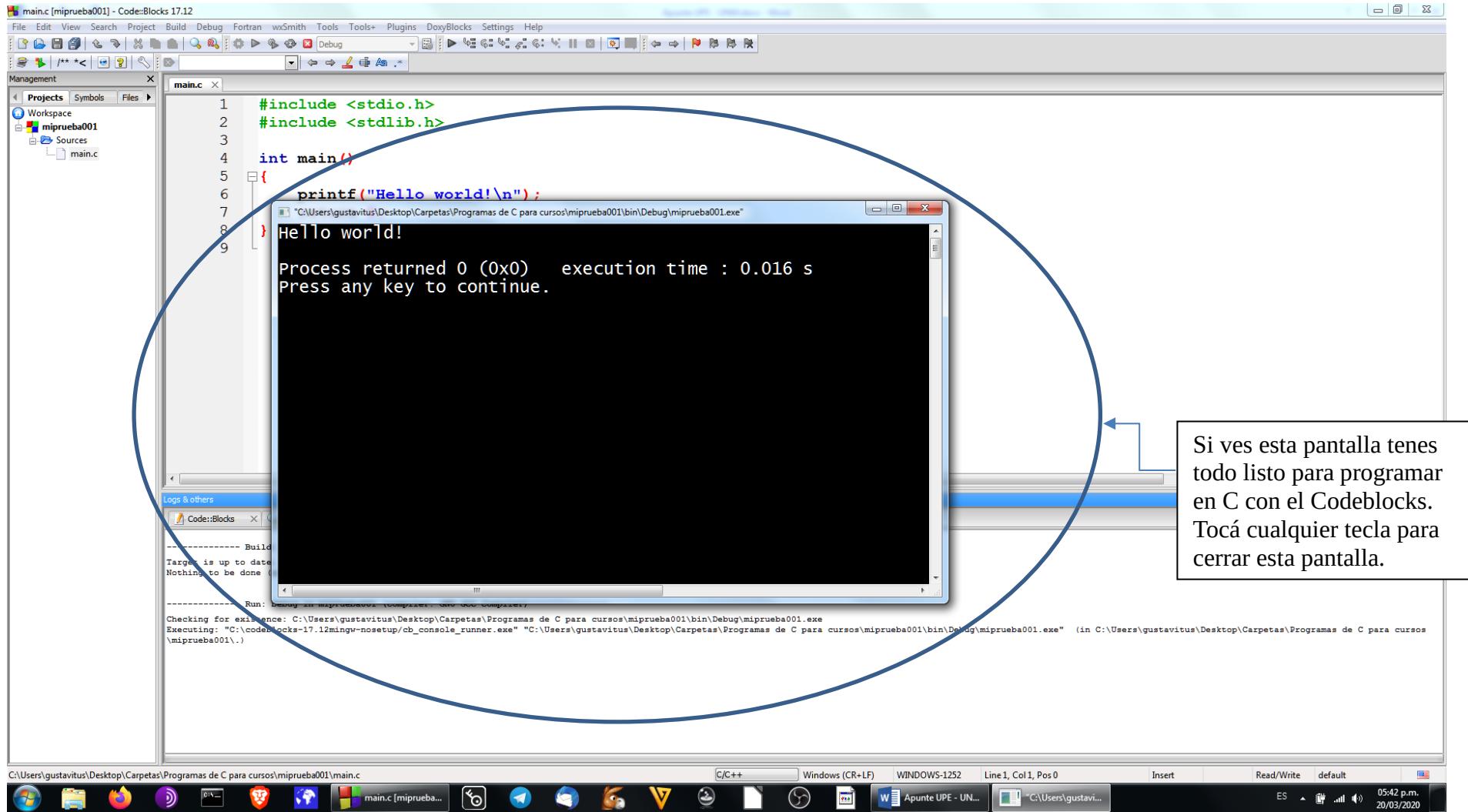
Code::Blocks Search results Cccccc Build log Build messages Cscope Debugger DoxyBlocks

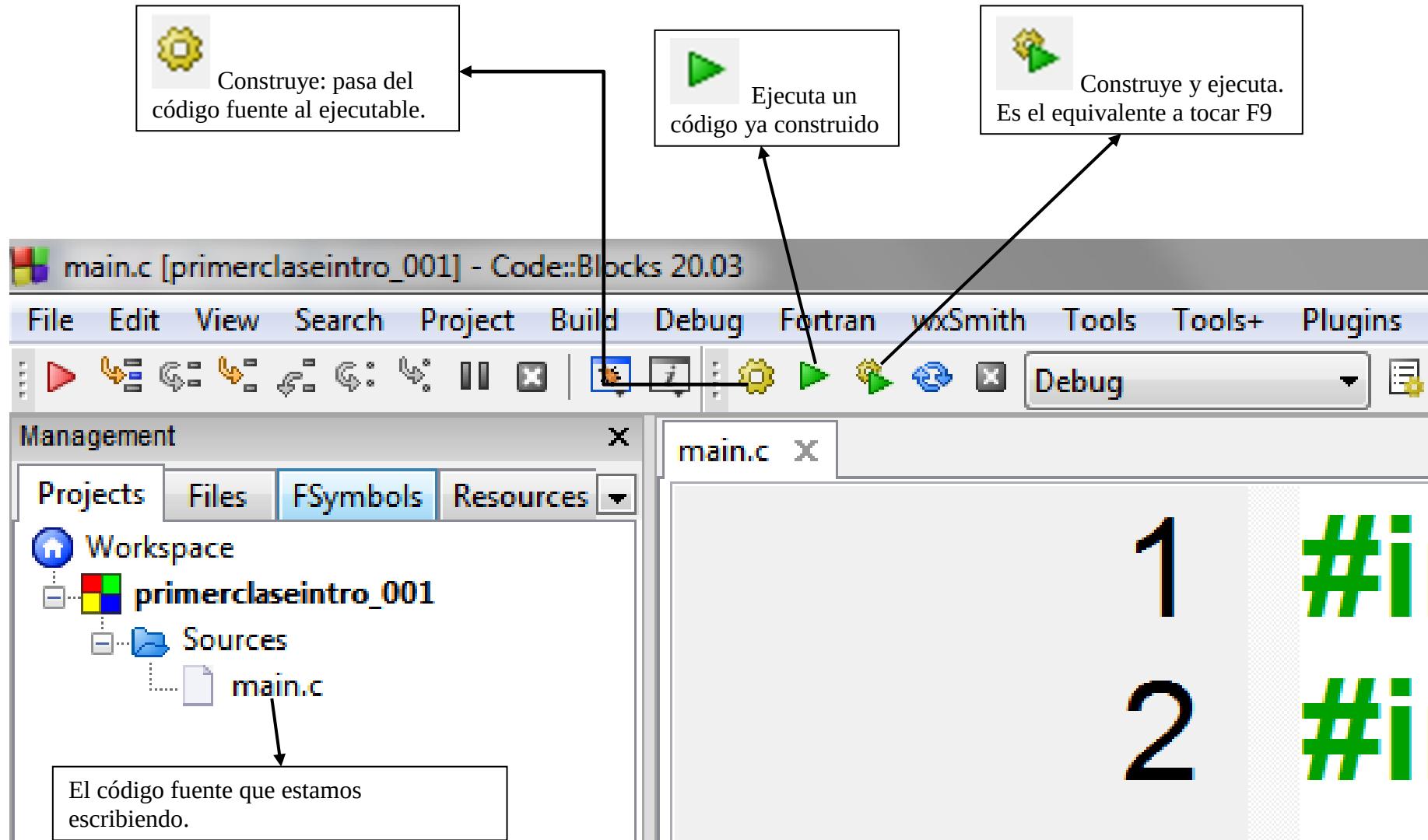
File Line Message

C:\Users\gustavitus\Desktop\Carpetas\Programas de C para cursos\miprueba001\main.c C/C++ Windows (CR+LF) WINDOWS-1252 Line 1, Col 1, Pos 0 Insert Read/Write default

Apunte UPE - UN... ES 05:37 p.m. 20/03/2020

Deberemos ver este código. Si es así, tocá F9 y deberías ver la siguiente pantalla.





Detalle de la pantalla de codeblocks con los iconos mas importantes para comenzar a programar.

Nuestro primer código !!!

```
#include <stdio.h>

int main(void)
{
    printf("Hola mundo");
    return 0;
}
```

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

The diagram illustrates the inclusion of a header file. A callout bubble labeled "Header" points to the line "#include <stdio.h>".

¿Que es un “header”?

- Es un archivo de texto ASCII (**búscate una tabla ASCII, hay para celulares de ser necesario**).
- Está a nivel humano no a nivel de máquina. Es un texto. (No modificar nada de nada ahí!!).
- Posee prototipos de funciones (SABER!!).
- Posee definiciones de tipos de datos.
- Posee definiciones de constantes.

Prototipos de funciones (**¡saber!**)

Contiene tres (3) elementos:

- Nombre de la función
- Tipo de dato devuelto (solo uno)
- Tipo/s de dato/s que recibe la función (uno o más)

Formato:

`tipo_devuelto nombre (tipo/s_recibido/s);`

Creando programas en C con gcc bajo Linux (desde línea de comando)

Deberás seguir estos simples pasos:

- a) Con un editor de texto escribí el código fuente.
- b) Salvalo con un nombre y la extensión del archivo .c (ej. **codigo001.c**)
- c) Abrí el terminal (consola de linux) y escribe: **\$ gcc codigo001.c -o codigo001.exe**
- d) **Para ejecutarlo** si no hay error (se compiló y linkó correctamente) solo queda probarlo, para lo cual haremos en la misma consola: **\$./codigo001.exe**

Para probar por primera vez el compilador es útil escribir el siguiente código fuente:

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

Si como resultado de hacer **\$ gcc codigo001.c -o codigo001.exe** se obtiene que no se encuentra **stdio.h** una solución es: **sudo apt-get install build-essential**

En algunos casos, pedirá la clave de root del linux y luego se deberá seguir las indicaciones que aparecen en pantalla.

Estructura recomendada de un programa en C

No es la única pero entendemos que hasta conocer bien C, es la mejor...

Headers (todos los necesarios)

Definición de tipos de datos creados por usuario(si los hay)

Prototipos de funciones propias (si las hay creadas por el usuario)

```
int main (void)
{
```

 Declaración de variables locales a la función main

 Código (instrucciones)

```
        return 0;
```

```
}
```

```
tipo funcion1 (tipo,...)
```

```
{
```

...

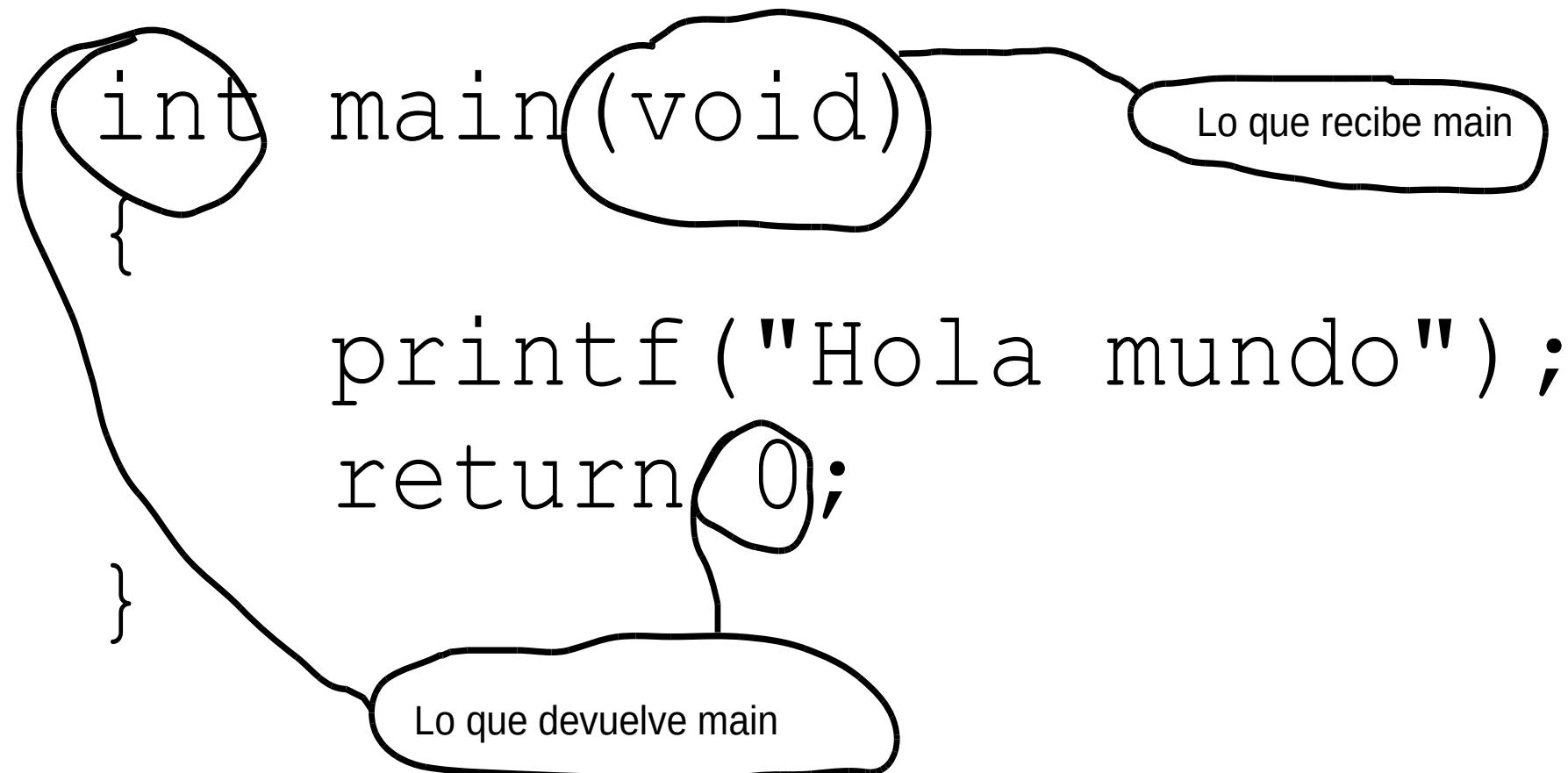
...

```
}
```

Nota muy importante: HOY LA FORMA QUE SE DEBE UTILIZAR ES: **int main(void)**

Otras formas, hoy muchos compiladores la consideran como ERROR y no permiten compilar.

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

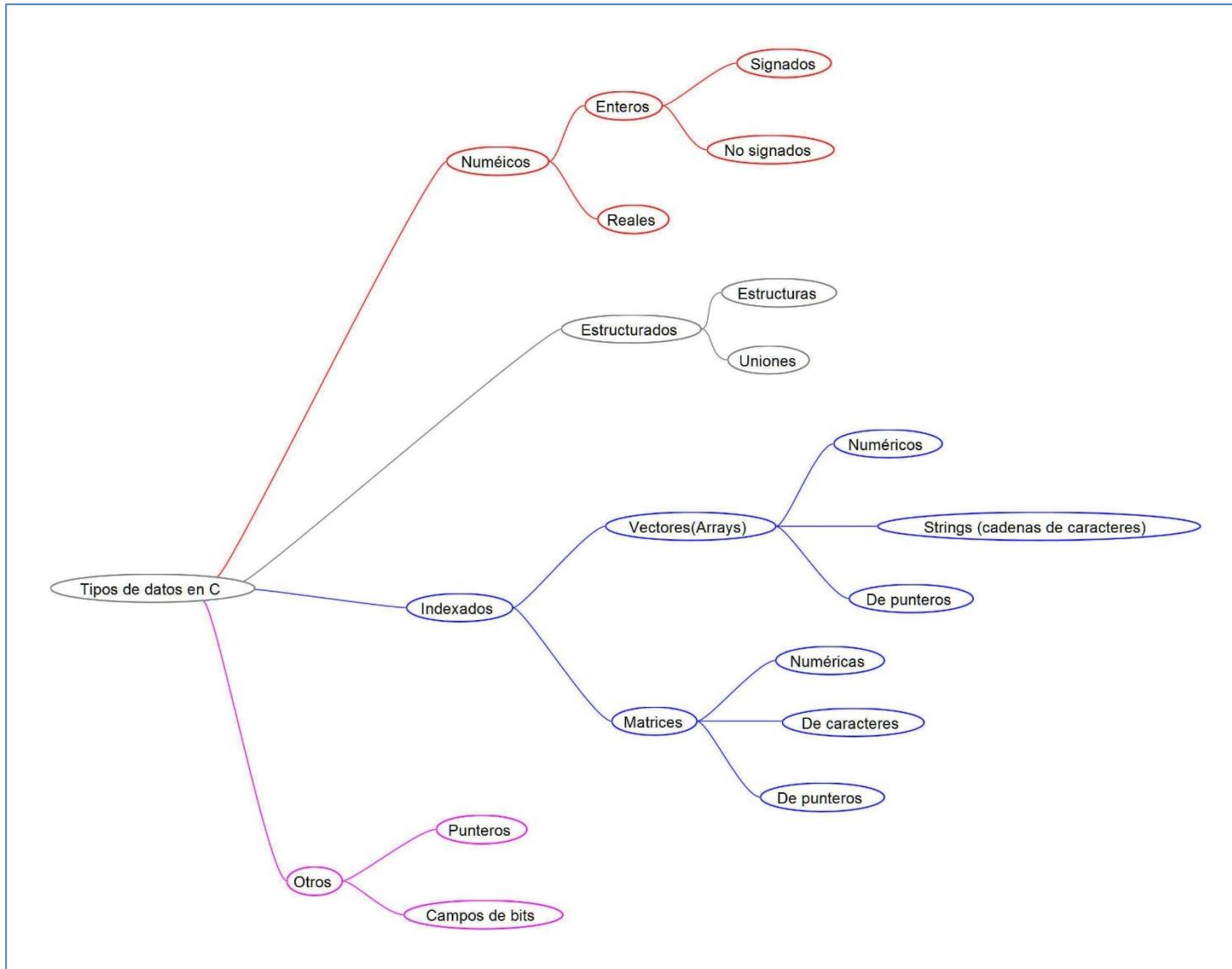


Tipos de datos en C (tema fundamental)

Los tipos de datos definen:

- El espacio en memoria utilizado para almacenar información (en bytes). Dicha información puede tener la forma de constante o variable.
- El rango de valores que es posible almacenar y su naturaleza.
- Que información se puede manejar con ese tipo de dato (Ej. Un puntero por ejemplo solo admite direcciones de memoria)

Gráfico de tipos de datos en C



Distintos tipos de datos standard en C y rangos de valores

Notación completa	Notación abreviada	Bytes	Bits	Rango de valores	Convenio al que responde	Observaciones
signed char	char	1	8	-128 a + 127	Complemento a 2	
unsigned char	no tiene	1	8	0 a 255	Magnitud y signo	
signed short int	short	2	16	-32768 a + 32767	Complemento a 2	
unsigned short int	unsigned short	2	16	0 a 65535	Magnitud y signo	
signed int	int	4 (en gral.)	32 (en gral.)	-2147483648 a +2147483647	Complemento a 2	El tamaño depende de la plataforma y del compilador. Use short y long.
unsigned int	unsigned int	4 (en gral.)	32 (en gral.)	0 a +4294967296	Magnitud y signo	
signed long int	long	4	32	-2147483648 a +2147483647	Complemento a 2	
unsigned long int	unsigned long	4	32	0 a +4294967296	Magnitud y signo	
signed long long int	long long	8	64	-9223372036854775808 a +9223372036854775807	Complemento a 2	En windows 64 bits se deben incluir, en general, instrucciones para su uso.
unsigned long long int	unsigned long long	8	64	0 a +18446744073709551616	Magnitud y signo	
float	no tiene	4	32	$\pm 1.175494351 \times 10^{-38}$ a $\pm 3.402823466 \times 10^{38}$	IEEE 754	El único, en general, definido en todos los compiladores y plataformas.
double	no tiene	8 o mas	64	$\pm 2.2250738585072014 \times 10^{-308}$ a $\pm 1.7976931348623158 \times 10^{308}$	IEEE 754	Pueden no estar definidos en ciertos compiladores y plataformas.
long double	no tiene	8 o mas	64	$\pm 2.2250738585072014 \times 10^{-308}$ a $\pm 1.7976931348623158 \times 10^{308}$	IEEE 754	

Rango de valores de tipos enteros y fórmulas para obtenerlos

Tipo de dato	Valor mínimo almacenable	Fórmula para el mínimo	Valor máximo almacenable	Formula para el máximo	Cantidad en bytes	Rango en fórmulas
char	-128	$-(2^7)$	+128	$(2^7) - 1$	1	$-(2^7) \text{ a } (2^7) - 1$
unsigned char	0	0	+255	$(2^8) - 1$	1	$0 \text{ a } (2^8) - 1$
short	-32768	$-(2^{15})$	+32767	$(2^{15}) - 1$	2	$-(2^{15}) \text{ a } (2^{15}) - 1$
unsigned short	0	0	+65535	$(2^{16}) - 1$	2	$0 \text{ a } (2^{16}) - 1$
long	-2147483648	$-(2^{31})$	+2147483647	$(2^{31}) - 1$	4	$-(2^{31}) \text{ a } (2^{31}) - 1$
unsigned long	0	0	+4294967296	$(2^{32}) - 1$	4	$0 \text{ a } (2^{32}) - 1$
long long	-9223372036854775808	$-(2^{63})$	+9223372036854775807	$(2^{63}) - 1$	8	$-(2^{63}) \text{ a } (2^{63}) - 1$
unsigned long long	0	0	+18446744073709551615	$(2^{64}) - 1$	8	$0 \text{ a } (2^{64}) - 1$

Los números reales en programación C (IEEE 754)

Como se pudo ver en tablas anteriores (que aquí volvemos a copiar) :

float	no tiene	4	32	$\pm 1.175494351 \times 10^{-38}$ a $\pm 3.402823466 \times 10^{38}$	IEEE 754	El único, en general, definido en todos los compiladores y plataformas.
double	no tiene	8 o mas	64	$\pm 2.2250738585072014 \times 10^{-308}$ a $\pm 1.7976931348623158 \times 10^{308}$	IEEE 754	Pueden no estar definidos en ciertos compiladores y plataformas.
long double	no tiene	8 o mas	64	$\pm 2.2250738585072014 \times 10^{-308}$ a $\pm 1.7976931348623158 \times 10^{308}$	IEEE 754	

...los datos de tipo Real se manejan dentro de C con los tipos float, double y long double, que se encuentran normados en la NORMA IEEE754.

Tenga en cuenta que según el sistema operativo y el hardware pueden cambiar de tamaño y de rango de valores. En general solo el float es garantizado como de 4 bytes en cualquier sistema. Use sizeof de ser necesario. (ej. **sizeof(double)** que devolverá la cantidad de bytes que ocupa en memoria).

No entraremos en detalle respecto a la norma. Solo haremos unos comentarios de índole práctico.
De ser necesario consulte la norma IEEE754.

En las siguientes capturas de pantalla de una app de Android llamada **Floating Point Konverter** (muy recomendable), vemos que en un float (32 bits) pasa lo siguiente:

The screenshot shows a user interface for a floating-point converter. At the top, there are two radio buttons: "32 Bit" (selected) and "64 Bit". Below them are four input fields: "Decimal" (12.5), "Exponential" (1.5625 · 2³), "Hexadecimal" (41480000), and "Actual decimal value" (12.5). There are also two output fields: "Rounding error" (0.0) and a bit representation section labeled "Bits". The bit representation table is as follows:

4	1	4	8	0	0	0	0
0	1	0	0	0	0	0	0
+ Exp.: 0x82	Mantissa: 0x480000						

Annotations with arrows explain the following concepts:

- "32 Bit significa tipo de dato float para nosotros." points to the "32 Bit" radio button.
- "Valor que queremos almacenar" points to the "Decimal" input field (12.5).
- "Valor que se almacena" points to the "Actual decimal value" output field (12.5).
- "Diferencia entre lo que queríamos almacenar y lo almacenado" points to the "Rounding error" output field (0.0).
- "Asi es como se almacena en la norma IEEE754. 0x significa en base 16 (hexadecimal)." points to the bit representation table.

Se puede ver que 12.5 se almacena correctamente en un dato tipo float.

Veamos ahora estos ejemplos:

32 Bit 64 Bit

Decimal:
5.1

Exponential:
 $1.274999976158142 \cdot 2^2$

Hexadecimal:
40A33333

Actual decimal value:
5.099999904632568359375

Rounding error:
9.5367431640625E-8

Bits:

4	0	A	3	3	3	3	3
0	1	0	0	0	0	1	0
+ Exp.: 0x81	Mantissa: 0x233333						

32 Bit 64 Bit

Decimal:
3.1

Exponential:
 $1.5499999523162842 \cdot 2^1$

Hexadecimal:
40466666

Actual decimal value:
3.099999904632568359375

Rounding error:
9.5367431640625E-8

Bits:

4	0	4	6	6	6	6	6
0	1	0	0	0	0	1	0
+ Exp.: 0x80	Mantissa: 0x466666						

Tanto 5.1 como 3.1 no se almacenaron exactamente como nosotros queríamos. Existe una diferencia.

32 Bit 64 Bit

Decimal:
0.45

Exponential:
 $1.799999523162842 \cdot 2^{-2}$

Hexadecimal:
3EE66666

Actual decimal value:
0.44999988079071044921875

Rounding error:
1.1920928955078125E-8

Bits:

3	E	E	6	6	6	6	6
0011111011100110011010011001100110							
+ Exp.: 0x7D	Mantissa: 0x666666						

32 Bit 64 Bit

Decimal:
0.9900000

Exponential:
 $1.9800000190734863 \cdot 2^{-1}$

Hexadecimal:
3F7D70A4

Actual decimal value:
0.990000095367431640625

Rounding error:
-9.5367431640625E-9

Bits:

3	F	7	D	7	0	A	4
0011111101111101011110000010100100							
+ Exp.: 0x7E	Mantissa: 0x7D70A4						

Tampoco se almacenan correctamente ni el 0.45 ni el 0.990000. Guarda el mas cercano a lo que queremos.

Enteros (integers) almacenados bajo norma IEEE 754 (tema optativo)

Como pudimos ver, los tipos de datos enteros (char, short, long, y los unsigned) tienen un límite máximo por sobre el cual se produce el llamado **efecto cíclico de la variable** (búsquelo en el capítulo de **Variables en C**).

Por este motivo podemos pensar en almacenar un valor entero en un float, que se puede pero no solo no es recomendable salvo que sea la única solución sino que además tendremos que tener en cuenta muchos detalles. De nuevo, todo está en la norma IEEE 754.

Veremos el más importante: rangos almacenables sin error.

La norma nos dice que un float puede almacenar un dato entero entre -16777216 a +16777216 esto es $0 \text{ a } \pm 2^{24}$ existiendo todos los valores enteros en dicho rango.

Para valores mayores se deberá considerar la siguiente tabla:

entre	Y hasta	solo pueden ser representados
$\pm 2^{24}$	$\pm 2^{25}$	Los múltiplos de 2
$\pm 2^{25}$	$\pm 2^{26}$	Los múltiplos de 4
$\pm 2^{26}$	$\pm 2^{27}$	Los múltiplos de 8
$\pm 2^{27}$	$\pm 2^{28}$	Los múltiplos de 16
$\pm 2^n$	$\pm 2^{n+1}$	Los múltiplos de 2^{n-23} , con $n > 23$
$\pm 2^{127}$	$\pm 2^{128}$	Los múltiplos de 2^{104}
Mayores a $\pm 2^{128}$		Se redondea a infinito (no pueden ser representados)

Veamos las siguientes capturas:

32 Bit 64 Bit

Decimal:
-16777216

Exponential:
-1.0 · 2²⁴

Hexadecimal:
CB800000

Actual decimal value:
-16777216

Rounding error:
0

Bits:

C	B	8	0	0	0	0	0	0
1	1	0	0	1	0	1	1	0
- Exp.: 0x97	Mantissa: 0x000000							

32 Bit 64 Bit

Decimal:
-16777217

Exponential:
-1.0 · 2²⁴

Hexadecimal:
CB800000

Actual decimal value:
-16777216

Rounding error:
-1

Bits:

C	B	8	0	0	0	0	0	0
1	1	0	0	1	1	0	0	0
-	Exp.: 0x97	Mantissa: 0x000000						

32 Bit 64 Bit

Decimal:
-16777218

Exponential:
-1.0000001192092896 · 2^{24}

Hexadecimal:
CB800001

Actual decimal value:
-16777218

Rounding error:
0

Bits:

C	B	8	0	0	0	0	0	1
1	1	0	0	1	0	1	1	0
-	Exp.: 0x97	Mantissa: 0x000001						

Veamos que pasó

- -16777216 se almacena correctamente
 - -16777217 no se almacena correctamente, almacena -16777216, no es múltiplo de 2
 - -16777218 se almacena correctamente, es múltiplo de 2.

Hay conocer que esto sucede, y si se necesitan detalles ir a la norma IEEE754.

Variables en C

Muy importante !!

Para todas y cada una de las variables que utilicemos en nuestro código, no importando para qué se use ni cuantas veces, se deberá reservar el espacio en memoria por medio de su declaración. Una variable se declara como sigue:

Tipo de dato identificador;

Tipo de dato:

Cualquiera preexistente (float, short int, char..., un tipo struct). Pueden ser tipos de datos creados por el usuario (estructura, unión).

Identificador:

- **Consta de uno o más caracteres.**
- **No puede ser una palabra reservada. (palabras reservadas)**
- **El primer carácter debe ser una letra o el carácter subrayado (_), mientras que, todos los demás pueden ser letras, dígitos o el carácter subrayado (_).**
- **Las letras pueden ser minúsculas o mayúsculas del alfabeto inglés.**
- **Así pues, no está permitido el uso de las letras 'ñ' y 'Ñ', ni letras acentuadas.**

No pueden existir dos identificadores iguales dentro de la misma función, es decir, dos elementos no pueden nombrarse de la misma forma, lo cual no quiere decir que un identificador no pueda aparecer más de una vez en un programa, en distintas funciones.

Ejemplos

Expresión	short A;	char maximo, minimo;	unsigned short suma;	float promedio;
Tipo de dato	short	char	unsigned short	float
Identificador	A	maximo y minimo	suma	promedio
Memoria reservada	2 bytes	1 byte por cada variable	2 bytes	4 bytes
Rango de valores permitidos	-32768 a +32767	-128 a +127	0 a 65535	Precisión de 6 a 7 dígitos
Convenio de almacenamiento	Complemento a 2	Complemento a 2		IEEE 754 / 854

Máximo y mínimo no tienen el acento en la tabla anterior, que en castellano si llevan, porque los identificadores en C usan el alfabeto inglés que no tiene acentos. Tengalo muy en cuenta a la hora de escribir su código. Tampoco existe la ñ en inglés.

No puede ser identificador **suma de valores** ya que no puede haber espacios entre las palabras. Tampoco puede ser **1deEnero** ya que no puede empezar con 1 por ser un número.

Name	Value	Type	Address
A	5	short	00000000002EF806
maximo	65	char	00000000002EF805
minimo	65	char	00000000002EF804
suma	29	unsigned short	00000000002EF802
promedio	2.89893	float	00000000002EF7FC

En memoria se almacenan así las variables de la tabla (muy importante):



Comentario muy importante: Las direcciones hoy en día se escriben en hexadecimal, base 16 (¡aprendalo!)

OJO!!

Hay tipos de datos que su tamaño en memoria depende del hardware y del sistema operativo en cuestión (por ejemplo si es de 32 o 64 bits).

¿Como podemos saber para cada caso el valor de la cantidad de bytes que ocupa un determinado tipo de dato?

sizeof !!!

sizeof es una palabra reservada del lenguaje C que permite obtener el tamaño en byte de una variable o un tipo de dato.

sizeof(tipo de dato)

Ejemplo: **sizeof (float)**, **sizeof (short)**, **sizeof(A)**, que devolverán un valor entero en bytes.

Tenga en cuenta que todo el tiempo el lenguaje va siendo actualizado y revisado. Hoy en ciertos sistemas ya se tienen enteros de 128 bits e incluso mas grandes con librerías especiales.

Tenemos ya en muchos compiladores y para muchos sistemas operativos el **long long int** y el **unsigned long long int** que extiende el rango de aplicación de los enteros a 64 bits (8 bytes).

La correcta elección de los tipos de datos a usar en un código es de vital importancia. Esto surge de un análisis de la información a manejar en el código.

Saber tipos de datos es algo fundamental en programación!!

Veamos el siguiente ejemplo de uso de sizeof para ver que tamaño ocupa cada tipo de dato en memoria. Los resultados en algún caso pueden variar según en que sistema operativo se pruebe, si es de 32 o 64 bits, el compilador usado, etc.

```
#include<locale.h> //para poder usar setlocale
#include<stdio.h>

int main(void)
{
    setlocale(LC_ALL, "Spanish");
    /*
        setlocale es optativa para asegurar poder escribir con printf
        caracteres en castellano, tal como ñ o vocales con acento.
    */
    short PEPE;

    printf("\n\nEl tamaño de un dato char es: %d",sizeof(char));
    printf("\n\nEl tamaño de un dato int es: %d",sizeof(int));
    printf("\n\nEl tamaño de un dato short es: %d",sizeof(short));
    printf("\n\nEl tamaño de un dato long es: %d",sizeof(long));
    printf("\n\nEl tamaño de un dato long long es: %d",sizeof(long long));
    printf("\n\nEl tamaño de un dato float es: %d",sizeof(float));
    printf("\n\nEl tamaño de un dato double es: %d",sizeof(double));
    printf("\n\nEl tamaño de un dato long double es: %d",sizeof(long double));
    printf("\n\nEl tamaño de un puntero es: %d",sizeof(void*)); //depende del hardware.

    printf("\n\nEl tamaño ocupado por la variable PEPE es: %d",sizeof(PEPE));

    return 0;
}
```

Visibilidad y alcance de variables

En C, las variables pueden ser declaradas en cuatro lugares del módulo del programa:

- **Fuera de todas las funciones del programa, son las llamadas variables globales, accesibles desde cualquier parte del programa. (Solo casos especiales, no recomendables en general)**
- **Dentro de una función, son las llamadas variables locales, accesibles tan solo por la función en las que se declaran. (Recomendables SIEMPRE!!)**
- **Como parámetros a la función, accesibles de igual forma que si se declararan dentro de la función.**
- **Dentro de un bloque de código del programa, accesible tan solo dentro del bloque donde se declara. Esta forma de declaración puede interpretarse como una variable local del bloque donde se declara.**

El “ciclismo” de las variables enteras y los tipos de datos. (muy importante!!)

Se dice que **las variables enteras son cíclicas**, esto es, alcanzado su máximo posible, si se les suma 1 van a su mínimo posible. Por eso tengo que conocer el rango de valores que se puede almacenar.

Si tenemos una variable **char**, por ser **signada** y ocupar **8 bits**, su rango de valores almacenables válidos irán desde **-128 a +127**.

Si la variable contiene 127 y le sumo 1, el nuevo valor será -128 **sin darnos ningún error, lo que significará un grave problema**.

Si en cambio la variable contiene -128 y le resto 1 (o le sumo -1) ahora el nuevo valor de la variable será 127. De nuevo no da ningún error.

Esta situación la deberemos tener siempre muy en cuenta a la hora de usar variables enteras.

Esto se nos podrá dar cuando tengamos sumadores y contadores, además de cuando realicemos operaciones matemáticas.

suma = suma + dato;

contador = contador + 1;

resultado= A * B;

Deberemos seleccionar muy bien el tipo de dato a la hora de definir las variables suma, contador y resultado, considerando el máximo y mínimo resultado posible.

Es responsabilidad del programador que esto no suceda en caso de no ser algo que se quiera que suceda. De no agregarse código que verifique este problema los daños pueden ser catastróficos. El C no verifica que una variable “de la vuelta”.

En algunas situaciones esto puede ser muy útil, pero en general es un problema que debemos siempre evitar.

Algunos ejemplos de declaración e inicialización de variables en C

Declaración	Comentario
short a=10;	Hemos declarado una variable entera signada que ocupa 2 bytes en memoria y se le asigna el valor inicial 10.
unsigned char valor;	Hemos declarado una variable entera no signada que ocupa 1 byte en memoria
unsigned short B=20;	Se declara una variable no signada que ocupa 2 bytes en memoria y a la que se le da el valor incial de 20. Por ser variable este valor podrá ser modificado dentro del código.
unsigned short x=-50	Se declara una variable no signada que ocupa 2 bytes en memoria y se le da el valor incial de -50. Este valor es incorrecto para el tipo de dato que hemos definido. Si imprimimos el contenido de x en pantalla veremos que se almacenó 65486. ¿De donde sale este valor?
float X;	Hemos declarado una variable real que ocupa 4 bytes en memoria y que contiene un valor indeterminado si es local a una función. Si no damos un valor inicial, las variables locales contendrán cualquier valor posible, y no necesariamente cero. Esto sucede independientemente del tipo de dato.
char letra;	Hemos declarado una variable entera signada que ocupa 1 byte en memoria y no se le asigna valor inicial por lo que puede tener cualquier valor entre -128 a +127. No necesariamente se usa solo para letras el char.
char contador = 0;	Hemos declarado una variable entera signada que ocupa 1 byte en memoria y se le asigna un valor inicial cero. Como toda variable entera, si no se tiene cuidado, se puede producir el efecto de pasar del máximo valor al minimo valor. Solo sucede con las variables enteras. Si nuestro contador posee el valor +127 y se le suma 1 el próximo valor será -128. Este efecto puede sernos útil en algunos casos, pero en otros muy peligroso. Las variables reales (float, double y long double) se comportan muy diferente y no tienen este "efecto" (ver norma IEEE 754)
float M = 2.56872;	Hemos declarado una variable real que ocupa 4 bytes en memoria y que se le ha asignado un valor inicial de 2.56827. Esta en norma IEEE 754.

Es de preferir declarar una variable por cada línea de código y no mas de una variable por línea. Ejemplo:

Declaraciones	Observacion
short a=10; short b=2; short c; short d=21;	Es lo que se prefiere y se deberia hacer. Hay normas que piden hacer esto siempre. Estas normas se inscriben dentro de lo que se llama "código seguro". El código seguro es fundamental en robotica, industria automotriz, software para aeronaves, energía nuclear, etc.
short a=10, b=2,c,d=21;	No es lo que se deberia hacer. Se pueden cometer errores difíciles de encontrar.

RECUERDE QUE AL DECLARAR VARIABLES LO QUE SE HACE ES RESEVAR ESPACIO CONTINUO Y CONTIGUO EN MEMORIA. TODA VARIABLE EN C, NO IMPORTANDO SU USO DEBE SER DECLARADA PARA QUE SE RESERVE EL ESPACIO EN MEMORIA.

Especificadores de almacenamiento de los tipos de datos (tema optativo)

Una vez explicada la declaración de variables y su alcance, vamos a proceder a explicar como es posible modificar el alcance del almacenamiento de los datos. Ello es posible realizarlo mediante los especificadores de almacenamiento. Existen **cuatro** especificadores de almacenamiento. Estos especificadores de almacenamiento, cuando se usan, deben preceder a la declaración del tipo de dato de la variable. Estos especificadores de almacenamiento son:

Especificador de almacenamiento	Efecto
auto	Variable local (por defecto)
extern	Variable externa
static	Variable estática
register	Variable registro

El **especificador auto** se usa para declarar que una variable local existe solamente mientras estemos dentro de la subrutina o bloque de programa donde se declara, pero, dado que **por defecto toda variable local es auto**, no suele usarse.

El **especificador extern** se usa en el desarrollo de programas compuestos por varios módulos. **Extern** se usa sobre las variables globales del módulo, de forma que si una variable global se declara como **extern**, el compilador no crea un almacenamiento para ella en memoria, sino que, tan solo tiene en cuenta que dicha variable ya ha sido declarada en otro modulo del programa y es del tipo de dato que se indica.

El **especificador static** actúa según el alcance de la variable(en la sección de funciones dentro de este apunte se verá su uso en detalle):

- Para variables locales, el **especificador static** indica que dicha variable local debe almacenarse de forma permanente en memoria, tal y como si fuera una variable global, pero su alcance será el que correspondería a una variable local declarada en la subrutina o bloque. El principal efecto que provoca la declaración como **static** de una variable local es el hecho de que la variable conserva su valor entre llamadas a la función.
- Para variables globales, el **especificador static** indica que dicha variable global es local al módulo del programa donde se declara, y, por tanto, no será conocida por ningún otro módulo del programa.

El **especificador register** se aplica solo a variables locales de tipo char e int. Dicho especificador indica al compilador que, caso de ser posible, mantenga esa variable en un registro de la CPU y no cree por ello una variable en la memoria. Se pueden declarar como **register** cuantas variables se deseen, pues el compilador ignorara dicha declaración caso de no poder ser satisfecha. El uso de variables con especificador de almacenamiento **register** permite colocar en registros de la CPU variables muy frecuentemente usadas, tales como contadores de bucles, etc.

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

/* Variables globales */

short global_A;
long global_B;
float global_C;

int main(void)
{
/* Variables locales a main */
    char local_X;
    short local_Y;
    float local_Z;

    return 0;
}
```

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

/* Variables globales */

short global_A;
long global_B=125000;
float global_C;

int main(void)
{
/* Variables locales a main */
    char local_X=2;
    short local_Y;
    float local_Z=9.25;

    return 0;
}
```

LEER!!!

Nota: En este curso **nunca usaremos variables globales.** **No las necesitaremos.** Todos los códigos se podrán resolver sin utilizarlas.

Hay un caso, que no se verá en este curso, que es la **interrupción de hardware** (digitales II y superiores) que se hace obligatorio el uso de variables globales. En el resto de las situaciones **no se deberán utilizar por ser peligrosas ya que su contenido puede ser modificado desde cualquier función.**

Usted debe conocer que existen, y sus características pero no las debe usar en este curso.

Mezclando tipos de datos (muy importante)

Operaciones entre distintos tipos de datos

En C se pueden realizar operaciones utilizando en las mismas, mezclas de tipos de datos. Se verá a continuación como esto influye en el resultado de la operación.

Dado:

```
short u=10,v=3,w;  
float x=20.0, y=7.0, z;
```

se verá que se obtiene como resultados de las siguientes divisiones:

Variables	Operación	Resultado
short u=10,v=3,w	w=u/v	w=3
short u=10,v=3 float z	z=u/v	z=3.000000
short w float x=20.0, y=7.0	w=x/y	w=2
float x=20.0, y=7.0, z;	z=x/y	z=2.857143
short u=10, w float y	w=u/y	w=1
float y=7.0, z; short u=10	z=u/y	z=1.428571

Se puede observar que en el primer caso la división es totalmente entera, perdiéndose los decimales en la operación de división y en su almacenamiento en la variable **w**.

En el segundo caso sucede lo mismo, pero esta vez se almacena en una variable tipo **float**, convirtiéndose el resultado **entero a float** para su almacenamiento.

En el tercer caso la división es totalmente flotante pero se pierden los decimales en su almacenamiento en la variable **w**.

El cuarto caso la división y el almacenamiento del resultado es del tipo **float** por lo que el resultado que se almacena es previamente la variable **a** de entera a flotante y luego realizándose la operación. Es decir que cuando se combinan varios tipos, la operación se realiza en el formato mas grande. En el caso en que el resultado se almacene en una variable entera se pierden los decimales, dando en el ultimo caso el resultado correcto.

Casteo

(tema muy importante!!)

Con el objetivo de dar solución exacta a la división entre dos enteros se debe realizar lo siguiente:

```
float R;  
int A=5, B=2;
```

R = (float)A / B;

Ahora en R quedará 2.5 y no 2.0 como antes.

La instrucción **(float)A**, hace que A sea “almacenado temporalmente” como **float**. Como la operación se debe hacer con el **mismo tipo de dato**, B también **pasa a ser float temporalmente**. Por lo tanto **la división se hace entre float y el resultado es correcto**. Cuando la operación termina, **A y B vuelven a ser enteros**. Ademas en este caso R deberá ser siempre declarado como float, ya que es donde se guardará el resultado correcto.

Veamos como hacer la misma cuenta de diferentes maneras. Veremos tres formas que dan bien y una que dà mal. Se presenta la conversión de grados Fahrenheit a grados Celsius.

```
#include<stdio.h>

int main(void)
{
    float F;
    float C;

    printf("Ingrese una temperatura en grados farenheit: ");
    scanf("%f",&F);

    C = 5/9*(F-32); //mal
    printf("\nLa temperatura en grados Celsius es C = 5/9*(F-32)= %.2f\n", C);

    C= (float)5/(float)9*(F-32);
    printf("\nLa temperatura en grados Celsius es C= (float)5/9*(F-32)= %.2f\n\n", C);

    C = (F-32)*5/9;
    printf("\nLa temperatura en grados Celsius es C = (F-32)*5/9= %.1f\n", C);

    C= 5/9.0*(F-32);
    printf("\nLa temperatura en grados Celsius es C=5.0/9*(F-32)= %.2f\n\n", C);

    return 0;
}
```

Veamos que si hacemos **5.0** es lo mismo que hacer **(float)5**. Ambas formas hacen que esa constante **5** sea tratada como **float** y no como **entera**. Vea también que el orden como se hace la cuenta puede en muchas situaciones influir en el resultado final.

Ahora también podemos castear de float a long (por ejemplo). Hay que tener muy en cuenta la perdida de la parte decimal y en muchos casos falsos resultados. Esto deberá usted siempre verificarlo.

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

int main(void)
{
    float A=9.434;
    float B=2.415;

    short R;

    R=(long)A%(long)B; /* El casteo es porque el operador % solo aplica entre dos enteros */

    printf("Recuerde que el operador %% es el resto de la division entre dos enteros\n\n");
    /* Los dos %% son para poder imprimir el % en la pantalla */

    printf("Por eso casteamos los float a long\n\n");
    printf("El resto entre %ld y %ld es %hd\n", (long)A,(long)B,R); /* Aqui tambien casteamos!! */

    return 0;
}
```

Nota: Hay muchas aplicaciones sumamente útiles del concepto de casteo en programación no siendo la única la división entre enteros.

```
/*Ejecute este código y vea que resultados se obtiene*/
/*Podria decir que paso?/
```

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

int main(void)
{
    unsigned long entero;

    printf("El tamano del unsigned long es: %Ld\n",sizeof(entero));
    entero=0x89ABCDEF; //Número escrito en hexadecimal con 32 bits

    printf("El entero contiene el valor hexadecimal: %X\n",entero);
    printf("El entero contiene el valor decimal: %Lu\n",entero);
    printf("Si lo casteo como (unsigned short): %X\n",(unsigned short)entero);
    printf("Si lo casteo como (unsigned char): %X\n",(unsigned char)entero);
    return 0;
}
```

Constantes en C

Una constante es un tipo de dato que contiene un valor que nunca cambiará ni se podrá cambiar al momento de la ejecución del código.

En ciencia se tienen muchos ejemplos, algunos muy conocidos, como lo son el número PI (3.141592...), el número e (2,7183.....), la velocidad de la luz en el vacío, la constante de Planck, el cero absoluto en grados centígrados, etc. Tambien el valor del IVA (21 % o 10,5% según el caso). Otras decenas de constantes de física, química, biología, etc.

Tambien puede ser constante un valor limite en nuestro código, como por ejemplo la cantidad máxima de valores a ingresar desde teclado.

Tambien tendremos como constantes a las direcciones donde se encuentra en primer elemento de un vector de cualquier tipo. Como adelanto, el nombre de un vector, es un puntero constante al primer elemento de dicho vector. Un puntero es un tipo de dato que solo puede almacenar direcciones. Ya se verá este tema mas adelante. De poderse cambiar dicha dirección, “perderíamos” el vector dentro de la memoria sin poder en muchos casos volver a encontrarlo. Sería un enorme problema esto.

Por lo que vemos muchas veces es una cuestión de seguridad que debemos tener siempre muy en cuenta. Los códigos tienen que ser seguros en todos sus aspectos siempre y nunca debemos olvidarnos de esto.

Usar constantes en general es algo muy muy útil. Hace mucho mas claros muchos códigos y por lo tanto son más fáciles de modificar y mantener.

Ejemplos de constantes y su uso en C

#define N 500

N es una **constante simbólica** que toma el valor 500 en todos los lugares del código donde escriba N.

#define es una opción de precompilador (antes de compilar se ejecuta).

Vea que no lleva el signo igual (=) ni ningún otro.

Ojo no poner en general punto y coma (;).

Esta forma de definir constantes es la que preferiremos por sobre otras.

```
#define PI 3.1415927
int main(void)
{
    float area;
    float radio;
    area=PI*radio*radio;
    long=2*PI*radio;
    printf("El area es %f",area);

    return 0;
}
```

Si modificamos PI en el #define se modificará el valor en todos los lugares donde lo usemos y esto es muy útil para evitar errores, además que ganamos mucho tiempo en la creación de nuestros códigos.

Además, #define funciona como el “buscar y reemplazar” de muchos editores. Basicamente lo que hace es buscar la cadena PI y reemplazarla por la cadena **3.1415927**, que son textos a este nivel antes de ser compilados. Por lo tanto no se ocupa ningún espacio en memoria para PI usando #define. Es una ventaja. Ademas su valor es como “constante” global. En verdad no es una constante sino una macro.

Podemos crear tantas constantes de esta forma como se necesiten.

Ahora lo mismo usando **const**:

```
int main(void)
{
    const float PI=3.1415927;
    float area;
    float radio;
    area=PI*radio*radio;
    long=2*PI*radio;
    printf("El area es %f",area);

    return 0;
}
```

Ahora **PI** solo es válido dentro de la función main. Con **const** se reserva espacio en memoria además de necesitar un tipo de dato (**float** en este caso).

Las constantes con const podrán ser del mismo tipo que cualquier variable (struct, float, unsigned short, etc.).

En el siguiente fragmento de código veremos algunas alternativas de uso.

```
struct persona {  
    char nombre[50];  
    int edad;  
};  
  
int main()  
{  
    const short codigo=5453;  
    const char Texto1[]{"Hola mundo\n"};  
    const struct persona empleado={"Lucas",45};  
    printf(texto1);  
    Texto1[0]='M';
```

NOOOOOOO!! Texto1 es una constante
no se le puede cambiar su contenido!!!

A ninguna constante, no importa como la definamos (la creemos) se le puede modificar su contenido, es una constante.

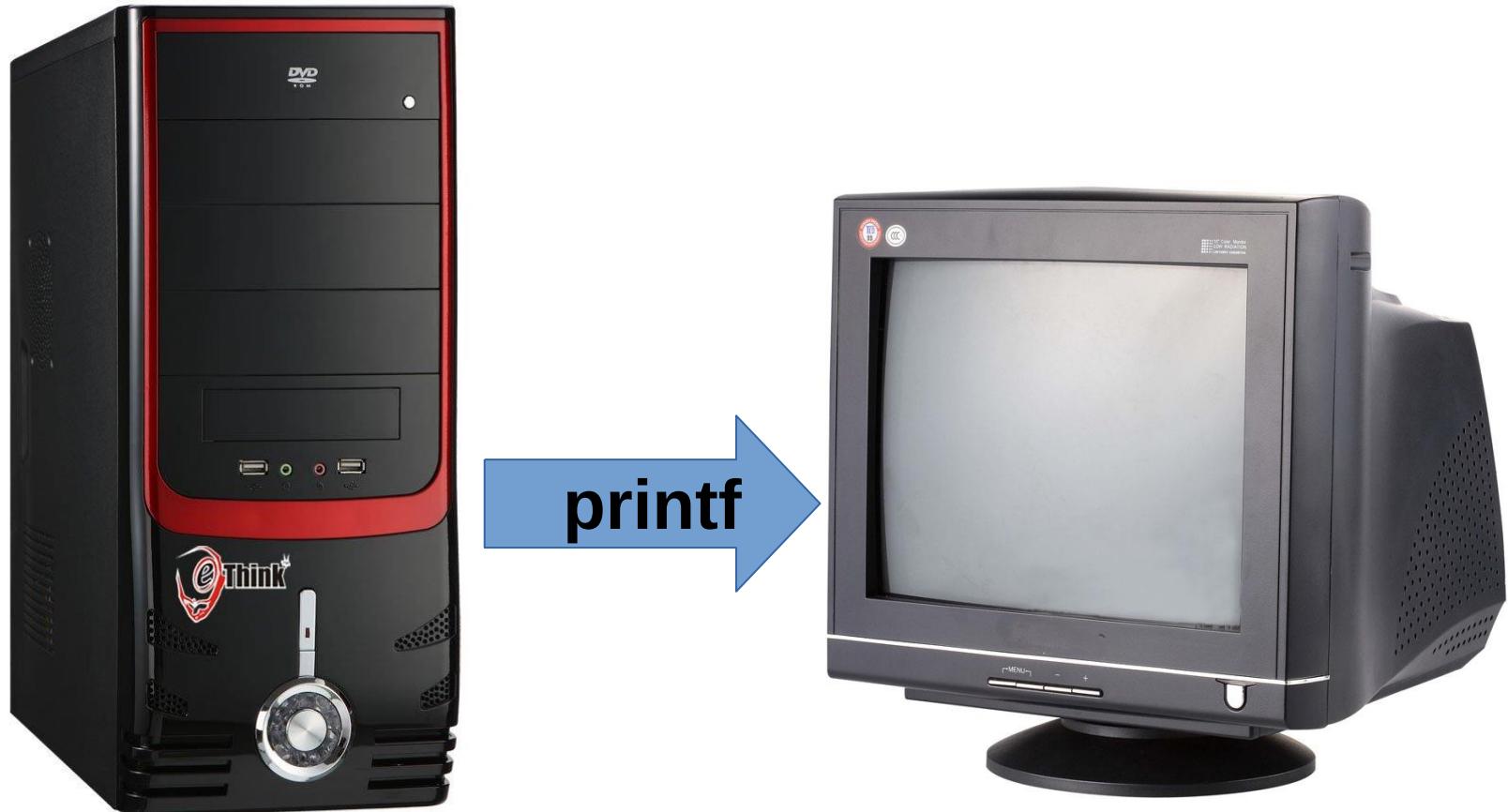
Un caso muy importante en programación es **el nombre de los vectores estáticos en C** que son **PUNTEROS CONSTANTES, ESTO ES, AL MOMENTO DE RESERVARSE EL ESPACIO EN MEMORIA PARA ALMACENAR DICHO VECTOR SE ASIGNA UNA DIRECCION INNICIAL QUE NO PUEDE NI DEBE SER MODIFICA!!!** (Jamas se olvide de este concepto ya que es muy muy importante!!). **PERDER LA DIRECCION INICIAL DE VARIABLES Y CONSTANTES ES PERDER EL DATO Y EN GENERAL PUEDE SER IRECUPERABLE LA INFORMACIÓN.**

La asignación de la dirección inicial la hace “automáticamente” nuestro programa, yo no asigno en general ninguna dirección. No está bien hacer eso y puede ser muy peligroso en programas bajo Windows, Linux y otros sistemas operativos.

Lo tendrá que tener muy en cuenta a la hora de “pasar” vectores a funciones...

Entradas y salidas en C

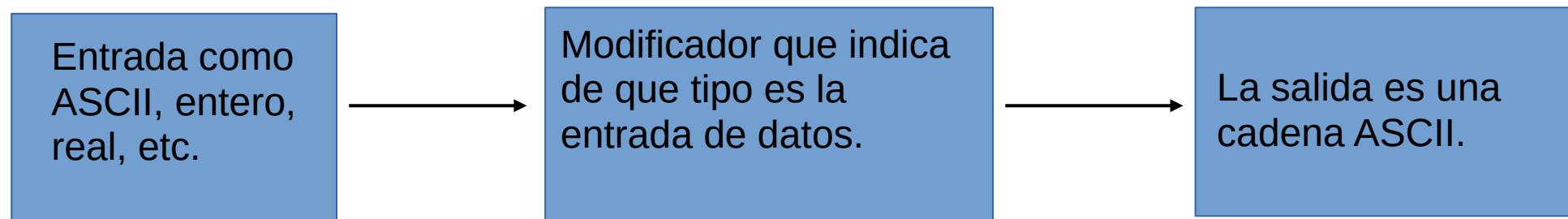
Salida en la PC, es lo que llamamos en C stdout !!



Funciones de entradas y salidas básicas I

Salida: printf

En la PC permite enviar la información al monitor. La entrada es de cualquier tipo estándar y la salida siempre es una cadena ASCII. También es una familia de funciones.



Ejemplo:

```
short A=5;  
printf("%hd", A);
```

No se usa el & porque se quiere imprimir el contenido de la variable, no la dirección.

Entrada en la PC, es lo que llamamos en C `stdin` !!!



Funciones de entradas y salidas básicas II

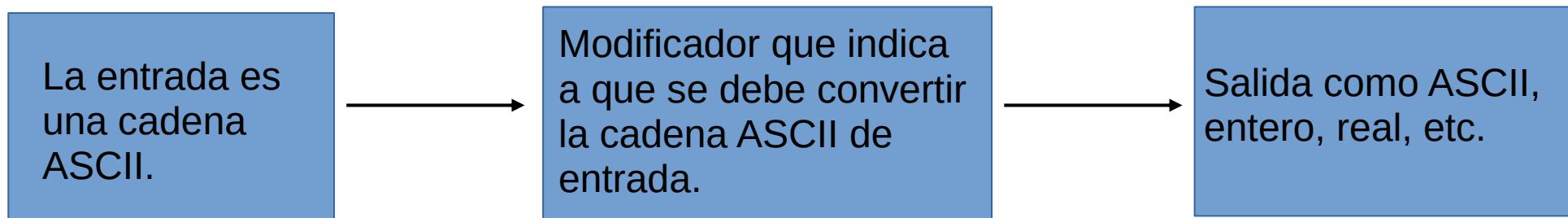
Entrada: scanf

Permite ingresar valores a variables de cualquier tipo estándar. El ingreso se completa con la tecla **ENTER**.

Ejemplo:

```
short A;  
scanf("%hd", &A);
```

El operador & permite obtener la dirección de memoria de la variable A.



Scnf es una familia de funciones que en todos los casos el origen de los datos es una cadena ASCII.
Modificadores de formato: %hd, %f, %c, %s, etc

Para poder utilizar correctamente **printf** y **scanf** siempre tengo que poner:

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

En la PC printf y scanf interactúan con el “monitor” y con el “teclado” respectivamente, pero en verdad tienen que ser vistas como funciones de entrada y salida a nivel de carácter o cadena de caracteres. En el caso de los microcontroladores, **printf** y **scanf** normalmente están relacionadas con el puerto serie. Tambien hay que notar que son funciones de conversión de datos. Vemos que si queremos “imprimir en pantalla” un dato entero, primero **printf** lo convierte en una cadena de caracteres. Existen funciones sumamente útiles de la familia del **printf** y el **scanf** como los son **fprintf**, **sprintf**, **fscanf** y **sscanf** funcionando las que empiezan con **f** con archivos, y las que empiezan con **s** con string.

Modificadores usuales de formato para printf y scanf

	Especificador para printf							
Longitud	d i	u o x X	f F e E g G a A	c	s	p	n	
(ninguno)	int	unsigned int	double	int	char*	void*	int*	
hh	signed char	unsigned char					signed char*	
h	short int	unsigned short int					short int*	
l	long int	unsigned long int			wint_t	wchar_t*	long int*	
ll	long long int	unsigned long long int					long long int*	
j	intmax_t	uintmax_t					intmax_t*	
z	size_t	size_t					size_t*	
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*	
L			long double					

	Especificador para scanf						
Longitud	d i	u o x	f e g a	c s [] [^]	p	n	
(ninguno)	int*	unsigned int*	float*	char*	void**	int*	
hh	signed char*	unsigned char*				signed char*	
h	short int*	unsigned short int*				short int*	
l	long int*	unsigned long int*	double*	wchar_t*		long int*	
ll	long long int*	unsigned long long int*				long long int*	
j	intmax_t*	uintmax_t*				intmax_t*	
z	size_t*	size_t*				size_t*	
t	ptrdiff_t*	ptrdiff_t*				ptrdiff_t*	
L			long double*				

Se usaran de todos estos solo algunos, los que se han remarcado en rojo. No todos son de uso “cotidiano”. Se deberá buscar en libros y en los sitios web que se recomiendan al comienzo de este apunte los detalles de cada uno de ellos. No todos se pueden usar directamente bajo entorno Windows tal como hh o ll. Para ello hay que agregar algunas líneas de código adicionales.

Los modificadores más comunes para printf y scanf(resumen)

en el modificador	se usa con
printf %hd	entero signado de 2bytes: short
scanf %hd	entero signado de 2bytes: short
printf %hu	entero sin signo de 2bytes: unsigned short
scanf %hu	entero sin signo de 2bytes: unsigned short
printf %ld	entero signado de 4bytes: long
scanf %ld	entero signado de 4bytes: long
printf %lu	entero sin signo de 4bytes: unsigned long
scanf %lu	entero sin signo de 4bytes: unsigned long
printf %f	se usa con float y doublé (trata al dato como un doublé)
scanf %f	se usa con float.
scanf %lf	se usa con doublé. ¡Ojo con esto!
printf %c	se usa con char para imprimir una letra en pantalla
scanf %c	se usa con char para ingresar una letra por teclado
printf %s	se usa con cadenas de caracteres: string
scanf %s	se usa con cadenas de caracteres: string

Un simple ejemplo de una suma de dos números: uso de printf y scanf.

```
#include <stdio.h>

int main(void)
{
    /* Haremos C=A + B */
    /* A, B, C son variables donde almaceno información que puede cambiar*/
    /* Hay que reservar memoria para cada una de las variables, SIEMPRE!!*/

    short A;
    short B;
    short C; /* short signada --> %hd */

    printf("Ingrese el valor de A: ");
    scanf("%hd",&A); /* &A nos da la direccion donde vive A */

    printf("Ingrese el valor de B: ");
    scanf("%hd",&B); /* &B nos da la direccion donde vive B */

    C = A + B; /* sumamos el contenido de A y B lo guardamos en C */
    printf("%hd", C); /* %hd indica que el dato C es un short signado */

    return 0;
}
```

Imprimir en pantalla la ñ y vocales acentuadas (tema optativo)

El siguiente código permite escribir en castellano en la salida por medio de la función printf.

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

int main()
{
    setlocale(LC_ALL, "spanish");

    printf("\n¡En el año 1967 se estrenó una muy buena canción!\n");
    printf("\n¿Cuál crees que es?\n"); return 0;
}
```

Vemos que para esto y al comienzo de la función main hemos escrito:

```
setlocale(LC_ALL, "spanish");
```

Tambien tenemos que usar `#include <locale.h>`

Ahora podremos utilizar **vocales acentuadas, la ñ y signos de puntuación del idioma castellano.**

Ahora si queremos ingresar texto en castellano con scanf, gets o fgets se verá el tema mas adelante cuando se vea strings.

Estructuras

(tipo de dato struct)

Es un tipo de dato creado por el usuario con el objetivo de resolver un problema concreto.

Se crea con tipos de datos preexistentes (short, float, long, char, otra estructura definida anteriormente, vectores, etc.)

Su definición es siempre global (fuera de cualquier función, incluida main). Esto es porque así se puede intercambiar “valores” entre funciones.

¿Por que y cuando usar estructuras?

1. Es un tipo de dato que nos permitirá agrupar información interrelacionada entre si.

Como ejemplo podemos dar:

- Los datos de personas: nombre, apellido, dirección, teléfono, mail, DNI, CUIL, etc
- Datos de automóviles: marca, modelo, año, color, patente, propietario, etc
- Datos de productos: marca, nombre, tipo, código, cantidad, precio de venta, etc.

2. Tambien veremos que cuando se quiera devolver más de un valor numérico desde una función será de gran utilidad. Esto se verá en este curso (**¡muy importante!**).

3. Es una muy buena forma de almacenar datos en archivos binarios.

4. Es mucho mas practico usar estructuras que vectores sueltos interrelacionados. El ordenamiento de información, por ejemplo, se simplifica muchísimo.

5. Son muchas las ventajas de usar estructuras en muchos códigos. Ademas permiten generar tipos de datos mas complejos e incluso generar listas dinámicas en forma sencilla.

6. Son la base de lo que se denomina “programacion orientada a objetos”.

7. Podremos crear tanto constantes como variables.

Por todo esto y mucho mas que no hemos dicho, es fundamental conocer bien este tema.

Ejemplo de uso:

```
#include<stdio.h>

Declaración del nuevo tipo de dato. No se reserva ni ocupa memoria. Es una descripción para que el compilador se entere que forma tiene este nuevo tipo de dato.

struct datos{
    char edad;
    float sueldo;
    long DNI;
};

int main(void)
{
    -----
    -----
}
```

struct datos es el **nombre del nuevo tipo de dato**, y se une a los ya existentes (short, long, float, etc).

NO OCUPA ESPACIO EN MEMORIA HASTA QUE DECLARE VARIABLES (o constantes) DE ESTE TIPO. CON ESTO SE LE INFORMA AL SISTEMA AL COMPILADOR QUE FORMA TIENE NUESTRO NUEVO TIPO DATO.

char edad es el **tipo y nombre** de uno de los tres (3) **campos o miembros de esta estructura**. Los otros son **float sueldo** y **long DNI**.

Puedo crear tantos campos como necesite. Puedo crear tantas estructuras como necesite.

Creación y uso de variables de tipo estructura.



La nueva variable se llama **informacion**, contiene 3 campos y en principio ocupa la suma de lo que ocupa cada uno de los tipos de datos de sus miembros (char 1 byte, float 4 bytes, long 4 bytes) esto es 9 bytes. Pero no siempre esto es así, por eso deberemos usar **sizeof** para determinar exactamente cuanto ocupa (**sizeof(informacion)**).

Accediendo a los miembros o campos de la estructura.

```
informacion.edad=45;  
informacion.sueldo=12459.25;
```

... esto es: **nombre de la variable, punto, campo...**

Otro ejemplo de uso

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

```
struct datos{  
    char edad;  
    float sueldo;  
    long DNI;  
};
```

```
int main(void)  
{  
    struct datos informacion;  
    -----  
    informacion.edad=45;  
    informacion.sueldo=12459.25;  
    informacion.DNI=19292393;  
  
    return 0;  
}
```

Name	Value	Type	Address
informacion	{...}	struct datos	00000000002AF7D8
edad	45	char	00000000002AF7D8
sueldo	12459.3	float	00000000002AF7DC
DNI	19292393	int	00000000002AF7E0

También puedo usar printf y scanf con estructuras !!

```
printf("%hd", informacion.edad);
```

```
printf("%f",informacion.sueldo);
```

Hemos impreso el contenido, y como se puede ver no puedo imprimir nada solo nombrando el nombre de la variable que en nuestro caso es informacion (sin acento porque es el identificador).

Ahora vamos a cargar los campos desde teclado, para esto usamos scanf

```
scanf("%hd",&informacion.edad);
```

```
scanf("%f",&informacion.sueldo);
```

No se olvide en el scanf en & para datos numéricos!!!

Estructuras anidadas

```
struct fecha{
    unsigned short dia;
    unsigned short mes;
    unsigned short anio;
};

struct ficha{
    unsigned short edad;
    float sueldo;
    struct fecha nacimiento;
};

}
```

```

#include <stdio.h>

struct fecha{
    unsigned short dia;
    unsigned short mes;
    unsigned short anio;
};

struct ficha{
    unsigned short edad;
    float sueldo;
    struct fecha nacimiento;
};

int main(void)
{
    struct ficha p;

    p.edad=25;
    p.sueldo=18745.25;
    p.nacimiento.dia=18;
    p.nacimiento.mes=9;
    p.nacimiento.anio=1947;

    printf("Edad: %hu\n",p.edad);
    printf("Sueldo: $%.2f\n",p.sueldo);
    printf("Fecha de nacimiento %hu/%hu/%hu\n\n",p.nacimiento.dia,p.nacimiento.mes,p.nacimiento.anio);

    return 0;
}

```

Name	Value	Type	Address
p	{...}	struct ficha	000000000019FCB0
edad	25	unsigned short	000000000019FCB0
sueldo	18745.3	float	000000000019FCB4
nacimiento	{...}	struct fecha	000000000019FCB8
dia	18	unsigned short	000000000019FCB8
mes	9	unsigned short	000000000019FCBA
anio	1947	unsigned short	000000000019FCBC
p.nacimiento.dia	18	unsigned short	000000000019FCB8
p.nacimiento.mes	9	unsigned short	000000000019FCBA
p.nacimiento.anio	1947	unsigned short	000000000019FCBC
p.edad	25	unsigned short	000000000019FCB0
p.sueldo	18745.3	float	000000000019FCB4

Para llegar al miembro dia, debo hacer: p.nacimiento.dia, esto es: “seguir la flecha”, en este caso la azul...



Hacer graficos como este, con tipos de datos complejos, ayuda mucho a no equivocarse. Tengalo siempre en cuenta. Además sirve para documentar su código, lo que es sumamente importante.

Estructuras (struct) es un tema fundamental en la moderna programación. No debe desconocer este tema bajo ningún concepto.

Estructuras de control

Las estructuras de control nos permitirán definir en que orden se ejecutan nuestras instrucciones e incluso definir si cierto grupo de instrucciones se ejecutará o no. Además podemos definir si un cierto conjunto de instrucciones se volverá a repetir hasta que una condición deje de ser verdadera o también decir cuantas veces un conjunto de instrucciones se va a repetir. Las estructuras de control son fundamentales en cualquier lenguaje de programación. Sin ellas los algoritmos no existirían...

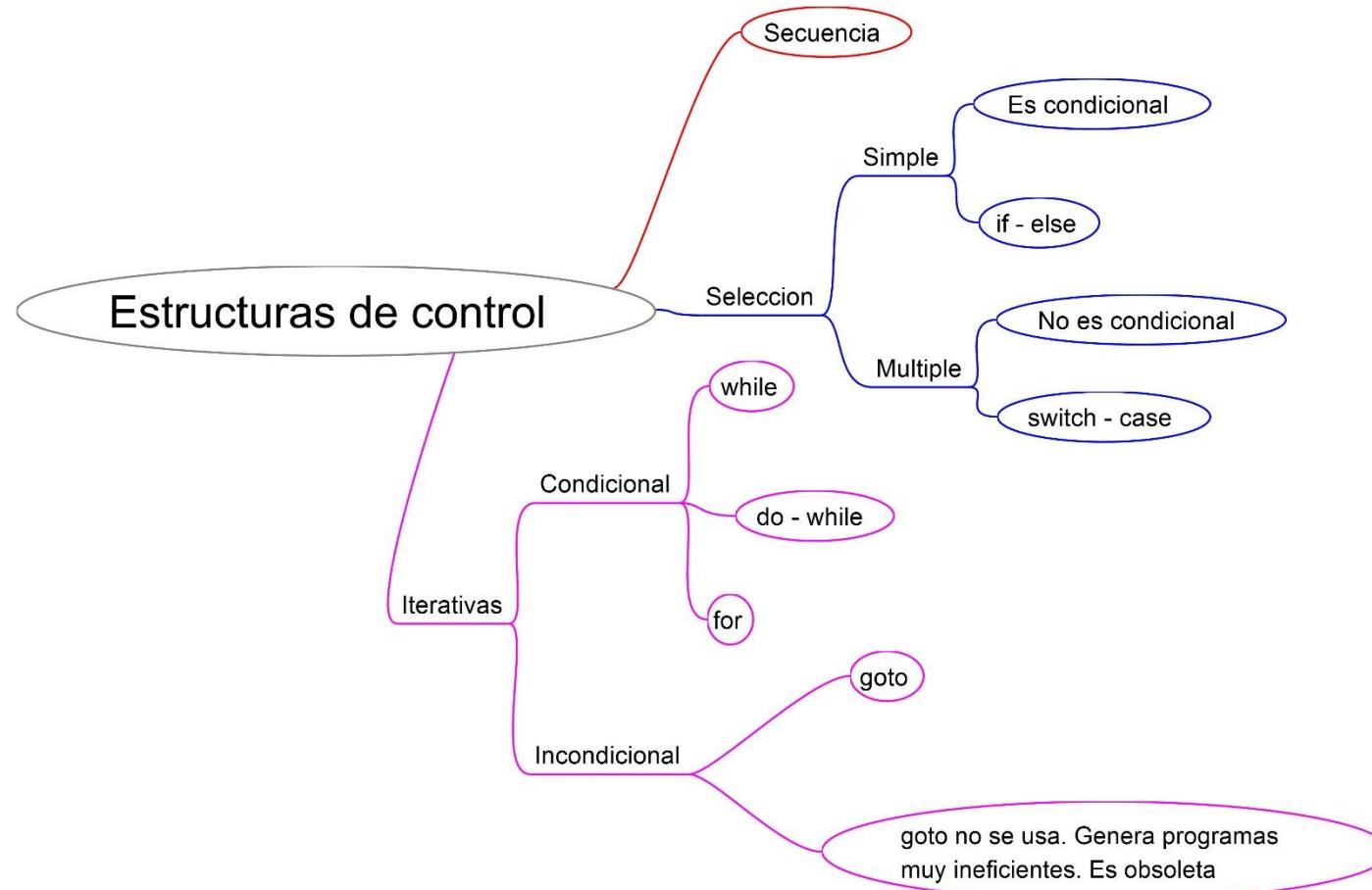
En C existen básicamente:

- Secuencia
- Selección
- Iteración

Goto es algo que también existe pero es obsoleto e innecesario su uso en la moderna programación. NOSOTROS NUNCA USAREMOS GOTO, NUNCA LO NECESITAREMOS... NADIE USA GOTO...

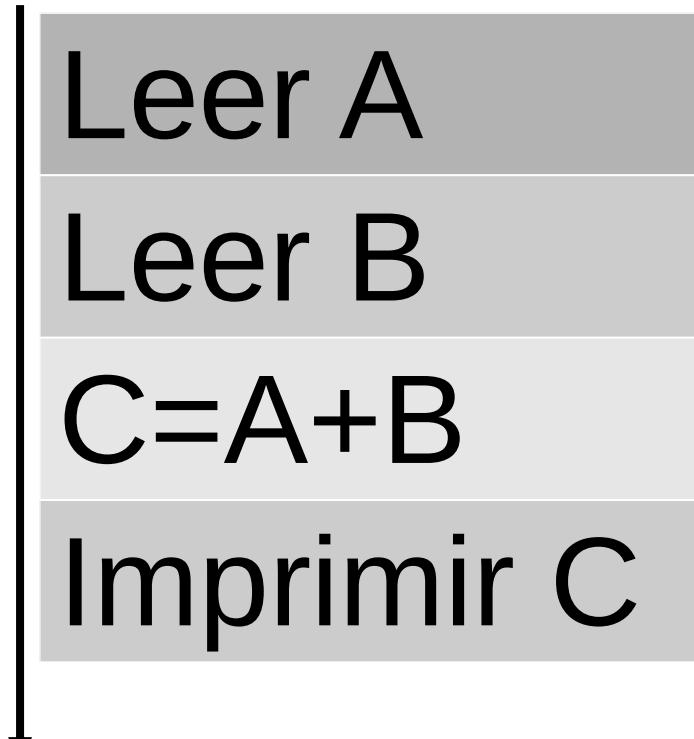
Estructuras de control en C

Cualquier problema informático se puede resolver con una o varias de las siguientes estructuras de control de flujo:



Secuencia (de instrucciones)

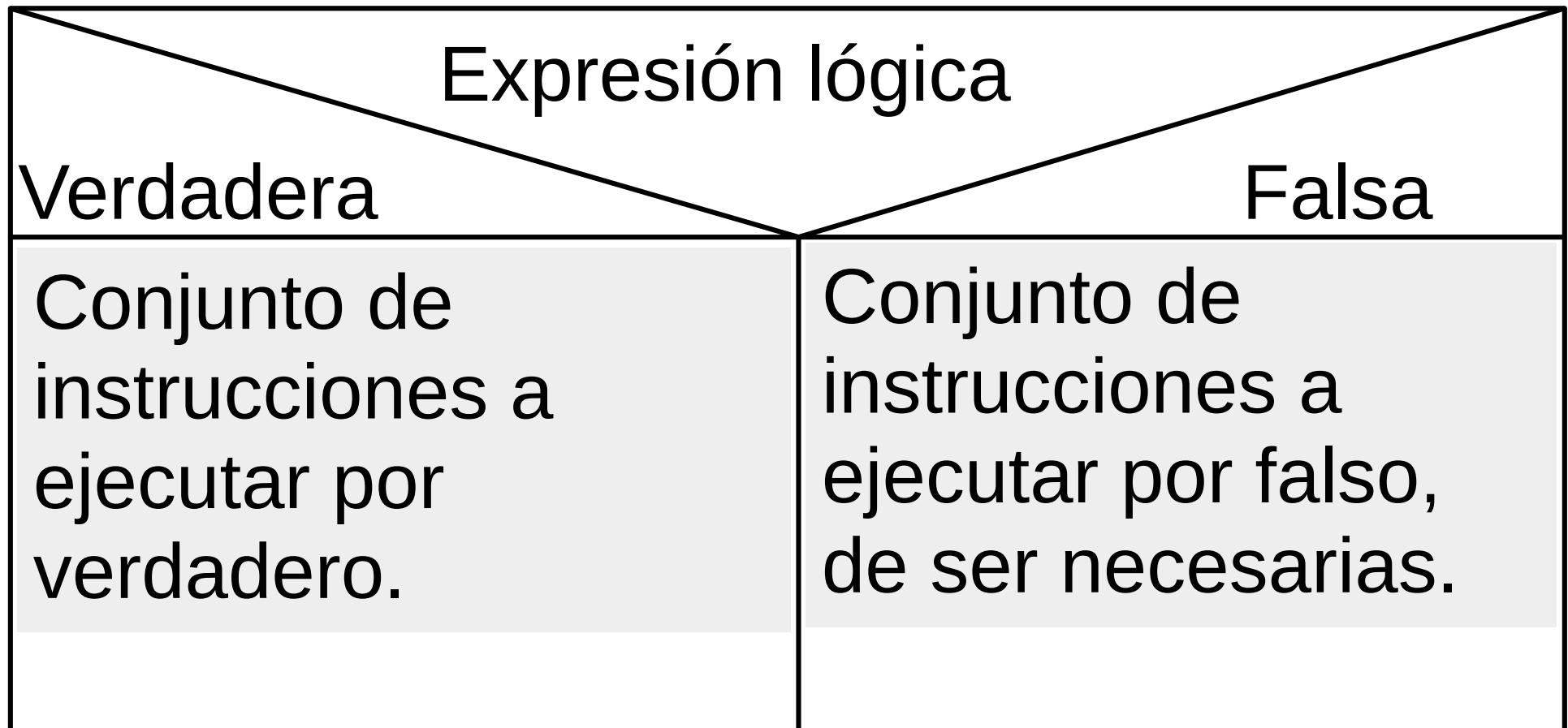
Veamos el siguiente ejemplo de la suma de dos números:



No se vuelve a repetir ninguna instrucción nuevamente y todas se ejecutan sin necesidad de cumplir con alguna condición.

Selección (condicional)

Se ejecuta una instrucción o conjunto de instrucciones si se cumple una determinada expresión lógica.



Expresiones relacionales

Expresión	Comentario / explicación	Ejemplo
A > B	¿A contiene un valor mayor que B?	A= 5 y B=4 dará verdadero
A < B	¿A contiene un valor menor que B?	A= 5 y B=4 dará falso
A>=B	¿A contiene un valor mayor o igual que B?	A= 5 y B=4 dará verdadero
A<=B	¿A contiene un valor menor o igual que B?	A= 5 y B=4 dará falso
A==B	¿A contiene el mismo valor que B?	A= 5 y B=4 dará falso
A!=B	¿A contiene un valor distinto que B?	A= 5 y B=4 dará verdadero
A	Si A contiene un valor distinto de cero la respuesta lógica es verdadero.. Exactamente cero es falso.	A= 5 dará verdadero
!A	Si A contiene un valor exactamente igual a cero es verdadera la respuesta lógica. Cualquier otro valor distinto de cero, dará como resultado falso.	A= 5 dará falso
A=B	Ojo!! Se asigna primero el contenido de B a A y luego se evalúa A.	A= 5 y B=4 primero A tomará el valor 4 y luego se evalua A quedando el resultado lógico como verdadero.

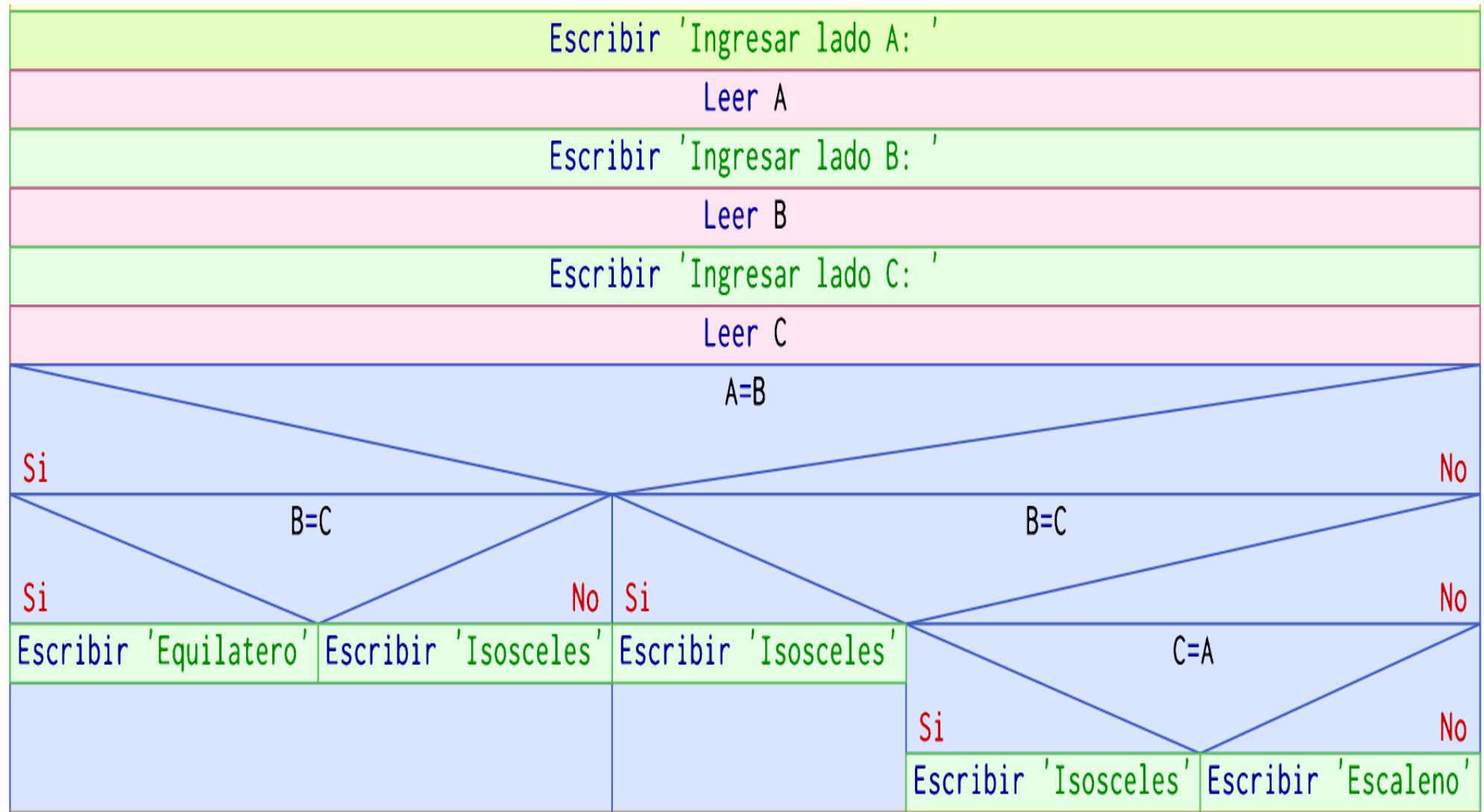
```
if(expresión)
{
    Instrucción o conjunto de instrucciones a
    realizar si la expresión es verdadera.
}
else
{
    Instrucción o conjunto de instrucciones a
    realizar si la expresión es falsa.
}
```

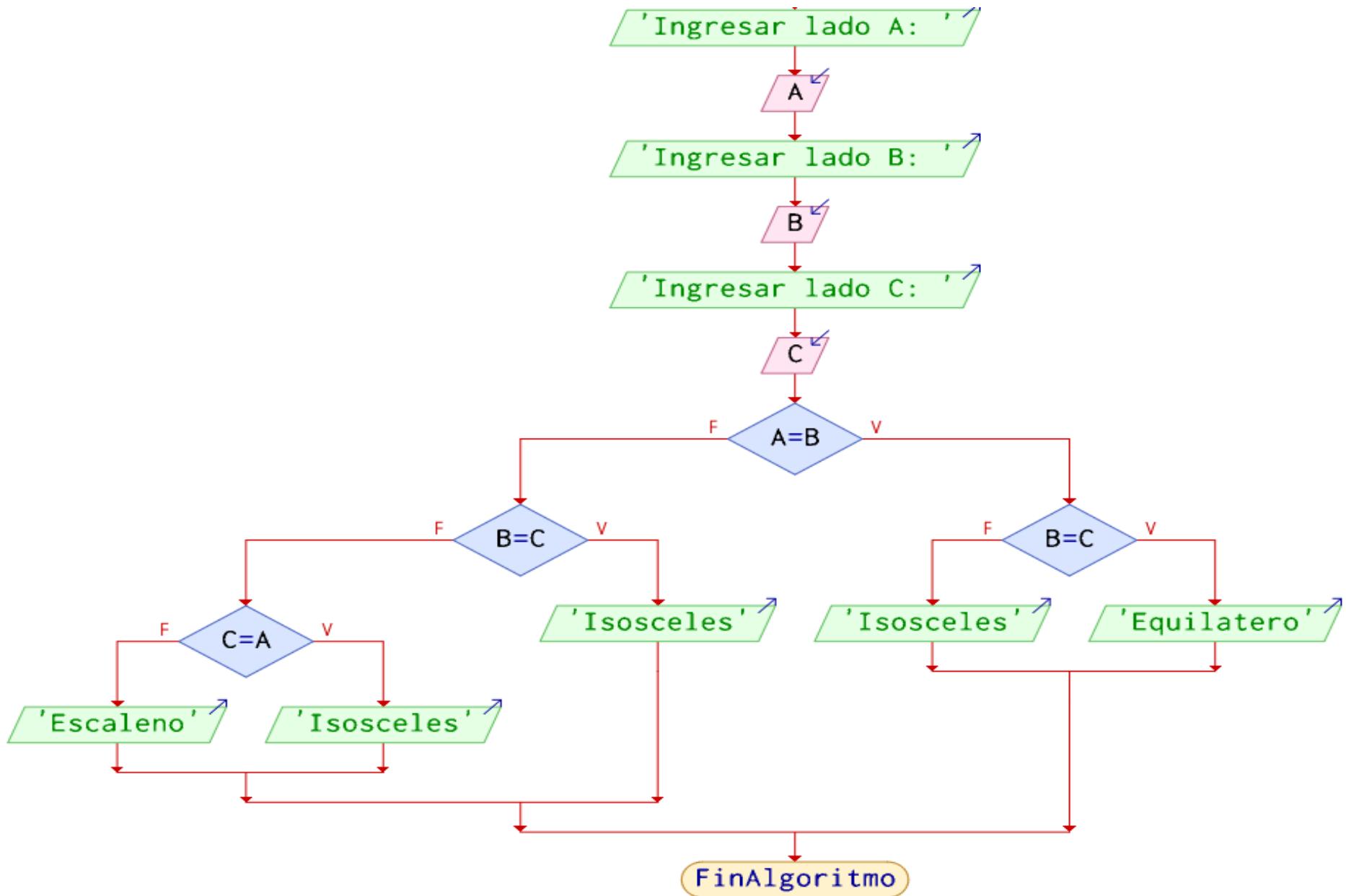
Cuatro ejemplos simples de uso de la estructura de control if.

<pre>if(A > B) printf("A es mayor a B"); else printf("B es mayor o igual a A");</pre>	<pre>if(B >=A) printf("B contiene un valor mayor o igual a A"); else printf("B contiene un valor distinto al de A");</pre>
<pre>if(A) printf("A contiene un valor distinto de cero"); else printf("A contiene un valor igual a cero");</pre>	<pre>if(!A) printf("A contiene un valor igual a cero"); else printf("A contiene un valor distinto a cero");</pre>

La sentencia if siempre se escribe en minúscula al igual que la sentencia else. Else puede faltar si no es necesario, es optativo. El verdadero JAMAS puede faltar.

Dados tres números reales, longitud de los lados de un triángulo, indicar si es equilátero, isósceles o escaleno.





```
#include <stdio.h>

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

    printf("Ingrese el valor de a: ");
    scanf("%f", &a);

    printf("Ingrese el valor de b: ");
    scanf("%f", &b);

    printf("Ingrese el valor de c: ");
    scanf("%f", &c);

    if (a==b)
        if(b==c)
            printf("El triangulo es equilátero");
        else
            printf("El triangulo es isósceles");
    else
        if(b==c)
            printf("El triangulo es isósceles");
        else
            if(c==a)
                printf("El triangulo es isosceles");
            else
                printf ("El triangulo es escaleno");
}

return 0;
```

AND (&&) y OR (||): relacionantes para expresiones lógicas.

Si queremos unir en una única expresión una o más expresiones individuales tenemos dos operadores que permiten realizar expresiones sumamente complejas (cuidado que pueden ser difíciles de evaluar):

operador	operación	comentario
&	And (y lógico)	Todas las expresiones deben ser verdaderas para que el resultado sea verdadero. Con que alguna de la expresiones sea falsa ya toda la expresión será falsa.
	Or (o lógico)	Todas las expresiones deben ser falsas para que el resultado sea falso. Con que alguna de la expresiones sea verdadera ya toda la expresión será verdadera.

Tablas lógicas de las funciones AND y OR para dos variables

AND			OR		
Si una expresión es:	Y la otra expresión es:	El resultado final será:	Si una expresión es:	Y la otra expresión es:	El resultado final será:
FALSO	FALSO	FALSO	FALSO	FALSO	FALSO
FALSO	VERDADERO	FALSO	FALSO	VERDADERO	VERDADERO
VERDADERO	FALSO	FALSO	VERDADERO	FALSO	VERDADERO
VERDADERO	VERDADERO	VERDADERO	VERDADERO	VERDADERO	VERDADERO

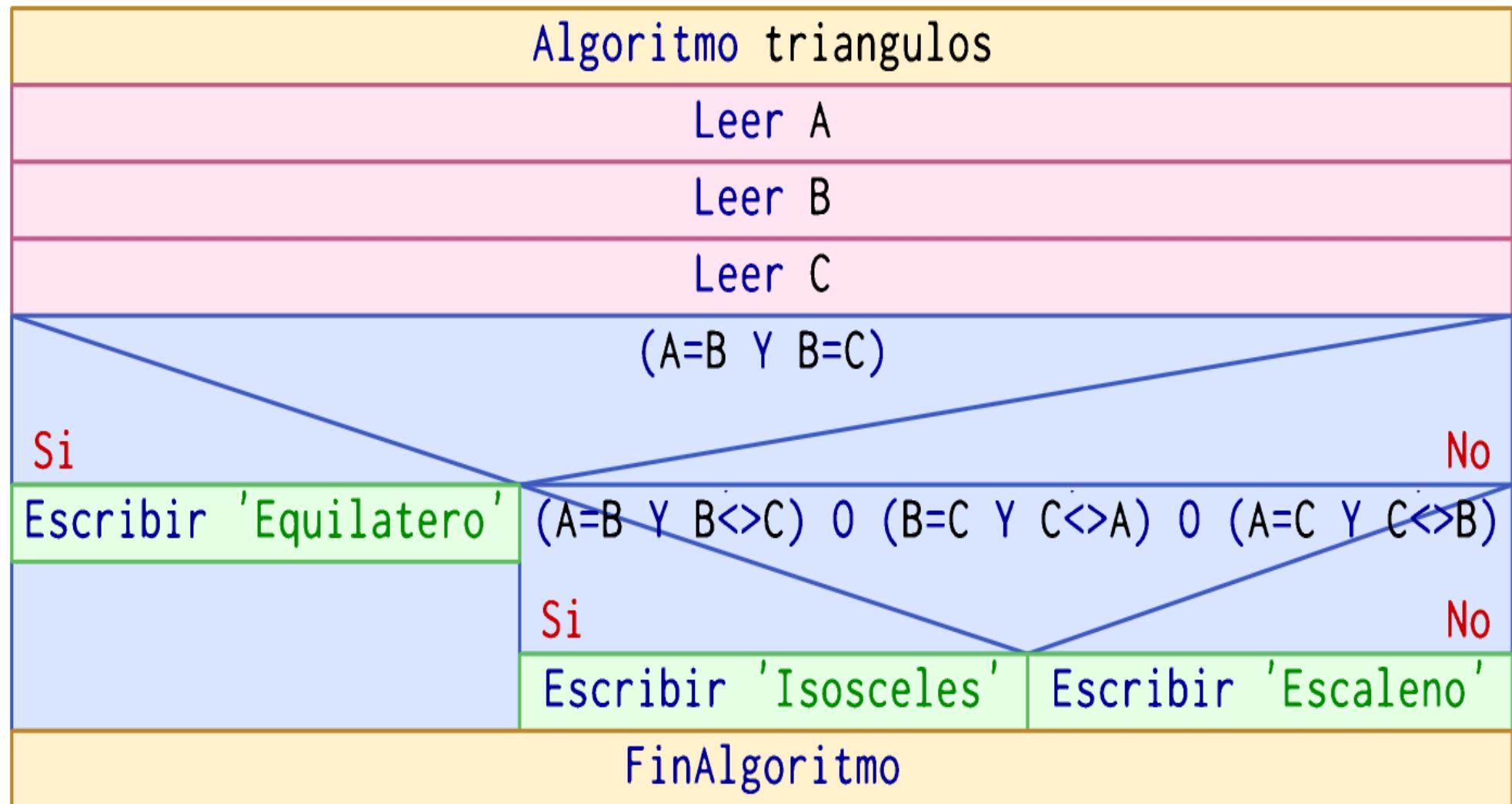
Un ejemplo donde vemos una expresión que contiene AND y OR:

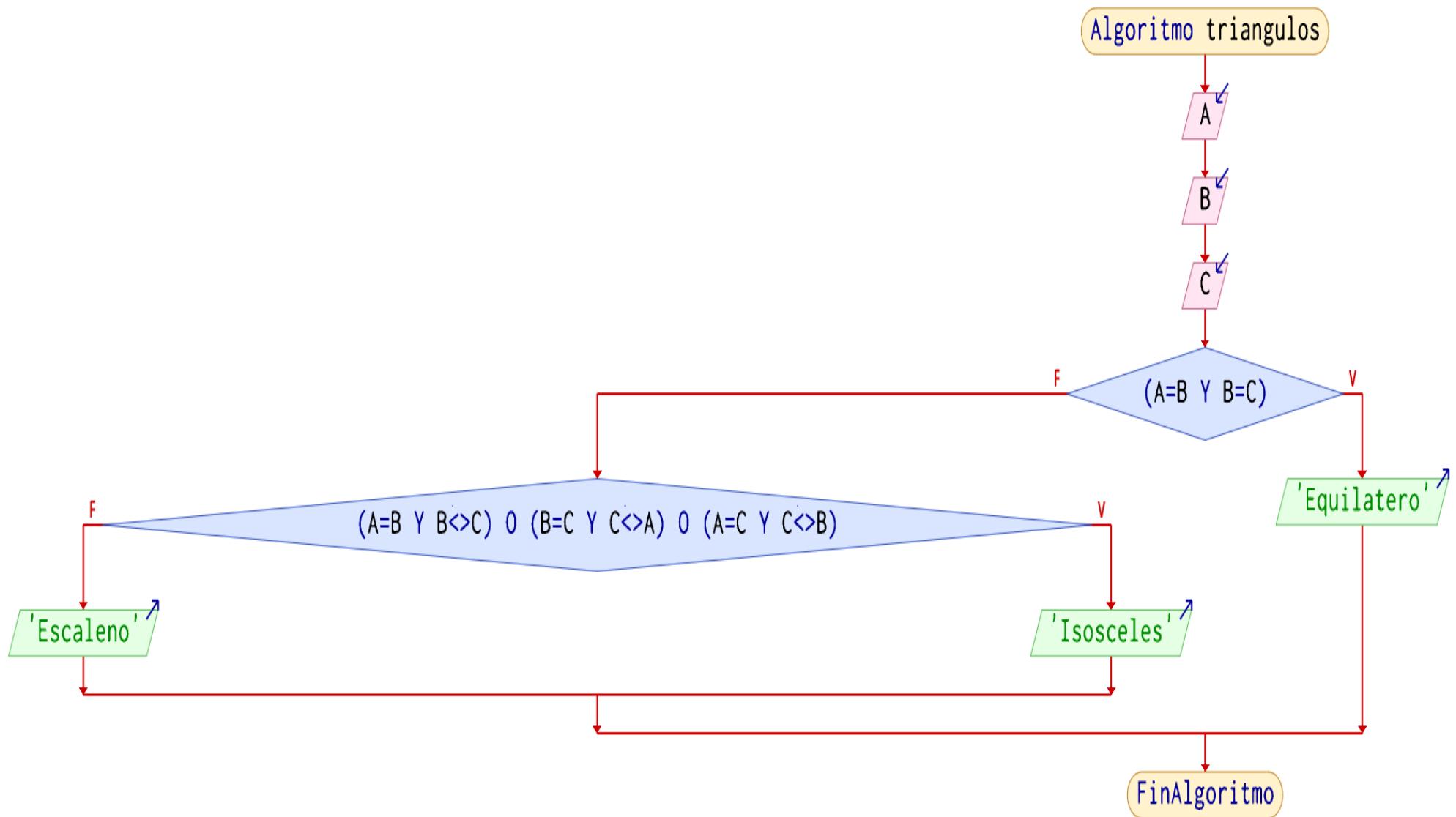
```
if((A > B) && (C < 2) || (D != 0))
{
    Instrucción o conjunto de instrucciones a realizar si la expresión es verdadera.
}
else
{
    Instrucción o conjunto de instrucciones a realizar si la expresión es falsa.
}
```

Ojo al evaluar que operador se debe utilizar para resolver un determinado problema !! Detengase a evaluar todas las posibles combinaciones de valores de las variables que se encuentran en las expresiones.

Muchas veces estos operadores harán mas complicada la lectura de nuestro código haciendo que se nos puedan pasar algunos errores o se hagan las expresiones difíciles de evaluar para todas las posibles combinaciones de sus variables como ya hemos mencionado.

Resolveremos nuevamente el problema de los triángulos





```
#include <stdio.h>

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

printf("Ingrese el valor de a: ");
scanf("%f", &a);

printf("Ingrese el valor de b: ");
scanf("%f", &b);

printf("Ingrese el valor de c: ");
scanf("%f", &c);

if (a==b && b==c)
    printf("El triangulo es equilatero");
else
    if ((a==b && b!=c) || (b==c && c!=a) || (a==c && c!=b))
        printf("El triangulo es isosceles");
    else
        printf ("El triangulo es escaleno");
return 0;
}
```

Algunos errores típicos utilizando **if – else**

Vamos a describir algunos errores que se dan en el uso de if

1. Si queremos saber si el contenido de tres variables es el mismo:

```
if( A==B==C) ← MAL!!!
{
}
```

no es valida la expresión, ya que siempre se compara de a dos!!

En cambio deberíamos hacer alguna de las dos siguientes opciones:

```
a. if(A==B && B==C) ← BIEN!!
{
}
```

```
b. if(A==B)
    if(B==C)
    {
}

}
```

BIEN!!

Nota: No siempre el compilador te dice que la expresión no es válida. Hay que recordar que siempre se compara de a dos y no de a tres o mas como en la expresión (**A==B==C**).

2. Si queremos saber si tres variables cumplen con alguna condición deberemos tener en cuenta lo dicho, esto es, **no podemos hacer if (A && B && C > 10)**. Se pretende saber si A, B y C contienen un valor mayor a 10, pero esta mal asi hecho.

En su lugar deberemos hacer: **if(A>10 && B>10 && C>10)**

Algunos ejemplos prácticos de uso de la estructura condicional “IF”

- 1) Ingresar un entero por teclado y saber si es par o impar. Se ven cuatro formas de obtener el mismo resultado, ninguna mejor que otra.

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

int main()
{
    short numero;
    short resto;

    printf("Ingrese numero: ");
    scanf("%hd",&numero);

    /*
        Forma 1
    */
    if(numero%2==0)
        printf("El numero es par!\n");
    else
        printf("El numero es impar!\n");

    /*
        Forma 2
    */
    if(numero%2) //Ojo, si el resultado es cero es FALSO
        printf("El numero es impar!\n");
    else
        printf("El numero es par!\n");
}
```

```
/*
    Forma 3
*/
if(resto=numero%2) //Ojo, si el resultado es cero es FALSO
    printf("El numero es impar!\n");
else
    printf("El numero es par!\n");

/*
    Forma 4
*/
if(!(numero%2)) //El ! invierte el resultado lógico, esto es si era falso dará verdadero
    printf("El numero es par!\n");
else
    printf("El numero es impar!\n");

return 0;
}
```

- 2) ¡Evitando problemas!: Sabemos que no podemos dividir por cero y tampoco calcular raíces cuadradas de valores negativos, en el conjunto de números reales.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h> //para la funcion raíz cuadrada sqrt

int main()
{
    short A=7;
    short B=5;
    float numero=0.0;
    float resultado;

    if(B != 0)
    {
        resultado=(float)A/B;//Castear para que dé correcto el resultado
        printf("El resultado es: %f\n",resultado);
    }
    else
        printf("No se puede dividir por cero!!\n");

    if(numero>=0)
    {
        resultado=sqrt(numero); //sqrt calcula la raíz cuadrada.
        printf("La raiz es: %f\n",resultado);
    }
    else
        printf("No se puede calcular raiz\n");

    return 0;
}
```

- 3) Este ejemplo busca conocer si un determinado numero ingresado por teclado está entre dos límites (límites incluidos) o no. Se muestra dos formas distintas de obtener el mismo resultado, ninguna mejor que la otra.

```
#include <stdio.h>

int main(void)
{
    short linferior=5;
    short lsuperior=20;
    short numero;

    printf("Ingrese numero: ");
    scanf("%hd",&numero);

    if(linferior<=numero)
    {
        if(lsuperior>=numero)
        {
            printf("Esta en el intervalo!!\n");
        }
        else
        {
            printf("Esta fuera de rango!!\n");
        }
    }
    else
    {
        printf("Esta fuera de rango!!\n");
    }

    if((linferior<=numero)&&(lsuperior>=numero))
        printf("Esta en el intervalo!!\n");
    else
        printf("Esta fuera de rango!!\n");
    return 0;
}
```

Iterativas(bucles o lazos)

En C tenemos 3 iterativas que nos permitirán **repetir el mismo conjunto de instrucciones**, siempre y cuando una expresión sea o permanezca verdadera.

Tenemos:

- **while**
- **do – while**
- **for**

Observeremos el siguiente ejemplo que plantearemos en pseudocódigo:

Sin estructuras iterativas	Con estructuras iterativas
<pre>Algoritmo suma_5_enteros_ingresados_por_teclado Definir suma Como Entero Definir dato Como Entero suma ← 0 Leer dato suma ← suma+dato Escribir suma FinAlgoritmo</pre>	<pre>Algoritmo suma_5_enteros_ingresados_por_teclado Definir suma Como Entero Definir dato Como Entero Definir contador Como Entero suma ← 0 contador ← 0 Mientras contador<5 Leer dato suma ← suma+dato contador ← contador+1 Escribir suma FinAlgoritmo</pre>

En este ejemplo hemos sumado cinco (5) valores enteros ingresados por teclado. NO menos de cinco, no más de cinco.

Si sumamos una cantidad distinta de cinco, cuando no usamos estructuras iterativas deberíamos agregar o quitar estas instrucciones:

```
Leer dato
suma ← suma+dato
```

Imagine ahora sumar 50000 números, tendríamos que repetir sin iterativas 50000 veces ese código. Con iterativas cambiamos la condición contador < 5 por contador < 50000 y nada más. Sin iterativas solo podríamos hacer la suma si conocemos EXACTAMENTE cuantos valores se van a sumar, lo que no siempre sucede. Las estructuras iterativas nos darán gran flexibilidad a la hora de escribir nuestros códigos.

while (mientras que...): Permite repetir una instrucción o conjunto de instrucciones, siempre y cuando la expresión sea y/o permanezca verdadera.

while(expresión)

Instrucción o conjunto de instrucciones a ser repetidas (cuerpo del lazo while)

for: Permite repetir una instrucción o conjunto de instrucciones, siempre y cuando la expresión sea y/o permanezca verdadera.

```
for(campo1; campo2; campo3)
```

Instrucción o conjunto de instrucciones a ser repetidas
(cuerpo del lazo for)

Explicación lazo for

Campo1: Se utiliza habitualmente para inicializar las variables que se utilizan en el lazo. Se ejecuta una sola vez, esto es, la primera vez que se ingresa a la estructura for y después es ignorado.

Campo2: Se usa exclusivamente para la expresión lógica del lazo.

Campo3: Se utiliza habitualmente para el incremento / decremento de las variables del lazo.

Nota: Todos los campos son optativos pudiendo faltar uno, dos o todos.

“Operadores pre y post”

Existen 4 operadores matemáticos de incremento y decremento que se dividen en dos grupos.

Queremos hacer: variable = variable + 1

++variable : Primero el contenido se incrementa en 1 y luego se usa con el nuevo valor. (pre-incremento)

variable++ : Primero se usa con el actual contenido y luego se incrementa en 1. (post-incremento)

Queremos hacer: variable = variable -1

--variable : Primero el contenido se decrementa en 1 y luego se usa con el nuevo valor. (pre-decremento)

variable-- : Primero se usa con el actual contenido y luego se decrementa en 1. (post-decremento)

Tenga muy en cuenta los detalles de cada operador y no los use en cualquier parte del código.
Tenga en cuenta que en muchas expresiones puede ser difícil de evaluar el resultado de la misma. Uselos con cuidado.

Sumar los números enteros del 1 al 20 y hacerlo con un lazo while.

```
dato=1; 1
while(dato < 21) 2
{
    suma = suma + dato; 3
    dato++; 4
}
```

1 Se ejecuta una sola vez.
2 Se evalúa la expresión.
Si **2** es verdadera va a **3**. Sino termina.
Se ejecuta **4**. Luego se va a **2**.
Comienza nuevamente el ciclo.

Sumar los números enteros del 1 al 20 y hacerlo con un lazo for.

```
1           2           4
for(dato=1; dato < 21; dato++)
{
    suma = suma+dato;3
}
```

1 Se ejecuta una sola vez.
2 Se evalúa la expresión.
Si **2** es verdadera va a **3**. Sino termina.
Se ejecuta **4**. Luego se va a **2**.
Comienza nuevamente el ciclo.

Sumar los números enteros del 1 al 20 y hacerlo con un lazo while.

```
#include <stdio.h>

int main(void)
{
    unsigned short dato;
    unsigned short suma=0;

    dato=1;
    while(dato<21)
    {
        suma=suma+dato;
        dato++;
    }

    printf("\nLa suma es: %hd\n\n",suma);

    return 0;
}
```

Sumar los números enteros del 1 al 20 y hacerlo con un lazo for.

```
#include <stdio.h>

int main(void)
{
    unsigned short dato;
    unsigned short suma=0;

    for(dato=1;dato<21;dato++)
        suma=suma+dato;

    printf("\nLa suma es: %hd\n\n",suma);

    return 0;
}
```

Sumar los números enteros del 1 al 20 y hacerlo con un lazo while.

Sumar los números enteros del 1 al 20 y hacerlo con un lazo for.

Algoritmo Sumatoria

dato \leftarrow 1

Mientras dato < 21

 suma \leftarrow suma + dato

 dato \leftarrow dato + 1

Escribir suma

FinAlgoritmo

Algoritmo Sumatoria

Definir Suma, i Como Entero

Suma \leftarrow 0

Escribir 'Nuevo programa'

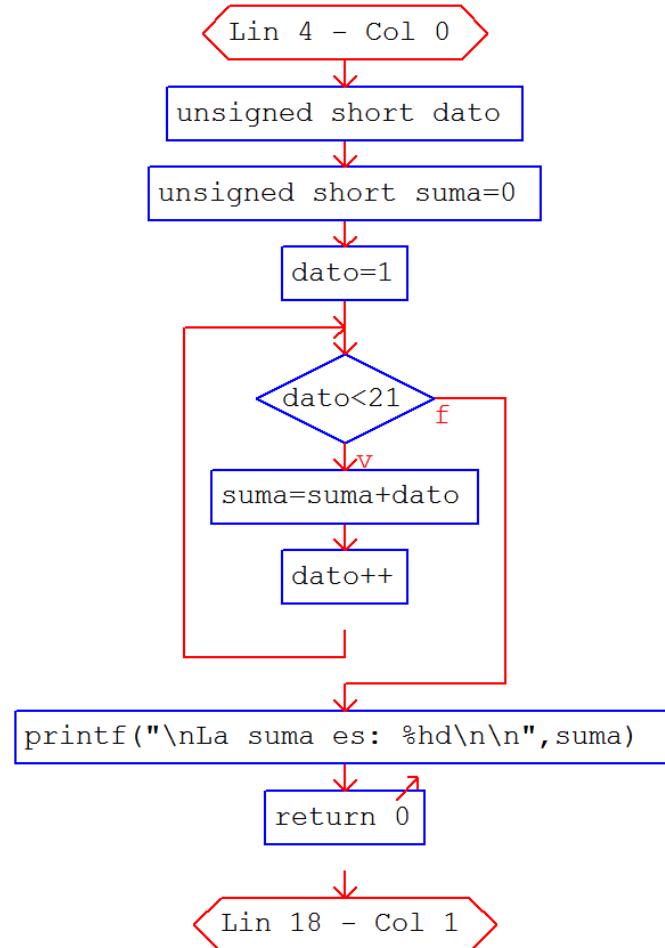
Para i Desde 0 Hasta 20 Con Paso

 Suma \leftarrow Suma + i

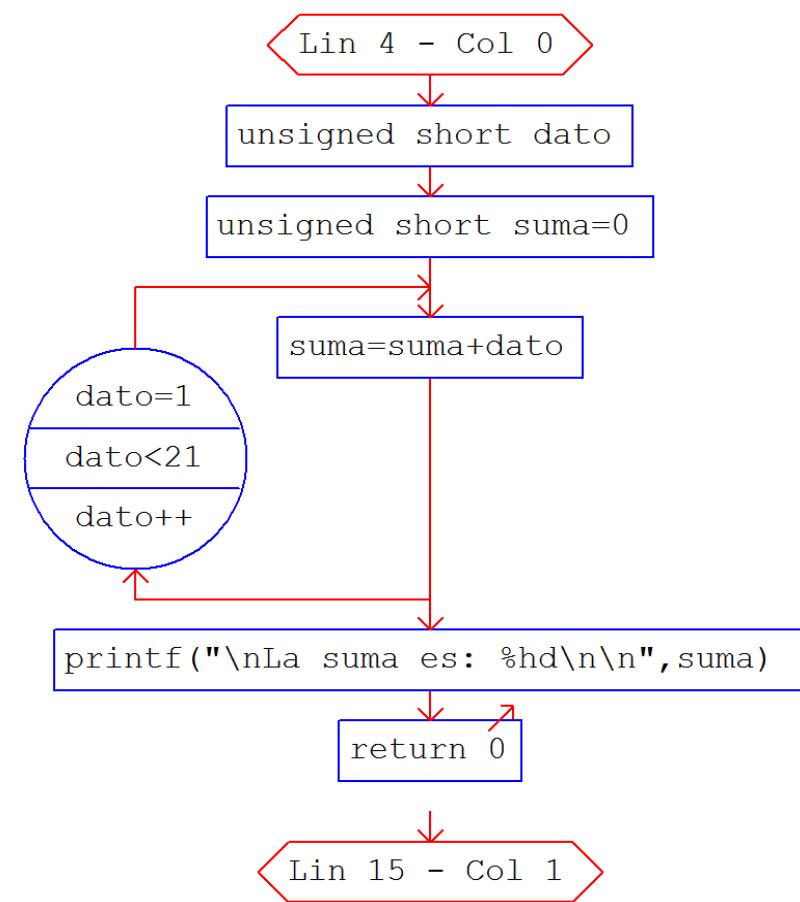
Escribir 'La suma es: ', Suma

FinAlgoritmo

Sumar los números enteros del 1 al 20 y hacerlo con un lazo while.



Sumar los números enteros del 1 al 20 y hacerlo con un lazo for.



Cálculo de factorial de un número en forma iterativa.

Calculo del factorial de 5 usando un lazo while.	Calculo del factorial de 5 usando un lazo for.
<pre>valor=5; factorial=1; while(valor>1) { factorial=factorial*valor; valor--; } printf("%llu",factorial); //llu long long</pre>	<pre>valor=5; factorial=1; for(;valor>1;valor--) factorial=factorial*valor; printf("%llu",factorial);</pre>

Como se dijo en otro apartado de este apunte, llu no funciona directamente en Windows, si en Linux, debiéndose agregar código adicional para esto.

Por lo tanto utilice long en lugar de long long para probar el código. Para ejecutarlo además, deberá declarar una función main y las variables, además de los #include que se consideren necesarios.

Veamos el diagrama de Chapin de estos códigos...

Factorial con while

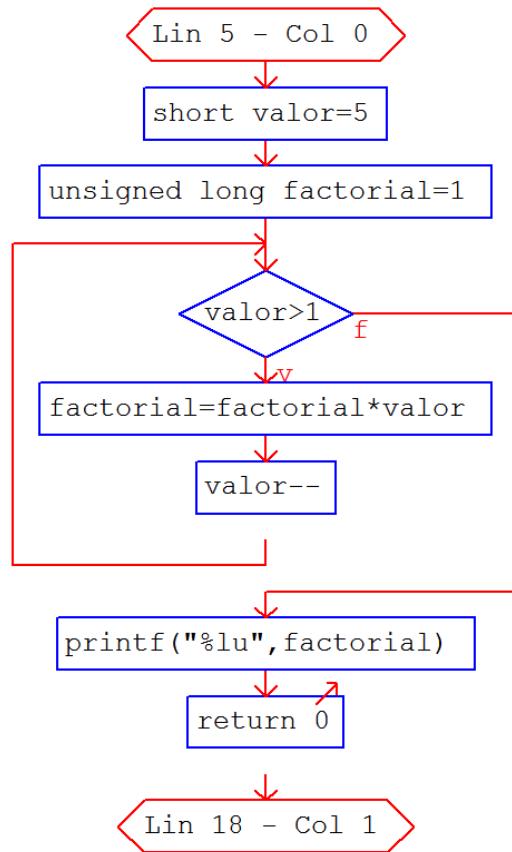
```
Algoritmo factorial_while
    valor ← 5
    factorial ← 1
    Mientras valor>1
        factorial ← factorial*...
        valor ← valor-1
    Escribir factorial
    FinAlgoritmo
```

Factorial con for

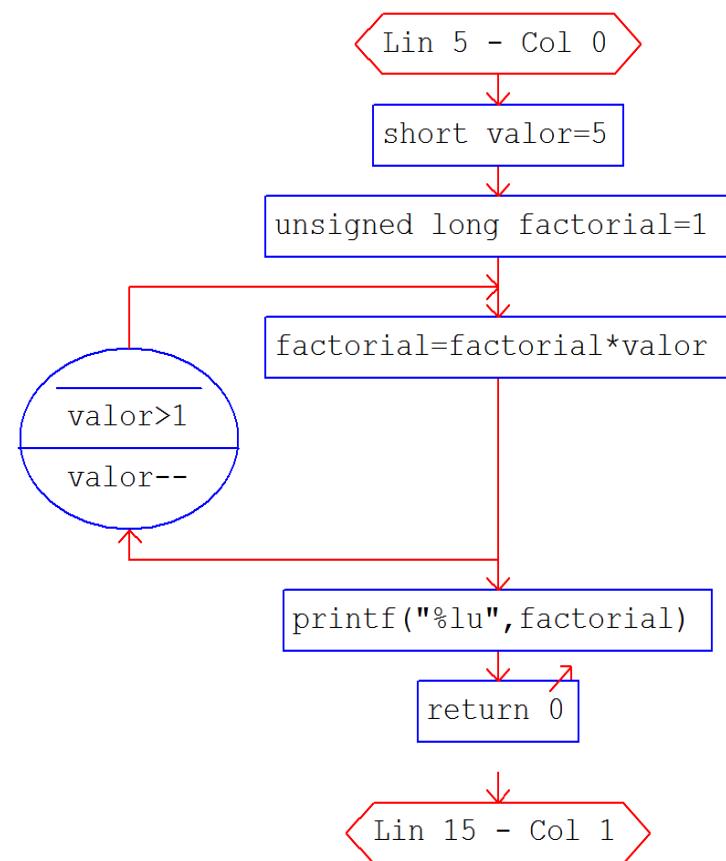
```
Algoritmo sin_titulo
    valor ← 5
    factorial ← 1
    Para valor Desde 5 Hasta 2 Con Paso -1
        factorial ← factorial*...
    Escribir factorial
    FinAlgoritmo
```

... y ahora los mismos códigos vistos desde el diagrama ANSI

Factorial con while



Factorial con for



do-while: Permite repetir una instrucción o conjunto de instrucciones, siempre y cuando la expresión sea y/o permanezca verdadera.

do

Instrucción o conjunto de instrucciones a ser repetidas
(cuerpo del lazo do-while)

while(expresión)

Un ejemplo típico de uso del do-while es rechazar valores que no cumplen con una determinada condición. Por ejemplo, el factorial se calcula con valores enteros positivos, incluido el cero.

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

int main()
{
    short valor;
    unsigned long factorial=1;

    do{
        printf("Ingrese un valor entero: ");
        scanf("%hd",&valor);
    }while(valor<0);

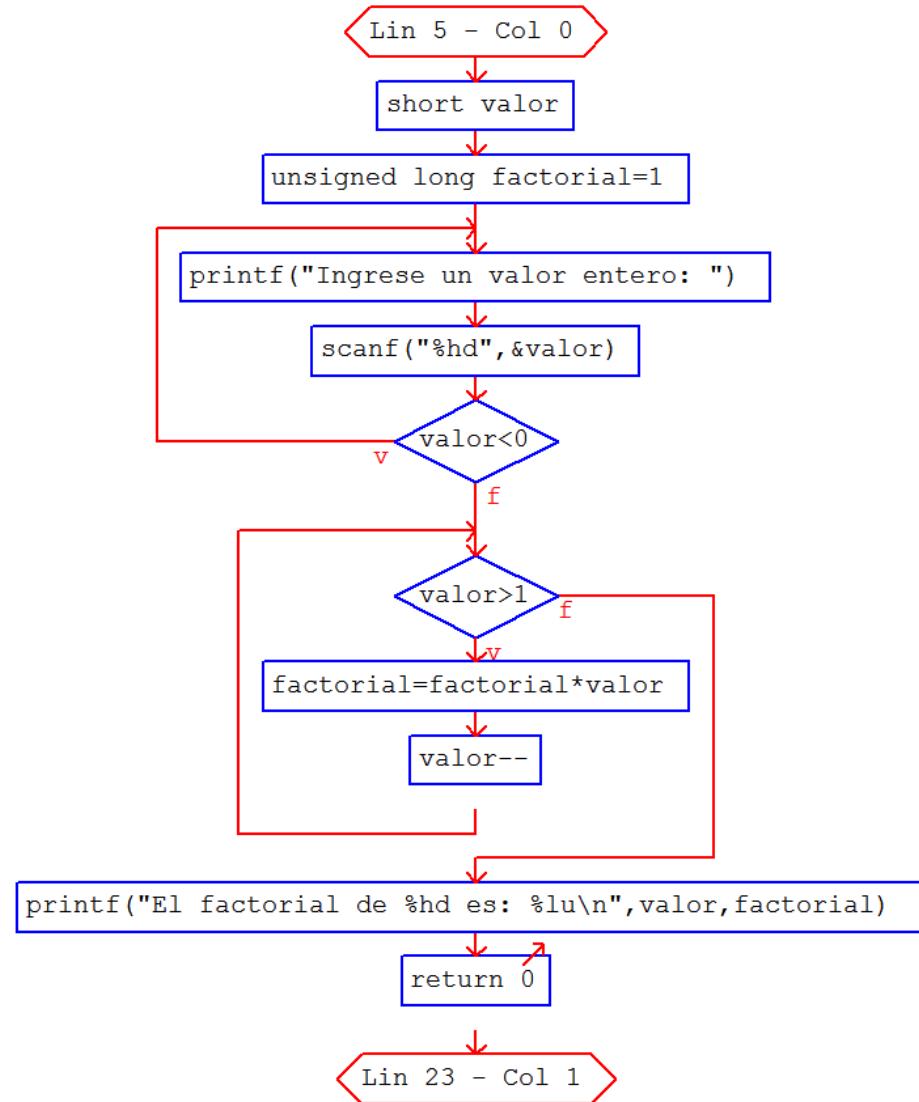
    while(valor>1)
    {
        factorial=factorial*valor;
        valor--;
    }

    printf("El factorial de %hd es: %lu\n",valor,factorial);

    return 0;
}
```

El “do-while” lo que hace en nuestro código es pedir un valor entero por teclado. **Si es mayor o igual a cero el valor ingresado por teclado, no cumple con la condición de valor < 0** y por lo tanto continua el código ejecutando el **while(valor >1)** y todo el código que sigue. Pero si se ingresa por ejemplo -5 (menos cinco), cumple la condición y vuelve a pedir un nuevo valor por teclado. El problema que tiene este código es que, si el usuario nunca ingresa un valor no negativo se quedará indefinidamente “girando” dentro del lazo do-while. Para evitar esto deberíamos permitir solamente una cierta cantidad de ingresos equivocados por teclado y luego de terminar con dicho lazo resolver que hacer. Si es un programa sencillo podríamos imprimir en pantalla que el valor ingresado por teclado no cumple con las condiciones que nos plantea la matemática y terminar el programa.

Aquí va el diagrama del código anterior.



Esto lo podríamos hacer como sigue, para lo cual planteamos que se pueda equivocar como máximo tres veces al ingresar el valor por teclado.

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

int main()
{
    short valor;
    unsigned long factorial=1;
    short equivocaciones=0;

    do{
        equivocaciones++;
        printf("Ingrese un valor entero: ");
        scanf("%hd",&valor);
    }while(valor<0 && equivocaciones<3);

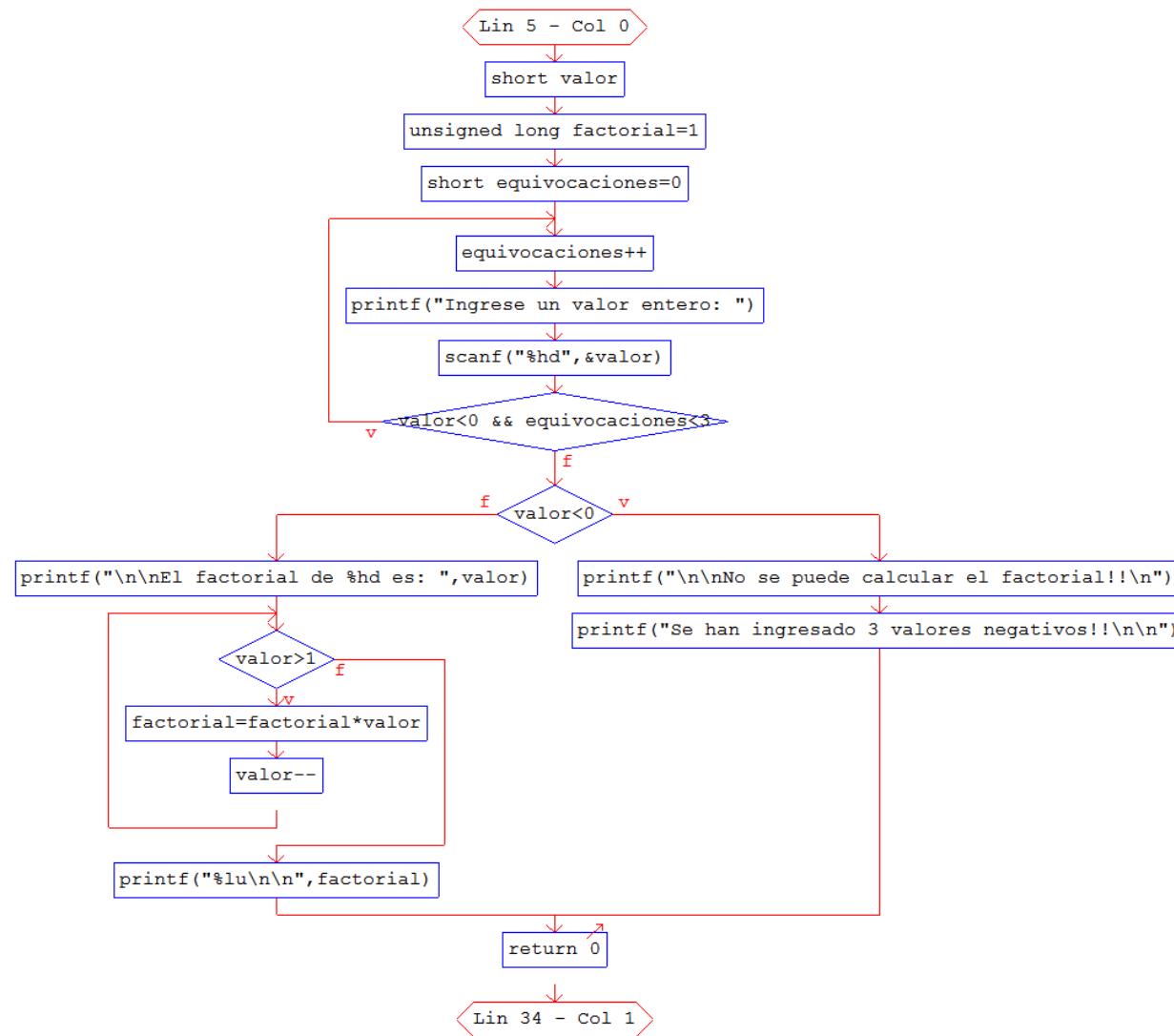
    if(valor<0)//Vemos por que termino el lazo do-while
    {
        printf("\n\nNo se puede calcular el factorial!!\n");
        printf("Se han ingresado 3 valores negativos!!\n\n");
    }
    else
    {
        printf("\n\nEl factorial de %hd es: ",valor);

        while(valor>1)
        {
            factorial=factorial*valor;
            valor--;
        }
        printf("%lu\n\n",factorial);
    }

    return 0;
}
```

Se ha remarcado el código do-while y la nueva variable que usamos para contar la cantidad de “vueltas” que se da dentro del lazo. No es la única forma de hacer esto. Es fundamental en la práctica “validar” la información. Piense que siempre el usuario puede cometer errores.

Aquí te presentamos el diagrama de flujo del programa anterior. Es muy importante graficar nuestras ideas. El diagrama sirve tanto como una forma de “pensar” nuestro código y una vez realizado también analizarlo.



Estos dos códigos cumplen con pedir valores enteros por teclado hasta que el valor ingresado sea cero o positivo. Ninguno es mejor que el otro...

... este código es...	... equivalente a este código
<pre>#include <stdio.h> #include <stdlib.h> int main() { short N; printf("Ingrese un entero: "); scanf("%hd",&N); while(N<0) { printf("Ingrese un entero: "); scanf("%hd",&N); } printf("El valor ingresado es: %hd",N); return 0; }</pre>	<pre>#include <stdio.h> #include <stdlib.h> int main() { short N; do{ printf("Ingrese un entero: "); scanf("%hd",&N); }while(N<0); printf("El valor ingresado es: %hd",N); return 0; }</pre>

Comparación de las tres únicas iterativas que tiene el lenguaje C

while	for	do - while
Repite un determinado conjunto de instrucciones	Repite un determinado conjunto de instrucciones	Repite un determinado conjunto de instrucciones
Si la primera vez que se evalúa la expresión es falsa no se ejecuta el cuerpo del while.	Si la primera vez que se evalúa la expresión es falsa no se ejecuta el cuerpo del for.	Si la primera vez que se evalúa la expresión es falsa se ejecuta el cuerpo del do – while y termina.
Se mantiene en el lazo por expresión verdadera.	Se mantiene en el lazo por expresión verdadera.	Se mantiene en el lazo por expresión verdadera.
Termina o no se ejecuta si la expresión es falsa.	Termina o no se ejecuta si la expresión es falsa.	Termina o se ejecuta una sola vez si la expresión es falsa.

Observe los colores de las celdas (por fila). A igual color, igual comportamiento...

Lazos infinitos en C

Se deben siempre evitar en los programas para PC y similares, salvo casos muy específicos y excepcionales. **En este curso no se van a utilizar en ningún caso lazos infinitos**, debiéndose evitar siempre caer en esta situación. En microcontroladores, Arduino por ejemplo, son obligatorios. **Conozca el tema y aprenda en que ámbito se usan y en cual no.**

Se pueden hacer de varias maneras. Lo típico es:

```
while(1)
{
}
```

```
for(; ; )
{
}
```

Ojo !!! Una forma de llegar al lazo infinito es no hacer que la/s variable/s que forman parte de la expresión lógica no varíen dentro del cuerpo del lazo. Esto es que quede siempre la expresión como verdadera sin tener posibilidad de ser falsa en algún momento.

Correcto	Incorrecto
<pre>dato=1; while(dato < 21) { suma = suma+dato; dato++; }</pre>	<pre>dato=1; while(dato < 21) { suma = suma+dato; /*falta dato++*/ }</pre>

Un ejemplo desde la matemática

Vamos a resolver en forma iterativa este sistema de dos ecuaciones lineales. **Siga la ejecución del código para ver como es el método de resolución.**

$$\begin{aligned} 2x - y &= 1 \\ -x + 2y &= 1 \end{aligned}$$

Para esto lo vamos a transformar, usando álgebra básica, en:

$$\begin{aligned} x &= (1 + y) / 2 \\ y &= (1 + x) / 2 \end{aligned}$$

Un posible código podría ser el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h> //para usar fabs

int main()
{
    float Xactual;
    float Yactual;
    float Xanterior;

    printf("Resolucion de ecuaciones en forma iterativa\n\n");
    printf("Ingrese un valor de arranque para X: ");
    scanf("%f",&Xactual);

    do{
        Xanterior=Xactual;
        Yactual=(1+Xanterior)/2;
        Xactual=(1+Yactual)/2;
        printf("\nXactual= %f \t Yactual= %f",Xactual,Yactual);
    }while(fabs(Xactual-Xanterior)>1E-6); //fabs es el valor absoluto de un numero float o double

    printf("\n\nX= %f Y= %f\n\n",Xactual,Yactual);

    return 0;
}
```

Switch – case (selección multiple)

- Salta a una **etiqueta (label)**.
- **No reemplaza al if.**
- **No es condicional.** No puedo usar expresiones lógicas.
- **La variable de control es siempre un entero o una única letra. No floats, no strings...**
- **En los casos que pueda utilizarse, es más rápido que hacer if (anidados o no anidados). Es más eficiente en tiempo de ejecución.**

Comparación de un código con if (condicional) y switch – case (incondicional)

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

int main()
{
    short A;

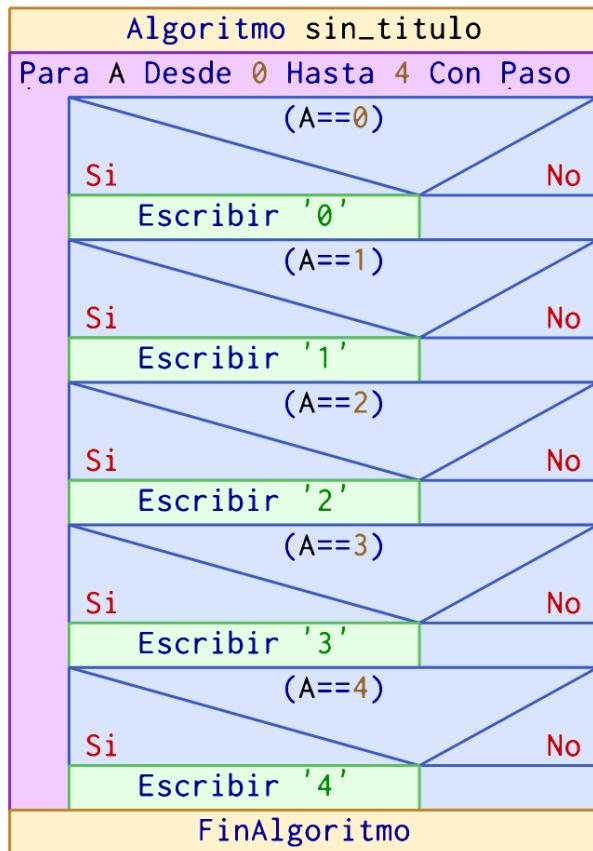
    for(A=0;A<5;A++)
    {
        if(A==0)
            printf("0\n");
        if(A==1)
            printf("1\n");
        if(A==2)
            printf("2\n");
        if(A==3)
            printf("3\n");
        if(A==4)
            printf("4\n");
    }
    return 0;
}
```

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

int main()
{
    short A;

    for(A=0;A<5;A++)
    {
        switch(A)
        {
            case 0:
                printf("0\n");break;
            case 1:
                printf("1\n");break;
            case 2:
                printf("2\n");break;
            case 3:
                printf("3\n");break;
            case 4:
                printf("4\n");break;
        }
    }
    return 0;
}
```

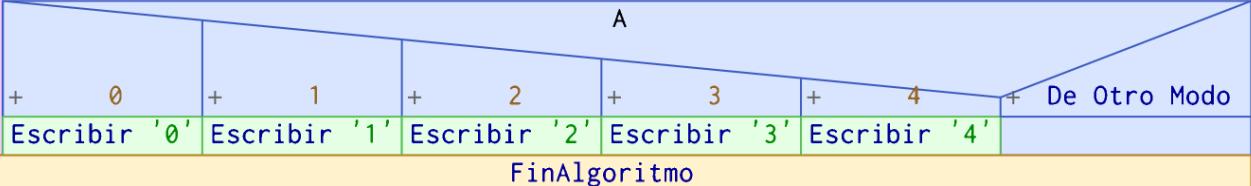
Con if



Con switch - case

Algoritmo switch_case

Para A Desde 0 Hasta 4 Con Paso 1



Otro código comparando if con switch – case. El alumno deberá analizar el código para ver como funciona y que hace.

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

int main()
{
    short Letra;

    for(Letra='a';Letra<'f';Letra++)
    {
        if(Letra=='a')
            printf("a\n");
        if(Letra=='b')
            printf("b\n");
        if(Letra=='c')
            printf("c\n");
        if(Letra=='d')
            printf("d\n");
        if(Letra=='e')
            printf("e\n");
    }
    return 0;
}
```

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

int main()
{
    short Letra;

    for(Letra='a';Letra<'f';Letra++)
    {
        switch(Letra)
        {
            case 'a':
                printf("a\n");break;
            case 'b':
                printf("b\n");break;
            case 'c':
                printf("c\n");break;
            case 'd':
                printf("d\n");break;
            case 'e':
                printf("e\n");break;
        }
    }
    return 0;
}
```

Un típico menú realizado con switch – case llamando a funciones (tema que se verá en otro apartado de este apunte)

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

short funcion01(void);
short funcion02(void);
short funcion03(void);

int main(void)
{
    char control;

    do{
        fflush(stdin);
        printf("Ingrese una opcion a, b o c: ");
        scanf("%c",&control);
        switch(control)
        {
            case 'a': funcion01(); break;
            case 'b': funcion02(); break;
            case 'c': funcion03(); break;
            default: printf("\nNinguno de los anteriores\n\n");break;
        }
    }while(control<'d');

    return 0;
}
```

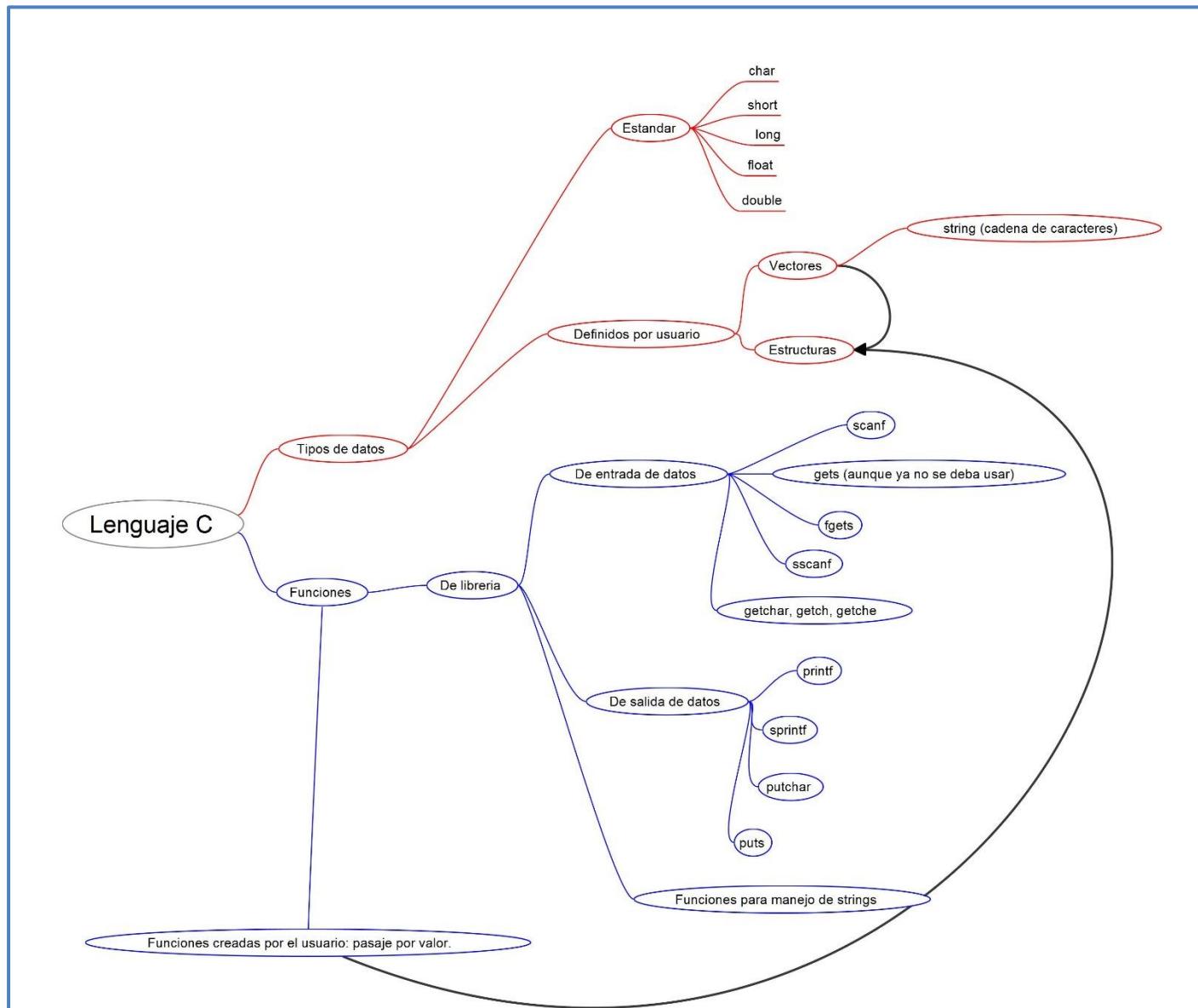
```
short funcion01(void)
{
    printf("\nSoy funcion 1\n\n");
    return 1;
}

short funcion02(void)
{
    printf("\nSoy funcion 2\n\n");
    return 2;
}

short funcion03(void)
{
    printf("\nSoy funcion 3\n\n");
    return 3;
}
```

Visión general de lo que vimos y vamos a ver (que debo saber)

Si nunca has programado, este resumen tal vez te sirva. En el siguiente gráfico veremos lo que deberías saber en INTRODUCCION AL DESARROLLO DE SOFTWARE o en INFORMÁTICA I. Es lo “mínimo”, desde mi punto de vista, que jamás deberías dejar de saber.



No se debe tomar el gráfico como “lo único que debo saber”. Es solo una referencia MÍNIMA e incompleta. Lo que si dice el gráfico es que lo que está ahí es un **si o si** lo debo saber. **Es la base de la programación es C.** También se debe conocer una serie de algoritmos fundamentales, tales como:

- **Búsqueda de máximos y mínimos**
- **Cálculo de promedios**
- **Cálculo de porcentajes**
- **Busqueda secuencial y binaria**
- **Busqueda con inserción**
- **Ordenamiento de datos**
- **Otros algoritmos de uso común (factorial, minimo común múltiplo, Fibonacci, etc)**

Es fundamental conocer perfectamente el uso de **struct**, en particular para devolver mas de un dato desde una función.

Struct es la base de la programación orientada a objetos.

Hay que manejar perfectamente vectores tanto numéricos, de char (strings) y de estructuras.

Los vectores y las cadenas de caracteres (strings) son fundamentales en programación. En particular en strings debemos saber como copiar un string en otro, como comparar dos strings, como saber cuantos caracteres válidos contiene, entre otras cosas. No existen, en general, códigos que no usen string. **Los strings son fundamentales en programación.** Conocer las funciones que manejan strings implica conocer sus fortalezas y debilidades (limites y capacidades en su uso). Scanf es una funcion que tiene muchas opciones para el manejo de strings (ingreso de strings desde teclado) y es fundamental conocer alguna de ellas para poder resolver problemas de uso frecuente. Un string, por ejemplo, puede contener una frase o un dato alfanumérico (por ejemplo una patente de automóvil).

Funciones es otro tema que no se debe desconocer. **Los códigos C están “armados” en base a funciones.**

Funciones en C

Funciones en matemática

(tema optativo no de la asignatura para conocimiento general)

Definición:

En matemática, una función f es una relación entre un conjunto dado X (el dominio) y otro conjunto de elementos Y (el codominio) de forma que a cada elemento x del dominio le corresponde un único elemento del codominio $f(x)$.

La notación matemática es:

$$f: X \rightarrow Y$$

La flecha representa la relación que existe entre el conjunto entrada y el conjunto salida:

Ej: $Y = X + 2$

$X + 2$ es la relación que existe entre la entrada, X y la salida Y .

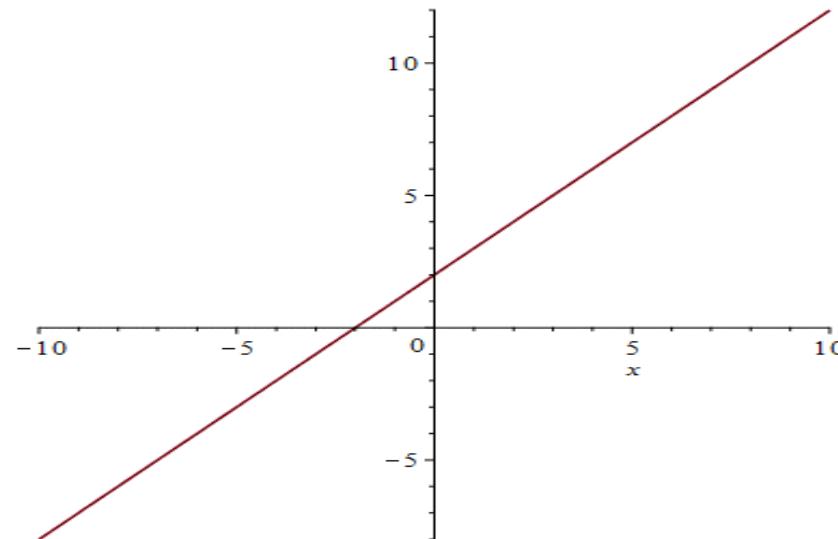
En matemática a los que nosotros llamamos **tipos de datos en programación** se le suele llamar **conjunto numérico**.

A la variable **X** en matemática se le suele llamar variable independiente y a la variable **Y** se le suele llamar dependiente.

Tendremos matemáticamente los siguientes tipos de funciones (no se nombran todos):

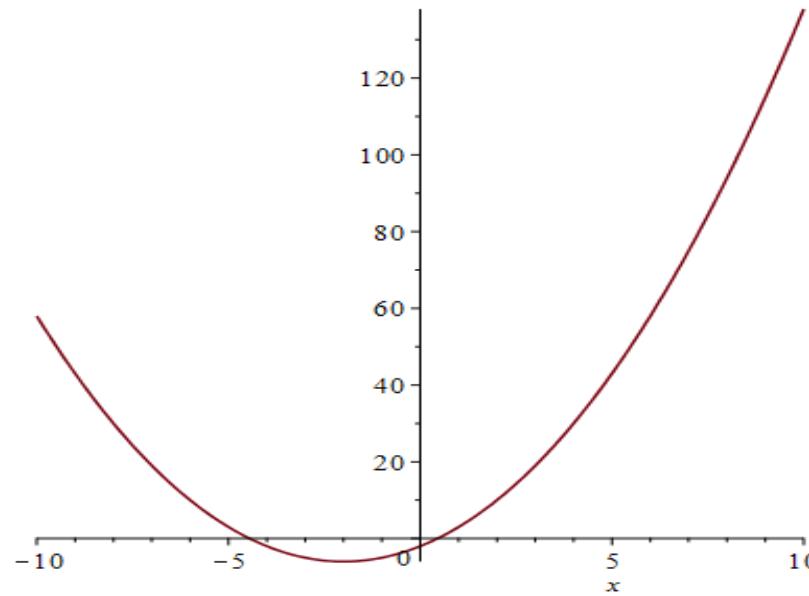
- Función lineal

$$y = x + 2$$



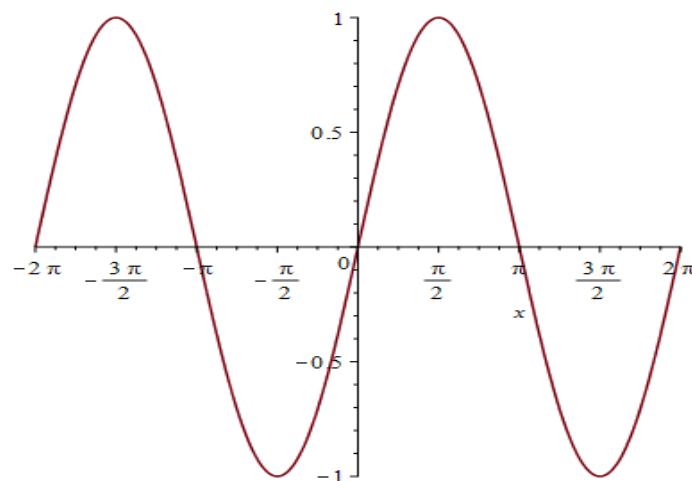
- Función cuadrática

$$y = x^2 + 4x - 2$$

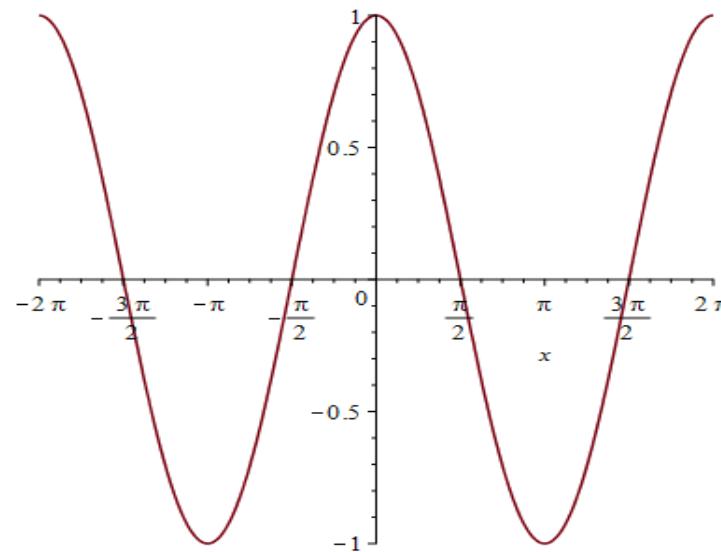


- Funciones trigonométricas

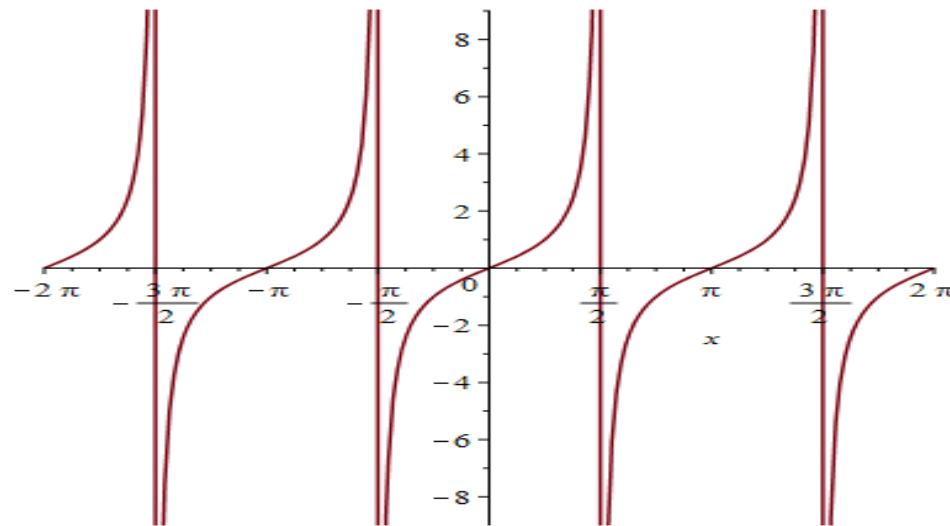
- $y = \operatorname{sen}(x)$



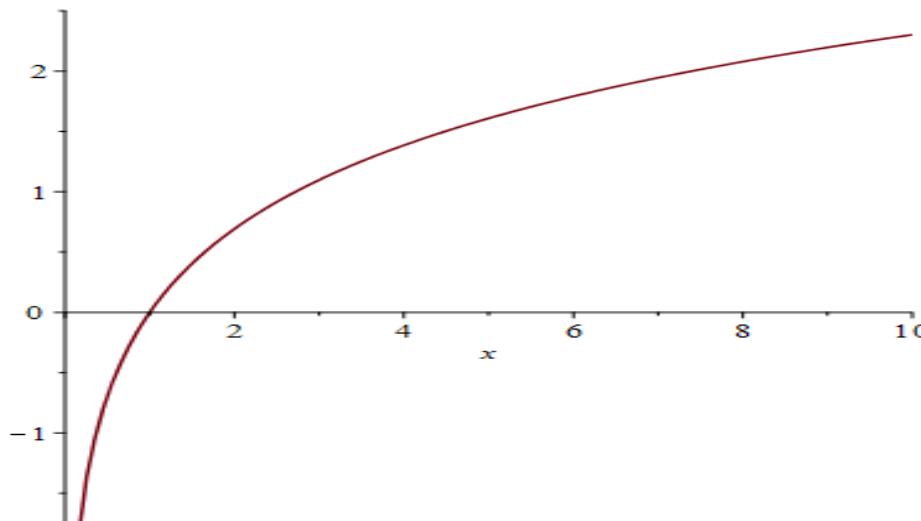
- $y = \operatorname{cosen}(x)$



- $y = \operatorname{tangente}(x)$



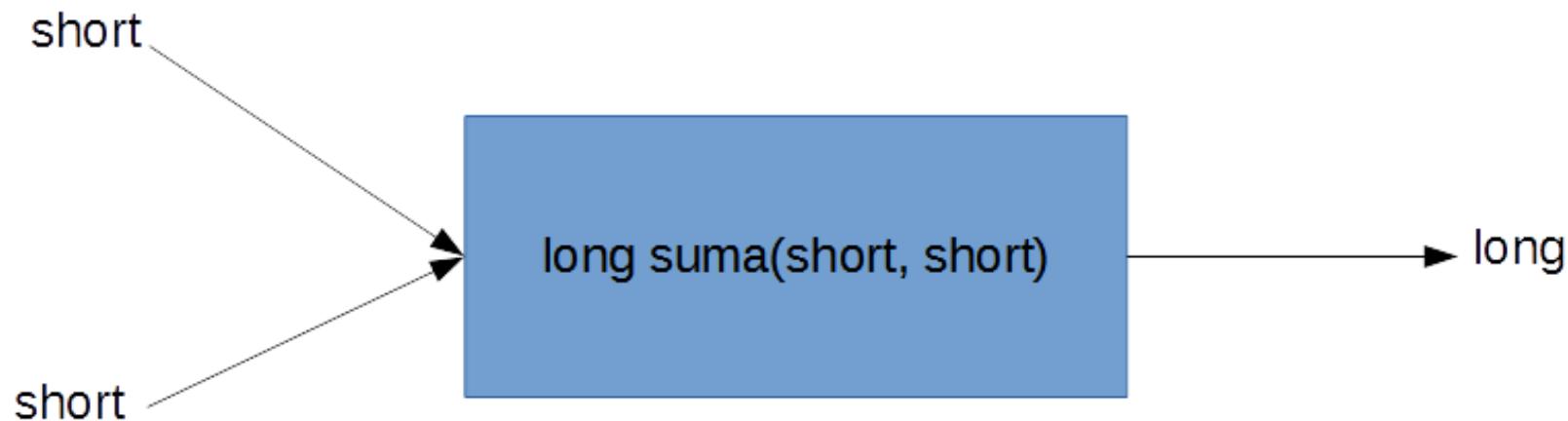
- Función logaritmo base e: $y = \ln(x)$



Más adelante volveremos a las funciones matemáticas desde el lenguaje C.

Concepto general de funciones desde la programación en C

Cuando utilizemos funciones dentro de nuestro código, las podremos llegar a ver muchas veces como “**cajas negras**” ya que, en la mayoría de los casos, **NO SABREMOS**, como dicha función está implementada (no conocemos su código), pero **SI SABREMOS**, que hace, que recibe y que devuelve por medio de su prototipo o de su header donde están los prototipos o de otra información.



Una función es un bloque de código independiente que se ubica en memoria. Una función que ya conocemos es main() de la que ya hemos escrito su código. También hemos usado printf y scanf de las que **NO SABEMOS** su código pero si que reciben y que devuelven. Ahora veremos cómo escribir nuestras funciones en general...

```
int main(void)
{
    printf("Hola mundo");
    return 0;
}
```

← Función main

Código de la función printf

← Función printf

Las funciones son la mejor forma de “modularizar” un código. Las funciones realizan tareas en general muy específicas: printf imprime en la pantalla, scanf lee del teclado, sqrt(x) de math.h calcula la raíz cuadrada, pow (m, n) también de math.h calcula m elevado a la n y así cientos de funciones.

Es por lo tanto fundamental aprender a hacer nuestras propias funciones. NO ES BUENO TENER UN MAIN GIGANTE. El main debería ser pequeño y que en general llame a las funciones que realizan las tareas específicas. Un main gigante es muy difícil de modificar y mantener, además de que puede ser factible que sea muy grande el número de errores. Debemos dividir nuestro código en tareas específicas por medio de funciones.

Además cuando se trabaja en equipo es más fácil dividir las tareas a realizar. Uno hace una función de ingreso, otro una función que imprima en pantalla, otro procesa en una función los datos, etc. Cada uno deberá informar a los otros el nombre de la función, que tipo de dato recibirá la función y que devuelve la función. Con esto es suficiente para utilizarla, de hecho no sabemos cuál es el código de printf o scanf y no tenemos problemas a la hora de usarla. Además las funciones suelen estar documentadas para conocer de ellas sus características y sus limitaciones de uso. Hay funciones que solo trabajan con valores numéricos, otras solo pueden trabajar con cadenas de caracteres (string), otras solo con estructuras, etc.

DOCUMENTAR EN TODO LO POSIBLE LAS FUNCIONES ES UNA MUY BUENA PRÁCTICA.

CARACTERÍSTICAS DE LAS FUNCIONES en C

Todas las funciones en C tienen un nombre, al menos un tipo de dato que reciben (algunas veces este es void), y un tipo de dato que devuelven, que también puede ser void.

Void básicamente es un comodín, un tipo de dato que ocupa cero bytes y que en el caso de usarse en la funciones, representa que la función no va a recibir ningún dato desde ningún lado y/o que no va a devolver ningún dato a nadie.

A esta información de:

- nombre de la función
- que tipo de dato recibe la función
- que tipo de dato devuelve la función

se denomina *prototipo de la función*. (muy importante!!)

SIEMPRE QUE ESCRIBAMOS NUESTRA FUNCION DEBEREMOS ESCRIBIR EL PROTOTIPO DE LA MISMA. NUNCA DEBEMOS DEJAR DE HACERLO. POR MEDIO DEL PROTOTIPO EL COMPILADOR SE ENTERA DE QUE LA FUNCION EXISTE Y QUE FORMA TIENE.

LOS PROTOTIPOS DE LAS FUNCIONES QUE NO HEMOS ESCRITO NOSOTROS SE ENCUENTRAN EN LOS .H (STDIO.H, MATH.H, ETC).

COMO ESCRIBIR MIS PROPIAS FUNCIONES

(tema fundamental en programación)

Supongamos que se nos pide escribir una función que reciba como parámetro dos valores enteros y devuelva cero (0) si son iguales, uno (1) si el primero es mayor a al segundo, o menos uno (-1) si el segundo es mayor que el primero.

Primero vemos el prototipo:

short **compara**(**short**, **short**);

La función devuelve un **short**, se llama **compara** y recibe **dos enteros short** (guiarse por los colores).

Veamos un posible código de la función:

```
short compara(short A, short B)
{
    if(A > B)
        return 1;
    if(A == B)
        return 0;
    if(A < B)
        return -1;
}
```

Por medio de RETURN se devuelve el valor desde la función. Además return termina la función, esto es, si se ejecuta return, el código que sigue, no se ejecuta. Si sale de nuestra función por return 1, todo el código por debajo no es ejecutado. Esto es muy importante.

Otra forma que puede tomar la función es la siguiente:

```
short compara(short A, short B)
{
    if(A > B)
        return 1;
    if(A==B)
        return 0;

    return -1;
}
```

Vemos que el último “if” era innecesario ya que era la “opción que nos quedaba”.

Es ya de notar que la “llamada a la función” no se enterará de este cambio. Es una caja negra la función. Recibe valores y devuelve valores sin conocerse en general como lo hace. Como ya dijimos lo único que debemos saber en general es su prototipo.

Un posible código completo para prueba es el siguiente:

```
#include <stdio.h>

short compara(short, short); // prototipo de la función

int main(void)
{
    short resultado; // variable donde guardaremos lo devuelto por la función

    resultado=compara(6,16); // llamada a la función con dos valores

    if(resultado==0)
        printf("Los valores son iguales\n");
    if(resultado==1)
        printf("El primero es mayor al segundo\n");
    if(resultado==-1)
        printf("El segundo es mayor al primero\n");
    return 0;
}

short compara(short A, short B) // la implementación del código de la función
{
    if(A > B)
        return 1;
    if(A==B)
        return 0;

    return -1;
}
```

Cuando hacemos la llamada a la función (la usamos), en este ejemplo en la función compara, A toma el valor 6 y B toma el valor 16, siempre en ese orden. Luego al código de la función lo podemos pensar como si lo hiciéramos en el main. Luego, la función mediante return devuelve 1, 0 o -1 según corresponda.

Vean además que la función la hemos escrito inmediatamente abajo del main. Hay otras formas pero no las veremos en este curso.

Otra forma de hacer la misma función donde veremos que solo cambia el código de la función compara pero en main no hay ningún cambio.

```
#include <stdio.h>

short compara(short, short);

/** la funcion main */

short compara(short A, short B)
{
    short para_retornar;

    if(A > B)
        para_retornar= 1;
    if(A==B)
        para_retornar= 0;
    if(A<B)
        para_retornar=-1;

    return para_retornar;
}
```

Un par de reglas que son importantes tener en cuenta a la hora de escribir funciones:

- 1) Las funciones deben ser específicas en todo lo posible. Esto es una función que ingresa datos no imprime, una que imprime no ingresa datos, una que procesa la información no ingresa ni imprime los datos. Las funciones deben ser lo más cortas posible.
- 2) Siempre que se pueda, la función deberá devolver uno o más valores por éxito y uno por fracaso. Esto no es siempre posible, pero de serlo será muy bueno hacerlo así.
- 3) Una función debería en todo lo posible, evitar llamar a otras funciones. Hace más lento el código. De nuevo, esto no siempre es posible, pero cuando se pueda, es una buena práctica.
- 4) Las funciones pueden devolver un UNICO TIPO DE DATO. Gracias al uso del tipo de dato estructura puedo devolver varios valores y de distinto tipo. Pensemos una estructura con campos float, long, short, etc. Usar estructuras (struct) es una gran manera de saltar la limitación de la función con respecto a devolver un único tipo de dato.

Un ejemplo sencillo de uso de funciones

<pre>#include<stdio.h> long suma(short,short); //prototipo de mi funcion suma int main() { short A, B; long S; printf("Ingrese el primer numero: "); scanf("%hd",&A); printf("Ingrese el segundo numero: "); scanf("%hd",&B); S=suma(A, B); printf("La suma es: %ld",S); return 0; }</pre>	<pre>long suma(short X, short Y) { long M; M = X + Y; return M; }</pre>
---	---

Otros ejemplos de funciones

a) Una función que recibe un valor entero y no devuelve nada. Todo lo hace internamente la función.

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

```
void imprime_en_pantalla(short);
```

En el **prototipo** de la función no es necesario colocar el nombre de las variables. Si se hace no pasa nada. Además la definición del prototipo siempre termina con punto y coma (;).

```
int main(void)
```

```
{
```

```
    short valor_ingresado;
```

Tipo de dato que va a recibir la función. En este caso short.

```
printf("Ingrese un valor entero: ");
```

```
scanf("%hd",&valor_ingresado);
```

```
imprime_en_pantalla(valor_ingresado);
```

El **void** significa que la función no va a devolver nada. NO se usa **return** para este caso en la función.

```
return 0;
```

```
}
```

La implementación de la función (su código) no tiene punto y coma aquí. OJO con eso!!

```
void imprime_en_pantalla(short dato)
```

```
{
```

```
    printf("El valor ingresado fue: %hd\n",dato);
```

```
}
```

Dato en la función va a tomar el valor que tenga la variable **valor_ingresado**.

- b) Una función que no recibe nada y devuelve un entero. La función pide un dato por teclado y es impreso en pantalla en el main.

```
#include <stdio.h>

short lee_el_teclado(void);

int main(void)
{
    short valor_ingresado;
    valor_ingresado=lee_el_teclado();
    printf("El valor ingresado fue: %hd",valor_ingresado);
    return 0;
}

short lee_el_teclado(void)
{
    short dato;
    printf("Ingrese un dato entero: ");
    scanf("%hd",&dato);
    return dato;
}
```

Tipo de dato devuelto por la función.
En este caso un **short**.

Como la función no espera ningún valor los paréntesis, **que son obligatorios**, van vacíos.

Variable que la función va a devolver.

c) Y ahora usamos ambas funciones en un solo código

```
#include <stdio.h>

short lee_el_teclado(void);
void imprime_en_pantalla(short);

int main(void)
{
    short valor_ingresado;

    valor_ingresado=lee_el_teclado();
    imprime_en_pantalla(valor_ingresado);
    return 0;
}

short lee_el_teclado(void)
{
    short dato;

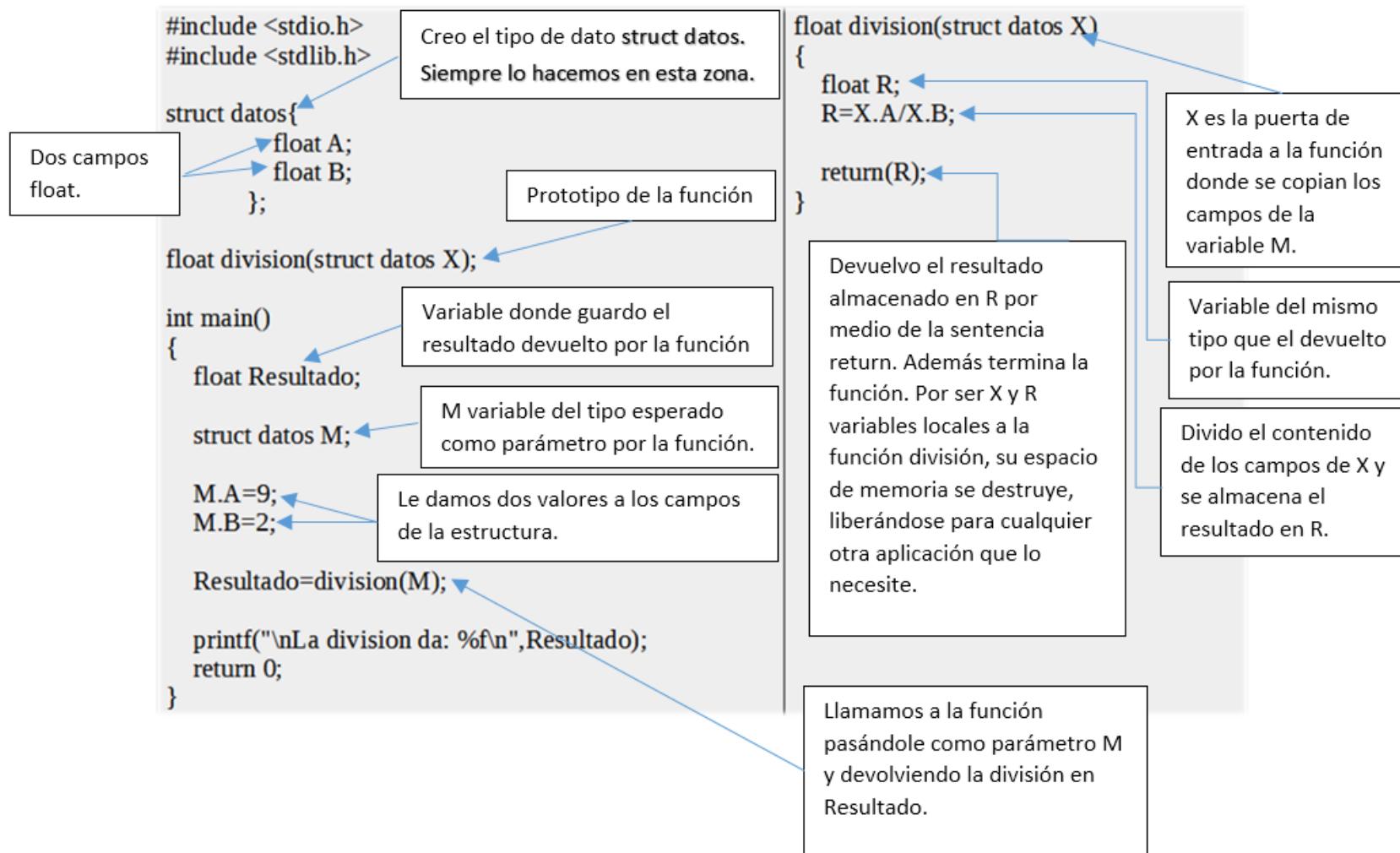
    printf("Ingrese un dato entero: ");
    scanf("%hd",&dato);
    return dato;
}

void imprime_en_pantalla(short dato)
{
    printf("El valor ingresado fue: %hd\n",dato);
}
```

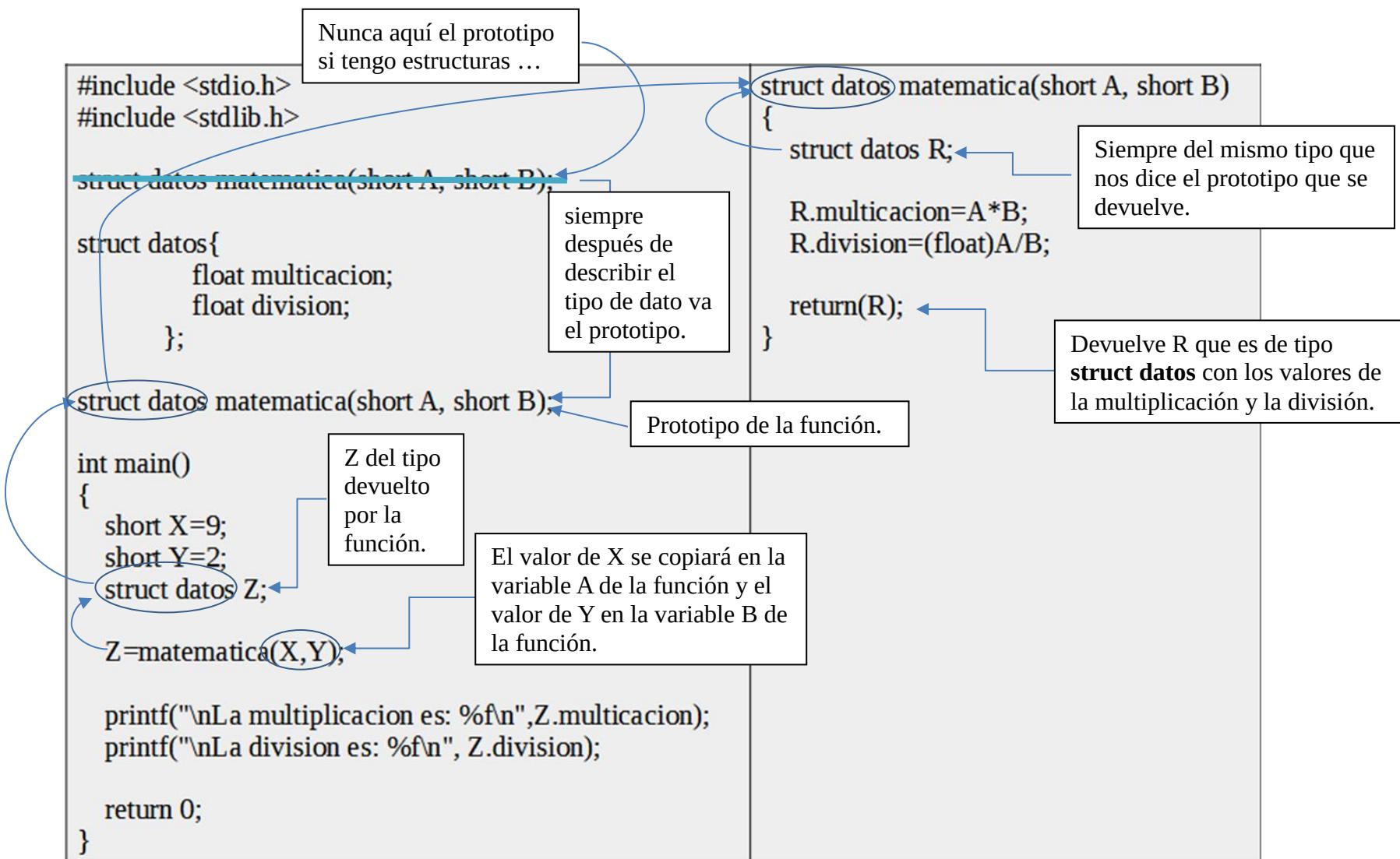
Funciones y estructuras

(...como devolver mas de un valor desde una función...)

Veamos cómo hacer una función que reciba una estructura (struct)



Ahora veremos un código muy importante: una función que devuelve una estructura, quebrando la limitación de las funciones a devolver un solo valor. Las funciones siempre devolverán un único tipo de dato.



Ejemplo para EDA1 e informática II. Recibe la función un vector de estructuras. Se debería ver después de saber bien vectores y punteros, no antes. Aquí solo a modo de ejemplo y adelanto del tema.

<pre>#include <stdio.h> #include <stdlib.h> #define tamano 10 struct datos{ short valor; short cuadrado; }; void imprimir(struct datos *Z); int main() { short i; struct datos M[tamano]; for(i=0;i<tamano;i++) { M[i].valor=i+1; M[i].cuadrado=(M[i].valor)*(M[i].valor); } imprimir(M); return 0; }</pre>	<pre>void imprimir(struct datos *Z) { short i; for(i=0;i<tamano;i++) { printf("%hd\t%hd\n",Z[i].valor,Z[i].cuadrado); } }</pre>
---	---

Ejemplo para EDA1 e informática II. Ordena un vector de estructuras. Saber bien vectores y punteros!!!

<pre>#include <stdio.h> #include <stdlib.h> #define tam 5 struct info { char NyA[81]; short edad; }; void ordenar(struct info *, short); int main(void) { struct info X[tam]; short i; for(i=0; i<tam; i++) { fflush(stdin); printf("Ingrese un nombre: "); gets(X[i].NyA); printf("Edad: "); scanf("%hd", &X[i].edad); } printf("\n"); ordenar(X, tam); for(i=0; i<tam; i++) printf("Nombre: %s \t edad: %hd\n", X[i].NyA, X[i].edad); return 0; }</pre>	<pre>void ordenar(struct info *V, short tamano) { short i, j; struct info auxi; for(i=0; i<tamano-1; i++) { for(j=i+1; j<tamano; j++) { if(V[i].edad>V[j].edad) { auxi=V[i]; V[i]=V[j]; V[j]=auxi; } } } }</pre>
--	--

Funciones matemáticas en C

Para poder utilizar las funciones de la librería standard de C deberemos incluir ***math.h***.

Si las funciones son trigonometrías los angulos se expresan, tanto como argumentos como también resultado en radianes.

Recordemos que **π radianes son 180 grados**. 360 grados serán 2π radianes.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h> /*La incluimos para usar las funciones matematicas*/

int main()
{
    float angulo_en_radian;
    float angulo_en_grados;
    float y;

    printf("Funciones trigonometricas\n");

    angulo_en_grados=45;
    angulo_en_radian=angulo_en_grados*M_PI/180; /*M_PI esta definida en math.h*/

    printf("\n%f grados es %f radianes\n",angulo_en_grados,angulo_en_radian);

    y=sin(angulo_en_radian);
    printf("sin(%f)= %f\n",angulo_en_radian,y);

    y=cos(angulo_en_radian);
    printf("cos(%f)= %f\n",angulo_en_radian,y);
```

```
y=tan(angulo_en_radian);
printf("tan(%f)= %f\n",angulo_en_radian,y);

y=atan(1); /**que es atan?*/
printf("atan(%f)= %f\n",1.0,y);

printf("\n\nConstantes en math.h\n");

printf("M_PI= %1.15f\n",M_PI);
printf("M_E= %1.15f\n",M_E);

printf("\n\nRaiz cuadrada\n");
printf("sqrt(16)= %.0f",sqrt(16));

printf("\n\nPotencia x elevada a la y\n");
printf("pow(2,5)= %.0f\n\n",pow(2,5));

return 0;
}
```

Vamos con algunos ejemplos matemáticos

```
/*Programa que calcula el valor de una función y la derivada en un punto dado*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <conio.h>  
#include <math.h>  
double funcion(double x);  
double derivada(double x);  
  
int main(void)  
{  
    double y,x;  
    char tecla;  
  
    do{  
        printf("\nIngrese el punto donde calcular la funcion y su derivada: ");  
        scanf("%lf",&x);  
        y=funcion(x);  
        printf("\nValor funcion: %lf\n",y);  
        y=derivada(x);  
        printf("Valor derivada: %lf\n\n",y);  
        printf("\nNuevo valor de x? (s / n): ");  
        tecla=getch();  
    }while(tecla != 'n');  
  
    return 0;  
}
```

```

/*Calculo de raices de una funcion por el metodo de Newton – Raphson*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>

double funcion(double x);
double derivada(double x);

int main(void)
{
    double xanterior,xactual, error=(double)1E-7;
    int tecla;
    long iteraciones,contador=0;

    do{
        printf("\n\nIngrese el punto de arranque: "); scanf("%lf",&xactual);
        printf("Ingrese la cantidad maxima de iteraciones: "); scanf("%ld",&iteraciones);

        do
        {
            xanterior=xactual;
            xactual=xanterior-(funcion(xanterior))/derivada(xanterior);
            printf("Contador: %ld\t\ttraiz: %f\n",contador,xactual);
            contador++;
        }while((fabs(xactual-xanterior)>error)&&(contador<iteraciones));

        printf("\n\nUna raiz real de la funcion es: %f\n",xactual);
        contador=0;
        printf("\n\nSi quiere seguir pulse la letra s minuscula u otra para terminar: ");
        tecla=getch();
    }while(tecla=='s');
    return 0;
}

```

```
double funcion(double x)
{
    double y;
    y=x*x*x+2;
    return y;
}

double derivada(double x)
{
    double y;
    double deltax=(double)1E-9;
    y=((long double)funcion(x+deltax)-(long double)funcion(x))/deltax;
    return y;
}
```

/*Son las mismas funciones del ejemplo anterior, las hemos reutilizado!!!!, no las hemos vuelto a escribir!!!*/

Variables de tipo static en funciones en C

Muchas veces necesitaremos que una función, cada vez que sea llamada, no pierda el valor que tenía una cierta variable. Para exemplificar, vamos a realizar un contador por medio de una función.

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

void funcion(void)
{
    int i = -10;
    printf("%d", i);
    i++;
}

void main(void)
{
    do{
        funcion();
    }while(!kbhit());
}
```

Si ejecutamos este programa tal cual está, el valor que se imprimirá cada vez que se llame a la función, será siempre –10, sin tener ningún efecto el incremento de i.

Para solucionar este “problema” usamos la palabra reservada static.

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

void funcion(void)
```

void main(void)
{
 do{
 funcion();
 }while(!kbhit());
}

```
void funcion(void)
{
    static int i = -10;
    printf("%d", i);
    i++;
}
```

Con punteros, cuando los vea, se tendrá una solución mejor y diferente a este “problema”.

Funciones para el ingreso de
caracteres por teclado
y
comparación de caracteres

1) ¿Con qué función ingreso un único carácter por teclado?

Tenemos varias opciones que se diferencian entre si por necesitar o no una indicación de fin de ingreso que normalmente es la tecla **ENTER**. Veamos algunas de ellas...

Con ENTER

FORMA 1: Usando **scanf**.

```
char letra;  
printf("Ingrese una letra: ");  
scanf("%c",&letra);  
printf("\nLa letra fue: %c\n",letra);
```

FORMA 2: Usando **getchar**.

```
char letra;  
printf("Ingrese una letra: ");  
letra=getchar();  
printf("\nLa letra fue: %c\n",letra);
```

FORMA 3: Usando **fgetch**.

```
char letra;  
printf("Ingrese una letra: ");  
letra=fgetchar();  
printf("\nLa letra fue: %c\n",letra);
```

Sin ENTER

FORMA 1: Usando getch.

```
char letra;  
printf("Ingrese una letra: ");  
letra= getch();  
printf("\nLa letra fue: %c\n",letra);
```

FORMA 2: Usando getche.

```
char letra;  
printf("Ingrese una letra: ");  
letra= getche();  
printf("\nLa letra fue: %c\n",letra);
```

Para el correcto uso de estas funciones se deberá incluir alguno o todos estos headers: **stdio.h**, **conio.h**, **curses.h**, **ncurses.h**, según que compilador se tenga y que sistema operativo se esté utilizando. Muchos de ellos se deben descargar desde Internet para su uso ya que algunos no son estándar, sino que han sido creados para que muchas funciones se puedan volver a utilizar, como es el caso de las que se encontraban bajo sistema operativo MSDOS en conio.h que, bajo Linux, se reemplaza por curses.h o ncurses.h, según disponibilidad.

2) ¿Cómo comparo caracteres?

Puedo tener la necesidad de saber si una variable contiene un determinado carácter o no. Por ejemplo quiero saber si contiene el carácter A.

Puedo hacer esto:

if(letra == 'A')

ó

if(letra == 65)

siendo el resultado final exactamente el mismo.

Puedo querer saber si un determinado carácter se encuentra dentro de un determinado rango de la tabla ASCII, por ejemplo saber si es una mayúscula.

Para esto puedo hacer:

if((letra >= 65) && (letra <= 90))

ó

if((letra >= 'A') && (letra <= 'Z'))

ambas formas generan exactamente el mismo resultado.

Nota: Estas mismas expresiones pueden ser utilizadas en el **while**, **do – while** y **for**.

Ejemplos:

```
char letra;  
letra = getch();  
  
while(letra != 'm')  
{  
    letra = getch();  
}
```

```
char letra;  
  
do{  
    letra = getch();  
}while(letra != 'm');
```

¿Qué hacen estos códigos?

Recursividad

(EDA1 e Informática II)

Definición: Un proceso es recursivo si forma parte de si mismo, esto es si se define en función de si mismo.

Ambito de Aplicación:

General

Problemas cuya solución se puede hallar solucionando el mismo problema pero con un caso de menor tamaño.

Razones de uso:

- Problemas “casi” irresolubles con las estructuras iterativas.
- Soluciones elegantes.
- Soluciones más simples.

Ventajas de la Recursión ya conocidas

- Soluciones simples, claras.
- Soluciones elegantes.
- Soluciones a problemas complejos.

Desventajas de la Recursión: INEFICIENCIA

–Sobrecarga asociada con las llamadas a subalgoritmos.

- Una simple llamada puede generar un gran numero de llamadas recursivas. ($\text{Fact}(n)$ genera n llamadas recursivas).
Peligro de desbordar el stack del micro!!.. (Los primeros virus informáticos hacían esto). La recursividad usa mucha memoria RAM.
- ¿La claridad compensa la sobrecarga?: En muchas situaciones NO!!

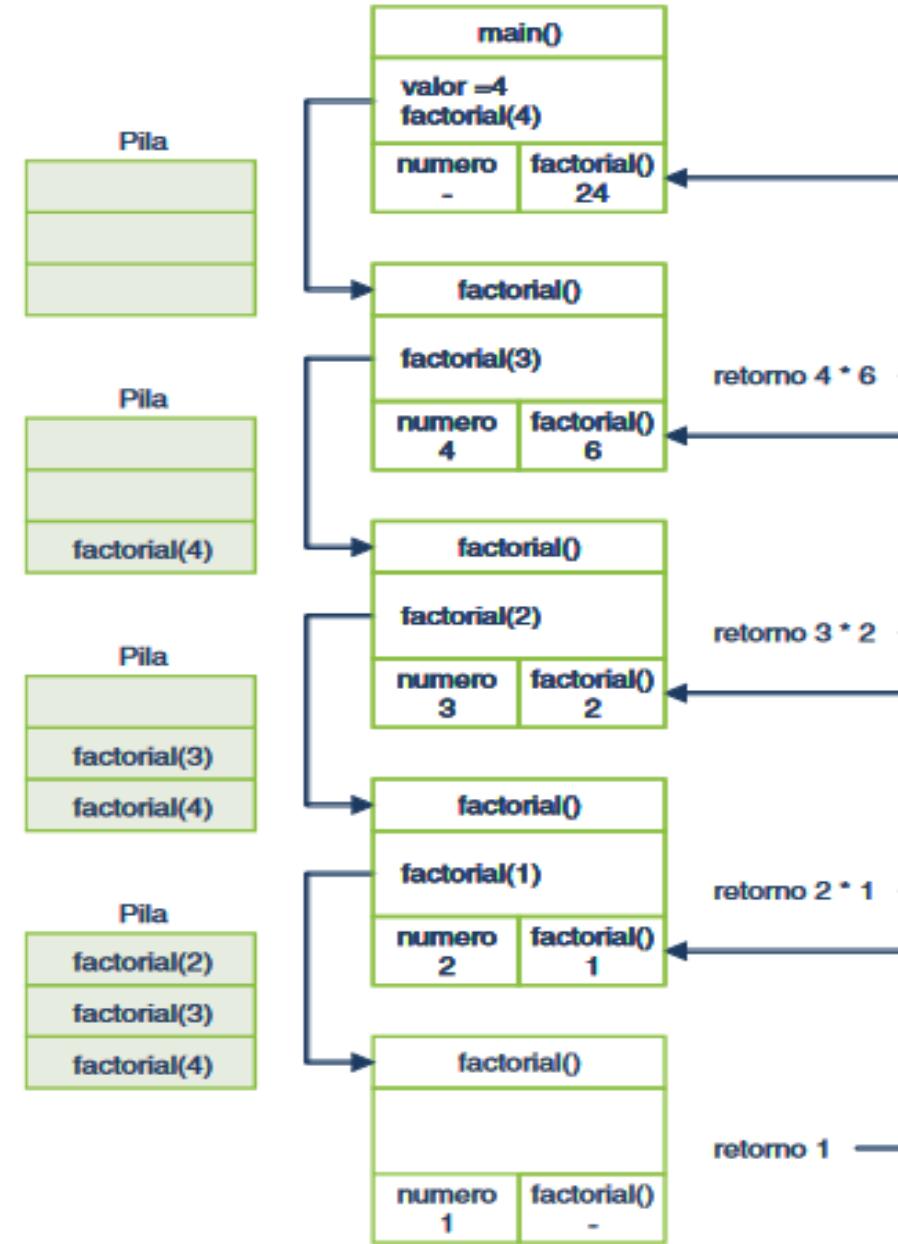
El valor de la recursividad reside en el hecho de que se puede usar para resolver problemas sin fácil solución iterativa. Luego se preferirá soluciones no recursivas, de haberlas.

–La ineficiencia inherente de algunos algoritmos recursivos. Son en general muy muy lentos.

LA RECURSIVIDAD SE DEBE USAR CUANDO SEA REALMENTE NECESARIA, ES DECIR, CUANDO NO EXISTA UNA SOLUCIÓN ITERATIVA SIMPLE O ESTA NO EXISTE..

La función factorial en forma recursiva

```
long factorial(short x)
{
    if (x<2)
        return 1;
    else
        return x*factorial(x-1);
}
```



Vamos a medir el tiempo que tarda en hacer el calculo de cual es el elemento de la posición n de la serie de Fibonacci.

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

time_t inicial, final;

long int fibonacci_norecu (long int n)
{
    int actual, ant1, ant2,i;
    ant1 = ant2 = 1;
    if ((n == 0) || (n == 1))
        actual = 1;
    else
        for (i=2; i<=n; i++)
    {
        actual = ant1 + ant2;
        ant2 = ant1;
        ant1 = actual;
    }
    return actual;
}

long int fibonacci_recu (long int n)
{
    if ((n == 0) || (n == 1))
        return 1;
    else
        return fibonacci_recu(n-1) + fibonacci_recu(n-2);
}

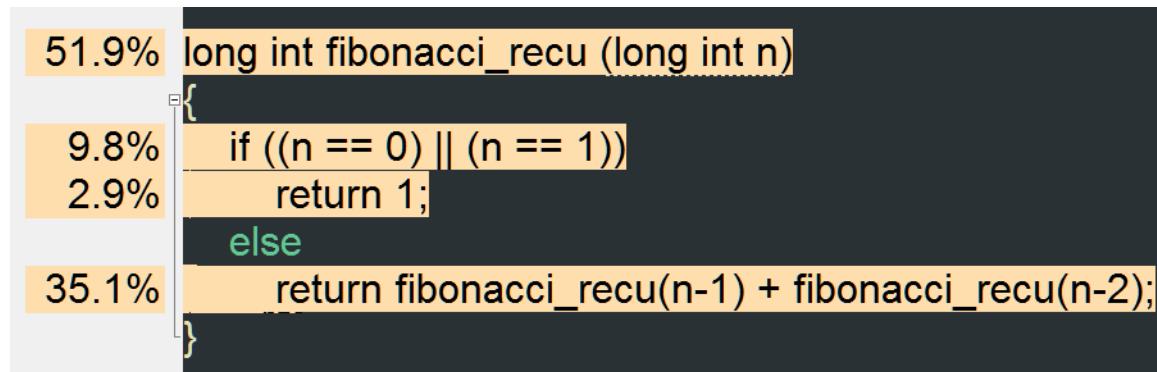
int main()
{
    double secs;
    long int posicion=0;

    for(posicion=20; posicion<=40; posicion+=1)
    {
        inicial=clock();
        printf("%ld= %ld \n", posicion,fibonacci_recu(posicion));
        //printf("%ld= %ld \n", posicion,fibonacci_norecu(posicion));
        final=clock();

        secs = (double)(final- inicial) / CLOCKS_PER_SEC;
        printf("en %.16g segundos\n", secs);
    }
    return 0;
}
```

En esta imagen vemos que el 99.7% del tiempo total de ejecución se lo lleva nuestra función Fibonacci recursiva.

Se probó llamándola con valores entre 10 y 45.



Se ve en la ejecución que a medida que se la llama con un n más grande el tiempo de ejecución crece.

Ejemplo de funciones recursivas para el calculo del factorial y de x^y

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

long factorial(long x)
{
    if (x<2)
        return 1;
    else
        return x*factorial(x-1);
}

long potencia( long base, long exponente)
{
    if(exponente == 0)
    {
        return 1;
    }
    else
    {
        return base * potencia(base, exponente -1); // llamada recursiva
    }
}

int main()
{
    long numero, exponente;

    printf("Ingrese un numero entero: ");
    scanf("%d",&numero);

    printf("\nIngrese un exponente: ");
    scanf("%d",&exponente);

    printf("%d\n",factorial(numero));
    printf("%d\n",potencia(numero,exponente));

    return 0;
}
```

Entero a binario por recursión: se genera una tabla del 0 al 32

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

void DecimalAbinario(short N)
{
    if(N>1)
    {
        DecimalAbinario(N / 2);
        printf("%hd",N%2);
    }
    else
        printf("%hd",N);
}

int main()
{
    short N;
    for(N=0;N<=32;N++)
    {
        DecimalAbinario(N);
        printf("\n");
    }
    return 0;
}
```

Calculo del resultado de hacer i^n , siendo i la unidad imaginaria.

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

void complejo(short potencia)
{
    switch(potencia)
    {
        case 0:printf("1");return;
        case 1:printf("i");return;
        case 2:printf("-1");return;
        case 3:printf("-i");return;
        default: complejo(potencia-4);
    }
}

int main()
{
    complejo(8);

    return 0;
}
```

Cálculo de máximo común divisor entre dos números con dos métodos recursivos diferentes.

```
#include<stdio.h>

short MCD1(short x, short y);
short MCD2(short x, short y);

int main()
{
    short num1=0;
    short num2=0;

    printf("MAXIMO COMUN DIVISOR\n");

    printf("Primer numero: ");
    scanf("%hd",&num1);

    printf("Segundo numero: ");
    scanf("%hd",&num2);

    printf("\nEl resultado es con MCD1: %hd\n", MCD1(num1, num2));
    printf("\nEl resultado es con MCD2: %hd\n", MCD2(num1, num2));

    return 0;
}
```

```
short MCD1(short x, short y)
{
    if(y==0)
        return x;
    MCD1(y, x%y);
}

short MCD2(short x, short y)
{
    short AUX;
    if(y>x)
    {
        AUX=y;
        y=x;
        x=AUX;
    }
    if(!y)
        return x;
    MCD2(y,x-y);
}
```

Vectores (Arrays)

(tema fundamental en programación!!)

Definición:

Un vector o array es un tipo de dato que permite almacenar información del mismo tipo (short, long, float, struct, etc) en forma contigua en memoria de solo lectura (en ROM será un vector de constantes) o en memoria de lectura-escritura (en la llamada RAM o RWM (read write memory) lo que hará que sea un vector de variables. Como comentario en RAM también podremos crear vectores de constantes.

Es un **tipo de dato indexado**. Para acceder a un dato almacenado necesitaremos un índice como ya veremos.

Nosotros veremos en este capítulo **vectores estáticos** o también llamados compactos. Se pueden crear tambien **vectores dinámicos** de una forma distinta a la que ahora veremos.

Definimos un vector como estático a aquel que se define su tamaño en tiempo de compilación sin poderlo variar (aumentar o disminuir) o redefinirlo en tiempo de ejecución. (muy importante)

Tampoco se puede crear en tiempo de ejecución ni liberar el espacio de memoria utilizado para el mismo.

Al vector que se puede crear y/o modificar en tiempo de ejecución, incluso liberar su espacio se denomina **dinámico**.

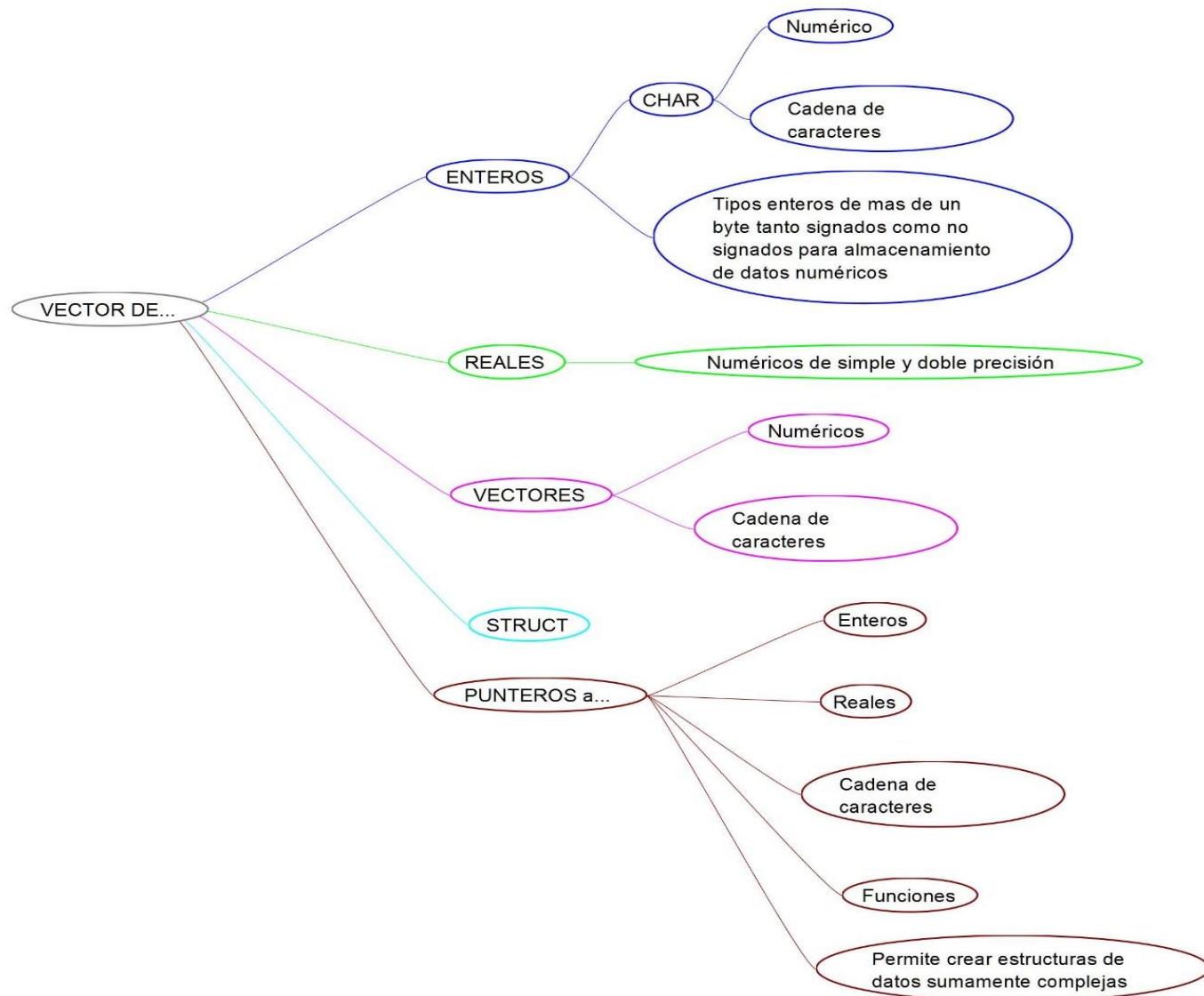
Podemos ver en la figura, una captura de una planilla de cálculo “tipo Excel” en la que se ha completado una columna A entre los índices 1 y 10 con valores enteros. Podemos decir que un vector es, en primera aproximación al tema, una columna de dicha planilla. Veremos que en lenguaje C, **SIEMPRE EL PRIMER INDICE ES CERO (0) y no uno (1) como en la figura**.

En C el vector se crearía como **short A[10]**, siendo diez (10) la mayor cantidad de datos que se puede almacenar. **SIEMPRE DEBEMOS RESERVAR CUANTOS DATOS VAMOS A ALMACENAR. SI RESERVAMOS PARA DIEZ DATOS NO PODEMOS, NI DEBEMOS, ESCRIBIR NI LEER MAS DE DIEZ DATOS DEL VECTOR. ESCRIBIR MAS DATOS QUE LOS ESPACIOS RESERVADOS LLEVA A LO QUE SE DENOMINA DESBORDAMIENTO DEL MISMO, QUE JAMAS DEBE SUCEDER. ES ALGO MUY MUY GRAVE DESBORDAR UN VECTOR!!!!**



	A	B	C	D	E	F
1	2					
2	9					
3	7					
4	-445					
5	-20					
6	1					
7	4					
8	-10					
9	30					
10	-32					
11						
12						
13						
14						
15						
16						

¿De que tipo pueden ser los vectores(los más comunes...)?



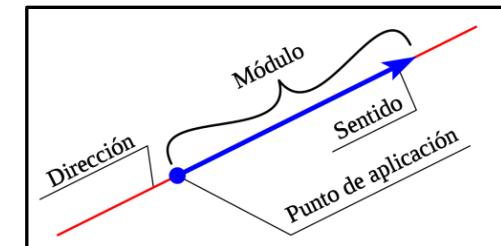
¿Para qué me sirven los vectores?

En **matemática**, un vector es un segmento de una recta que tiene como características tener una dirección (recta de aplicación), un sentido y un módulo. Se usa mucho en ciencia y tecnología.

Tambien en **matemática**, una fila o una columna de una matriz puede ser considerada como un vector, esto es un vector fila o un vector columna.

Ej. Vemos una matriz de 4×4 con datos enteros.

$$\begin{vmatrix} 1 & 8 & -4 & -56 \\ 9 & -1 & 2 & 20 \\ -6 & 16 & 11 & 29 \\ 3 & 14 & 91 & -16 \end{vmatrix}$$



De la matriz por podemos obtener el vector fila: $\begin{bmatrix} -6 & 16 & 11 & 29 \end{bmatrix}$

Tambien de la matriz podemos obtener el vector columna:

$$\begin{bmatrix} -4 \\ 2 \\ 11 \\ 91 \end{bmatrix}$$

En C veremos que no hay diferencia a nivel de código para definir un vector fila o un vector columna. Se almacenan y crean igual. Es la interpretación de quien los use lo que los hacen vector fila o vector columna. Ej **short A[4]** puede ser vector fila o columna según interpretación humana.

Un vector en programación sirve para almacenar información en forma continua. (por ejemplo datos todos short, long, float, etc).

Dichos datos, una vez almacenados, pueden por ejemplo ser ordenados de mayor a menor o puedo buscar un dato en el mismo, o usarlos en un cálculo matemático o calcular el promedio de los mismos, buscar cuales son pares o múltiplos de 3, etc.

La idea, en general, es ingresar la información una única vez y después usarla todas las veces que sea necesario.

Un buffer también es un vector, en general de posiciones de 1 byte (char). Sirve para almacenar información temporal. Muchas veces vemos una película desde algún sitio de internet. Partes de dicha película se van almacenando en memoria para que no existan saltos en la imagen.

Existen muchos ejemplos de usos de vectores. Obviamente para resolver cálculos matemáticos son fundamentales.

Tambien veremos que una cadena de caracteres (string) es un vector de char que incluyen un carácter especial llamado NULL. En C no existe el tipo de dato string.

Tambien podemos manejar matrices de 2 o mas dimensiones que almacenen cualquier tipo de dato.

En muchas situaciones existe una relación muy estrecha entre punteros y vectores.

VAMOS A VER EN DETALLE LOS VECTORES...

Si tenemos:

```
short vector[8]={2,16,-4,29,234,12,0,3};
```

lo podemos ver así:

Dirección	1500	1502	1504	1506	1508	1510	1512	1514
	2	16	-4	29	234	12	0	3
Contenido	vector[0]	vector[1]	vector[2]	vector[3]	vector[4]	vector[5]	vector[6]	vector[7]

Podemos decir lo siguiente de este vector que hemos definido:

- Es de short, entero de 2 bytes signado.
- Se almacena en forma contigua. NO PUEDE HABER ESPACIO INTERMEDIO ENTRE ELEMENTOS.
- Cada dato ocupa 2 bytes. (SI ES DE SHORT!!)
- No hay datos distintos en este caso a un short, ni lo puede haber. TODOS DEL MISMO TIPO.
- Los índices van de 0(cero) SIEMPRE, a (cantidad -1), en este caso hasta el índice 7 ya que puedo almacenar hasta 8 enteros de tipo short.
- EL NOMBRE DEL VECTOR, EN ESTE CASO VECTOR, CONTIENE LA DIRECCION INICIAL EN MEMORIA DE DICHO VECTOR, EN ESTE CASO 1500. Es la misma dirección que el elemento vector[0]. **CONCEPTO MUY MUY IMPORTANTE QUE NO DEBEMOS NUNCA OLVIDAR!!!**
- No puedo acceder a un elemento del vector, para leerlo o escribirlo sin usar índice. Siempre debo usar índice, para leer o escribir un valor.

- Si hago `sizeof(vector)` me da 16 (bytes). Esto es $8*sizeof(short)$, ya que `sizeof(short)` es el tamaño que ocupa un único dato de tipo short en memoria, o sea 2bytes.
- Si hago `sizeof(vector[0])` me da la cantidad de memoria utilizada por dicho elemento y es lo mismo que hacer `sizeof(short)`.
- **NUNCA DEBO ESCRIBIR MAS CANTIDAD DE DATOS QUE LOS RESERVADOS, NUNCA...** Se pueden producir grandes problemas, desde perdidas de datos hasta cuelgues y reinicios del sistema.
- Tampoco debo leer datos desde “posiciones” inexistentes, si reservé para 8 datos leo los índices de 0 a 7, el índice 8 y mayores, NO EXISTEN, por mas que se me permita hacerlo. **LEO INFORMACION FALSA SI LEO INDICES MAYORES AL MAXIMO PERMITIDO.**
- **ES RESPONSABILIDAD DEL PROGRAMADOR NO ESCRIBIR, NI LEER ZONAS DE MEMORIA NO RESERVADAS. ES RESPONSABILIDAD DEL PROGRAMADOR VERIFICAR LOS LIMITES DE USO DE CUALQUIER VARIABLE.** PIENSE SI ESTA PROGRAMANDO UN SISTEMA DE FRENO AUTOMATICO DE UN TREN, UN EQUIPO MEDICO, UN CONTROL DE UN ROBOT DE SOLDADO EN UNA FABRICA, EL SALIR FUERA DE RANGO PUEDE CAUSAR PROBLEMAS Y ACCIDENTES SUMAMENTE GRAVES DEL CUAL USTED SERÁ RESPONSABLE COMO PROFESIONAL.

¿Como hace el compilador para encontrar cada uno de los elementos del vector?

Debemos siempre recordar que en el nombre de nuestro vector se “guarda” la dirección inicial del mismo. Si nuestro vector se define como short X[10], el primer elemento de dicho vector estará “apuntado” por X. **X es un puntero constante** a la primera dirección de nuestro vector y por ser constante no se podrá modificar su contenido.

Considerando esta información cada elemento se encuentra aplicando la siguiente fórmula:

Dirección_del_elemento = dirección_inicial + indice x sizeof(tipo_de_dato)

Los vectores podrán ser de short, char, long, estructuras, etc ademas de poder ser de más de una dimensión, tal como las matrices.

Dirección	1500	1502	1504	1506	1508	1510	1512	1514
	2	16	-4	29	234	12	0	3
Contenido	vector[0]	vector[1]	vector[2]	vector[3]	vector[4]	vector[5]	vector[6]	vector[7]

Ejemplos de uso de vectores (primeros pasos...)

Ejemplo 1:

1. short X[5];
- 2.
3. X[1]= 25;
4. printf("Lo ingresado en X[1] fue: %hd\n",X[1]);
5. printf("Ingrese un valor para X[3]: ");
6. scanf("%hd",&X[3]);
7. printf("\nLo ingresado en X[3] fue: %hd\n",X[3]);

En la linea 1 vemos que **hemos reservado espacio para almacenar en forma contigua en memoria 5 elementos de tipo short**. Por lo tanto si hacemos sizeof(X) obtendremos como resultado 10 bytes.

En la linea 3 vemos como al vector X en el indice 1 le asignamos el valor 25 que lo imprimimos en la linea 4.

En la 6 vemos que para leer un valor desde teclado y asignarlo a la posición 3 debemos usar &X[3].
En la linea 7 lo imprimimos en pantalla.

Nota: Cuando programamos los números que están encolumnados a la izquierda, 1, 2, 3, etc no los ponemos. En este caso se usan para numerar la línea para poder identificarla.

Modificaremos el ejercicio anterior para ver otras formas de definir un vector estático.

Ejemplo 2:

1. **short X[]={2, 9, 6, 3, 5};**
2. **unsigned short indice;**
- 3.
4. **unsigned short tamano=sizeof(X)/sizeof(short);**
- 5.
6. **for(indice=0;indice<tamano;indice++)**
7. **printf("%hd ",X[indice]);**

En la linea 1 ahora reservamos memoria sin indicar en los [] un valor. Esto solo lo podemos hacer cuando decimos que elementos iniciales va a contener el vector como lo hemos hecho con {2, 9, 6, 3, 5}. Es decir la posicion de indice cero va a contener 2, la de indice 1 va a contener 9 y asi sucesivamente. Es decir nuestro vector es de tamaño 5.

Por ser de tamaño 5, los indices van de 0 a 4 !! Nunca puede ser negativo ni mayor a la cantidad menos uno. El compilador no verifica que al ejecutarse el indice del vector no supere la cantidad máxima de memoria reservada. Es responsabilidad plena y absoluta del programador impedir que el rango del indice supere los valores permitidos. Otra cosa que no hay que perder de vista es que el indice cero siempre existe.

La linea 4 obtiene por medio de la formula la cantidad de elementos que posee el vector.

Ejemplo 3

```
#include <stdio.h>
#include <stdlib.h>
#define tamano 10
```

Constante simbólica. Donde se use tamano el valor numérico será en este caso 10.

```
int main(void)
{
    short X[tamano]={2,9,6,3,5};
    unsigned short indice;

    for(indice=0;indice<tamano;indice++)
        printf("%hd ",X[indice]);

    return 0;
}
```

Se crea el espacio de, en este caso, de 20 bytes contiguos en memoria RAM, con el objetivo de almacenar 10 datos de tipo short. Desde X[0] a X[4] se los inicializa con los valores que figuran. El resto, de X[5] a X[9] tienen el valor entero cero (0) automáticamente.

X[índice] es el valor numérico almacenado en el vector X en la posición índice. Índice va de 0 a 9 en este caso, esto es de 0 a (tamano - 1).

Algoritmos básicos con vectores

Ejemplo 1: Realice un programa que cargue un vector de enteros y luego calcule e imprima el promedio de los datos almacenados.

```
#include <stdio.h>
#define tamano 5

int main(void)
{
    short vector[tamano];
    short indice;
    short suma=0;
    float promedio;

    /*Almaceno 5 valores enteros en el vector*/
    for(indice=0;indice<tamano;indice++)
    {
        printf("\nIngrese un valor entero: ");
        scanf("%hd",&vector[indice]);
    }
    /*fin carga del vector*/

    /*calculo de la sumatoria*/
    for(indice=0;indice<tamano;indice++)
    {
        suma=suma+vector[indice];
    }
    /*fin calculo sumatoria*/

    promedio=(float)suma/indice;
    printf("\n\nEl promedio es: %f",promedio);

    return 0;
}
```

#define tamano 5 define una **CONSTANTE SIMBOLICA**, en este caso **tamano**, con un valor 5. Recuerde que para los nombres de constantes y variables se usa el alfabeto inglés. En cada lugar donde figura **tamano** se deberá leer, para este caso un valor 5. Cambiando el valor 5 en el **#define** por otro, dicho valor cambia en todos los lugares donde figura **tamano**.

Creamos un vector que tiene como **cantidad de elementos enteros de tipo short el valor de la constante tamano**; en este caso **reservamos espacio para 5 datos short**, esto es, 10 bytes contiguos.

Usamos un lazo, en este caso **for**, pero pudo haber sido también un **while**, tanto para escribir desde el índice cero del vector hasta el índice **tamano -1** como también para leer cada una de dichas posiciones. En este caso escribimos y leemos desde la primera hasta la última posición reservada. NO es obligatorio hacer esto, yo puedo escribir y leer los datos del vector en cualquier orden y no estoy obligado a escribir todas las posiciones reservadas. Puedo reservar para 100 datos pero usar 20, 5, 40, 80, 100 etc. Lo que nunca puedo hacer es usar mas de lo que he reservado. Si reservé para 100 datos, existen los índices del 0 al 99, pero el 100 o mayores no.

Observe el **cambio** en el cálculo del promedio. Se cambia porque la división se hace entre valores enteros y el resultado no es un entero.

Ejemplo 2: Realice un programa que cargue un vector de enteros e imprima el valor máximo y el mínimo almacenado en el mismo. Vea que se toma SIEMPRE EL PRIMER VALOR INGRESADO O EL PRIMER DATO DEL VECTOR COMO VALOR INICIAL PARA REALIZAR LA COMPARACIONES. NO HAY QUE INVENTAR VALORES INICIALES PORQUE SE PUEDEN PRODUCIR ERRORES GRAVES EN LA BUSQUEDA DEL MÁXIMO Y EL MÍNIMO (es muy importante tomar siempre en cuenta esto).

```
#include <stdio.h>
#define tamano 5 //La cantidad del elementos del vector

int main(void)
{
    short vector[tamano];
    short indice;
    short maximo;
    short minimo;

    /*Almaceno 5 valores enteros en el vector*/
    for(indice=0;indice<tamano;indice++)
    {
        printf("\nIngrese un valor entero: ");
        scanf("%hd",&vector[indice]);
    }
    /*fin carga del vector*/

    /*calculo del maximo y el minimo*/
    for(indice=0;indice<tamano;indice++)
    {
        if(indice==0)/*El primero de la lista*/
        {
            maximo=vector[indice];
            minimo=vector[indice];
        }

        if(maximo<vector[indice])//Calculo del maximo
            maximo=vector[indice];

        if(minimo>vector[indice])//Calculo del minimo
            minimo=vector[indice];
    }
    /*fin calculo de maximo y minimo*/

    printf("\n\nEl minimo es: %hd",minimo);
    printf("\nEl maximo es: %hd",maximo);

    return 0;
}
```

Otra forma de calcular el máximo y el mínimo:

Una forma	Otra forma
<pre>if(indice==0)/*El primero de la lista*/ { maximo=vector[indice]; minimo=vector[indice]; } if(maximo<vector[indice]) //Calculo del maximo maximo=vector[indice]; if(minimo>vector[indice]) //Calculo del minimo minimo=vector[indice];</pre>	<pre>if(índice == 0 vector[índice] > maximo) { maximo=vector[índice]; } if(índice == 0 vector[índice] < minimo) { minimo=vector[índice]; }</pre>

En las dos formas se toma el primer dato (el de índice cero) como máximo y mínimo para comenzar a comparar. El primero, por ser en ese momento el único dato es el máximo y el mínimo, y eso está bien. Los demás valores leídos los reemplazarán o no. No hay que “inventar” valores iniciales para máximo y mínimo como ya se ha dicho. Puede llevar a grandes y graves errores.

Ejemplo 3: Realice un programa que obtenga: cuantos pares, cuantos impares tiene almacenados un vector y además obtenga el porcentaje de valores pares respecto del total.

#include <stdio.h> #define tamano 5 int main(void) { short vector[tamano]; short indice; short cuentapar=0; short cuentaimpar=0; float porcentaje_pares; for(indice=0;indice<tamano; indice++) { printf("\nIngrese un valor entero: "); scanf("%hd",&vector[indice]); } }	for(indice=0;indice<tamano;indice++) { if(vector[indice]%2==0) cuentapar++; else cuentaimpar++; } porcentaje_pares=(float)cuentapar/(cuentaimpar+cuentapar); printf("\nLa cantidad de pares es: %hd",cuentapar); printf("\nLa cantidad de impares es %hd",cuentaimpar); printf("\nEl porcentaje de pares es: %.2f %%",porcentaje_pares*100); return 0;
---	---

Búsquedas y Ordenamientos

Búsqueda secuencial en vectores

Busqueda secuencial es la mas simple e intuitiva de todas. Aquí vemos una forma de buscar en un vector recorriéndolo totalmente. NO se necesita tener el vector ordenado. Esto lo distingue de la búsqueda binaria.

Si el vector no está ordenado, la búsqueda secuencial es la única forma de realizar la búsqueda. Cuando buscamos lo que hacemos es ver si existe lo buscado y en caso de existir, donde se encuentra (índice del vector)

```
printf("Que valor esta buscando?: ");
scanf("%hd",&buscado);

indice=0;

while(buscado!=vector[indice] && indice < cantidad)
{
    indice++;
}

if(vector[inice]== buscado)
    printf("\nEncontrado en &hd\n",indice);
else
    printf("\nNo encontrado\n");
```

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main(void)
{
    short vector[]={2,4,5,8,1,-2,-4,16,6};
    short cantidad=sizeof(vector)/sizeof(short);
    short buscado;
    short indice=0;

    do{
        printf("Que valor esta buscando?: ");
        scanf("%hd",&buscado);

        while(buscado!=vector[indice] && indice <cantidad)
        {
            indice++;
        }
        if(vector[indice]==buscado)
            printf("\nEncontrado en %hd\n",indice);
        else
            printf("\nNo encontrado\n");

        printf("Si quiere buscar otro valor presione s: \n");
    }while(getch()=='s');

    return 0;
}

```

Ordenamientos de datos en vectores

Si lo que esta en el índice i es mayor a lo que esta en el índice j se intercambian.

Ordenamiento por pivot

9	2	6	-8	1
i	j			
2	9	6	-8	1
i		j		
2	9	6	-8	1
i			j	
-8	9	6	2	1
i				j
-8	9	6	2	1
	i	j		

Quedó ubicado,
no lo toco mas...

Cuando comienza
una nueva “pasada”
es siempre $j=i+1$

-8	6	9	2	1
i		j		
-8	2	9	6	1
i				j
-8	1	9	6	2
	i	j		
-8	1	6	9	2
i			j	
-8	1	2	9	6
	i		j	

Y nos queda finalmente:

-8	1	2	6	9
----	---	---	---	---

//Ordenamiento por metodo del pivot

```
for(i = 0; i < cantidad -1; i++)
    for(j = i+1; j<cantidad; j++)
        if(vector[i] > vector[j])
    {
        auxiliar = vector[i];
        vector[i] = vector[j];
        vector[j] = auxiliar;
    }
```

El intercambio de datos y la variable auxiliar.

Swapping in action



```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    short vector[]={2,-9,8,6,-22,-1,-4,11,-11,3,49,27},auxiliar;
    short i, j,u;
    short cantidad=sizeof(vector)/sizeof(short);
    printf("Metodo del pivot\n\n");
    printf("Cantidad de datos a ordenar: %hd\n",cantidad);
    printf("Cantidad de comparaciones esperadas: %hd\n\n",((cantidad*cantidad)-cantidad)/2);
    printf("Vector original \n");
    for(u=0;u<cantidad;u++)
        printf("%hd ",vector[u]);

    printf("\n\nVemos paso a paso el ordenamiento \n\n");
    for(i=0;i<cantidad -1;i++)
    {
        for(j=i+1;j<cantidad;j++)
        { if(vector[i]>vector[j])
            {
                auxiliar=vector[i];
                vector[i]=vector[j];
                vector[j]=auxiliar;
            }
        for(u=0;u<cantidad;u++)
            printf("%hd ",vector[u]);
        printf("\n");
    }
}
return 0;
}

```

Ordenamiento por burbuja

9	2	6	-8	1
i	i+1			
2	9	6	-8	1
	i	i+1		
2	6	9	-8	1
		i	i+1	
2	6	-8	9	1
			i	i+1
2	6	-8	1	9
i	i+1			

2	6	-8	1	9
	i	i+1		
2	-8	6	1	9
		i	i+1	
2	-8	1	6	9
i	i+1			
-8	2	1	6	9
	i	i+1		
-8	1	2	6	9
i	i+1			

Y nos queda finalmente:

-8	1	2	6	9
----	---	---	---	---

//Ordenamiento por metodo de burbuja

```
R=0;
do{
    R++;
    no_ordenado=0;
    for(i=0;i<cantidad-R;i++)
        if(vector[i]>vector[i+1])
        {
            auxiliar=vector[i];
            vector[i]=vector[i+1];
            vector[i+1]=auxiliar;
            no_ordenado=1;
        }
    }while(no_ordenado);
```

```

#include <stdio.h>

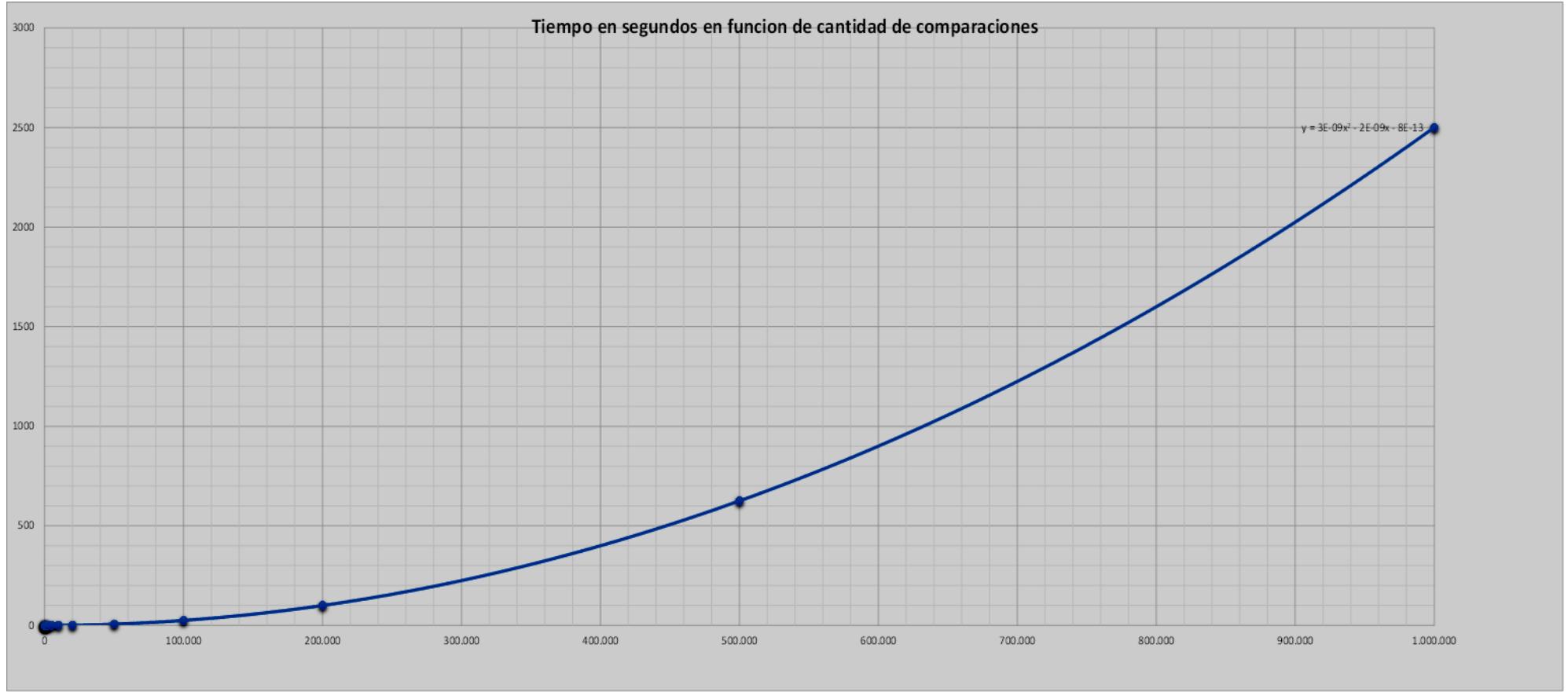
int main(void)
{
    short vector[]={10,9,8,7,6,5,4,3,2,1},auxiliar;
    //short vector[]={1,2,3,4,5,6,7,8,9,10},auxiliar;
    short i,u, no_ordenado=0,R=0;
    short cantidad=sizeof(vector)/sizeof(short);
    printf("Metodo de burbuja\n\n");
    printf("Cantidad de datos a ordenar: %hd\n",cantidad);
    printf("Cantidad de comparaciones esperadas maximas: %hd\n\n",((cantidad*cantidad)-cantidad)/2);
    printf("Vector original \n");
    for(u = 0 ;u < cantidad; u++)
        printf("%hd ",vector[u]);

    printf("\n\nVemos paso a paso el ordenamiento \n\n");
    do
    {
        R++;
        no_ordenado=0;
        for(i=0; i<cantidad-R; i++)
        {
            if(vector[i] > vector[i+1])
            {
                auxiliar = vector[i];
                vector[i] = vector[i+1];
                vector[i+1] = auxiliar;
                no_ordenado = 1;
            }
            for(u=0; u<cantidad; u++)
                printf("%hd ", vector[u]);
            printf("\n");
        }
    }while(no_ordenado);
    return 0;
}

```

Tabla comparativa de tiempos para ordenar un vector con el método del pivot y de burbuja (lectura optativa)

Datos a ordenar		Cantidad de comparaciones		Tiempo estimado de ordenamiento (considerando una computadora Intel Core i3 3217U @ 1.80GHz)					
En agosto de 2021	Cantidad de datos	Pivot	Burbuja peor caso	en segundos	en minutos	en horas	en días	en meses	en años
	5	10	10	0,00000005	8,33333E-10	1,38889E-11	5,78704E-13	1,92901E-14	1,60751E-15
	10	45	45	0,000000225	3,75E-09	6,25E-11	2,60417E-12	8,68056E-14	7,2338E-15
	20	190	190	0,00000095	1,58333E-08	2,63889E-10	1,09954E-11	3,66512E-13	3,05427E-14
	50	1225	1225	0,000006125	1,02083E-07	1,70139E-09	7,08912E-11	2,36304E-12	1,9692E-13
	100	4950	4950	0,00002475	4,125E-07	6,875E-09	2,86458E-10	9,54861E-12	7,95718E-13
	200	19900	19900	0,0000995	1,65833E-06	2,76389E-08	1,15162E-09	3,83873E-11	3,19895E-12
	500	124750	124750	0,00062375	1,03958E-05	1,73264E-07	7,21933E-09	2,40644E-10	2,00537E-11
	1.000	499500	499500	0,0024975	0,000041625	6,9375E-07	2,89063E-08	9,63542E-10	8,02951E-11
	2.000	1999000	1999000	0,009995	0,000166583	2,77639E-06	1,15683E-07	3,8561E-09	3,21341E-10
	5.000	12497500	12497500	0,0624875	0,001041458	1,73576E-05	7,23235E-07	2,41078E-08	2,00899E-09
	10.000	49995000	49995000	0,249975	0,00416625	6,94375E-05	2,89323E-06	9,6441E-08	8,03675E-09
	20.000	199990000	199990000	0,99995	0,016665833	0,000277764	1,15735E-05	3,85783E-07	3,21486E-08
	50.000	1249975000	1249975000	6,249875	0,104164583	0,001736076	7,23365E-05	2,41122E-06	2,00935E-07
	100.000	4999950000	4999950000	24,99975	0,4166625	0,006944375	0,000289349	9,64497E-06	8,03747E-07
	200.000	19999900000	19999900000	99,9995	1,666658333	0,027777639	0,001157402	3,85801E-05	3,215E-06
	500.000	1,25E+11	1,25E+11	624,99875	10,41664583	0,173610764	0,007233782	0,000241126	2,00938E-05
	1.000.000	5E+11	5E+11	2499,9975	41,666625	0,69444375	0,028935156	0,000964505	8,03754E-05
CABA (aprox)	2.800.000	3,92E+12	3,92E+12	19599,993	326,66655	5,4444425	0,226851771	0,007561726	0,000630144
	10.000.000	5E+13	5E+13	249999,975	41666,6625	69,4444375	2,893518229	0,096450608	0,008037551
Argentina	45.000.000	1,0125E+15	1,0125E+15	5062499,888	84374,99813	1406,249969	58,5937487	1,953124957	0,162760413
	100.000.000	5E+15	5E+15	24999999,75	416666,6625	6944,444375	289,351849	9,645061632	0,803755136
Japon	126.000.000	7,938E+15	7,938E+15	39689999,69	661499,9948	11024,99991	459,3749964	15,31249988	1,276041657
	1.000.000.000	5E+17	5E+17	2499999998	41666666,63	694444,4438	28935,18516	964,5061719	80,37551432
La India	1.395.000.000	9,73012E+17	9,73012E+17	4865062497	81084374,94	1351406,249	56308,59371	1876,953124	156,4127603
China	1.446.000.000	1,04546E+18	1,04546E+18	5227289996	87121499,94	1452024,999	60501,04162	2016,701387	168,058449
Poblacion mundial	7.900.000.000	3,1205E+19	3,1205E+19	1,56025E+11	2600416666	43340277,77	1805844,907	60194,83024	5016,235853



El tiempo promedio que surge de comparar el tiempo total con respecto a la cantidad de comparaciones en un Intel Core i3 3217U @ 1.80GHz, es de 1140 ps (pico segundos) o 1,14 ns (nano segundos). A pesar de esto en mi computadora no podría ordenar en una vida el padrón electoral de la India, que para 1446 millones tardaría unos 168 años. Estas comparaciones sirven para tener una forma objetiva de valorar los algoritmos que se usarán. **EL ANALISIS DEL PROBLEMA Y EL VALORAR DISTINTAS SOLUCIONES ES FUNDAMENTAL ANTES DE ESCRIBIR UN CÓDIGO.** Tenga en cuenta siempre esto.

Ordenamiento utilizando un vector de índices sin mover la información

```
#include<stdio.h>
#include<conio.h>
#define N 20

int main(void)
{
    int vector[N]={12,6,4,18,9,1,7,13,2,10,3,14,19,20,16,5,11,15,8,17};
    int indice[N];
    int i,j, aux;

    for(i=0;i<N;i++)
        indice[i]=i;

    for(i=0;i<N-1;i++)
        for(j=i+1;j<N;j++)

            if(vector[indice[i]]>vector[indice[j]])
            {
                aux=indice[i];
                indice[i]=indice[j];
                indice[j]=aux;
            }

    printf("Vector ordenado      Vector desordenado\n");
    for(i=0;i<N;i++)
        printf("%7d      %7d\n",vector[indice[i]],vector[i]);

    return 0;
}
```

Búsqueda binaria o dicotómica

Es uno de los mejores algoritmos que se conoce ya que su eficiencia es muy grande a tal punto que es el algoritmo de búsqueda mas utilizado en la práctica.

Para que podamos utilizar la búsqueda binaria, tendremos que tener ordenado nuestro vector (en forma ascendente o descendente), en función del dato a buscar.

Además, normalmente se aplica a datos sin repetición como por ejemplo buscar un DNI, un legajo, un código de producto, etc. Es el método más utilizado en la práctica, por su rapidez para saber donde está lo que estamos buscando (índice) o informar que no existe.

Tabla de eficiencia de los algoritmos de búsqueda en vectores: lo mensuramos en función de la cantidad de comparaciones.

Cantidad de datos	Cantidad de comparaciones	
	Secuencial	Binaria
5	5	3
10	10	4
20	20	5
50	50	6
100	100	7
200	200	8
500	500	9
1000	1000	10
2000	2000	11
5000	5000	13
10000	10000	14
100000	100000	17
1000000	1000000	20
10000000	10000000	24
100000000	100000000	27
1000000000	1000000000	30
10000000000	10000000000	34

Para mil millones de datos la búsqueda binaria solo necesita 30 comparaciones para saber si el dato buscado no está o si está, en donde. La diferencia es “abismal” entre la búsqueda binaria y la secuencial. El precio que se debe pagar es ordenar la información. Podriamos encontrar los datos de un votante de la India en segundos.

Búsqueda binaria o dicotómica

```
inferior=0;
superior=(sizeof(vector)/sizeof(short)) -1;
medio=superior/2;

printf("\nIngrese valor a buscar: ");
scanf("%hd",&buscado);

while(vector[medio]!=buscado && superior >=inferior)
{
    if(buscado > vector[medio])
        inferior=medio+1;
    else
        superior=medio-1;

    medio=(superior+inferior)/2;
}

if(buscado==vector[medio])
    printf("Encontrado en %hd \n",medio);
else
    printf("No existe \n");

printf("\npresione s para continuar: ");
```

```

//Busqueda binaria

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

int main(void)
{
    short vector[]={-5,-2,-1,2,4,5,6,9,12,15,22,25,45,56,67,102,140};
    short buscado;
    short superior;
    short inferior=0;
    short medio;

    do{
        superior=(sizeof(vector)/sizeof(short))-1;
        medio=superior/2;
        printf("\nIngrese valor a buscar: ");
        scanf("%hd",&buscado);
        while(vector[medio]!=buscado && superior >=inferior)
        {
            if(buscado > vector[medio])
                inferior=medio+1;
            else
                superior=medio-1;
            medio=(superior+inferior)/2;
        }
        if(buscado==vector[medio])
            printf("Encontrado en %hd \n",medio);
        else
            printf("No existe \n");
        printf("\nPresione s para continuar: ");

    }while(getch()=='s');

    return 0;
}

```

Algoritmo de búsqueda con inserción (muy importante conocerlo)

Imaginen que tienen un texto y tienen que armar una lista de las letras que aparecen en el mismo y cuantas, o tienen una carrera de autos que cada vez que la primera vez que se pasa por la meta se anota el tiempo de la vuelta pero la segunda vez se modifica dicho tiempo en función si es menor al que ya estaba. De nuevo, cada auto se asienta una sola vez en la lista. Es un algoritmo muy importante en programación y por eso lo vamos a ver con algunos ejemplos.

Ejemplo: Una forma de hacer el algoritmo. Tener en cuenta que se lo debe adaptar a cada situación particular. Cuando se tiene llena la lista termina.

```
#include <stdio.h>
#include <stdlib.h>
#define maximo 5

int main(void)
{
    short lista[maximo];
    short vacio=0, actual=0, haylugar=1;
    short ingreso;

    printf("Ingrese un entero signado distinto de cero: ");
    scanf("%hd",&ingreso);

    while(haylugar && ingreso!=0) /*Buscamos si esta*/
    {
        while(vacio!=actual && ingreso != lista[actual])/*Vemos si esta*/
            actual++;

        if(haylugar && actual==vacio) //si no esta, inserta.
        {
            lista[vacio]=ingreso;
            vacio++;
            if(vacio==maximo)
            {
                haylugar=0; /*no se permite agregar mas valores*/
                printf("\nSe completo la lista \n");
            }
        }
        else
            printf(..Ya estaba el %hd\n",ingreso);
    }

    if(haylugar)
    {
        printf("Ingrese un entero signado distinto de cero: ");
        scanf("%hd",&ingreso);
        actual=0;
    }
    printf("\n\nLa lista quedo con...\n");
    for(actual=0; actual<vacio; actual++)
        printf("%hd\n",lista[actual]);
}

return 0;
}
```

Vector de estructuras

Veamos esta porción de código:

Porción de código con la estructura y la creación del vector en main()				Como se almacena el vector creado en memoria																																																									
<pre>#define cantidad 4 struct datos{ short entero; float real; }; int main(void) { struct datos valores[cantidad]; }</pre>				<table border="1"> <thead> <tr> <th>Name</th><th>Value</th><th>Type</th><th>Address</th></tr> </thead> <tbody> <tr> <td>valores</td><td>[...]</td><td>struct datos[4]</td><td>0000000000022FA70</td></tr> <tr> <td>[0]</td><td>{...}</td><td>struct datos</td><td>0000000000022FA70</td></tr> <tr> <td>entero</td><td>0000</td><td>short</td><td>0000000000022FA70</td></tr> <tr> <td>real</td><td>0.00000</td><td>float</td><td>0000000000022FA74</td></tr> <tr> <td>[1]</td><td>{...}</td><td>struct datos</td><td>0000000000022FA78</td></tr> <tr> <td>entero</td><td>0000</td><td>short</td><td>0000000000022FA78</td></tr> <tr> <td>real</td><td>0.00000</td><td>float</td><td>0000000000022FA7C</td></tr> <tr> <td>[2]</td><td>{...}</td><td>struct datos</td><td>0000000000022FA80</td></tr> <tr> <td>entero</td><td>0000</td><td>short</td><td>0000000000022FA80</td></tr> <tr> <td>real</td><td>0.00000</td><td>float</td><td>0000000000022FA84</td></tr> <tr> <td>[3]</td><td>{...}</td><td>struct datos</td><td>0000000000022FA88</td></tr> <tr> <td>entero</td><td>0000</td><td>short</td><td>0000000000022FA88</td></tr> <tr> <td>real</td><td>0.00000</td><td>float</td><td>0000000000022FA8C</td></tr> </tbody> </table>		Name	Value	Type	Address	valores	[...]	struct datos[4]	0000000000022FA70	[0]	{...}	struct datos	0000000000022FA70	entero	0000	short	0000000000022FA70	real	0.00000	float	0000000000022FA74	[1]	{...}	struct datos	0000000000022FA78	entero	0000	short	0000000000022FA78	real	0.00000	float	0000000000022FA7C	[2]	{...}	struct datos	0000000000022FA80	entero	0000	short	0000000000022FA80	real	0.00000	float	0000000000022FA84	[3]	{...}	struct datos	0000000000022FA88	entero	0000	short	0000000000022FA88	real	0.00000	float	0000000000022FA8C
Name	Value	Type	Address																																																										
valores	[...]	struct datos[4]	0000000000022FA70																																																										
[0]	{...}	struct datos	0000000000022FA70																																																										
entero	0000	short	0000000000022FA70																																																										
real	0.00000	float	0000000000022FA74																																																										
[1]	{...}	struct datos	0000000000022FA78																																																										
entero	0000	short	0000000000022FA78																																																										
real	0.00000	float	0000000000022FA7C																																																										
[2]	{...}	struct datos	0000000000022FA80																																																										
entero	0000	short	0000000000022FA80																																																										
real	0.00000	float	0000000000022FA84																																																										
[3]	{...}	struct datos	0000000000022FA88																																																										
entero	0000	short	0000000000022FA88																																																										
real	0.00000	float	0000000000022FA8C																																																										

Vemos que cada estructura se almacena una detrás de la otra sin espacios en el medio. Observe las direcciones en hexadecimal y en cuanto difieren unas de las otras. Relacione con los tipos de datos. Observe que el "salto" entre el campo entero al campo flotante es 4 bytes en lugar de 2 bytes (mire el gráfico de la columna derecha). Esto pasa porque el compilador optimiza el acceso a la información. Si en lugar de short entero hubiésemos usado long entero, el espacio ocupado hubiese sido el mismo (para este caso; en otros hay que verificarlo usando sizeof). Si ocupan lo mismo con long ganamos mas rango de valores para almacenar...

Ejemplo completo del uso de un vector de estructuras en C

```
#include <stdio.h>
#define cantidad 5

struct datos{
    short entero;
    float real;
};

int main(void)
{
    struct datos valores[cantidad];
    short indice=0;

    for(indice=0;indice<cantidad;indice++)
    {
        printf("Ingrese un entero: ");
        scanf("%hd",&valores[indice].entero);
        printf("Ingrese un real: ");
        scanf("%f",&valores[indice].real);
    }

    printf("\n\nImprimimos lo ingresado...\n\n");

    for(indice=0;indice<cantidad;indice++)
    {
        printf("Entero: %hd\n",valores[indice].entero);
        printf("Real: %f\n\n",valores[indice].real);
    }
    return 0;
}
```

Realice un programa que, sin saberse cuantos datos se van a ingresar, se genere un vector de estructuras el que contendrá, para cada madera procesada, color, largo, ancho y superficie. Luego de ingresada dicha información, se deberá imprimir la superficie de cada una de dichas maderas. Los colores válidos son rojo, verde o azul. No habrá mas de 100 maderas procesadas.

```
#include <stdio.h>
#include <stdlib.h>
#define tamano 100

struct madera{
    char color;
    float largo;
    float ancho;
    float superficie;
};

int main(void)
{
    struct madera vector[tamano];
    short indice=0;
    char auxi_color;
    short cantidad;
    fflush(stdin);

    printf("\nIngrese el color de la madera: Rojo, Verde, Azul: ");
    auxi_color=toupper(getchar());

    while((auxi_color=='R'||auxi_color=='V'||auxi_color=='A')&&indice<tamano)
    {
        vector[indice].color=auxi_color;

        printf("\nIngrese el largo de la madera: ");
        scanf("%f",&vector[indice].largo);
        printf("\nIngrese el ancho de la madera: ");
        scanf("%f",&vector[indice].ancho);

        indice++;

        if(indice<tamano)
        {
            fflush(stdin);
            printf("\nIngrese el color de la madera: Rojo, Verde, Azul: ");
            auxi_color=toupper(getchar());
        }
    }

    cantidad=indice;//guardo la cantidad de datos almacenados
    for(indice=0;indice<cantidad;indice++)//Calculo de las superficies
    {
        vector[indice].superficie=vector[indice].largo*vector[indice].ancho;
    }

    for(indice=0;indice<cantidad;indice++)//Imprimo las superficies
    {
        printf("\nSuperficie= %f",vector[indice].superficie);
    }

    return 0;
}
```

Strings

(vector de caracteres + NULL)

Definición de string (Cadena de caracteres)

En informática cuando decimos **string** en general estamos hablando de textos o alfanuméricos. Cuando se dice texto se esta referenciando a lo que conocemos como tal en nuestra vida cotidiana. Lo escrito en este apunte es un texto (sin considerar graficos ni fotos).

Un alfanumérico es la combinacion de letras y números: Ej. "Uas2762", "AAb 339 AAc", etc. En este caso los números no son números, sino el “dibujo del simbolo” del número. Todo lo que haremos esta relacionado con la “tabla ASCII”. En la tabla ASCII el “símbolo de los números” están desde la posición 48 que contiene el dibujo del cero a la 57 que contiene el dibujo del nueve. Debo convertirlo a numero para poder usarlo para una operación matemática. Como comentario para comprender mejor esto, en la posición 65 de la tabla ASCII se encuentra el “dibujo” de la A (mayúscula). En otros idiomas en esa posición en la tabla podrá existir otro símbolo distinto. Esto queda muy bien reflejado en el UNICODE de idiomas orientales (chino, japonés, etc). El UNICODE en un futuro no muy lejano reemplazará al ASCII. De hecho hoy lo usamos en forma transparente en Windows y otros sistemas operativos.

Hay que tener muy en cuenta que es el humano o una “maquina” quien interpreta los símbolos y quien le da interpretación y sentido a ese dibujo.

Existen programas denominados genéricamente OCR (Optical Character Recognition), esto es “reconocimiento óptico de caracteres”. Convirten estos dibujos en texto. (busque usted que es un OCR para completar el tema).

Un string se define por medio de un vector de caracteres (VECTOR DE CHAR). NO EXISTE EL TIPO DE DATO STRING EN C. El uso de dicho vector lo convertirá en un string.

```
char texto[12];
```

Texto es un vector de char de 12 posiciones (índice de 0 a 11), esto es, almacena en cada posición datos de 1byte. Puede almacenar tanto números, como utilizarlo para almacenar un string.

Los string poseen SIEMPRE UN NULL. El NULL es el primer carácter de la tabla ASCII y es numéricamente cero.

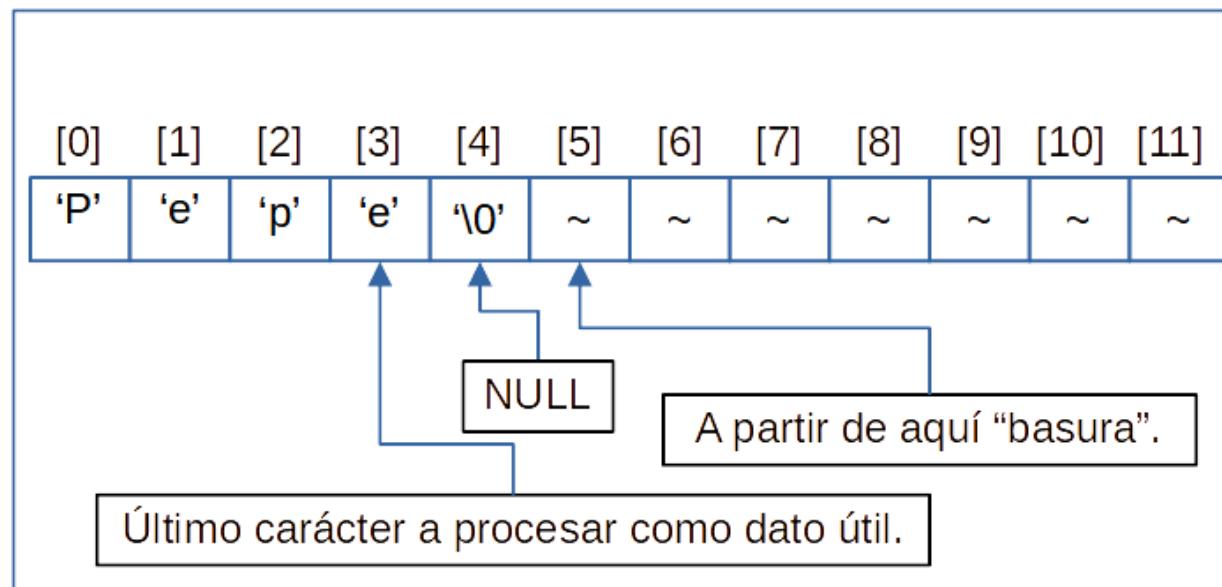
El NULL en C se utiliza para indicar cual es el ultimo carácter a tener en cuenta en el string, siendo todos los que están por “detrás” del NULL irrelevantes (no se consideran como información), o lo que comúnmente se dice: “basura”.

En el dimensionado del vector de char se debe tener en cuenta la cantidad de caracteres que queremos almacenar y luego sumar uno más para el NULL y otro más si usamos fgets para el '\n', esto es como norma general, al menos 2 más de los espacios necesarios para almacenar el texto más largo que vayamos a utilizar.

Veamos algunos ejemplos de calculo del tamaño:

Texto	Cuantas letras tiene el texto a almacenar	Cuantos caractes mas para el NULL y el \n si usamos fgets.	Cantidad mínima de caracteres del vector de char
Hola	4	2	6
Hola mundo	10	2	12
123456	6	2	8
AAA 124 bcd	11	2	13

Asi se almacena el string en el vector de char usando scanf (con fgets antes del NULL hay un /n):



Las funciones de “ingreso” de string desde teclado y otros dispositivos agregan automaticamente el NULL. Tanto scanf usando “%s” como las que se encuentran en la “string.h” agregan el NULL o consideran el NULL como limite de la información útil almacenada. Normalmente el NULL se ubica en la posición donde “tocamos” el ENTER. Pero ojo, scanf además lo hace cuando primero se encuentra el ENTER, el espacio o el tabulador. En el siguiente apartado se verá todo esto.

Nunca se debe perder de vista que un string es siempre un vector de char y todas las acciones que se hagan en el string se están haciendo en un vector. Puedo ingresarlo con funciones para string y luego manejarlo letra a letra o ingresar al vector letra a letra agregando “a mano” el NULL y luego manejarlo como string. De hecho así sucede con las funciones especializadas aunque no lo “veamos”.

ADEMÁS NO HAY QUE PERDER DE VISTA QUE EL NOMBRE DE UN VECTOR ES UN PUNTERO, ESTO ES CONTIENE UNA DIRECCIÓN Y ESA DIRECCIÓN ES LA PRIMERA DE DICHO VECTOR, LA DEL ELEMENTO DE INDICE CERO. ESTE CONCEPTO ES VALIDO PARA CUALQUIER VECTOR DE CUALQUIER TIPO!!!!

Esto dará en la práctica como resultado que no se pueda copiar el contenido de un string a otro string simplemente utilizando el operador de asignación “=” sino que se deberá hacer mediante una función específica. Tampoco puedo comparar el contenido de dos strings sin usar una función específica.

Funciones para ingresar strings por teclado

¿Con que función ingreso una cadena de caracteres (string) por teclado?

De nuevo tenemos varias alternativas, pero en este caso hay algunas funciones que se prefieren por sobre otras por motivos de seguridad, ya que se puede llegar a desbordar el vector contenedor del string y causar efectos imprevisibles que en muchas situaciones pueden afectar al resto del sistema.

FORMA 1: Usar la función **gets** es algo que ya no se prefiere ya que no tiene forma de conocer el tamaño del vector contenedor del string y por lo tanto no tiene forma de cortar el ingreso de caracteres. **LOS NUEVOS CONVENIOS DE C NO RECOMIENDAN SU USO (y algunos no lo permiten).**

```
char texto[40];
gets(texto);
```

gets reemplaza con el **NULL** pura y exclusivamente al **newline(LF)** (carácter 10 de la tabla ASCII). El **ENTER** está formado por el carácter **10 + 13. (LF + CR)**.

En muchos compiladores nuevos ya no se la va a incluir. De todas formas hay que conocerla, sabiendo sus limitaciones y los peligros que pueda crearnos y reemplazarla por otras donde se pueda.

FORMA 2: La función **gets_s** desde el convenio C11. Agrega a la función gets un nuevo parámetro que es el tamaño del vector que va a recibir al string leido desde teclado.

```
char texto[40];
gets_s(texto,40);
```

FORMA 3: Podemos usar la función **scanf**. Tiene la misma limitación que gets al no saber el tamaño del vector donde se almacenará la información.

Ahora el **NULL** se ubicará en el vector cuando encuentre (el primero que aparezca en la cadena de entrada) un **ENTER (LF)**, **ESPACIO (carácter 32)** o **TAB (el horizontal es el carácter 9 y el vertical el carácter 11)**.

```
char texto[40];  
scanf("%s",texto);
```

FORMA 4: La función **fgets** es una de las funciones que se preferirá utilizar a partir de ahora ya que tiene como uno de sus parámetros el tamaño del vector de almacenamiento del string.

```
char texto[40];  
fgets(texto, 40, stdin);
```

stdin es la corriente de entrada de teclado, pudiéndose utilizar otras como por ejemplo la que se genera desde un archivo abierto. El 40 representa en este caso la máxima cantidad de caracteres que serán aceptado, ignorando el resto de los que se ingresen.

Con scanf tambien podemos impedir el desborde de nuestro vector

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

int main(void)
{
    char texto[10];
    short i;

    printf("\nIngrese una palabra de mas de 5 caracteres \n");

    scanf("%5s",texto); /*Permite ingresar hasta 5 caracteres*/
    /**
     * 5s indica que almacenará desde indice 0 a indice 4
     * los caracteres ingresados y en indice 5 colocará un NULL.
     * Impide al igual que fgets y gets_s el desborde del vector,
     * si se usa correctamente, claro está...
     */
    printf("\nLos caracteres que realmente se almacenaron son: %s\n\n",texto);
    printf("\n\n");
    /*Ahora los imprimimos con %c caracter a caracter*/
    for(i=0;i<strlen(texto);i++)
        printf("%c\n",texto[i]);
    return 0;
}
```

Diferencia entre scanf, gets, gets_s y fgets

Vamos a resumir el comportamiento de estas funciones:

	Coloca el NULL en el vector cuando encuentra	Que devuelve strlen si ingreso PEPE?	Deja en el vector algo mas que el texto ingresado?	Puede causar problemas con otras funciones el vector generado?
scanf	Espacio, Enter ('\n') o Tabulador.	Los caracteres de PEPE hasta el NULL: 4	NO	NO
gets	Enter ('\n')	Los caracteres de PEPE hasta el NULL: 4	NO	NO
gets_s	Enter ('\n')	Los caracteres de PEPE hasta el NULL: 4	NO	NO
fgets	Enter ('\n')	Los caracteres de PEPE hasta el NULL: 5	SI, antes del NULL el \n	Resultado no esperado con strcat (concatenar dos strings). Problema al comparar strings.

Si ingresamos “Pepe” con scanf, gets, gets_s nuestro string nos queda así:

'P'	'e'	'p'	'e'	NULL	basura	basura	basura	basura	basura
-----	-----	-----	-----	------	--------	--------	--------	--------	--------

Si ingresamos “Pepe” con fgets nuestro string nos queda así:

'P'	'e'	'p'	'e'	'\n'	NULL	basura	basura	basura	basura
-----	-----	-----	-----	------	------	--------	--------	--------	--------

El '\n' que nos queda en el vector al usar **fgets** nos “obligará” a pensar en un espacio más para que lo contenga dentro del vector declarado, ademas del NULL. Ojo cuando comparemos dos strings ingresados con distintas funciones...

Si uso fgets para ingresar el string debo tener en cuenta que existe un ‘\n’ no previsto, que si no es tomado en cuenta siempre nos dará que son distintos !! Veamos esta situación:

<pre>#include <stdio.h> #include <string.h> int main(void) { char texto1[]{"fin"}; char texto2[20]; printf("Ingrese un texto: "); fgets(texto2,20,stdin); if(strcmp(texto1,texto2)==0) { printf("Contenido igual\n"); } else { printf("Contenido distinto\n"); } return 0; }</pre>	<pre>#include <stdio.h> #include <string.h> int main(void) { char texto1[]{"fin\n"}; char texto2[20]; printf("Ingrese un texto: "); fgets(texto2,20,stdin); if(strcmp(texto1,texto2)==0) { printf("Contenido igual\n"); } else { printf("Contenido distinto\n"); } return 0; }</pre>
---	---

El código de la izquierda fallará siempre. Para que funcione, en el de la derecha hemos puesto: “fin\n”, que es lo correcto y no “fin” sin el \n.

El algoritmo de búsqueda con inserción aplicado a un vector de estructuras y a un string para contar letras de un texto.

Se ingresa un texto y se arma un listado de cada una de las letras que lo conforman con la cantidad de veces que cada una se repite.

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

struct datos{
    char letra;
    short cantidad;
};

int main(void)
{
    struct datos vector[128];

    char texto[300];
    short indice=0, vacio=0,i=0;

    printf("Ingrese un texto (<300 caracteres): ");
    fgets(texto,300,stdin);

    printf("\n");

    while(texto[i])
    {
        for(indice=0;indice<vacio;indice++)
        {
            if(texto[i]==vector[indice].letra)
            {
                vector[indice].cantidad++;
                indice=vacio;
            }
        }
        if(indice==vacio)
        {
            if((texto[i]>='a' && texto[i]<='z')||(texto[i]>='A' && texto[i]<='Z')||texto[i]==' ')
            {
                vector[vacio].letra=texto[i];
                vector[vacio].cantidad=1;
                vacio++;
            }
        }
        i++;
    }

    for(indice=0;indice<vacio;indice++)
        printf("%hd\t%c\t%hd\n",vector[indice].letra,vector[indice].letra,vector[indice].cantidad);

    return 0;
}
```

Ejemplo completo de ordenamiento de un vector de estructuras, sin usar funciones creadas por el usuario, con un miembro vector(precio es un vector que acepta 13 float) y miembro string.

El siguiente ejemplo maneja productos, con la siguiente información por cada uno: designación, string de 50 caracteres válidos, origen, string con 60 caracteres válidos, y un vector de float, precio, de 13 elementos para guardar los 12 precios del año del producto (se desperdicia el índice cero para poder relacionar directamente índice con el numero de mes, siendo 1 enero y 12 diciembre).

<pre>#include <stdio.h> #include <string.h> #define cuantos 5 /* Probamos con solo 5 productos */ struct base{ char designacion[52]; char origen[62]; float precio[13]; /* almacena el precio de cada mes del año. En 1 el de enero*/ }; int main(void) { struct base productos[100]; struct base auxiliar; char texto1[]="Ingrese la designacion del producto: "; char texto2[]="Ingrese la origen del producto: "; char texto3[]="Ingrese el precio: "; short i , j, indice; short mes; for(indice=0;indice<cuantos;indice++) { printf(texto1); fflush(stdin); fgets(productos[indice].designacion,sizeof(productos[indice].designacion),stdin); printf("Designacion: %s",productos[indice].designacion); productos[indice].designacion[strlen(productos[indice].designacion)-1]=0; printf(texto2); fflush(stdin); fgets(productos[indice].origen,sizeof(productos[indice].origen),stdin); printf("Origen: %s",productos[indice].origen); productos[indice].origen[strlen(productos[indice].origen)-1]=0; for(mes=1;mes<13;mes++) { printf("Ingrese precio de mes %hd: ",mes); scanf("%f",&productos[indice].precio[mes]); printf("Precio: %f\n",productos[indice].precio[mes]); } } }</pre>	<pre>printf("Ingrese el mes por el que quiera ordenar los precios: 1-12"); scanf("%hd",&orden_mes); for(i=0;i<cuantos-1;i++) for(j=i+1;j<cuantos;j++) if(productos[i].precio[orden_mes] > productos[j].precio[orden_mes]) { auxiliar=productos[i]; productos[i]=productos[j]; productos[j]=auxiliar; } printf("\n\n"); for(indice=0;indice<cuantos;indice++) /* Imprime todos los datos ordenados */ { printf("Producto: %s\n",productos[indice].designacion); printf("Origen: %s\n",productos[indice].origen); for(mes=1;mes<13;mes++) printf("Precio: %f\n", productos[indice].precio[mes]); printf("-----\n"); } return 0; }</pre>
---	--

Veamos mas en detalle... acá ordenamos en función de un determinado mes, entre 1 y 12. Vea que solo quedará ordenado para ese mes y desordenado para el resto.

```
for(i=0;i<cuantos-1;i++)
{
    for(j=i+1;j<cuantos;j++)
    {
        if(productos[i].precio[orden_mes] > productos[j].precio[orden_mes])
        {
            auxiliar=productos[i];
            productos[i]=productos[j];
            productos[j]=auxiliar;
        }
    }
}
```

Ahora ordenamos por el campo *designación* en es un string (para *origen* seria lo mismo a que también es un string)

```
for(i=0;i<cuantos-1;i++)
{
    for(j=i+1;j<cuantos;j++)
    {
        if(strcmp(productos[i].designacion,productos[j].designacion)<0)
        {
            auxiliar=productos[i];
            productos[i]=productos[j];
            productos[j]=auxiliar;
        }
    }
}
```

Vemos que el único cambio es en la expresión del if y nada mas. Lo demás no cambia.

Otra modificación que podríamos hacer es reemplazar a fgets con scanf:

La línea: `fgets(productos[indice].designacion,sizeof(productos[indice].designacion),stdin);`

se puede reemplazar por: `scanf("%50[^\\n]s", productos[indice].designacion);`

Set de caracteres para códigos mayores a 80h (128 en decimal)

El set de caracteres 850 de la consola de Windows

cp850

Latin 1, Europa Occidental ISO-8859-1

iso-8859-1

	·0	·1	·2	·3	·4	·5	·6	·7	·8	·9	·A	·B	·C	·D	·E	·F
8 ·	U+0080	U+0081	U+0082	U+0083	U+0084	U+0085	U+0086	U+0087	U+0088	U+0089	U+008A	U+008B	U+008C	U+008D	U+008E	U+008F
9 ·	U+0090	U+0091	U+0092	U+0093	U+0094	U+0095	U+0096	U+0097	U+0098	U+0099	U+009A	U+009B	U+009C	U+009D	U+009E	U+009F
A ·	U+00A0	í	¢	£	¤	¥	ƒ	§	„	©	á	«	¬	®	—	—
B ·	ó	±	²	³	’	μ	¶	·	„	¹	º	»	¼	½	¾	¿
C ·	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D ·	Đ	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	Þ
E ·	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F ·	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

En Windows al “Latin-1” se lo denomina Windows-1252. Ojo que como verá hay diferencias.

windows-1252

	·0	·1	·2	·3	·4	·5	·6	·7	·8	·9	·A	·B	·C	·D	·E	·F
8 ·	€ U+20AC	… U+201A	,	f U+0192	” U+201E	… U+2026	† U+2020	‡ U+2021	^ U+02C6	%o U+2030	Š U+0160	< U+2039	Œ U+0152	… U+017D	ž U+017D	… U+017E
9 ·	„ U+2018	‘ U+2019	’ U+201C	“ U+201D	• U+2022	— U+2013	— U+2014	~ U+02DC	TM U+2122	š U+0161	> U+203A	œ U+0153	… U+017E	ž U+017E	ÿ U+0178	
A ·	i U+00A0	¢ U+00A1	£ U+00A2	¤ U+00A3	¥ U+00A4	₩ U+00A5	₪ U+00A6	₪ U+00A7	₪ U+00A8	© U+00A9	ª U+00AA	« U+00AB	¬ U+00AC	® U+00AD	— U+00AE	— U+00AF
B ·	º U+00B0	± U+00B1	² U+00B2	³ U+00B3	‘ U+00B4	µ U+00B5	¶ U+00B6	· U+00B7	· U+00B8	¹ U+00B9	º U+00BA	» U+00BB	¼ U+00BC	½ U+00BD	¾ U+00BE	¿ U+00BF
C ·	À U+00C0	Á U+00C1	Â U+00C2	Ã U+00C3	Ä U+00C4	Å U+00C5	Æ U+00C6	Ç U+00C7	È U+00C8	É U+00C9	Ê U+00CA	Ë U+00CB	Ì U+00CC	Í U+00CD	Î U+00CE	Ï U+00CF
D ·	Đ U+00D0	Ñ U+00D1	Ò U+00D2	Ó U+00D3	Ô U+00D4	Õ U+00D5	Ö U+00D6	× U+00D7	Ø U+00D8	Ù U+00D9	Ú U+00DA	Û U+00DB	Ü U+00DC	Ý U+00DD	Þ U+00DE	Þ U+00DF
E ·	à U+00E0	á U+00E1	â U+00E2	ã U+00E3	ä U+00E4	å U+00E5	æ U+00E6	ç U+00E7	è U+00E8	é U+00E9	ê U+00EA	ë U+00EB	ì U+00EC	í U+00ED	î U+00EE	ï U+00EF
F ·	ð U+00F0	ñ U+00F1	ò U+00F2	ó U+00F3	ô U+00F4	õ U+00F5	ö U+00F6	÷ U+00F7	ø U+00F8	ù U+00F9	ú U+00FA	û U+00FB	ü U+00FC	ý U+00FD	þ U+00FE	ÿ U+00FF

¿Cómo puedo ingresar un texto en español e imprimirllo en pantalla?

Debo tener muy en claro en que set de caracteres se ha codificado la información a imprimir en pantalla para no encontrarme con “símbolos” extraños o incluso sin que no exista símbolo en alguna posición de algún string.

```
#include <stdio.h>
// Añade paquete de idiomas
#include <locale.h>
#include <windows.h>

int main(void)
{
    // Declaración de variables
    char texto[255];//Ojo que lo usamos con la función fgets

    // Establecer el idioma a español
    setlocale(LC_ALL, "spanish"); // Cambiar locale en Linux y Windows
    SetConsoleCP(1252); // Solo para Windows, necesaria
    SetConsoleOutputCP(1252); // Solo para Windows, necesaria
    SetConsoleTitle("Texto en español"); // Probado en Windows 7, optativa
    /* Ahora se pueden mostrar los caracteres en español, signos de puntuación incluidos*/

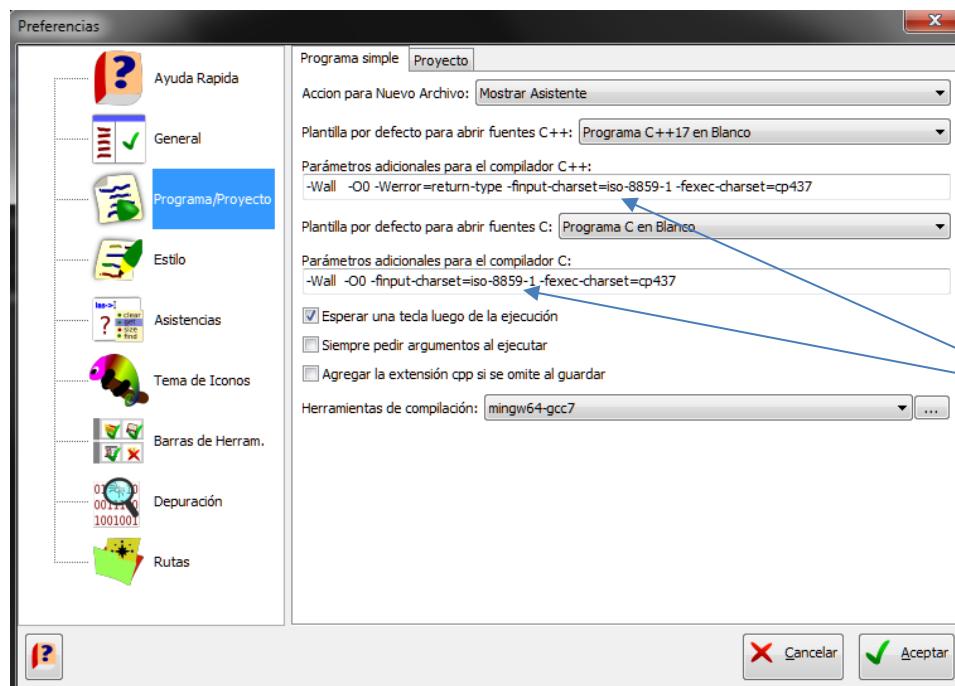
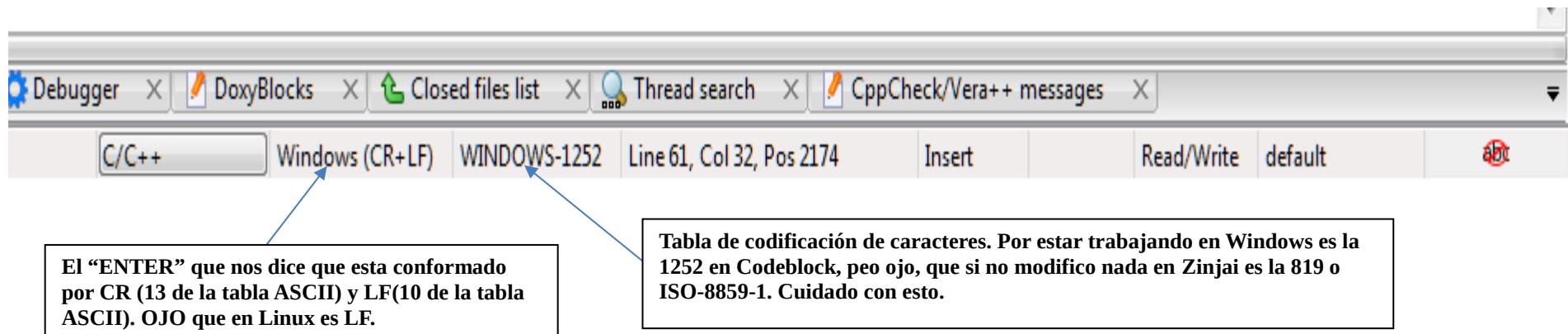
    printf("\n¡Éxito!. Se muestran los caracteres especiales del español.\n\n");

    printf("Introduce un texto en español (ñ, acentos, etc): ");
    fgets(texto,sizeof(texto),stdin);

    printf("\nEl texto es: %s\n\n", texto);

    return 0;
}
```

Si observamos la parte inferior derecha del IDE Codeblocks vemos lo siguiente:



En ZINJAI vemos la codificación iso-8859-1 (819 como también se la conoce).

Conocer como está almacenada la información es fundamental para trabajar correctamente con ella. Esto es muy importante con archivos, particularmente de texto. No puede desconocer este tema.

¿Cómo trabajar con char numéricos bajo entorno Windows 64 bits?

Cuando queremos trabajar con números enteros, estamos usando short como el tipo de dato que ocupa menos espacio en memoria. Si necesitamos almacenar una edad, estamos haciendo:

- `short edad`, o
- `unsigned short edad`

Estas variables, en general, son “muy grandes” para almacenar la edad medida en cantidad de años. Lo lógico sería usar:

- `char edad`, o
- `unsigned char edad`

Pero las variables numéricas char en Windows sufren el problema de no tener un modificador propio para `scanf` ni para `printf` lo que nos dá error a la hora de usarlas. Es por la forma como Microsoft implementó la librería dinámica (dll) asociada a `printf` y `scanf`. No se debe considerar un error sino un criterio de implementación propio de Microsoft.

Veamos el siguiente ejemplo:

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

int main(void)
{
    char A,B,C;

    printf("\nIngrese un valor numerico char: ");
    scanf("%hd",&A);

    printf("\nIngrese un valor numerico char: ");
    scanf("%hd",&B);

    printf("\nIngrese un valor numerico char: ");
    scanf("%hd",&C);

    printf("\nLo ingresado como char es: %hd",A);
    printf("\nLo ingresado como char es: %hd",B);
    printf("\nLo ingresado como char es: %hd",C);

    return 0;
}
```

Observemos una posible salida por consola del mismo:

```
Ingrese un valor numerico char: 1
Ingrese un valor numerico char: 2
Ingrese un valor numerico char: 5
Lo ingresado como char es: 0
Lo ingresado como char es: 0
Lo ingresado como char es: 5
Process returned 0 (0x0)    execution time : 4.508 s
Press any key to continue.
```

Vemos que el resultado obtenido es incorrecto para las dos primeras variables.

La solución que se va a proponer es solamente aplicable al compilador mingw64 no al de 32 bits. En Linux no es necesario agregar nada para obtener el resultado correcto.

Vamos a incluir la siguiente línea: **#define scanf __mingw_scanf** con lo que indicaremos que queremos usar la implementación de scanf de mingw y no la de Microsoft. De esta forma vamos a tener el %hhd y el %hhu. %hhd indicará a scanf que queremos ingresar un dato numérico de 1byte signado (char) mientras que si usamos %hhu indicará a scanf que queremos ingresar un dato numérico de 1byte no signado (unsigned char). Si queremos tener una solución mas completa, necesitaremos implementar printf también en formato mingw para lo cual haremos: **#define scanf __mingw_printf**. Ya veremos que esto nos aportará otras soluciones sumamente necesarias a la hora de realizar ciertos códigos.

Este es el código correcto que deberemos usar con el compilador mingw de 64 bits

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

#define scanf __mingw_scanf
#define printf __mingw_printf

int main(void)
{
    char A,B,C;

    printf("\nIngrese un valor numerico char: ");
    scanf("%hhd",&A);

    printf("\nIngrese un valor numerico char: ");
    scanf("%hhd",&B);

    printf("\nIngrese un valor numerico char: ");
    scanf("%hhd",&C);

    printf("\nLo ingresado como char es: %hhd",A);
    printf("\nLo ingresado como char es: %hhd",B);
    printf("\nLo ingresado como char es: %hhd",C);

    return 0;
}
```

```
Ingrese un valor numerico char: 1
Ingrese un valor numerico char: 2
Ingrese un valor numerico char: 5
Lo ingresado como char es: 1
Lo ingresado como char es: 2
Lo ingresado como char es: 5
Process returned 0 (0x0)  execution time : 3.510 s
Press any key to continue.
```

De nuevo: tengo que tener un Windows 64 bits y un compilador 64 bits o no podré usar esta solución. En Linux no es necesario hacer esto ya que puedo usar sin mas el modificador %hhd.

¿Como usar long long en entornos Windows 64 bits con compiladores de 64 bits?

El siguiente código imprimirá en pantalla los valores límites del long long signado y del long long no signado. Ademas se podrá ver por medio de sizeof cuanto ocupa en memoria.

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

#define printf __mingw_printf
#define scanf __mingw_scant

int main(void)
{
    long long positivo=9223372036854775807; //Es el mayor long long positivo
    long long negativo=-9223372036854775808; //Es el mayor long long negativo
    unsigned long long no_signado=18446744073709551615; //Es el mayor long long no signado

    setlocale(LC_ALL, "spanish"); //Para escribir en castellano

    printf("Tamaño del long long signado es: %lld bytes\n",sizeof(positivo));
    printf("Tamaño del long long no signado es: %llu bytes\n\n",sizeof(no_signado));
    printf("El long long int contiene: %lld\n",positivo);

    printf("El long long int contiene: %lld\n",negativo);

    printf("El unsigned long long int contiene: %llu\n",no_signado);

    return 0;
}
```

Con **%lld** tendremos el manejo de los **long long signados** y con **%llu** tendremos los **long long no signados** tanto con **printf** como **scanf**.

¿Cómo usar las constantes matemática de math.h en Windows 64 con el compilador mingw64?

La cabecera math.h contiene los valores de las siguientes constantes:

#define M_E	2.7182818284590452354
#define M_LOG2E	1.4426950408889634074
#define M_LOG10E	0.43429448190325182765
#define M_LN2	0.69314718055994530942
#define M_LN10	2.30258509299404568402
#define M_PI	3.14159265358979323846
#define M_PI_2	1.57079632679489661923
#define M_PI_4	0.78539816339744830962
#define M_1_PI	0.31830988618379067154
#define M_2_PI	0.63661977236758134308
#define M_2_SQRTPI	1.12837916709551257390
#define M_SQRT2	1.41421356237309504880
#define M_SQRT1_2	0.70710678118654752440

Para poder usarlas correctamente haremos:

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

#define _USE_MATH_DEFINES
#include <math.h>

#define printf __mingw_printf
#define scanf __mingw_scanf

int main(void)
{
    printf("\n\nPi= %1.19f\n", M_PI);
    printf("E= %1.19f\n", M_E);

    return 0;
}
```

Punteros

¿Qué es un puntero?

Un puntero es un tipo de dato que solo puede contener una dirección.

Puede ser una constante o una variable y puede contener la dirección de una constante o una variable.

El tamaño en memoria del puntero **solo depende del hardware y no del tipo de dato “apuntado”**. Depende de la cantidad de memoria máxima a la que el micro puede acceder.

Para declarar el puntero haremos:

float *p1; //p1 puntero que va a contener la dirección de un float

char *p2; //p2 puntero que va a contener la dirección de un char

Si lo implementamos en una PC de 32 bits tanto p1 como p2 será un espacio en memoria de 4 bytes (32 bits). Además va a depender de que la compilación sea en 32 o 64 bits.

Las direcciones se manejan por una cuestión de practicidad en el sistema hexadecimal.

Veamos un primer ejemplo del uso de punteros

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

int main(void)
{
    float A = -2.9876535;
    char B = 'X';
    char texto[ ]="Hola mundo\n";

    float *p1;
    char *p2;

    p1 = &A;
    p2 = &B;

    printf("\nTamanio puntero a float: %hd\n", sizeof(p1));
    printf("Tamanio puntero a char: %hd\n\n", sizeof(p2));
    printf("Tamanio del float: %hd\n", sizeof(A));
    printf("Tamanio del char: %hd\n", sizeof(B));
    printf("Tamanio del texto: %hd\n\n", sizeof(texto));
    printf("Contenido de lo apuntado por p1: %f y contenido del float %f\n", *p1,A);
    printf("Contenido de lo apuntado por p2: %c y contenido del char %c\n\n", *p2,B);

    p2=texto;
    printf("Texto: %s",p2);

    return 0;
}
```

Vamos a ver que representa cada línea de código que hemos escrito:

Definimos 3 variables, una float, una char y un string y le damos un valor inicial.

```
float A = -2.9876535;
char B = 'X';
char texto[]="Hola mundo\n";
```

Creamos 3 variables de tipo puntero para almacenar la dirección de nuestras variables float, char y string.

```
float *p1;
char *p2;
```

Le decimos a nuestros punteros que almacenen la dirección en donde viven nuestras variables.

```
p1 = &A;
p2 = &B;
```

Vemos que p1 va a contener la dirección donde vive el float y p2 va a contener la dirección donde vive la variable char.

ESTE PASO ES FUNDAMENTAL, NO PUEDO USAR UN PUNTERO SI ESTE NO CONTIENE LA DIRECCION DE UNA DETERMINADA VARIABLE O CONSTANTE. SIN ESTE PASO EL PUNTERO CONTIENE UNA DIRECCION QUE PUEDE SER PELIGROSA. NO DEBO NUNCA OLVIDAR ESTE PASO QUE SEGÚN EL CASO PUEDE TENER UNA FORMA DIFERENTE. SIN ESTE PASO EL PUNTERO ES UN “PUNTERO DESCONTROLADO”.

Vamos a ver ahora cuanto ocupa cada puntero en memoria usando sizeof

```
printf("\nTamanio puntero a float: %hd\n", sizeof(p1));
printf("Tamanio puntero a char: %hd\n\n", sizeof(p2));
```

En mi caso ambos son de 8 bytes esto es 64 bits. Yo lo estoy ejecutando en una máquina cuyo micro es de 64 bits (un I3 de Intel) con un sistema operativo de 64 bits. También es común en la PC que de 4 esto es 32 bits. Lo que tenemos que tener muy en claro que NO IMPORTA EL TIPO DE DATO DEL QUE EL PUNTERO TENGA LA DIRECCION SINO DEL HARDWARE Y DEL SISTEMA OPERATIVO, INCLUSO DEL COMPILADOR. SI APUNTO A UNA VARIABLE DE UN BYTE COMO ES UN CHAR O A UN FLOAT EL TAMAÑO DE LA DIRECCION EN MI CASO ES DE 8 BYTES. ES DE 8 BYTES TAMBIEN SI APUNTO A UN STRUCT O A UN VECTOR NUMERICO O UN STRING.

Para verificar lo dicho, imprimo la cantidad de bytes que ocupa cada una de mis variables

```
printf("Tamanio del float: %hd\n", sizeof(A));  
printf("Tamanio del char: %hd\n", sizeof(B));  
printf("Tamanio del texto: %hd\n\n",sizeof(texto));
```

Ahora imprimo los contenidos de las variables usando y no usando punteros

```
printf("Contenido de lo apuntado por p1: %f y contenido del float %f\n", *p1,A);  
printf("Contenido de lo apuntado por p2: %c y contenido del char %c\n\n", *p2,B);
```

Vemos que usamos *p1 y *p2. *p1 significa “el contenido de la dirección contenida en p”, en este caso el contenido del flotante. Con el mismo criterio, *p2 significa “el contenido de lo apuntado por p2”, en este caso la letra X.

Veremos por último

```
p2=texto;  
printf("Texto: %s",p2);
```

p2 era un puntero a char, y si lo pensamos bien un string es una sucesión de elementos char, cada uno con su dirección. Al hacer p2=texto, estoy almacenando la primera de ellas que es la que necesita printf para comenzar desde ahí, a imprimir la cadena hasta encontrar un NULL.

POR LO TANTO UN PUNTERO A CHAR PUEDE CONTENER TANTO LA DIRECCION DE UN UNICO CHAR COMO LA PRIMER DIRECCION DE UN STRING. RECORDEMOS QUE EL NOMBRE DE CUALQUIER VECTOR ES LA DIRECCION INCIAL DEL PRIMER ELEMENTO DE DICHO VECTOR.

Un puntero puede apuntar a un puntero (en mucha bibliografía se denominan punteros dobles)

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

```
int main(void)
{
```

```
    short A=12;
    short *p;
    short **pp;
```

```
    pp = &p;
    p = &A;
```

Name	Value	Type	Address
A	12	short	00000000001AF866
= p	00000000001AF866	short *	00000000001AF858
*p	12	short	00000000001AF866
= pp	00000000001AF858	short **	00000000001AF850
= *pp	00000000001AF866	short *	00000000001AF858
**pp	12	short	00000000001AF866

```
printf("Contenido de A: %hd\n",A);
printf("Contenido de *p: %hd\n",*p);
printf("Contenido de **pp: %hd\n\n",**pp);
```

```
printf("Direccion de A: %p\n",&A);
printf("Direccion de p: %p\n",&p);
printf("Direccion de pp: %p\n\n",(&pp));
```

```
printf("Contenido de p: %p\n",p);
printf("Contenido de *pp: %p\n",*pp);
printf("Contenido de pp: %p\n\n",pp);
```

```
return 0;
```

```
}
```

Otro ejemplo con punteros

En el siguiente ejemplo sumamos dos enteros usando punteros.

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

int main(void)
{
    short A=5;
    short B=9;
    short suma;

    suma=*(&A)+*(&B);
    printf("La suma es: %hd",*&suma);

    return 0;
}
```

Name	Value	Type	Address
A	5	short	00000000001CFE26
B	9	short	00000000001CFE24
suma	14	short	00000000001CFE22
&A	00000000001CFE26		
&B	00000000001CFE24		

Para tener en cuenta desde ahora, ojo con la nomenclatura de punteros:

Declaracion	Explicacion
short *P	P es un puntero a un entero signado de 16 bits
short *P[10]	P es un vector de 10 punteros a enteros signados de 16 bits
short (*P)[10]	P es un puntero a un vector de 10 enteros signados de 16 bits

Cuidado con las declaraciones de punteros. En los siguientes capítulos veremos más en detalle todo esto.

Punteros y estructuras

Sabemos que para acceder a los miembros o campos de una estructura lo podemos hacer con el *operador punto*.

También lo podemos hacer mediante punteros por medio del *operador flecha*.

Se puede definir en forma dinámica una estructura. La función que hace esto se llama malloc, función que devuelve un puntero a la primer dirección de memoria de la estructura.

Si lo hacemos en forma estática, el siguiente código muestra como se usa el operador flecha.

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

struct info{
    unsigned short codigo;
    char nombre[7];
    float sueldo;
};

int main(void)
{
    struct info datos={897,"Pepe",56876.27};
    struct info *pinfo;
    pinfo=&datos;
    printf("codigo: %hu\n",pinfo->codigo);
    printf("nombre: %s\n",pinfo->nombre);
    printf("sueldo: %.2f\n\n",pinfo->sueldo);
    return 0;
}
```

Name	Value	Type	Address
■ datos	{...}	struct info	000000000022F8A8
codigo	897	unsigned short	000000000022F8A8
■ nombre	"Pepe"	char[7]	000000000022F8AA
[0]	80	char	000000000022F8AA
[1]	101	char	000000000022F8AB
[2]	112	char	000000000022F8AC
[3]	101	char	000000000022F8AD
[4]	0	char	000000000022F8AE
[5]	0	char	000000000022F8AF
[6]	0	char	000000000022F8B0
sueldo	56876.3	float	000000000022F8B4
■ pinfo	000000000022F8A8	struct info *	000000000022F8A0
■ *pinfo	{...}	struct info	000000000022F8A8
codigo	897	unsigned short	000000000022F8A8
■ nombre	"Pepe"	char[7]	000000000022F8AA
[0]	80	char	000000000022F8AA
[1]	101	char	000000000022F8AB
[2]	112	char	000000000022F8AC
[3]	101	char	000000000022F8AD
[4]	0	char	000000000022F8AE
[5]	0	char	000000000022F8AF
[6]	0	char	000000000022F8B0
sueldo	56876.3	float	000000000022F8B4

El siguiente código usa el operador flecha para acceder a las funciones de manejo de fecha y hora de C. Se usa los define para mejorar la lectura del código. (Probado en Windows 7)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#define C1 info->tm_year+1900
#define C2 info->tm_mon+1
#define C3 info->tm_mday
#define C4 info->tm_hour
#define C5 info->tm_min
#define C6 info->tm_sec

int main(void)
{
    time_t rawtime;
    struct tm *info;
    char cadena[30];
    short veces;
    for(veces=0;veces<5;veces++)
    {
        time( &rawtime );
        info = localtime( &rawtime );

        printf("Fecha completa: %s", asctime(info));
        sprintf(cadena,"archivo%hd%02hd%02hd%02hd%02hd",C1,C2,C3,C4,C5,C6);
        printf("%s\n",cadena);

        sleep(2); /*2 segundos de demora – windows.h, también _sleep(2) */
    }
    return(0);
}
```

Name	Value	Type	Address
rawtime	1661800066	unsigned int	000000000024FEDC
info	000000013F57E430	struct tm *	000000000024FED0
*info	{...}	struct tm	000000013F57E430
tm_sec	46	int	000000013F57E430
tm_min	7	int	000000013F57E434
tm_hour	16	int	000000013F57E438
tm_mday	29	int	000000013F57E43C
tm_mon	7	int	000000013F57E440
tm_year	122	int	000000013F57E444
tm_wday	1	int	000000013F57E448
tm_yday	240	int	000000013F57E44C
tm_isdst	0	int	000000013F57E450
cadena	"archivo20220829160746"	char[30]	000000000024FEB2
veces	0	short	000000000024FEB0

Punteros y vectores

Si tenemos:

```
short vector[ ]={2,16,-4,29,234,12,0,3};
```

lo podemos ver así:

Dirección	1500	1502	1504	1506	1508	1510	1512	1514
Contenido	vector[0]	vector[1]	vector[2]	vector[3]	vector[4]	vector[5]	vector[6]	vector[7]

y así vemos también la dirección de cada elemento

Dirección	&vector[0]	&vector[1]	&vector[2]	&vector[3]	&vector[4]	&vector[5]	&vector[6]	&vector[7]
Dirección	vector	vector+1	vector+2	vector+3	vector+4	vector+5	vector+6	vector+7

Y ahora vemos lo almacenado...

Dirección	&vector[0]	&vector[1]	&vector[2]	&vector[3]	&vector[4]	&vector[5]	&vector[6]	&vector[7]
Contenido	*vector	*(vector+1)	*(vector+2)	*(vector+3)	*(vector+4)	*(vector+5)	*(vector+6)	*(vector+7)

¿Qué significa vector+1 o vector+4?

Significa que a la dirección inicial del vector la incrementamos en la cantidad de bytes ocupada por cada uno de los elementos. Si hacemos vector+1 estamos haciendo lo siguiente: vector + 1 * sizeof(short) por ser short el tipo de dato almacenado en nuestro vector. Entonces la nueva dirección es vector + 2 bytes. En el caso de vector + 4 tendremos vector + 4 * sizeof(short) esto es vector + 8 bytes.

Nunca perder de vista que vector es un puntero constante; no puedo cambiar la dirección almacenada en él.

Con el “nombre” de nuestro vector, por ser un puntero constante, NUNCA podremos hacer vector+=1 o vector++ ya que lo que estaríamos intentando hacer es vector = vector + 1 * sizeof(short), esto es, estamos queriendo modificar la dirección almacenada de nuestro puntero constante y eso esta **PROHIBIDO!**

Por esto necesitamos sumarle una constante como cuando hacemos vector+1 o usar una variable como por ejemplo cuando hacemos vector + i.

Esta última expresión nos da la formula general vector + i*sizeof(short), donde i es el “índice” de mi vector, nueva dirección que no se almacena en ninguna parte (por ahora...).

Dirección de los elementos de un vector

Supongamos que previamente se ha definido un vector denominado tabla. (ejemplo: short tabla[50])

El nombre de un vector es una *constante de tipo puntero* o lo que es lo mismo, *la dirección del primer elemento* (dirección inicial) en memoria de dicho vector. (muy importante este concepto!!).

- Si el array es unidimensional

Para obtener la dirección de comienzo de la tabla o lo que es lo mismo la dirección del primer elemento, se puede obtener de las siguientes formas:

```
tabla  
&tabla  
&tabla[0]
```

Si se desea obtener la dirección del segundo elemento de la tabla haremos lo siguiente:

```
tabla + 1  
&tabla + 1
```

En general para obtener la dirección del elemento *i* se tiene estas alternativas:

```
tabla + i  
&tabla + i
```

- Si el vector es bidimensional (matriz)

Para obtener la dirección de comienzo de la tabla o lo que es lo mismo la dirección del primer elemento, se puede obtener de una de las siguientes maneras:

```
tabla  
&tabla  
&tabla[0]  
&tabla[0][0]
```

Si se desea obtener la dirección del elemento (2,0) de la tabla, es decir, el primer elemento de la fila 2, tenemos las siguientes alternativas:

```
tabla +2  
*(tabla + 2)  
&tabla[2]  
&tabla[2][0]
```

Si se desea la dirección del elemento (2,3) de la tabla, sería:

```
(*(tabla + 2)+3)  
(tabla[2]+3)  
&tabla[2][3]
```

En general para obtener la dirección del elemento (i,j) seria:

```
(*(tabla + i) + j)  
(tabla[i] +j)  
&tabla[i][j]
```

Contenido de los elementos de un vector

Cuando se declara una vector, se puede acceder a sus elementos mediante indexación (utilizando índices) o utilizando punteros.

La ventaja de acceder mediante punteros es su mayor velocidad de acceso frente al cálculo del índice.

- Si es un vector unidimensional (short tabla[5]={-8,2,-6,4,1};)

Se puede acceder al primer elemento de varias maneras:

tabla[0]
*tabla

Se puede acceder al elemento 3 de varias maneras:

tabla[3]
*(tabla+3)

Se puede acceder al elemento i de varias formas:

`tabla[i]`
`*(tabla[i])`

- Si es un vector bidimensional (matriz) (Ejemplo: short tabla[7][5])

Se puede acceder al primer elemento de la fila 0 de varias formas:

`tabla[0][0]`
`*(tabla)`

Se puede acceder al primer elemento de la fila i de una de las siguientes formas:

`*(tabla+i)`
`tabla[i][0]`

Se puede acceder por ejemplo al elemento (i, j) usando una de las dos formas:

`tabla[i][j]`
`*(tabla + i)+j`

Por medio de uno o más punteros puedo acceder al contenido de un mismo vector

```
#include <stdio.h>
#include <stdlib.h>
#define cantidad 5

int main(void)
{
    /**Creo un vector numerico de 5 elementos y lo cargo con valores*/
    short vector_numerico[cantidad]={5,-2,4,-6,3};

    /**Creo un puntero a short*/
    short *puntero_a_short;

    /**Creamos una variable que la usaremos como indice*/
    unsigned short indice;

    /** Le digo al puntero donde vive el vector.
    El nombre del vector contiene la direccion del primer elemento
    del mismo.*/
    puntero_a_short=vector_numerico;

    /** Vamos a imprimir los valores del vector en pantalla */
    for(indice=0;indice<cantidad;indice++)
    {
        printf("El contenido en indice %hu es %hd\n",indice, *puntero_a_short);
        puntero_a_short++;
    }

    printf("\n\n");/**Bajo dos lineas para separar la informacion*/
```

```
/** Otra alternativa posible es la siguiente para imprimir los datos  
Pero OJO!! EL PUNTERO NOS QUEDO APUNTANDO A UN DATO DENTRO DEL VECTOR QUE  
NO EXISTE. NO TENEMOS INDICE 5 EN EL VECTOR.  
POR LO TANTO LE TENEMOS QUE VOLVER A INFORMAR AL PUNTERO NUEVAMENTE  
DONDE EMPIEZA EL VECTOR EN MEMORIA. SINO, VAN A IMPRIMIR CUALQUIER COSA!!  
*/
```

```
puntero_a_short=vector_numerico;  
  
for(indice=0;indice<cantidad;indice++)  
{  
    printf("El contenido en indice %hu es %hd\n",indice, *(puntero_a_short+indice));  
}  
return 0;  
}
```

Existe una marcada dualidad entre puntero y vector que es muy útil a la hora de trabajar con vectores.

Como detalle muy importante se debe hacer notar lo que se denomina aritmética de puntero.

Si tengo un puntero a short este cada vez que yo le sume 1 (uno) “saltará” de a dos direcciones, si es un puntero a char lo hará de a uno y si es a un float de a 4 (o el tamaño que tenga el float definido en el convenio IEEE 754) es decir salta tantos bytes como tamaño de memoria ocupe. Puedo también sumar 5, 7 o cualquier otro entero siempre y cuando esté dentro de los índices válidos del vector.

De ser una estructura como “mínimo” saltará a una nueva dirección que resulta de sumar el tamaño de cada uno de los campos o miembros de la estructura.

Por ejemplo por ocupar la siguiente estructura 56 bytes se debe entender el +1 o el ++ como un salto de 56 bytes para encontrar la siguiente estructura.

```
struct tipo1{  
    short entero;  
    char nombre[50];  
    float real;  
};
```

Veamos un vector de estructuras y su uso con punteros

```

#include <stdio.h>
#include <stdlib.h>
#define cantidad 5

struct tipo1{
    short entero;
    char nombre[50];
    float real;
};

int main(void)
{
    struct tipo1 mi_variable[cantidad]={{2,"Pepe",45.65},{6,"Pipo",76.29}};

    struct tipo1 *puntero_a_estructura;

    unsigned short indice;

    puntero_a_estructura=mi_variable;

    printf("Entero: %hd \n",puntero_a_estructura->entero);
    printf("Nombre: %s \n",puntero_a_estructura->nombre);
    printf("Real: %f\n\n",puntero_a_estructura->real);

    puntero_a_estructura++;/*apunto a la siguiente estructura en mi vector*/
    printf("Entero: %hd \n",puntero_a_estructura->entero);
    printf("Nombre: %s \n",puntero_a_estructura->nombre);
    printf("Real: %f\n\n",puntero_a_estructura->real);

    puntero_a_estructura++;/*estamos en el indice 2 de mi vector, no perder esta cuenta*/
}

```

Name	Value	Type	Address
mi_variable	[...]	struct tipo1[5]	000000000015FD28
[0]	{...}	struct tipo1	000000000015FD28
entero	2	short	000000000015FD28
nombre	"Pepe"	char[50]	000000000015FD2A
real	45.6500	float	000000000015FD5C
[1]	{...}	struct tipo1	000000000015FD60
entero	6	short	000000000015FD60
nombre	"Pipo"	char[50]	000000000015FD62
real	76.2900	float	000000000015FD94
[2]	{...}	struct tipo1	000000000015FD98
entero	0	short	000000000015FD98
nombre	""	char[50]	000000000015FD9A
real	0.000000	float	000000000015FDCC
[3]	{...}	struct tipo1	000000000015FDD0
[4]	{...}	struct tipo1	000000000015FE08
puntero_a_estructura	000000000015FD28	struct tipo1 *	000000000015FD20
*puntero_a_estructura	{...}	struct tipo1	000000000015FD28
entero	2	short	000000000015FD28
nombre	"Pepe"	char[50]	000000000015FD2A
real	45.6500	float	000000000015FD5C

Name	Value	Type	Address
mi_variable	[...]	struct tipo1[5]	000000000015FD28
[0]	{...}	struct tipo1	000000000015FD28
entero	2	short	000000000015FD28
nombre	"Pepe"	char[50]	000000000015FD2A
real	45.6500	float	000000000015FD5C
[1]	{...}	struct tipo1	000000000015FD60
entero	6	short	000000000015FD60
nombre	"Pipo"	char[50]	000000000015FD62
real	76.2900	float	000000000015FD94
[2]	{...}	struct tipo1	000000000015FD98
[3]	{...}	struct tipo1	000000000015FDD0
[4]	{...}	struct tipo1	000000000015FE08
puntero_a_estructura	000000000015FD60	struct tipo1 *	000000000015FD20
*puntero_a_estructura	{...}	struct tipo1	000000000015FD60
entero	6	short	000000000015FD60
nombre	"Pipo"	char[50]	000000000015FD62
real	76.2900	float	000000000015FD94

/* habiamos definido solamente el contenido de los indices 0 y 1.
Ahora cargaremos el contenido de la estructura en el indice 2 */

```
printf("Ingrese un entero: ");
scanf("%hd",&puntero_a_estructura->entero); /* Ojo no olvidar el & */
printf("Ingrese un nombre: ");
scanf("%s",puntero_a_estructura->nombre);
printf("Ingrese un real: ");
scanf("%f",&puntero_a_estructura->real); /* Ojo no olvidar el & */

printf("\nEntero: %hd \n",puntero_a_estructura->entero);
printf("Nombre: %s \n",puntero_a_estructura->nombre);
printf("Real: %f\n\n",puntero_a_estructura->real);

/*Busque usted la manera de cargar de indice 0 a indice 4 por teclado usando un for */

return 0;
}
```

Apuntando a un vector de estructuras con un puntero a char: vamos a ver al vector de estructuras como un vector de tipos de datos distintos.

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

#define cantidad 5

struct tipo1{
    short entero;
    char nombre[50];
    float real;
};

int main(void)
{
    struct tipo1 mi_variable[cantidad]={{2,"Pepe",45.56},{6,"Pipo",76.76}};

    char *puntero_a_char=(char *)mi_variable;
    short indice;

    setlocale(LC_ALL, "spanish"); //Para escribir en castellano

    printf("\nTamaño de la estructura: %lld\n",sizeof(mi_variable[0]));
}
```

```
for(indice=0; indice<6; indice++)
{
    switch(indice % 3)
    {
        case 0:
            printf("%p      %hd\n",puntero_a_char,*puntero_a_char);
            break;
        case 1:
            puntero_a_char+=sizeof(short);
            printf("%p      %s\n",puntero_a_char,puntero_a_char);
            break;
        case 2:
            puntero_a_char+=50;
            printf("%p      %f\n",puntero_a_char,*((float*)puntero_a_char));
            puntero_a_char+=4;//preparo para la siguiente estructura
            break;
    }
}
return 0;
}
```

Ordenamiento utilizando vector de direcciones sin mover físicamente los datos

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

#define borrar_pantalla system("cls")
#define N 20

int main(void)
{
    short vector[N]={2,6,4,8,9,1,7,13,12,10,-3,14,19,-20,16,5,11,-15,18,-17};
    short *indice[N];
    short i,j, *aux;

    borrar_pantalla;

    for(i=0; i<N; i++)
        indice[i]=&vector[i];

    printf("Comparacion entre contenido de indice y direcciones de cada elemento del vector\n\n");
    for(i=0; i<N; i++)
    {
        printf("\t\t%p ----- %p\n",indice[i],&vector[i]);
    }

    printf("\nIntecla para continuar....");
    getch();

    borrar_pantalla;
```

```

for(i=0; i<N-1; i++)
    for(j=i+1; j<N; j++)
        if(*indice[i]>*indice[j])
        {
            aux=indice[i];
            indice[i]=indice[j];
            indice[j]=aux;
        }

printf("Comparacion entre el contenido de lo apuntado por indice y vector original\n\n");

printf("Vector desordenado      Vector ordenado\n");
for(i=0; i<N; i++)
    printf("%7hd _____ %7hd\n",vector[i],*indice[i]);

return 0;
}

```

Pasaje de parámetros por referencia a funciones

Un simple programa con funciones y pasaje por referencia

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

```
void imprimir(short *);
```

```
int main(void)
```

```
{
```

```
short A;
```

```
A=12;
```

Name	Value	Type	Address
A	12	short	0000000001AFBB6
&A	00000000001AFBB6		

```
imprimir(&A);
```

```
}
```

```
return 0;
```

```
}
```

Name	Value	Type	Address
variable	00000000001AFBB6	short *	00000000001AFB90
*variable	12	short	00000000001AFBB6

```
void imprimir(short *variable)
```

```
{
```

```
printf("La variable contiene el valor: %hd",*variable);
```

```
}
```

Un código que nos puede dar problemas...

```
#include <stdio.h>

void imprimir(short *variable)
{
    printf("\nLa variable contiene el valor: %hd\n",*variable);
}

short* ingresar(void)
{
    static short M; //si le saco static no funciona!! OJO
    printf("\nIngrese un valor: ");
    scanf("%hd",&M);

    return &M;
}

imprimir(puntero);

return 0;
}
```

Una alternativa a las variables tipo static

Usando static	Usando pasaje por referencia
<pre>#include <stdio.h> #include <stdlib.h> void contador(void); int main() { short veces; for(veces=0;veces<10;veces++) contador(); return 0; } void contador(void) { static short cuenta=0; cuenta++; printf("\n%hd",cuenta); }</pre>	<pre>#include <stdio.h> #include <stdlib.h> void contador(short *); int main(void) { short cuenta=0; short veces; for(veces=0;veces<10;veces++) contador(&cuenta); return 0; } void contador(short *cuenta) { (*cuenta)++; printf("\n%hd",*cuenta); }</pre>

Funciones y vectores

Funciones de ordenamiento

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

void ordena(short *vector, short cantidad);

int main(void)
{
    short X[]={7, 5, 3, 9, 55, 32,12, -2, -8,16};
    short cuantos, indice;
    cuantos=sizeof(X)/sizeof(short);

    printf("Los imprimo en el orden original...\n\n");
    for(indice=0;indice<cuantos;indice++)
        printf("%hd\n",X[indice]);

    ordena(X,cuantos);

    printf("\n\ny ahora los imprimo ordenados...\n\n");
    for(indice=0;indice<cuantos;indice++)
        printf("%hd\n",X[indice]);

    return 0;
}
```

```
void ordena(short *vector, short cantidad)
{
    short auxiliar;
    short i, j;

    for(i=0;i<cantidad-1;i++)
        for(j=i+1;j<cantidad;j++)
            if(vector[i]>vector[j])
            {
                auxiliar=vector[i];
                vector[i]=vector[j];
                vector[j]=auxiliar;
            }
}
```

```

#include <stdio.h>
#include <stdlib.h>

void ordena(short *vector, short cantidad);

int main(void)
{
    short X[]={7,5,3,9,55,32,12,-2,-8,16};
    short cuantos, indice;
    cuantos=sizeof(X)/sizeof(short);

    printf("Los imprimo en el orden original...\n\n");
    for(indice=0;indice<cuantos;indice++)
        printf("%hd\n",X[indice]);

    ordena(X,cuantos);

    printf("\n\ny ahora los imprimo ordenados...\n\n");
    for(indice=0;indice<cuantos;indice++)
        printf("%hd\n",X[indice]);
}

```

```

void ordena(short *vector, short cantidad)
{
    short auxiliar;
    short R=0;
    short no_ordenado;
    short i, j;

    do{
        R++;
        no_ordenado=0;
        for(i=0;i<cantidad-R;i++)
            if(vector[i]>vector[i+1])
            {
                auxiliar=vector[i];
                vector[i]=vector[i+1];
                vector[i+1]=auxiliar;
                no_ordenado=1;
            }
    }while(no_ordenado);
}

```

Función de búsqueda binaria

```

//Busqueda binaria

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void busqueda_binaria(short *vector, short superior, short buscado, short *encontrado);

int main(void)
{
    short vector[]={-5,-2,-1,2,4,5,6,9,12,15,22,25,45,56,67,102,140};  

    short buscado;
    short superior;
    short encontrado;

    do
    {
        printf("\nIngrese valor a buscar: ");
        scanf("%hd",&buscado);

        superior=(sizeof(vector)/sizeof(short)) -1;
        busqueda_binaria(vector,superior, buscado, &encontrado);
        if(encontrado>=0)
            printf("Encontrado %hd en el indice %hd\n",buscado, encontrado);
        else
            printf("No encontrado el numero %hd\n",buscado);
        printf("\npresione s para continuar con un nuevo valor a buscar: ");
    }while(getch()=='s');

    return 0;
}

```

El vector siempre tiene que estar ordenado. NO SE PUEDE APLICAR BUSQUEDA BINARIA A VECTORES DESORDENADOS!!!

```
void busqueda_binaria(short *vector, short superior, short buscado, short *encontrado)
{
    short inferior=0;
    short medio=superior/2;

    while(vector[medio]!=buscado && superior >=inferior)
    {
        if(buscado > vector[medio])
            inferior=medio+1;
        else
            superior=medio-1;

        medio=(superior+inferior)/2;
    }

    if(buscado==vector[medio])
    {
        *encontrado=medio;
    }
    else
    {
        *encontrado=-1;
    }
}
```

Funciones y matrices

```
#include <stdio.h>
#include <stdlib.h>
#define maxcolumnas 5

void imprimir(short P[ ][maxcolumnas]);

int main(void)
{
    short matriz[2][5]={{1,2,3,4,5},{6,7,8,9,10}};

    imprimir(matriz)
    return 0;
}

void imprimir(short P[ ][maxcolumnas])
{
    short fila;
    short columnna;

    for(fila=0;fila<2;fila++)
    {
        printf("\n");
        for(columnna=0;columnna<maxcolumnas;columnna++)
            printf("%hd ",P[fila][columnna]);
    }
}
```

Ahora la pasamos la matriz con un puntero a un vector...

```
#include <stdio.h>
#include <stdlib.h>
#define maxcolumnas 5

void imprimir(short *P);

int main(void)
{
    short matriz[2][5]={{1,2,3,4,5},{6,7,8,9,10};

        imprimir((short *)matriz); //vectorizamos la matriz
        return 0;
}

void imprimir(short *P)
{
    short fila;
    short columna;

    for(fila=0;fila<2;fila++)
    {
        printf("\n");
        for(columna=0;columna<maxcolumnas;columna++)
            printf("%hd ",P[fila*maxcolumnas+columna]);//calculamos direccion de cada elemento
    }
}
```

“CASTEAMOS” PARA CONVERTIR DE
MATRIZ DE DOS DIMENSIONES A UN
VECTOR. La función espera un vector,
nunca perder en cuenta esto.

Funciones y string

```
void mi_gets(unsigned char *vector, unsigned short tamano)
{
    unsigned char letra;
    unsigned short indice=0;

    letra=getche();

    while(letra!=13 && indice<tamano) //distinto a enter
    {
        vector[indice]=letra;
        indice++;
        letra=getche();
    }

    vector[indice]=0;
}
```

```
void mi_puts(unsigned char *vector)
{
    unsigned short indice=0;

    while(vector[indice]!=0)
    {
        putchar(vector[indice]);
        indice++;
    }
}
```

```
unsigned short mi_strlen(unsigned char *vector)
{
    unsigned short contador=0;

    while(vector[contador]!=0)
        contador++;

    return contador;
}
```

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void mi_gets(unsigned char *vector, unsigned short tamano);
void mi_puts(unsigned char *vector);
unsigned short mi_strlen(unsigned char *vector);

int main(void)
{
    unsigned char vector[50];

    printf("Ingrese un texto: ");
    mi_gets(vector,40);
    printf("\n\nEl texto ingresado es: ");
    mi_puts(vector);
    printf("\n\nEl tamaño del texto ingresado es: %hu",mi_strlen(vector));
    return 0;
}

unsigned short mi_strlen(unsigned char *vector)
{
    unsigned short contador=0;

    while(vector[contador]!=0)
        contador++;

    return contador;
}

```

```

void mi_gets(unsigned char *vector, unsigned short tamano)
{
    unsigned char letra;
    unsigned short indice=0;

    letra=getche();

    while(letra!=13 && indice<tamano)
    {
        vector[indice]=letra;
        indice++;
        letra=getche();
    }
    vector[indice]=0;
}

void mi_puts(unsigned char *vector)
{
    unsigned short indice=0;

    while(vector[indice]!=0)
    {
        putchar(vector[indice]);
        indice++;
    }
}

```

```

short mi_strcpy(char *destino, char *fuente)
{
    short indice=0;

    while(fuente[indice]!=0)
    {
        destino[indice]=fuente[indice];
        indice++;
    }

    destino[indice]=0;

    return 1;
/** Indica que la funcion termino correctamente.
NO es necesario.
La funcion podría devolver void aunque no se recomienda.
*/
}

```

```

#include <stdio.h>
#include <stdlib.h>
short mi_strcpy(char *destino, char *fuente);
int main()
{
    char fuente[]="Hola mundo";
    char destino[40]; //mayor o igual que fuente

    if(mi_strcpy(destino,fuente));
        printf(destino);
    return 0;
}

short mi_strcpy(char *destino, char *fuente)
{
    short indice=0;

    while(fuente[indice]!=0)
    {
        destino[indice]=fuente[indice];
        indice++;
    }
    destino[indice]=0;

    return 1;
}

```

```

short mi_strcmp(char *cadena1, char *cadena2)
{
    short indice=0;

    while((cadena1[indice] != 0 && cadena2[indice] != 0) &&(cadena1[indice] == cadena2[indice]))
    {
        indice++;
    }
    return(cadena1[indice] - cadena2[indice]);
}

```

```

#include <stdio.h>
#include <stdlib.h>

short mi_strcmp(char *cadena1, char *cadena2);

int main()
{
    printf("%d",mi_strcmp("Baca","Vaca"));
    return 0;
}

short mi_strcmp(char *cadena1, char *cadena2)
{
    short indice=0;

    while((cadena1[indice]!=0 && cadena2[indice]!=0) &&(cadena1[indice]== cadena2[indice]))
    {
        indice++;
    }
    return(cadena1[indice]-cadena2[indice]);
}

```

```
short mi_token(char *vector, char buscado, short reset)
{
    static short indice=0;
    short encontrado=-1;

    if(reset==1)
        indice=0;

    while(vector[indice]!=0 && vector[indice]!=buscado)
    {
        indice++;
    }

    if(vector[indice]==buscado)
    {
        encontrado=indice;
        indice++; //preparo el indice para la proxima llamada a la funcion
        return(encontrado);
    }

    if(vector[indice]==0)
    {
        indice=0;
        return(-1); //se acabo el vector con -1
    }
}
```

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

short mi_token(char *vector, char buscado, short reset);

int main(void)
{
    char texto[]={ "Este texto, que es de prueba, se usa como ejemplo." };
    char caracter;
    short resultado;

    printf("\nTexto donde buscar: %s\n\n",texto);

    printf("Ingrese el caracter a buscar: ");
    caracter=getchar();

    }resultado=mi_token(texto,caracter,1); //fuerzo indice al comienzo del vector con 1

    while(resultado!=-1)
    {
        printf("%hd ",resultado);
        resultado=mi_token(texto,caracter,0);
    }

    printf("\n\n");

    return 0;
}

```

```

short mi_token(char *vector, char buscado, short reset)
{
    static short indice=0;
    short encontrado=-1;

    if(reset==1)
        indice=0;

    while(vector[indice]!=0 && vector[indice]!=buscado)
    {
        indice++;
    }

    if(vector[indice]==buscado)
    {
        encontrado=indice;
        indice++;
        return(encontrado);
    }

    if(vector[indice]==0)
    {
        indice=0;
        return(-1); //se acabo el vector con -1
    }
}

```

Vector de strings (vector de vectores)

Un vector de strings no es ni más ni menos que una matriz de caracteres de dos dimensiones. La primera representará cuantos strings distintos tendremos, mientras que la segunda representará cuantos caracteres máximos podrá tener cada uno de los strings, sin olvidarnos del NULL.

Si tenemos:

```
char NyA[40][91];
```

Significará que tenemos 40 **nombres y apellidos** distintos de 90 caracteres máximos cada uno, más el NULL lo que nos da 91.

Si queremos ingresar cada uno de los strings podemos hacer lo siguiente:

```
for(cual=0;cual>40;cual++)  
    gets(NyA[cual]);
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define cuantos 5
#define caracteres 51

int main(void)
{
    char nombres[cuantos][caracteres]={"Pepe","Maria","Diego","Ana Laura","Orlando"};
    short i;

    printf("Direccion de la matriz: %p\n",nombres);
    printf("Direccion de cada uno de los nombres\n\n");

    for(i=0;i<cuantos;i++)
        printf("nombres[%hd]: %p\n",i,nombres[i]);

    printf("\nTamanio total en memoria: %hd\n\n",sizeof(nombres));
    for(i=0;i<cuantos;i++)
        printf("nombres[%hd]: %s\n",i,nombres[i]);
    printf("\n\n");

    for(i=0;i<cuantos;i++)
        printf("strlen(nombres[%hd]): %hd\n",i,strlen(nombres[i]));
    return 0;
}

```

Vector de punteros a cadena de caracteres (ejemplo)

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

int main(void)
{
    char texto1[ ]="Texto1\n";
    char texto2[ ]="Texto2\n";
    char texto3[ ]="Texto3\n";

    char *puntero_a_texto[3];

    short indice;

    puntero_a_texto[0]=texto1;
    puntero_a_texto[1]=texto2;
    puntero_a_texto[2]=texto3;

    for(indice=0;indice<3;indice++)
    {
        printf("%s",*(puntero_a_texto+indice));
    }

    return 0;
}
```

Name	Value	Type	Address
text01	"Texto1\n"	char[8]	00000000002CFC60
text02	"Texto2\n"	char[8]	00000000002CFC58
text03	"Texto3\n"	char[8]	00000000002CFC50
puntero_a_texto	[...]	char *[3]	00000000002CFC38
[0]	00000000002CFC60	char *	00000000002CFC38
[1]	00000000002CFC58	char *	00000000002CFC40
[2]	00000000002CFC50	char *	00000000002CFC48
indice	0	short	00000000002CFC36

... y además ahora vemos un determinado carácter...

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

int main(void)
{
    char texto1[ ]="Texto1\n";
    char texto2[ ]="Texto2\n";
    char texto3[ ]="Texto3\n";

    char *puntero_a_texto[3];

    short indice;

    puntero_a_texto[0]=texto1;
    puntero_a_texto[1]=texto2;
    puntero_a_texto[2]=texto3;

    for(indice=0;indice<3;indice++)
    {
        printf("%s",*(puntero_a_texto+indice));
    }

    printf("El sexto caracter del tercer texto es: %c\n",puntero_a_texto[2][5]);
    printf("El sexto caracter del tercer texto es: %c\n",*((puntero_a_texto+2)+5));
    return 0;
}
```

Name	Value	Type	Address
indice	0	short	000000000002BFE36
puntero_a_texto	[...]	char *[3]	000000000002BFE38
text01	"Texto1\n"	char[8]	000000000002BFE60
text02	"Texto2\n"	char[8]	000000000002BFE58
text03	"Texto3\n"	char[8]	000000000002BFE50
puntero_a_texto[2][5]	51	char	000000000002BFE55
*((puntero_a_texto+2)+5)	51	char	000000000002BFE55

Vector de punteros a cadena de caracteres

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *dias[7]={"Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "Sabado", "Domingo"};
    printf("%s",*(dias+3));
    return 0;
}
```

Name	Value	Type	Address
■ dias	[...]	char *[7]	000000000013FAF0
■ [0]	000000013F3E8443	char *	000000000013FAF0
■ [1]	000000013F3E843C	char *	000000000013FAF8
■ [2]	000000013F3E8432	char *	000000000013FB00
■ [3]	000000013F3E842B	char *	000000000013FB08
	"Jueves"	char	000000013F3E842B
■ [4]	000000013F3E8423	char *	000000000013FB10
■ [5]	000000013F3E841C	char *	000000000013FB18
■ [6]	000000013F3E8414	char *	000000000013FB20
*(dias + 3)	000000013F3E842B	char *	000000000013FB08
((dias+3))	74	char	000000013F3E842B
dias[3]	000000013F3E842B	char *	000000000013FB08
dias[3][0]	74	char	000000013F3E842B

Pasaje de un vector de strings a una función

Si ahora queremos pasar nuestro vector de strings a una función, haremos lo siguiente:

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

#define cuantos 5
#define caracteres 51

void imprimir(char (*V)[51], short);

int main(void)
{
    char  nombres[cuantos][caracteres]={"Pepe","Maria","Diego","Ana Laura","Orlando"};

    imprimir(nombres,cuantos);
    return 0;
}

void imprimir(char (*V)[51], short cuantos_son)
{
    short i;

    for(i=0;i<cuantos_son;i++)
        printf("%s\n",V[i]);
}
```

...una modificación del código de la función imprimir...

```
void imprimir(char (*V)[51], short cuantos_son)
{
    short i;

    for(i=0;i<cuantos_son;i++)
        printf("%s\n",V+i);
}
```

Ejemplo completo ordenamiento de un vector de estructuras, con funcion, con un miembro string.

Ordena un vector de estructuras por miembro string.

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

struct datos
{
    short legajo;
    char nombre[25];
};

void imprime(struct datos *V, short N);
void ordena(struct datos *V, short N);

int main(void)
{
    struct datos vector[]=
    {
        {111,"Pepe"},{2341,"Juan"},{3444,"Carlos"},
        {4441,"Ana"},{5514,"Julia"},{6000,"Lucas"},
        {7199,"Federico"},{8880,"Gustavo"},
        {9000,"Eleonora"},{10000,"Orlando"},
        {11549,"Leo"},{12000,"Luis"},{13400,"Guillermo"},
        {14999,"Luciana"},{15900,"Luciana"},{18900,"Julia"},
        {19700,"Hector"},{20100,"Susana"}
    };

    short cantidad=sizeof(vector)/sizeof(vector[0]);

    ordena(vector, cantidad);
    imprime(vector, cantidad);

    return 0;
}
```

```
void ordena(struct datos *V, short N)
{
    struct datos aux;
    short i, j;
    for(i=0; i<N-1; i++)
        for(j=i+1; j<N; j++)
            if(strcmp(V[i].nombre,V[j].nombre)>0)
            {
                aux=V[i];
                V[i]=V[j];
                V[j]=aux;
            }
}

void imprime(struct datos *V, short N)
{
    short i;

    for(i=0; i<N; i++)
        printf("%hd ---> %s\n ",V[i].legajo, V[i].nombre);
}
```

Más sobre punteros y funciones

Se presentan a continuación cuatro funciones que hacen lo mismo pero de distinta forma. Se presentan a modo de ejemplo que las funciones pueden ser realizadas de distinta forma, obteniéndose el mismo resultado. Algunas de ellas se comprenderán mucho mejor en Informática II, EDA1 u otras asignaturas similares.

Se observa el uso de punteros, punteros a punteros y la necesidad en una de ellas del uso de la sentencia static.

<pre>#include <stdio.h> #include <stdlib.h> short *cargashort1(void); void cargashort2(short **puntero); void cargashort3(short *puntero); void cargashort4(short *puntero); int main(void) { short A; short *PunteroMain; A=*cargashort1(); printf("Cargashort1: %hd\n",A); cargashort2(&(PunteroMain)); printf("Cargashort2: %hd\n",*PunteroMain); cargashort3(&A); printf("Cargashort3: %hd\n",A); cargashort4(&A); printf("Cargashort4: %hd\n",A); return 0; }</pre>	<pre>short *cargashort1(void) { static short valor=5; return &valor; } void cargashort2(short **puntero) { (*puntero)=5; } void cargashort3(short *puntero) { *puntero=5; } void cargashort4(short* puntero) { short valor=5; *puntero=valor; }</pre>
--	--

Ahora vamos a crear y cargar un vector dentro de una función. Como vemos para esta forma de hacerlo necesitamos la sentencia **static**. **Se debe evitar el uso de static y el devolver desde una función direcciones de variables locales. Se debe hacer esto por medio del uso de punteros (pasaje por referencia).**

```
#include <stdio.h>
#include <stdlib.h>
#define tamano 25

short *carga(void);

int main(void)
{
    short i;
    short *V;

    V=carga();

    for(i=0; i<tamano; i++)
        printf("%hd ",V[i]);
    printf("\n\n");
    return 0;
}
```

```
short *carga(void)
{
    static short vector[tamano];
    short i;

    for(i=0; i<tamano; i++)
        vector[i]= i+1;

    return vector;
}
```

Buffer circular

Un buffer basicamente es en general un vector de bytes esto es en C un vector de datos tipo char. El vector se va llenando y descargando en el mismo sentido. Si llega al final del mismo vuelve a la posicion cero y vuelve a realizar la vuelta. Existen dos politicas básicas para el manejo de los buffers circulares.

- Si el buffer esta lleno indica esta situacion y no permite que se escriba un nuevo dato. No se puede perder ningun dato almacenado.
- Si el buffer esta lleno a la llegada de un nuevo dato sobreescribe. Esta situacion se puede dar en el caso de estar recibiendo video o sonido en vivo donde perder un dato no es un problema.

Los buffers circulares, tambien llamados colas circulares ya que lo primero que llega es lo primero que se saca, tienen dos funciones básicas, una de carga y otra de descarga y manejan dos indices, uno que apunta al lugar donde se debe escribir y otro que indica cual es el primero que se debe leer.

Se muestra a continuacion un ejemplo de buffer circular donde no se permite seguir escribiendo hasta que no exista al menos un espacio “vacio”. Tambien se verifica que no se quiera sacar datos cuando el buffer se encuentra vacio. Este tipo de estructura de dato es muy importante en programacion.

```

#include<conio.h>
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>

void carga(short *, short *, short *, short *);
void descarga(short *, short *, short *, short *);

int main(void)
{
    short N=1;
    char tecla='U';
    short buffer[10];
    unsigned short cantidad=0; //cantidad de datos en el buffer
    unsigned short icarga=0, idescarga=0; //indices de carga y descarga

    printf("Tecla E para cargar, tecla S para descargar, tecla Q para salir\n\n");

    while(N)
    {
        if(kbhit())//kbhit da cero si no hay tecla pulsada y distinto de cero si la hay
        {
            tecla=getch();
            if(toupper(tecla)=='E')
                carga(buffer,&icarga, &idescarga,&cantidad);
            if(toupper(tecla)=='S')
                descarga(buffer,&icarga, &idescarga,&cantidad);
            if(toupper(tecla)=='Q')
                N=0;
        }
    }
    return 0;
}

```

```

void carga(short *buffer, short *icarga, short *idescarga, short *cantidad)
{
    if(*cantidad<10)
    {
        printf("Ingrese un valor: ");
        scanf("%hd",&buffer[*icarga]);
        (*icarga)++;
        if(*icarga==10)
            *icarga=0;
        (*cantidad)++;
    }
    else
        printf("Buffer lleno\n");
}

void descarga(short *buffer, short *icarga, short *idescarga, short *cantidad)
{
    if(*cantidad>0)
    {
        printf("%hd\n",buffer[*idescarga]);
        (*idescarga)++;
        if(*idescarga==10)
            *idescarga=0;
        (*cantidad)--;
    }
    else
        printf("Buffer vacio\n");
}

```

Archivos en C

Archivo: Definición

Según el diccionario de la Real Academia Española

- **Edificio o local donde se conservan los documentos ordenados y clasificados que produce una institución, personalidad, etc., en el ejercicio de sus funciones o actividades.**
- **Mueble o caja que sirve para guardar documentos o fichas de manera ordenada.**

En el campo de la informática, se llama “archivo” al **elemento de información compuesto por una suma de registros** (combinaciones de bytes). Llevan este nombre por ser los equivalentes digitalizados de los archivos antes descriptos. Tanto es así que muchos de los archivos “en papel” se están actualmente digitalizando, para reducir su tamaño físico y facilitar su organización y búsqueda. Los archivos informáticos, en general, tienen algunas características en común:

- **Nombre.** Cada archivo es identifiable con un nombre, que no puede coincidir con otro que esté en la misma ubicación.
- **Extensión.** Los archivos llevan una extensión opcional, que muchas veces indica su formato.
- **Tamaño.** Como se dijo, están compuestos por una serie de bytes que determinan su tamaño. Puede alcanzar kilobytes, megabytes, gigabytes.
- **Descripción.** Además del nombre y la extensión, suelen tener otras características. Dentro de estas características puede aparecer la protección del archivo, lo que significa el permiso limitado para la lectura o modificación.

- **Ubicación.** Todos los archivos pertenecen a determinado lugar en la computadora (o circunstancialmente fuera de ella), el llamado espacio de almacenamiento. La mayoría se encuentran almacenados en discos rígidos, que están ordenados jerárquicamente en carpetas y subcarpetas. Existe necesariamente una ruta de acceso hacia ese lugar, que comienza con el disco al que se hace referencia (C:, D:).
- **Formato.** El modo en que el archivo será interpretado depende de su formato, entre los que están los formatos de texto, ejecutable, de datos, de imagen, de audio, de video, entre muchísimos otros.

En informática los archivos se dividen en dos tipos:

- **Archivos de texto**
- **Archivos binarios**

Un archivo ejecutable (**.com** o **.exe** bajo Windows) será siempre de tipo binario. Un archivo de configuración de un programa podrá ser de texto o binario. Un archivo de audio o video será siempre binario.

Tendremos funciones que se usan para ambos tipos de archivo tales como `fopen`, `feof` y `fclose` y otras que son exclusivas de cada tipo de archivo.

Archivos de texto

En los archivos de texto hay, en general, una “conversión” de la información. Un entero corto (short) se almacena en dos bytes en memoria bajo el convenio complemento a 2. Ahora, si lo almaceno en un archivo de texto, este entero será convertido a cadena de caracteres con lo que puede ocupar uno o mas bytes según de que valor numérico estemos hablando.

Ej: short A=2;

Ocupa en memoria por ser un short 2 bytes
La cadena de caracteres, sin el NULL es de 1 byte

Ej: short B=21687;

Ocupa en memoria por ser un short 2 bytes
La cadena de caracteres, sin el NULL es de 5 bytes

Ej: short C=-245;

Ocupa en memoria por ser un short 2 bytes
La cadena de caracteres, sin el NULL es de 4 bytes (hay que incluir el ASCII del signo negativo).

Para resumir, como short siempre serán 2 bytes NO IMPORTANDO EL CONTENIDO DE LA VARIABLE. Pero como texto será variable la cantidad de caracteres ASCII que se almacenan. Se debe tener en cuenta que el signo menos y el punto decimal de un valor real, también se deben tener en cuenta en la suma final de cuanto ocupan en un archivo de texto. Obviamente, el NULL, también lo podremos almacenar en el archivo. Es un carácter de la tabla ASCII.

Veamos otro ejemplo:

Float PI=3.1415927

Como binario ocupará 4 bytes siempre.

Como texto, la cantidad de cifras que le hayamos puesto, en este caso son 9, 8 cifras mas el punto decimal.

La ventaja del archivo de texto se podría decir que está en que es muy fácil de leer y editar. Desde el notepad de Windows, el Vi de Unix/Linux, hasta el Word de Microsoft, software libre como Writer de LibreOffice, etc. Aclarar que estamos hablando de texto plano, esto es solo los caracteres del texto. Aquí no existe si la letra es arial, de tal tamaño o de tal color. Es solo el texto sin ningún otro atributo. Si uso Word se debe almacenar como “solo texto”.

Los editores que usamos para programar (escribir el archivo fuente) en C son solo texto.

Con atributo se deberá pensar en un archivo binario.

El ASCII contiene caracteres especiales que representan fin de línea, fin de archivo, comienzo de texto, etc. Eso significa que, en general, esa combinación de unos y ceros, no puede ser usada para otra cosa que no sea el especificado por el código ASCII. Por ejemplo tenemos el LF, CR, NULL, ESC, DEL de uso actual. Otros han quedado obsoletos, ya que el ASCII se creó con el objetivo de transmitir información entre computadoras y terminales, e incluso controlar teletipos y otros sistemas similares.

Como comentario, hoy existe el UNICODE, que incluye todos los idiomas y lenguas conocidas, incluso las muertas. El C también permite el manejo de UNICODE.

Archivo binario

El archivo binario es el que almacena datos, en general, sin ningún tipo de conversión. Se guardaría tal cual está en memoria.

Es necesario para su correcta “lectura”, saber exactamente, y remarco, exactamente, como fue almacenada la información. Ejemplos son los formatos de música mp3, ogg o los mpg o mp4 de video, o los varios distintos formatos fotográficos. También los archivos ejecutables exe y com además de las dll. Sin el correcto conocimiento de la estructura del archivo no es posible su correcta “lectura”. Obviamente el software que los crea también es muy diferente al de los archivos de texto.

En nuestros ejemplos la “unidad de almacenamiento” será la estructura. Pero también podrá ser “cualquier bloque de memoria”. Por ejemplo podríamos almacenar los dos bytes de un short o los cuatro bytes de un float.

Funciones básicas para el manejo de archivos en C (texto y binarios)

- **fopen**

- Header: stdio.h
- Prototipo: FILE * fopen (const char * filename, const char * mode);

Abre el archivo cuyo nombre se especifica en el nombre de archivo (filename, que es un string) del parámetro y lo asocia con un flujo(stream) que puede identificarse en futuras operaciones mediante el puntero FILE devuelto.

El parámetro de modo (mode, que es un string) define las operaciones que se permiten en el flujo (stream) y cómo se realizan.

La secuencia devuelta está completamente almacenada (fully buffered) de forma predeterminada.

El puntero FILE devuelto puede desasociarse del archivo llamando a fclose o freopen. Todos los archivos abiertos se cierran automáticamente al finalizar el programa.

El entorno de ejecución admite al menos archivos FOPEN_MAX abiertos simultáneamente.

Ejemplo:

```
FILE *archivo;  
  
archivo=fopen("miarchivo.txt","rt");
```

- **feof**

- Header: stdio.h
- Prototipo: int feof (FILE * stream);

Comprueba si el indicador de fin de archivo asociado con el flujo está configurado, devolviendo un valor diferente de cero si es así. Devuelve cero si no llegó al fin de archivo.

Este indicador generalmente se establece mediante una operación previa en el flujo que intentó leer al final del archivo o después.

Tenga en cuenta que el indicador de posición interno del flujo puede apuntar al final del archivo para la siguiente operación, pero aún así, el indicador de fin de archivo puede no establecerse hasta que una operación intente leer en ese punto.

Este indicador se borra mediante una llamada a un aclarador, rebobinado, fseek, fsetpos o freopen. Aunque si el indicador de posición no se reposiciona en una llamada de este tipo, es probable que la próxima operación de E / S vuelva a configurar el indicador.

Ejemplo:

```
FILE *archivo;  
  
while(!feof(archivo))  
{  
  
}
```

- **fclose**

- Header: stdio.h
- Prototipo: int fclose (FILE * stream);

Cierra el archivo asociado con el flujo y lo disocia.

Todos los búferes internos asociados con la secuencia se disocian y se vacían: el contenido de cualquier búfer de salida no escrito se escribe y el contenido de cualquier búfer de entrada no leído se descarta.

Incluso si la llamada falla, la secuencia pasada como parámetro ya no estará asociada con el archivo ni con sus búferes.

Ejemplo:

```
FILE *archivo;  
  
fclose(archivo);
```

- **fseek**

- Header: stdio.h
- Prototipo: int fseek (FILE * stream, long int offset, int origin);

Establece el indicador de posición asociado con el flujo a una nueva posición.

Para las secuencias abiertas en modo binario, la nueva posición se define agregando desplazamiento a una posición de referencia especificada por origen.

Para las secuencias abiertas en modo de texto, el desplazamiento será cero o un valor devuelto por una llamada previa a ftell, y el origen necesariamente será SEEK_SET.

Si se llama a la función con otros valores para estos argumentos, el soporte depende del sistema particular y la implementación de la biblioteca (no portátil).

El indicador interno de fin de archivo de la secuencia se borra después de una llamada exitosa a esta función, y todos los efectos de las llamadas anteriores a ungetc en esta secuencia se eliminan.

En los flujos abiertos para actualización (lectura + escritura), una llamada a fseek permite cambiar entre lectura y escritura.

Posición (origin) utilizada como referencia para el desplazamiento. Se especifica mediante una de las siguientes constantes definidas en <stdio.h> exclusivamente para usarse como argumentos para esta función:

Constante	Posición de referencia
SEEK_SET	Comienzo del archivo
SEEK_CUR	Actual posición dentro del archivo
SEEK_END	Final de archivo *

* Se permite que las implementaciones de la biblioteca no admitan de manera significativa SEEK_END (por lo tanto, el código que lo usa no tiene portabilidad estándar real).

- **fseek**

- Header: stdio.h
- Prototipo: long int fseek (FILE * stream , int offset , int whence);

Devuelve el valor actual del indicador de posición del flujo (stream).

Para las secuencias binarias, este es el número de bytes desde el comienzo del archivo.

Para las secuencias de texto, el valor numérico puede no ser significativo, pero aún se puede usar para restaurar la posición a la misma posición más tarde usando fseek (si hay caracteres devueltos usando ungetc aún pendientes de lectura, el comportamiento no está definido).

- **rewind**

- Header: stdio.h
- Prototipo: void rewind (FILE * stream);

Establece el indicador de posición asociado con el flujo(stream) al comienzo del archivo.

Los indicadores internos de fin de archivo y error asociados a la secuencia se borran después de una llamada exitosa a esta función, y todos los efectos de las llamadas anteriores a ungetc en esta secuencia se eliminan.

En las transmisiones abiertas para actualización (lectura + escritura), una llamada para rebobinar permite cambiar entre lectura y escritura.

- **ungetc (se explica porque fue referenciada varias veces... no la vamos a usar)**

- **Header: stdio.h**
- **Prototipo: int ungetc (int character, FILE * stream);**

Un carácter se vuelve a colocar virtualmente en una secuencia de entrada, disminuyendo su posición de archivo interno como si se hubiera deshecho una operación getc previa.

Este carácter puede o no ser el que se lee del flujo en la operación de entrada anterior. En cualquier caso, el siguiente carácter recuperado de la secuencia es el carácter pasado a esta función, independientemente del original.

Sin embargo, tenga en cuenta que esto solo afecta las operaciones de entrada adicionales en esa secuencia, y no el contenido del archivo físico asociado con él, que no se modifica por ninguna llamada a esta función.

Algunas implementaciones de la biblioteca pueden admitir que esta función se llame varias veces, haciendo que los caracteres estén disponibles en el orden inverso en el que se volvieron a colocar. Aunque este comportamiento no tiene garantías de portabilidad estándar, y las llamadas adicionales simplemente pueden fallar después de cualquier número de llamadas más allá de la primera.

Si tiene éxito, la función borra el indicador de flujo de fin de archivo (si estaba configurado actualmente) y disminuye su indicador interno de posición de archivo si funciona en modo binario; En modo texto, el indicador de posición tiene un valor no especificado hasta que todos los caracteres devueltos con ungetc hayan sido leídos o descartados.

Una llamada a fseek, fsetpos o rewind en la secuencia descartará cualquier carácter previamente puesto de nuevo con esta función.

Si el argumento pasado para el parámetro de caracteres es EOF, la operación falla y la secuencia de entrada permanece sin cambios.

Vamos a ver algunos ejemplos de como usar archivos en C

- Como leer un archivo de texto:

Una forma de hacerlo es...

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

int main(void)
{
    FILE *archivo;
    char caracter;

    archivo = fopen("buffer.bak","rt");//rt= leer/texto

    if(archivo == NULL)
    {
        printf("ERROR");
        exit(0);
    }

    do{
        caracter = fgetc(archivo);
        printf("%c",caracter);
    } while (caracter != EOF); //lee hasta encontrar el character End Of File

    fclose(archivo);

    return 0;
}
```

- Como crear y escribir en un archivo de texto: una forma de hacerlo...

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

int main(void)
{
    FILE *archivo;
    char caracter;
    char msg[] = "Hola...\\n\\nEsto contiene el archivo de texto!!\\n\\n";
    int i = 0;

    //si el archivo existe lo sobreescribe
    //hay que crear una funcion que verifique esto de ser necesario

    archivo = fopen("mensaje.bak","wt");//escribe/texto

    if(archivo == NULL)
    {
        printf("ERROR de creacion");
        exit(0);//si hay error termina el programa
    }

    while (msg[i] != NULL)//lee texto hasta NULL
    {
        fputc(msg[i], archivo);//escribe caracter a caracter en el archivo.
        i++;
    }

    fclose(archivo);
    return 0;
}
```

- Otra forma de escribir en un archivo de texto: Si lo ejecuta verá que se obtiene el mismo resultado que con el ejemplo anterior.

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

int main(void)
{
    FILE *archivo;
    char msg[] = "Hola...\\n\\nEsto contiene el archivo de texto!!\\n\\n";
    int i = 0;

    //si el archivo existe lo sobreescrive
    //hay que crear una funcion que verifique esto de ser necesario

    archivo = fopen("mensaje.bak","wt");//escribe/texto

    if(archivo == NULL)
    {
        printf("ERROR de creacion");
        exit(0);//si hay error termina el programa
    }

    fputs(msg,archivo); //Escribe la cadena en el archivo

    fclose(archivo);
    return 0;
}
```

- Lee el archivo de texto creado en el ejemplo anterior

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

int main(void)
{
    FILE *archivo;
    char msg[100];
    int i = 0;

    archivo = fopen("mensaje.bak","rt");//lee/texto

    if(archivo == NULL)
    {
        printf("ERROR de lectura");
        exit(0);//si hay error termina el programa
    }

    while(!feof(archivo))
    {
        fgets(msg,100,archivo);
        printf("%s",msg);
        i++;
    }

    printf("La informacion fue leida en %hd veces\n\n",i);

    fclose(archivo);

    return 0;
}
```

- Lectura y escritura de una archivo binario

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

int main()
{
    short A=32;
    short B=2048;
    short C=-1;
    short D[5]={-8,65,25,-71,4096};

    short i;

    FILE *archivo1;
    FILE *archivo2;

    char nombrearchivo[]= "binario.bin";

    printf("\nCreamos un archivo de formato binario\n\n");

    archivo1=fopen(nombrearchivo,"wb");
    if(archivo1!=NULL)
    {
        fwrite(&A,sizeof(A),1,archivo1);
        fwrite(&B,sizeof(B),1,archivo1);
        fwrite(&C,sizeof(C),1,archivo1);
        fwrite(D,sizeof(D),1, archivo1);
        fclose(archivo1);
    }
    else
        printf("\nNO se puede crear archivo\n");
```

```

archivo1=fopen(nombrearchivo,"rb");
if(archivo1!=NULL)
{
    printf("\nLo abrimos para lectura en formato binario\n\n");

    while(!feof(archivo1))
    {
        fread(&A,sizeof(A),1,archivo1);
        fread(&B,sizeof(B),1,archivo1);
        fread(&C,sizeof(C),1,archivo1);
        fread(D,sizeof(D),1,archivo1);
    }

    fclose(archivo1); /* fuera del while!!! */

    printf("A= %hd\n",A);
    printf("B= %hd\n",B);
    printf("C= %hd\n",C);

    for(i=0;i<5;i++)
        printf("\nD[%d]= %hd",i,D[i]);
    }
else
    printf("\nNO se puede leer archivo\n");

return 0;
}

```

Si usamos el programa notepad++ vemos como quedaron archivados nuestros valores numéricos en formato binario. Cada entero ocupa 2 bytes. Queda al alumno verificar si están correctamente almacenados. Recuerde que los datos son enteros y en complemento a 2.

Dirección	0	1	2	3
00000000	00100000	00000000	00000000	00001000
00000004	11111111	11111111	11111000	11111111
00000008	01000001	00000000	00011001	00000000
0000000C	10111001	11111111	00000000	00010000

¿Era lo esperado? ¿Cómo se puede verificar si están bien los datos almacenados?

Estructura FILE

Si bien son muy similares y en general contienen los mismo campos, varian de compilador a compilador y según sea el sistema operativo. Se muestra a titulo de ejemplo y puede no ser la que se esta usando.

```
typedef struct _iobuf
{
    char*   _ptr;
    int     _cnt;
    char*   _base;
    int     _flag;
    int     _file;
    int     _charbuf;
    int     _bufsiz;
    char*   _tmpfname;
} FILE;
```

Campo Flags de la estructura FILE: De nuevo son similares de compilador a compilador. Tenga en cuenta que estos valores tal vez no corresponden al compilador que usted esta usando. Es nuevamente solo a modo de ejemplo.

```
#define __SLBF 0x0001      /* line buffered */
#define __SNBF 0x0002        /* unbuffered */
#define __SRD 0x0004         /* OK to read */
#define __SWR 0x0008         /* OK to write */
#define __SRW 0x0010         /* open for reading & writing */
#define __SEOF 0x0020         /* found EOF */
#define __SERR 0x0040         /* found error */
#define __SMBF 0x0080         /* _buf is from malloc */
#define __SAPP 0x0100         /* fdopen()ed in append mode */
#define __SSTR 0x0200         /* this is an sprintf/snprintf string */
#define __SOPT 0x0400         /* do fseek() optimisation */
#define __SNPT 0x0800         /* do not do fseek() optimisation */
#define __SOFF 0x1000         /* set iff _offset is in fact correct */
#define __SMOD 0x2000         /* true => fgetln modified _p text */
#define __SALC 0x4000         /* allocate string space dynamically */
#define __SIGN 0x8000         /* ignore this file in _fwalk */
```

Tabla de modos de archivos

MODO (O_TEXT)	ACCIÓN	TIPO	¿QUÉ DEVUELVE?			EFECTO
			Éxito	Archivo no existente	Error de acceso	
r	L	Texto	fp	NULL	NULL	No cambia
w	E	Texto	fp	fp	NULL	Se pierde
a	E	Texto	fp	fp	NULL	Se agrega
r+	L/E	Texto	fp	NULL	NULL	Se modifica
w+	E/L	Texto	fp	fp	NULL	Se pierde
a+	E/L	Texto	fp	fp	NULL	Se agrega
rb	L	Binario	fp	NULL	NULL	No cambia
wb	E	Binario	fp	fp	NULL	Se pierde
ab	E	Binario	fp	fp	NULL	Se agrega
rb+	L/E	Binario	fp	NULL	NULL	Se modifica
wb+	E/L	Binario	fp	fp	NULL	Se pierde
ab+	E/L	Binario	fp	fp	NULL	Se agrega

Ejemplo de archivos con vectores

Ejemplo 1:

```
#include <stdio.h>
#include <stdlib.h>
/**Como guardar y leer un vector numerico completo en un archivo binario*/
int main()
{
    short vector_inicial[]={1,2,3,4,5,6,7,8,9,10};

    short cantidad=sizeof(vector_inicial)/sizeof(vector_inicial[0]);
    short vector_leido[cantidad];
    FILE *archivo;
    short indice;
    archivo=fopen("Prueba001.bin","wb");
    if(archivo!=NULL)
    {
        fwrite(vector_inicial,sizeof(vector_inicial),1,archivo);
    }
    fclose(archivo);

    archivo=fopen("Prueba001.bin","rb");
    if(archivo!=NULL)
    {
        fread(vector_leido,sizeof(vector_leido),1,archivo);
    }
    fclose(archivo);
    for(indice=0;indice<cantidad;indice++)
        printf("%hd\n",vector_leido[indice]);

    return 0;
}
```

Ejemplo 2: Como leer y escribir un vector de estructuras

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

struct datos{
    short edad;
    char nombre[35];
};

/**Como guardar un vector de estructura en un archivo binario*/

int main()
{
    struct datos vector_inicial[]={{"1","Pepe"}, {"2","Juan"}, {"3","Ana"}, {"4","Carlos"}, {"5","Lucia"}, {"6","Marta"}};

    short cantidad=sizeof(vector_inicial)/sizeof(vector_inicial[0]);
    struct datos vector_leido[cantidad];
    FILE *archivo;
    short indice;

    archivo=fopen("Prueba001.bin","wb");

    if(archivo!=NULL)
    {
        for(indice=0;indice<cantidad;indice++)
        {
            fwrite(&vector_inicial[indice],sizeof(struct datos),1,archivo);
        }
    }

    fclose(archivo);
```

```

archivo=fopen("Prueba001.bin","rb");
if(archivo!=NULL)
{
    indice=0;
    fread(&vector_leido[indice],sizeof(struct datos),1,archivo);
    while(!feof(archivo))
    {
        printf("%hd ",vector_leido[indice].edad);
        printf("%s\n",vector_leido[indice].nombre);
        indice++;
        fread(&vector_leido[indice],sizeof(struct datos),1,archivo); //guardamos de a una estructura a la vez
    }
}

fclose(archivo);

cantidad=indice;

printf("\nCuantos se leyeron: %hd\n",indice);

for(indice=0;indice<cantidad;indice++)
    printf("%hd %s\n",vector_leido[indice].edad,vector_leido[indice].nombre);

return 0;
}

```

Ejemplo 3: Como leer y escribir un vector de estructuras (otra forma)

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

struct datos{
    short edad;
    char nombre[35];
};

/**Como guardar un vector de estructur completo en un archivo binario*/

int main()
{
    struct datos vector_inicial[]={{"1","Pepe"}, {"2","Juan"}, {"3","Ana"}, {"4","Carlos"}, {"5","Lucia"}, {"6","Marta"}};

    short cantidad=sizeof(vector_inicial)/sizeof(vector_inicial[0]);
    struct datos vector_leido[cantidad];
    FILE *archivo;
    short indice;

    archivo=fopen("Prueba001.bin","wb");
    if(archivo!=NULL)
    {
        fwrite(vector_inicial,sizeof(vector_inicial),1,archivo);
    }

    fclose(archivo);
```

```
archivo=fopen("Prueba001.bin","rb");

if(archivo!=NULL)
{
    fread(vector_leido,sizeof(vector_leido),1,archivo);
}
fclose(archivo);

for(indice=0;indice<cantidad;indice++)
{
    printf("%hd ",vector_leido[indice].edad);
    printf("%s\n",vector_leido[indice].nombre);
}

return 0;
}
```

Comparando las estructuras de control desde Assembler
(tema de lectura optativa. Es un tema avanzado y no es para gente que programa por primera vez)

Vamos a armar tablas donde se verá desde C y desde Assembler como quedan nuestras instrucciones. Esto nos permitirá visualizar mucho mejor las cosas y llegado el caso seleccionar usando un criterio técnico cual de las estructuras, en caso de poder usarse mas de una, es la mas apropiada para usar en mi código.

No es la idea aprender Assembler ya que nos llevaría todo un curso. Queda en usted, llegado el caso, comprender más en detalle los código que se publicarán.

Antes veremos una tabla con los distintos **saltos codicionales** (jump) en Assembler y su ámbito de aplicación (de nuevo, para servir únicamente de referencia). Como detalle **jmp** es un salto incondicional.

Tipo de salto	Ámbito de aplicación	Instrucción	Instrucción alternativa	Observación
Saltos condicionales aritméticos (se usan después de la instrucción cmp)	Aritmética signada (con números positivos, negativos y cero)	JL etiqueta	JNGE etiqueta	Salir a etiqueta si es menor.
		JLE etiqueta	JNG etiqueta	Salir a etiqueta si es menor o igual.
		JE etiqueta		Salir a etiqueta si es igual
		JNE etiqueta		Salir a etiqueta si es distinto.
		JGE etiqueta	JNL etiqueta	Salir a etiqueta si es mayor o igual.
		JG etiqueta	JNLE etiqueta	Salir a etiqueta si es mayor.
	Aritmética sin signo (con números positivos y cero)	JB etiqueta	JNAE etiqueta	Salir a etiqueta si es menor.
		JBE etiqueta	JNA etiqueta	Salir a etiqueta si es menor o igual.
		JE etiqueta		Salir a etiqueta si es igual.
		JNE etiqueta		Salir a etiqueta si es distinto.
		JAE etiqueta	JNB etiqueta	Salir a etiqueta si es mayor o igual.
		JA etiqueta	JNBE etiqueta	Salir a etiqueta si es mayor.
Saltos condicionales según el valor de los flags		JC etiqueta		Salir si hubo arrastre/préstamo (CF = 1).
		JNC etiqueta		Salir si no hubo arrastre/préstamo (CF = 0).
		JZ etiqueta		Salir si el resultado es cero (ZF = 1).
		JNZ etiqueta		Salir si el resultado no es cero (ZF = 0).
		JS etiqueta		Salir si el signo es negativo (SF = 1).
		JNS etiqueta		Salir si el signo es positivo (SF = 0).
		JP etiqueta	JPE etiqueta	Salir si la paridad es par (PF = 1).
		JNP etiqueta	JPO etiqueta	Salir si la paridad es impar (PF = 0).

Le estructura repetitiva while:

En C	En Assembler de Intel	Comentarios
<pre>int main(void) { short veces; veces=0; while(veces<15) veces++; return 0; }</pre>	<pre>main: push rbp mov rbp, rsp WORD PTR [rbp-2], 0 .L2: movzx eax, WORD PTR [rbp-2] add eax, 1 mov WORD PTR [rbp-2], ax WORD PTR [rbp-2], 14 .L3: cmp eax, 0 jle mov pop ret </pre> <p>Salvamos rsp en rbp</p>	<p>Los registros rbp y rsp son de 64bits. El rbp es un registro de uso general sin aplicación específica.</p> <p>El rsp es el “stack pointer” esto es un puntero que contiene una dirección dentro de la pila o stack. Las variables locales se almacenan dentro de la pila.</p> <p>La instrucción mov es “la asignación en Assembler”, y su sintaxis es destino, fuente.</p> <p>El [rbp-2] significa restar 2 al registro rbp. ¿Por qué 2?, bueno, porque el dato es un short que ocupa 2 bytes en memoria. Jmp salta a la etiqueta que corresponda incondicionalmente. En cambio el salto jle necesita una instrucción cmp (comparar). La instrucción add es la suma en Assembler.</p> <p>El registro eax es de 64bits mientras ax es de 16bits (como la variable veces).</p>

La estructura repetitiva do - while:

En C	En Assembler de Intel	Comentarios
<pre>int main(void) { short veces; veces=0; do{ veces++; }while(veces<15); return 0; }</pre>	<pre>main: push rbp mov rbp, rsp mov WORD PTR [rbp-2], 0 .L2: movzx eax, WORD PTR [rbp-2] add eax, 1 mov WORD PTR [rbp-2], ax cmp WORD PTR [rbp-2], 14 jle .L2 mov eax, 0 pop rbp ret</pre>	<p>En el próximo cuadro veremos que a nivel de Assembler son “muy similares”, salvo un salto que se hace.</p> <p>Salta a L2 si es menor o igual a 14...</p>

Comparacion de las estructuras repetitivas **while** vs **do – while** desde assembler

While	Do - while	Comentarios
<pre>main: push rbp mov rbp, rsp mov WORD PTR [rbp-2], 0 jmp .L2 .L3: movzx eax, WORD PTR [rbp-2] add eax, 1 mov WORD PTR [rbp-2], ax cmp WORD PTR [rbp-2], 14 jle .L3 mov eax, 0 pop rbp ret</pre>	<pre>main: push rbp mov rbp, rsp mov WORD PTR [rbp-2], 0 .L3: movzx eax, WORD PTR [rbp-2] add eax, 1 mov WORD PTR [rbp-2], ax cmp WORD PTR [rbp-2], 14 jle .L3 mov eax, 0 pop rbp ret</pre>	<p>Es la única diferencia que hay a nivel de Assembler el salto incondicional <code>jmp .L2</code>. El resto es exactamente igual!!</p>

La estructura repetitiva for:

En C	En Assembler de Intel	Comentarios
<pre>int main(void) { short veces; for(veces=0;veces<15;veces++); return 0; }</pre>	<pre>main: push rbp mov rbp, rsp mov WORD PTR [rbp-2], 0 jmp .L2 .L3: movzx eax, WORD PTR [rbp-2] add eax, 1 mov WORD PTR [rbp-2], ax .L2: cmp WORD PTR [rbp-2], 14 jle .L3 mov eax, 0 pop rbp ret</pre>	¿No vimos antes este código??

Comparacion de las estructuras repetitivas for y while

While	for	Comentarios
<pre> main: push rbp mov rbp, rsp mov WORD PTR [rbp-2], 0 jmp .L2 .L3: movzx eax, WORD PTR [rbp-2] add eax, 1 mov WORD PTR [rbp-2], ax cmp WORD PTR [rbp-2], 14 jle .L3 mov eax, 0 pop rbp ret </pre>	<pre> main: push rbp mov rbp, rsp mov WORD PTR [rbp-2], 0 jmp .L2 .L3: movzx eax, WORD PTR [rbp-2] add eax, 1 mov WORD PTR [rbp-2], ax cmp WORD PTR [rbp-2], 14 jle .L3 mov eax, 0 pop rbp ret </pre>	Como dijimos: “for y while” son lo mismo escritos en forma distinta en C, no es Assembler, que son los mismos códigos.

La estructura de selección if:

If en C	En Assembler de Intel	Comentarios
<pre>#include<stdio.h> int main(void) { short seleccion=1; if(seleccion==1) printf("Es 1"); if(seleccion==2) printf("Es 2"); if (seleccion==3) printf("Es 3"); return 0; }</pre>	<pre>.LC0: .string "Es 1" .LC1: .string "Es 2" .LC2: .string "Es 3" main: push rbp mov rbp, rsp sub rsp, 16 mov WORD PTR [rbp-2], 1 cmp WORD PTR [rbp-2], 1 jne .L2 mov edi, OFFSET FLAT:.LC0 mov eax, 0 call printf .L2: cmp WORD PTR [rbp-2], 2 jne .L3 mov edi, OFFSET FLAT:.LC1 mov eax, 0 call printf .L3: cmp WORD PTR [rbp-2], 3 jne .L4 mov edi, OFFSET FLAT:.LC2 mov eax, 0 call printf .L4: mov eax, 0 leave ret</pre>	Jne: salte si no es igual

La estructura de selección multiple switch – case:

Switch-case en C	En Assembler de Intel	Comentarios
<pre>#include<stdio.h> int main(void) { short seleccion=1; switch(seleccion) { case 1: printf("Es 1"); break; case 2: printf("Es 2"); break; case 3: printf("Es 3"); break; } return 0; }</pre>	<pre>.LC0: .string "Es 1" .LC1: .string "Es 2" .LC2: .string "Es 3" main: push rbp mov rbp, rsp sub rsp, 16 mov WORD PTR [rbp-2], 1 movsx eax, WORD PTR [rbp-2] cmp eax, 3 je .L2 cmp eax, 3 jg .L3 cmp eax, 1 je .L4 cmp eax, 2 je .L5 jmp .L3 .L4: mov edi, OFFSET FLAT:.LC0 mov eax, 0 call printf jmp .L3 .L5: mov edi, OFFSET FLAT:.LC1 mov eax, 0 call printf jmp .L3</pre>	<p>Je: salte si es igual Jg: salte si es mayor Jmp: salto incondicional</p>

```
.L2:  
    mov    edi, OFFSET FLAT:.LC2  
    mov    eax, 0  
    call   printf  
    nop  
.L3:  
    mov    eax, 0  
    leave  
    ret
```

Comparativa de las estructuras de selección if y switch – case:

If en assembler	Switch – case en assembler	Comentarios
<pre>.LC0: .string "Es 1" .LC1: .string "Es 2" .LC2: .string "Es 3" main: push rbp mov rbp, rsp sub rsp, 16 mov WORD PTR [rbp-2], 1 cmp WORD PTR [rbp-2], 1 jne .L2 mov edi, OFFSET FLAT:.LC0 mov eax, 0 call printf .L2: cmp WORD PTR [rbp-2], 2 jne .L3 mov edi, OFFSET FLAT:.LC1 mov eax, 0 call printf .L3: cmp WORD PTR [rbp-2], 3 jne .L4 mov edi, OFFSET FLAT:.LC2 mov eax, 0 call printf .L4: mov eax, 0 leave ret</pre>	<pre>.LC0: .string "Es 1" .LC1: .string "Es 2" .LC2: .string "Es 3" main: push rbp mov rbp, rsp sub rsp, 16 mov WORD PTR [rbp-2], 1 movsx eax, WORD PTR [rbp-2] cmp eax, 3 je .L2 cmp eax, 3 jg .L3 cmp eax, 1 je .L4 cmp eax, 2 je .L5 jmp .L3 .L4: mov edi, OFFSET FLAT:.LC0 mov eax, 0 call printf jmp .L3 .L5: mov edi, OFFSET FLAT:.LC1 mov eax, 0 call printf jmp .L3</pre>	Es difícil viendo los códigos de Assembler saber cual es mas eficiente.

	.L2: mov edi, OFFSET FLAT:.LC2 mov eax, 0 call printf nop .L3: mov eax, 0 leave ret	
--	--	--

Otra comparativa de if y switch – case desde el Assembler.

if	Switch-case
<pre>main.o: file format pe-x86-64 Disassembly of section .text: 0000000000000000 <main>: #include<stdio.h> int main(void) { 0: 55 push rbp 1: 48 89 e5 mov rbp,rsp 4: 48 83 ec 30 sub rsp,0x30 8: e8 00 00 00 00 call d <main+0xd> short seleccion=1; d: 66 c7 45 fe 01 00 mov WORD PTR [rbp-0x2],0x1 if(seleccion==1) 13: 66 83 7d fe 01 cmp WORD PTR [rbp-0x2],0x1 18: 75 0c jne 26 <main+0x26> printf("Es 1"); 1a: 48 8d 0d 00 00 00 00 lea rcx,[rip+0x0] # 21 <main+0x21> 21: e8 00 00 00 00 call 26 <main+0x26> if(seleccion==2) 26: 66 83 7d fe 02 cmp WORD PTR [rbp-0x2],0x2 2b: 75 0c jne 39 <main+0x39> printf("Es 2"); 2d: 48 8d 0d 05 00 00 00 lea rcx,[rip+0x5] # 39 <main+0x39> 34: e8 00 00 00 00 call 39 <main+0x39> if (seleccion==3) 39: 66 83 7d fe 03 cmp WORD PTR [rbp-0x2],0x3 3e: 75 0c jne 4c <main+0x4c> printf("Es 3"); 40: 48 8d 0d 0a 00 00 00 lea rcx,[rip+0xa] # 51 <main+0x51> 47: e8 00 00 00 00 call 4c <main+0x4c> return 0; 4c: b8 00 00 00 00 mov eax,0x0 } 51: 48 83 c4 30 add rsp,0x30 55: 5d pop rbp 56: c3 ret</pre>	<pre>main.o: file format pe-x86-64 Disassembly of section .text: 0000000000000000 <main>: #include<stdio.h> int main(void) { 0: 55 push rbp 1: 48 89 e5 mov rbp,rsp 4: 48 83 ec 30 sub rsp,0x30 8: e8 00 00 00 00 call d <main+0xd> short seleccion=1; d: 66 c7 45 fe 01 00 mov WORD PTR [rbp-0x2],0x1 switch(seleccion) 13: 0f bf 45 fe movsx eax,WORD PTR [rbp-0x2] 17: 83 f8 02 cmp eax,0x2 1a: 74 18 je 34 <main+0x34> 1c: 83 f8 03 cmp eax,0x3 1f: 74 21 je 42 <main+0x42> 21: 83 f8 01 cmp eax,0x1 24: 75 29 jne 4f <main+0x4f> { case 1: printf("Es 1"); 26: 48 8d 0d 00 00 00 00 lea rcx,[rip+0x0] # 2d <main+0x2d> 2d: e8 00 00 00 00 call 32 <main+0x32> break; 32: eb 1b jmp 4f <main+0x4f> case 2: printf("Es 2"); 34: 48 8d 0d 05 00 00 00 lea rcx,[rip+0x5] # 40 <main+0x40> 3b: e8 00 00 00 00 call 40 <main+0x40> break; 40: eb 0d jmp 4f <main+0x4f> case 3: printf("Es 3"); 42: 48 8d 0d 0a 00 00 00 lea rcx,[rip+0xa] # 53 <main+0x53> 49: e8 00 00 00 00 call 4e <main+0x4e> break; 4e: 90 nop } return 0; 4f: b8 00 00 00 00 mov eax,0x0 } 54: 48 83 c4 30 add rsp,0x30 58: 5d pop rbp 59: c3 ret</pre>

Anexo 1

Algoritmos básicos en C

Cálculo de un promedio con datos numéricos

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

#define cantidad 10

int main()
{
    short dato;
    float promedio;
    short suma=0, i=0;

    while(i<cantidad)
    {
        printf("Ingresar un entero: ");
        scanf("%hd",&dato);
        suma=suma+dato;
        i++;
    }

    promedio=(float)suma/cantidad;

    printf("\nEl promedio es: %f\n",promedio);

    return 0;
}
```

Cálculo de un promedio con datos almacenados en un vector numérico

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

#define cantidad 10

int main()
{
    short dato[cantidad]={17,12,45,12,3,6,-2,-11,32,15};
    float promedio;
    short suma=0, i=0;

    while(i<cantidad)
    {
        suma=suma+dato[i];
        i++;
    }

    promedio=(float)suma/cantidad;

    printf("\nEl promedio es: %f\n",promedio);

    return 0;
}
```

Máximos y mínimos: forma 1 de calcularlos

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

int main()
{
    short dato;
    short maximo, minimo;
    short i;

    for(i=0;i<10;i++) /se ingresan 10 numeros por teclado
    {
        printf("Ingrese un dato: ");
        scanf("%hd",&dato);

        if(i==0) //si i es cero es el primero
        {
            maximo=dato; //impongo el maximo
            minimo=dato; //impongo el minimo
        }
        else
        {
            if(maximo<dato)
                maximo=dato;
            if(minimo>dato)
                minimo=dato;
        }
    }

    printf("\nEl minimo es: %hd\n",minimo);
    printf("\nEl maximo es: %hd\n",maximo);

    return 0;
}
```

Máximos y mínimos: forma 2 de calcularlos

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

int main()
{
    short dato;
    short maximo, minimo;
    short i;

    for(i=0;i<10;i++) //se ingresan 10 numeros por teclado
    {
        printf("Ingrese un dato: ");
        scanf("%hd",&dato);

        if(i==0 || maximo<dato)
        {
            maximo=dato; //nuevo maximo
        }
        if(i==0 || minimo>dato)
        {
            minimo=dato; //nuevo minimo
        }
    }

    printf("\nEl minimo es: %hd\n",minimo);
    printf("\nEl maximo es: %hd\n",maximo);

    return 0;
}
```

Máximos y mínimos: forma 3 de calcularlos

El máximo y el mínimo se busca hasta que se ingrese cero por teclado, el que no se toma en cuenta.

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

int main()
{
    short dato;
    short maximo, minimo;
    short i=0;

    printf("Ingrese un dato: ");
    scanf("%hd",&dato);

    while(dato !=0)
    {
        if(i==0 || maximo<dato)
        {
            maximo=dato; //nuevo maximo
        }
        if(i==0 || minimo>dato)
        {
            minimo=dato; //nuevo minimo
            i=1;
        }
        printf("Ingrese un dato: ");
        scanf("%hd",&dato);
    }
    printf("\nEl minimo es: %hd\n",minimo);
    printf("\nEl maximo es: %hd\n",maximo);

    return 0;
}
```

Máximos y mínimos con vector numérico

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

#define cantidad 10

int main()
{
    short dato[cantidad]={17,12,45,12,3,6,-2,-11,32,15};
    short maximo, minimo;
    short i=0;

    while(i<cantidad)
    {
        if(i==0 || maximo<dato[i])
        {
            maximo=dato[i]; //nuevo maximo
        }
        if(i==0 || minimo>dato[i])
        {
            minimo=dato[i]; //nuevo minimo
        }
        i++;
    }

    printf("\nEl minimo es: %hd\n",minimo);
    printf("\nEl maximo es: %hd\n",maximo);

    return 0;
}
```

Una forma de comparar el contenido de dos strings usando switch - case

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

int main()
{
    char texto1[]{"Halas"};
    char texto2[]{"Holas"};

    short diferencia=0;
    short i=0;

    do{
        switch(texto1[i]-texto2[i])
        {
            case 0:
                break;
            default:
                diferencia=texto1[i]-texto2[i];
                break;
        }
        i++;
    }while((texto1[i]!=0 || texto2[i]!=0) && !diferencia);

    if(diferencia==0)
        printf("Iguales...");
    else
        printf("Distintos...");

    return 0;
}
```

Anexo 2

Tablas, diagramas
y
otros anexos

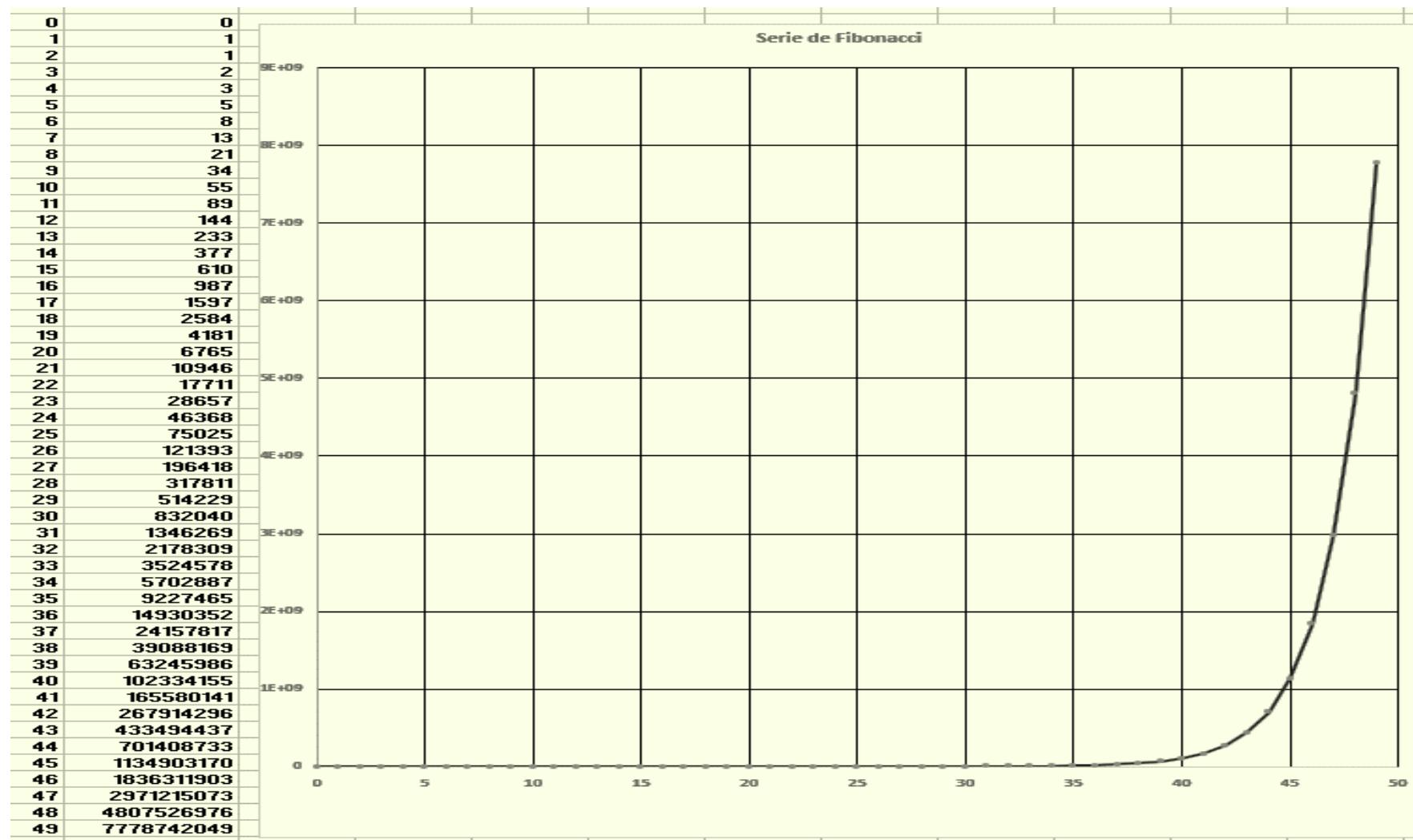
Tabla ASCII

Caracteres ASCII de control		Caracteres ASCII imprimibles						ASCII extendido					
00	NULL (carácter nulo)	32	espacio	64	@	96	'	128	ç	160	á	192	l
01	SOH (inicio encabezado)	33	!	65	A	97	a	129	ü	161	í	193	ł
02	STX (inicio texto)	34	"	66	B	98	b	130	é	162	ó	194	ł
03	ETX (fin de texto)	35	#	67	C	99	c	131	â	163	ú	195	ł
04	EOT (fin transmisión)	36	\$	68	D	100	d	132	ã	164	ñ	196	-
05	ENQ (consulta)	37	%	69	E	101	e	133	à	165	ñ	197	+
06	ACK (reconocimiento)	38	&	70	F	102	f	134	à	166	º	198	ä
07	BEL (timbre)	39	,	71	G	103	g	135	ç	167	º	199	å
08	BS (retroceso)	40	(72	H	104	h	136	ê	168	¿	200	ł
09	HT (tab horizontal)	41)	73	I	105	i	137	è	169	®	201	ł
10	LF (nueva línea)	42	*	74	J	106	j	138	è	170	¬	202	ł
11	VT (tab vertical)	43	+	75	K	107	k	139	í	171	½	203	ł
12	FF (nueva página)	44	,	76	L	108	l	140	í	172	¼	204	ł
13	CR (retorno de carro)	45	-	77	M	109	m	141	í	173	ı	205	=
14	SO (desplaza afuera)	46	.	78	N	110	n	142	À	174	«	206	‡
15	SI (desplaza adentro)	47	/	79	O	111	o	143	Á	175	»	207	¤
16	DLE (esc.vínculo datos)	48	0	80	P	112	p	144	É	176	„	208	ð
17	DC1 (control disp. 1)	49	1	81	Q	113	q	145	æ	177	„	209	đ
18	DC2 (control disp. 2)	50	2	82	R	114	r	146	Æ	178	„	210	€
19	DC3 (control disp. 3)	51	3	83	S	115	s	147	ò	179	—	211	€
20	DC4 (control disp. 4)	52	4	84	T	116	t	148	ö	180	—	212	€
21	NAK (conf. negativa)	53	5	85	U	117	u	149	ö	181	ä	213	§
22	SYN (inactividad sínc)	54	6	86	V	118	v	150	ú	182	À	214	í
23	ETB (fin bloque trans)	55	7	87	W	119	w	151	ù	183	Á	215	í
24	CAN (cancelar)	56	8	88	X	120	x	152	ÿ	184	®	216	í
25	EM (fin del medio)	57	9	89	Y	121	y	153	ö	185	—	217	—
26	SUB (sustitución)	58	:	90	Z	122	z	154	Ü	186	—	218	—
27	ESC (escape)	59	:	91	[123	{	155	ø	187	—	219	—
28	FS (sep. archivos)	60	<	92	\	124		156	£	188	—	220	—
29	GS (sep. grupos)	61	=	93]	125	}	157	ø	189	¢	221	—
30	RS (sep. registros)	62	>	94	^	126	~	158	×	190	¥	222	—
31	US (sep. unidades)	63	?	95	—			159	f	191	—	223	—
127	DEL (suprimir)							159	nbsp			255	nbsp

Tabla de factoriales

N	Factorial de N
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600
13	6227020800
14	87178291200
15	1307674368000
16	20922789888000
17	355687428096000
18	6402373705728000

Serie de Fibonacci





Muchas gracias !!!