Textos docentes | 177

Lecciones de Programación en C para estudiantes de ingeniería

Isabel María del Águila Cano

edual

Lecciones de Programación en C para estudiantes de ingeniería

texto:
Isabel María del Águila Cano

Textos Docentes n.º 177

edición:
Editorial Universidad de Almería, 2024
editorial@ual.es
www.ual.es/editorial
Telf/Fax: 950 015459
¤
ISBN: 978-84-1351-338-6

edual editorial UNIVERSIDAD DE ALMERÍA

En este libro puede volver al índice
pulsando el pie de la página

Índice

Conceptos básicos de la informática

Computadora y cálculos. Estructura

La **informática** según la RAE es

> Conjunto de conocimientos científicos y técnicos que hacen posible el tratamiento automático de la información por medio de los ordenadores.

- (Informática = INFORmación + autoMÁTICA)

En literatura anglosajona

- Computer Science, ciencia de los computadores
- Computer Engineering
- IT Engineering

En literatura francófona

- Informatique

La **programación** según la RAE es

> Acción o efecto de programar

Programar según la RAE es:

1. Formar programas, previa declaración de lo que se piensa hacer y anuncio de las partes de que se ha de componer un acto o espectáculo o una serie de ellos.
2. Idear y ordenar las acciones necesarias para realizar un proyecto.
3. **Preparar ciertas máquinas o aparatos para que empiecen a funcionar en el momento y en la forma deseados.**
4. **Elaborar programas para su empleo en computadoras.**

Máquinas programables

Una **máquina** es un cierto dispositivo físico capaz de realizar un cierto trabajo u operación.

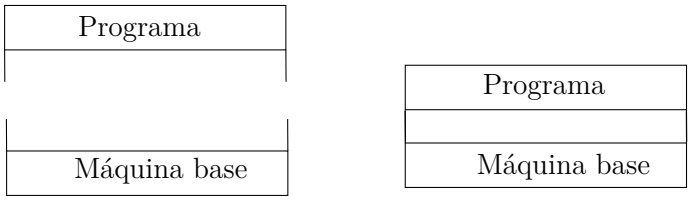
Este concepto puede extenderse a cuando consideramos máquinas que no existen físicamente, pero en las que puede describirse y concebirse su comportamiento, se denominan **máquinas virtuales**.

Atendiendo a como funcionan se pueden clasificar las máquinas en:

- Máquinas no automáticas o de control manual. Es preciso un agente externo u operador que desencadena las operaciones.

- Máquinas automáticas. Actúan por si solas, sin necesidad de operador, pero pueden reaccionar ante estímulos externos.

Existen otras máquinas automáticas denominadas **máquinas programables**, cuyo comportamiento fijo se completa con una parte, *programa*, que permite modificar el comportamiento de la máquina base. Tal como es muestra en la siguiente figura:



Basta con cambiar el programa para tener una nueva máquina con un comportamiento diferente.

Veamos un ejemplo sacado del mundo de los instrumentos musicales:

- Máquina no automática (requiere intervención manual constante): El órgano tradicional (como el órgano de tubos) es un buen ejemplo. Para que funcione, necesita que una persona accione los teclados y pedales constantemente, además de controlar el flujo de aire. Necesitando dos usuarios: el calcante tenía que bombear aire de manera constante y precisa para que el organista pudiera tocar. Con la llegada de motores eléctricos, los órganos dejaron de necesitar una segunda persona para accionar los fuelles, ya que el aire comenzó a ser proporcionado automáticamente por estos sistemas.
- Máquina automática (funciona sin intervención humana directa durante su operación, pero no es programable): La caja de música es un gran ejemplo. Una vez que se da cuerda, la máquina reproduce la música de manera automática, siguiendo un patrón fijo que no se puede modificar, que esta definido en el momento que se construye la caja de música.
- Máquina automática programable (puede programarse para realizar diferentes acciones sin intervención humana durante su operación): El organillo o pianola es un buen ejemplo de máquina automática programable. Estos instrumentos funcionan automáticamente siguiendo una secuencia predefinida, pero lo interesante es que se les puede cambiar la "programación" (rollos de papel perforado en la pianola o cilindros en el organillo) para reproducir diferentes melodías.

La diferencia clave entre cada tipo de máquina en cuanto a intervención humana y capacidad de programación. Por ejemplo el organillo tradicional (o órgano de manivela) es programable pero no es automático necesita al organillero que gire la manivela para que funcione. La pianola sí es completamente automática en cuanto a la ejecución de la música. Una vez que se coloca el rollo de papel perforado y se acciona, la pianola reproduce la música sin necesidad de intervención humana constante.

Por lo tanto, aunque ambos pueden considerarse máquinas programables, el organillo requiere intervención física del operador, mientras que la pianola puede funcionar sin que una persona esté continuamente involucrada durante la ejecución de la música.

Concepto de cómputo

La definición de diccionario de cómputo nos dice que es: *Cálculo para averiguar el resultado, el valor de algo.*

Un cómputo por tanto implica el *tratamiento de información*, numérica de forma tradicional, pero se puede extrapolar a cualquier otro tipo de información.

Cualquier computo se puede describir de diferentes maneras. Por ejemplo el computo para **Calcular la nota final** de un alumno que ha realizado tres exámenes a lo largo del curso, pero sabiendo que si en uno de los exámenes tiene un cero entonces su nota es cero:

- Una opción es utilizar una fórmula

$Nota \approx \sqrt[3]{nota_1 * nota_2 * nota_3}$

- Pero también se puede describir como un proceso:

```
1. Nota = 0;
2. Si nota1 no es cero Nota = Nota + nota1
   - Sino  Nota = 0 e ir a paso 5
3. Si nota2 no es cero Nota = Nota + nota2
   - Sino  Nota = 0 e ir a paso 5
4. Si nota3 no es cero Nota = Nota + nota3
   - Sino  Nota = 0 e ir a paso 5
5. Nota = Nota / 3
```

- O incluso una expresión lógica. Si el resultado de esta expresión es cierto entonces la nota es 0.

$Nota = \neg(((nota_1 + nota_2 + nota_3) = (nota_1 + nota_2) \vee ((nota_1 + nota_2 + nota_3) = (nota_1 + nota_3)) \vee ((nota_1 + nota_2 + nota_3) = (nota_2 + nota_3)))$

Concepto de computador

Un **computador** es una máquina programable para el tratamiento de información, es decir realización de cómputos. Posee unos elementos **fijos** o máquina de base que llamamos **hardware** y otros modificables que son los programas o **software**.

Los computadores actuales son máquinas con programa almacenado de forma que la modificación de los programas no implica la modificación de los elementos físicos y por tanto necesitamos sistemas de almacenamiento y de comunicación del ordenador con el entorno (recordemos los discos de una pianola).

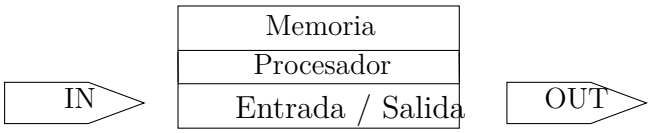
Así, una versión más descriptiva del concepto de computadora donde se indica el proceso seguido para las tareas de cómputo sería:

Un **computador o computadora** es una máquina formada por elementos de tipo **electrónico**, capaces de aceptar unos datos de **entrada**, realizar con ellos gran variedad de tareas (**operaciones**) y proporcionar la información resultante a través de un medio de **salida**, bajo el control de un **programa**, previamente **almacenado** en el propio computador. Esto puede incluir tanto dispositivos físicos como servidores, laptops, tablets, entre otros, así como también máquinas virtuales desplegadas en la nube.

Las **máquinas virtuales** en la nube son entornos computacionales virtuales que se ejecutan en infraestructuras de computación remota y están disponibles a través de Internet. Estas máquinas virtuales proporcionan capacidades informáticas similares a las de una computadora física, pero con la flexibilidad y escalabilidad adicionales ofrecidas por la computación en la nube.

Un **programa** será por lo tanto la descripción de un cómputo en un lenguaje de programación, que como se ha visto en el ejemplo de la nota media puede tener muchas formas o lenguajes diferentes, aunque nosotros nos centraremos en la definición de programas como un conjunto de **instrucciones**, lo que llamamos **programación estructurada**, sobre el ejemplo del cómputo de la nota media es una notacion en forma de proceso.

Gráficamente esta última definición de computadoras se recoge en la siguiente imagen



La memoria almacena datos y programas. Los dispositivos de Entrada/Salida permiten intercambiar información con el exterior. El procesador es el elemento de control que realiza las operaciones elementales del tratamiento de la información interna y de las operaciones de entrada/salida de la información al exterior (de acuerdo con los programas almacenados en la memoria). Es usual llamar **periféricos** a los dispositivos electrónicos que conforman las unidades de entrada y de salida.

Estructura de una computadora

Cada año, probablemente esperamos pagar al menos un poco más por la mayoría de los productos y servicios, sube la luz, el agua, las matrículas. Lo contrario ha ocurrido en los campos de la informática y las comunicaciones, especialmente en lo que se refiere a la electrónica que soporta estas tecnologías. A lo largo de los años los costes del hardware han disminuido rápidamente. Durante décadas, cada par de años, la potencia de procesamiento de los ordenadores aproximadamente es el doble. Esta notable tendencia suele denominarse **Ley de Moore**, llamada así por Gordon Moore, cofundador de Intel 1960.

En el campo de las comunicaciones se ha producido un crecimiento similar. Los costes se han desplomado a medida que la enorme demanda de ancho de banda de las comunicaciones (es decir, la capacidad de transmisión de información) ha traído una intensa competencia.

Los componentes electrónicos de una computadora son cada día más complejos, y muchos de ellos incluso se duplican. Existen numerosas combinaciones de dispositivos/componentes físicos que pueden ensamblarse para crear computadoras. Lo que es más, actualmente aparece el concepto de **computadores virtuales** o **máquinas virtuales**, que no necesitan *hardware* como tal. Se definen o alquilan componentes/unidades lógicas sobre servidores remotos para crear computadoras personalizadas virtuales. Es decir, la máquina base se define de forma virtual, definiendo las unidades lógicas que se despliegan sobre soportes físicos de distinta naturaleza.

Independientemente de las diferencias físicas, los ordenadores combinan distintos tipos de **unidades lógicas** o secciones: (Unidades de entrada, de salida, de memoria, de control, aritmético lógica y de memoria secundaria o almacenamiento)

Unidad de entrada

Esta unidad obtiene la información (datos y programas informáticos) de los dispositivos de entrada y la pone a disposición de las demás unidades para su procesamiento. Los computadores reciben la mayoría de las entradas de los usuarios a través de teclados, pantallas táctiles, ratones y touchpads.

Otras formas de entrada son:

- la recepción de comandos de voz,
- escanear imágenes y códigos de barras,
- leer datos de dispositivos de almacenamiento secundarios (como unidades de discos duros, Blu-ray Disc™ y memorias USB, también llamados "lápices de memoria"),
- recibir vídeo de una cámara web,
- recibir información de Internet (por ejemplo, cuando transmite vídeos de de YouTube® o descarga libros electrónicos de Amazon),
- recibir datos de posición de un dispositivo GPS,
- recibir información de movimiento y orientación de un acelerómetro (un dispositivo que responde a la aceleración hacia arriba/abajo, izquierda/derecha y adelante/atrás).
- recibir la voz de asistentes inteligentes como Apple Siri, Amazon Alexa y Google Home.

Unidad de salida

Esta unidad "envía" la información que el computador ha procesado y la coloca en varios dispositivos de salida para que esté disponible fuera del computador. La mayor parte de la información que sale de los ordenadores hoy en día:

- se muestra en pantallas,
- se imprime en papel ("ser verde" desaconseja esto),
- se reproduce en audio o vídeo en teléfonos inteligentes, tabletas, ordenadores y pantallas gigantes en estadios deportivos,
- se transmite por Internet o
- para controlar otros dispositivos, como coches que se conducen solos (y vehículos autónomos en general), robots y electrodomésticos "inteligentes".

También es habitual que la información se transmita a dispositivos de almacenamiento secundario, como las unidades de estado sólido (SSD), los discos duros, memorias USB y unidades de DVD. Son unidades donde se hace **persistente** la información, es decir, no se pierde aunque se apague la máquina.

Otras formas de salida son la vibración de teléfonos inteligentes, mandos de juegos y dispositivos de realidad virtual.

Unidad de memoria

Esta unidad es un "almacén" de acceso rápido y capacidad relativamente baja que mantiene la información introducida a través de la unidad de entrada, haciéndola disponible cuando sea necesario. La unidad de memoria también guarda la información procesada hasta que puede ser colocada en dispositivos de salida.

La información de la unidad de memoria es volátil. Se pierde cuando se apaga el ordenador. La unidad de memoria suele denominarse memoria principal, memoria primaria o RAM (Random Access Memory).

Se divide en posiciones (**palabras de memoria**) de un determinado tamaño. Se accede a cada posición a través de un número (**dirección de memoria**).

Las memorias principales de los ordenadores de sobremesa y portátiles contienen hasta 128 GB de RAM, aunque lo más habitual es entre 8 y 16 GB. 16 GB es lo más habitual. GB son las siglas de gigabytes; un gigabyte equivale aproximadamente a mil millones de bytes.

Un byte son ocho bits. Un bit (abreviatura de "dígito binario") es un 0 o un 1. EstoEstos conceptos se verán más adelante. Son las unidades en las que se mide la capacidad de almacenamiento o velocidad de trasmisión de información, son por tanto unidades para medir la cantidad de información.

Parámetros significativos de la memoria:

- Tamaño de la memoria expresado en Megabytes (1 GigaByte = 1024 MegaBytes; 1 MegaByte = 1024 Kilobytes; 1 Kilobyte = 1024 Bytes; 1 Byte = 8 bits; bit = dígito binario: 0 ó 1).
- Tiempo de acceso expresado en nanosegundos (1 ns = 10-9 segundos).
- Ancho de banda expresado en Megabytes transferidos a ó desde la memoria por segundo.

Unidad Central aritmético lógica (ALU)

Esta unidad "fabrica", es decir, realiza cálculos (por ejemplo, suma, resta, multiplicación y división) y toma decisiones (por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales). En los sistemas actuales, la ALU forma parte de la siguiente unidad lógica, la CPU.

Unidad central de proceso

Esta unidad es la "administrativa" coordina y supervisa el funcionamiento de las demás unidades, incluyendo habitualmente el procesador y la unidad aritmético lógica (ALU). La CPU indica:

- a la unidad de entrada, cuándo leer información en la unidad de memoria,
- a la ALU cuándo utilizar la información de la unidad de memoria en los cálculos, y
- a la unidad de salida cuándo enviar información desde la unidad de memoria a dispositivos de salida,
- cuando y como se accede a la memoria.

La mayoría de los ordenadores actuales tienen procesadores multinúcleo que implementan de forma económica múltiples procesadores en un único chip de circuito integrado. Estos procesadores pueden realizar muchas operaciones simultáneamente. Un procesador de doble núcleo tiene dos CPU, un procesador de cuatro núcleos tiene cuatro y un procesador octa-core tiene ocho. Intel tiene algunos procesadores con hasta 72 núcleos.

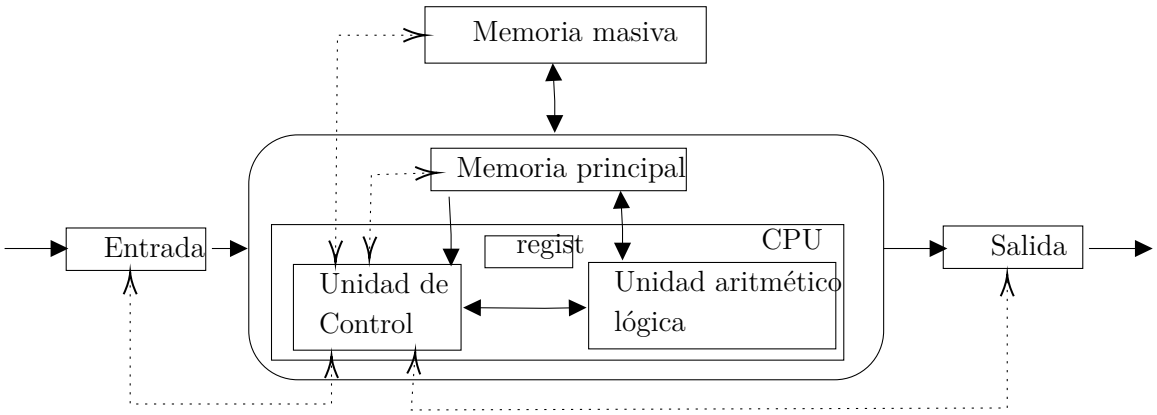
Unidad de almacenamiento secundario

Es la unidad de "almacenamiento" a largo plazo y de gran capacidad, siendo tanto de entrada como de salida. Los programas y datos utilizados activamente por las otras unidades se colocan en dispositivos de almacenamiento secundario (también llamada memoria secundaria) hasta que vuelvan a necesitarse, posiblemente horas, días, meses o incluso años después. También se llama memoria masiva.

La información en el almacenamiento secundario es persistente: se conserva incluso cuando el computador se apaga. Se tarda mucho más en acceder a la información del almacenamiento secundario que información de la memoria primaria, pero su coste por byte es mucho menor.

Ejemplos de dispositivos de almacenamiento secundarios son las unidades de estado sólido (SSD), las memorias flash USB, los discos duros y las unidades Blu-ray de lectura/escritura.

Muchas unidades actuales almacenan terabytes (TB) de datos. Un terabyte es aproximadamente un billón de bytes. Los discos duros de ordenadores de sobremesa y portátiles que tienen una capacidad de hasta 4 TB, y algunos discos duros recientes para ordenadores de sobremesa tienen una capacidad de hasta 20 TB.



Otros elementos que completan la arquitectura clásica de un ordenador son:

Buses: Conjuntos o grupos de hilos que interconectan las unidades de procesamiento y de control y la memoria, así como estas con los periféricos (unidades de entrada y salida), proporcionando un camino de comunicación para el flujo de datos entre las distintas unidades. La información se transmite en paralelo (en un instante de tiempo dado se encuentran en el bus todos los bits de un dato).

Registros: Dispositivos físicos que tienen una labor específica o que permiten la realización de operaciones por parte de la CPU. Los más importantes son:

- registro de instrucción (IR) almacena la instrucción que se está ejecutando

- contador de programa (PC) contiene la dirección de memoria de la siguiente instrucción a ejecutar
- puntero de pila (SP)
- r0-rD: registros de uso general (RF: banco de registros)

Representación de la información

De las definiciones de **dato** según la RAE, son del contexto de la computación la 1 y la 3. En concreto la tercera es sólo relativa a la informática.

1. m. Información sobre algo concreto que permite su conocimiento exacto o sirve para deducir las consecuencias derivadas de un hecho. A este problema le faltan datos numéricos.
2. m. Documento, testimonio, fundamento.
3. m. *Inform.* Información dispuesta de manera adecuada para su tratamiento por una computadora.

Siendo más específicos un **Dato** en informática es: una representación formalizada de hechos o conceptos susceptible de ser comunicada y/o procesada.

Existen distintos **tipos de datos** que por lo tanto tienen que ser representados de forma diferente:

- Numéricos (12, 28.5): reales o enteros
- Alfabéticos (Ana)
- Alfanuméricos: 23456X, M-6995
- Imágenes, sonido, video,...

Llamamos *representación de la información* a la forma en la que se almacenan y recogen los datos para poder ser procesados en la computadora. Se debe tener en cuenta que en los medios electrónicos de almacenamiento / procesamiento solo disponen de **dos estados**: (con permiso de las tecnologías cuánticas)

- interruptor (relé) abierto-cerrado
- paso o no paso de corriente
- magnetización en un sentido u otro

Esto se traslada a que la información procesada por un computador son siempre 1s y 0s. Lo que lleva a la necesidad de **traducir** cualquier dato a una combinación de esos dos símbolos es decir **notación binaria** o sistema de numeración binario.

Sistemas de numeración en informática

Los sistemas de numeración usuales en informática no son decimales, no utilizan la base 10 como los humanos.

Tipo	Base	Uso	Alfabeto
decimal	10	humanos	0 a 9
binario	2	computadoras	0, 1
octal	8	computadoras (códigos intermedios)	0 a 7
hexadecimal	16	computadoras (códigos intermedios)	0 a 9, A, B, C, D, E, F

De forma general un sistema de numeración base b: (Sistema de numeración posicional)

Alfabeto tiene **b** símbolos o cifras diferentes, con lo que un Número = {cifras}

Su valor depende de:

- Nº de cifras
- Valor de cada cifra
- Posición cifra dentro del número

Para el sistema de numeración decimal

$\text{Alfabeto}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Ejemplo de número decimal:

$4325,12 = 4000 + 300 + 20 + 5 + 0,1 + 0,02 = 4 * 10^3 + 3 * 10^2 + 2 * 10^1 + 5 * 10^0 + 1 * 10^{(-1)} + 2 * 10^{(-2)}$

Generalización para cualquier base b:

$\text{Alfabeto}_b = \{0, 1, 2, ..., b-1\}$

$N = \ldots\ n_4 n_3 n_2 n_1 n_0 n_{-1} n_{-2} n_{-3}\ \ldots = \ldots + n_4 * b^4 + n_3 * b^3 + n_2 * b^2 + n_1 * b^1 + n_0 * b^0 + n_{-1} * b^{-1} + n_{-2} * b^{-2} + \ldots$

Sistema de numeración base 2: .

Solo hay

1 0

tipos de personas:

Las que entienden binario
y las que no.

.

.

Nº decimal	Nº binario
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

Observando la tabla se comprueba que por ejemplo para representar el "6" en decimal, necesitamos 3 posiciones en binario. Cada una de estas posiciones es un **bit**.

Por tanto si se quieren representar en binario las posibles estados emocionales de una persona:

Estados= { Tranquilo, Feliz, Emocionado, Preocupado, Triste, Enojado, Sorprendido, Nervioso, Cansado, Motivado }

Son necesarios cuatro cifras binarias. 2^4 es el máximo número de elementos a representar, en este caso tenemos 10 estados.

BIT

- Los dígitos en el sistema de numeración binario se llaman bits (BIT= Binary digit).
- Es la unidad más pequeña de información, un uno o un cero.

Byte

- Es un conjunto de 8 bits.
- También se llama Octeto o Carácter (porque con un byte se suele codificar un carácter).

Con 1 bit se pueden representar hasta 2^1 elementos {0,1}

Con 2 bits se pueden representar hasta 2^2 elementos {00, 01, 10, 11}

Con 8 bits podemos representar hasta 256 elementos 2^8

```
0000 0000
```

hasta

```
1111 1111
```

Pregunta: En un lago hay una superficie cubierta de nenúfares y cada día esa extensión dobla su tamaño. Si tarda 48 días en cubrir el lago, ¿cuánto tarda en cubrir la mitad del lago?

Esta es la razón por la que la unidad de almacenamiento de información en informática (tamaño memoria, tamaño almacenamiento secundario, registros y buses) se mide en potencias de 2.

Múltiplos del byte

acrónimo	nombre	equivalencia
KB	Kilobyte	2^{10} bytes = 1024 bytes $\approx 10^3$ bytes
MB	Megabyte	2^{20} bytes = 1048576 bytes $\approx 10^6$ bytes
GB	Gigabyte	2^{30} bytes = 1073741824 bytes $\approx 10^9$ bytes
TB	Terabyte	2^{40} bytes $\approx 10^{12}$ bytes
PB	Petabyte	2^{50} bytes $\approx 10^{15}$ bytes
EB	Exabyte	2^{60} bytes $\approx 10^{18}$ bytes

qbits (cúbit)

Unidad relacionada con la computación cuántica. Los **qubits** pueden estar en una superposición de estados *{0, 1}*, lo que significa que pueden representar simultáneamente 0 y 1. Además, el estado de un qubit depende del estado de otro, lo que proporciona una potencialmente mayor capacidad de procesamiento y almacenamiento de información en comparación con los bits clásicos.

Aritmética del binario Lo primero es saber como trasformar número entre bases

De binario a decimal

$(110100)_2 = (1 * 2^5) + (1 * 2^4) + (1 * 2^2) = 32 + 16 + 4 = 52$

De decimal a binario

Parte entera: dividir sucesivamente por 2 y utilizar el último cociente y los restos de las divisiones para las cifras binarias (el último cociente es el bit más significativo y el primer resto es el bit menos significativo):

```
26 | 2
06 | 13 | 2
 0 |  1 | 6 | 2
       0 | 3 | 2
         1 | 1
```

$(26)_{10} = (11010)_2$

Parte fraccionaria: multiplicar sucesivamente por 2 las partes fraccionarias resultantes y utilizar las partes enteras en orden para las cifras binarias:

$$\begin{array}{r} 0.1\,8\,7\,5 \\ \times \quad 2 \\ \hline 0.3\,7\,5\,0 \end{array} \quad \begin{array}{r} 0.3\,7\,5 \\ \times \quad 2 \\ \hline 0.7\,5\,0 \end{array} \quad \begin{array}{r} 0.7\,5 \\ \times \quad 2 \\ \hline 1.5\,0 \end{array} \quad \begin{array}{r} 0.5 \\ \times \quad 2 \\ \hline 1.0 \end{array}$$

$(0,1875)_{10} = (0,0011)_2$

Operaciones en binario

a	b	a + b	a – b	a * b	a / b
0	0	0	0	0	indeterminado
0	1	1	1 y me adeudo 1	0	0
1	0	1	1	0	
1	1	0 y me llevo 1	0	1	1

Representación en complementos

Útiles para representar números negativos (se reducen las restas a sumas y se simplifica el hardware)

- Complemento **a la base-1** de un número N es el número que resulta de restar cada una de las cifras de N a la base menos 1 del sistema de numeración que se esté utilizando. Se obtiene cambiando 0 por 1 y 1 por 0.
- Complemento **a la base** de un número N es el número que resulta de restar cada una de las cifras de N a la base menos uno del sistema de numeración que se esté utilizando, y posteriormente sumar 1 a la diferencia obtenida.

Ejemplos:

Numero	Base	Complemento a base-1	Complemento a base
63	10	36	37
10010	2	01101	01110

Como calcular complementos en binario

- Complemento a 1: Cambiar los ceros por unos y los unos por ceros
- Complemento a 2: de derecha a izquierda se deja igual hasta el primer 1 y se cambian a partir de este.

Codificación de datos numéricos enteros

Las principales formas de representar los enteros son en **binario**, con o sin signo y utilizando el decimal codificado en binario (**BCD**)

BCD (decimal codificada en binario)

4 bits para representar las cifras: 0 1 2 3 4 5 6 7 8 9

Dado el número:98325

```
 9       8     3       2     5
1001   1000   0011   0010   0101
```

BINARIA

- Un numero entero puede ser representado en base dos (sistema de numeración binario)

  $25 = 16 + 8 + 1 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 11001$

- Módulo y signo
  - Se usa un bit adicional para representar el signo.
- Representación en complemento
  - Es más apropiada para el procesamiento de la información en un ordenador.

De esta forma tenemos:

Enteros sin signo

Con n bits tenemos 2^n números

Rango: $[0, 2^n - 1]$

Enteros con signo

Cuando se utiliza el bit más significativo para el signo se puede optar por tres alternativas, mantener la magnitud en binario o utilizar complemento a 1 o complemento a 2

1. *Signo y magnitud*

- Primer bit: Recoge el signo (0 = positivo, 1 = negativo)
- Resto de bits: valor absoluto número

  Rango: $[-(2^{n-1} - 1), 2^{n-1} - 1]$

2. *Complemento a 1*

- Primer bit: signo (0=positivo, 1=negativo)
- Resto de bits:
  - Si el número es positivo: valor absoluto del número
  - Si el número es negativo: complemento a 1 de valor absoluto

  Rango: $[-(2^{n-1} - 1), 2^{n-1} - 1]$

3. *Complemento a 2*

- Primer bit: signo (0=positivo, 1=negativo)
- Resto de bits:
  - número positivo: valor absoluto del número

- número negativo: complemento a 2 de valor absoluto

Rango: $[-2^{n-1}, 2^{n-1} - 1]$

EJEMPLO:

Supongamos que tenemos 4 bits para representar los números, podremos representar 16 elementos 2^4. Si son enteros positivos podremos recoger del 0 al 15 simplemente pasándolos a binario. Si utilizamos el bit más significativo para el signo podremos representar $[-7, 7]$. Si se utiliza el complemento a 1 tendremos dos representaciones para el 0 cosa que no pasa con el complemento a dos.

nº Decimal	Signo y magnitud	Complemento a 1	Complemento a 2
+7	0111	0111	0111
+6	0110	...	...
+5	0101	...	
+4	0100		
+3	0011		
+2	0010		
+1	0001		
+0	0000		
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8			1000

Codificación de datos numéricos reales

- Se representan por separado los tres elementos del número en notación científica: base, mantisa y exponente
  - Ejemplo: 0,123 x 10^{-4}
- Se representan por separado base (10), mantisa (0.123) y exponente (-4)
  - Permite representar rangos grandes de números usando pocos bits
- Actualmente se usa un sistema normalizado: norma IEEE 754
  - 32 bits
  - Base 2, está predeterminada, por lo que no es necesario codificarla
  - Signo: un bit, 0 = +, 1= -
  - Exponente: 8 bits (que utiliza una representación sesgada)

    - Mantisa: 23 bits

- Al representar de esta forma se pierde precisión

    - !Cuidado al comparar números! El ordenador sólo los considera iguales si todos los bits son iguales.
    - La norma IEEE 754 también contempla usar 64 bits (doble precisión)

- Enlace para ver la notación en bits de un número real
- Enlace a Conversor en línea
- Enlace para el estudio de las formas de representación de información numérica

Codificación de caracteres alfanuméricos

- Códigos de caracteres

    - **código** = correspondencia biunívoca entre un conjunto de caracteres y un conjunto de combinaciones de bits (tabla).
    - **caracteres**: numéricos, alfabéticos, alfanuméricos, especiales y de control.

- Código ASCII extendido

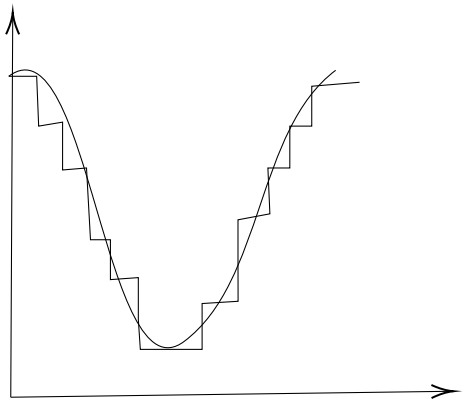
    - American Standard Code for Information Interchange
    - ASCII : solo 7 bits de los 8 que forman el byte
        - el bit más significativo (izquierda) no se utilizaba
        - Se pueden representar 2^7= 128 caracteres distintos
    - ASCII extendido: 8 bits
        - Con 8 bits (1 byte) se pueden representar $2^8 = 256$ caracteres diferentes, a continuación se muestran hasta 127

Decimal	8 bits	ASCII	Decimal	8 bits	ASCII	Decimal	8 bits	ASCII
0	00000000	[NUL]	1	00000001	[SOH]	2	00000010	[STX]
3	00000011	[ETX]	4	00000100	[EOT]	5	00000101	[ENQ]
6	00000110	[ACK]	7	00000111	[BEL]	8	00001000	[BS]
9	00001001	[TAB]	10	00001010	[LF]	11	00001011	[VT]
12	00001100	[FF]	13	00001100	[CR]	14	00001110	[SO]
15	00001111	[SI]	16	00010000	[DLE]	17	00010001	[DC1]
18	00010010	[DC2]	19	00010011	[DC3]	20	00010100	[DC4]
21	00010101	[NAK]	22	00010110	[SYN]	23	00010111	[ETB]
24	00011000	[CAN]	25	00011001	[EM]	26	00011010	[SUB]
27	00011011	[ESC]	28	00011100	[FS]	29	00011101	[GS]
30	00011110	[RS]	31	00011111	[US]	32	00100000	espacio
33	00100001	!	34	00100010	”	35	00100011	#
36	00100100	$	37	00100101	%	38	00100110	&
39	00100111	’	40	00101000	(	41	00101001	)
42	00101010	*	43	00101011	+	44	00101100	,

Decimal	8 bits	ASCII	Decimal	8 bits	ASCII	Decimal	8 bits	ASCII
45	00101101	-	46	00101110	.	47	00101111	/
48	00110000	0	49	00110001	1	50	00110010	2
51	00110011	3	52	00110100	4	53	00110101	5
54	00110110	6	55	00110111	7	56	00111000	8
57	00111001	9	58	00111010	:	59	00111011	;
60	00111100	<	61	00111101	=	62	00111110	>
63	00111111	?	64	01000000	@	65	01000001	A
66	01000010	B	67	01000011	C	68	01000100	D
69	01000101	E	70	01000110	F	71	01000111	G
72	01001000	H	73	01001001	I	74	01001010	J
75	01001011	K	76	01001100	L	77	01001101	M
78	01001110	N	79	01001111	O	80	01010000	P
81	01010001	Q	82	01010010	R	83	01010011	S
84	01010100	T	85	01010101	U	86	01010110	V
87	01010111	W	88	01011000	X	89	01011001	Y
90	01011010	Z	91	01011011	[	92	01011100	\
93	01011101	]	94	01011110	^	95	01011111	_
96	01100000	‘	97	01100001	a	98	01100010	b
99	01100011	c	100	01100100	d	101	01100101	e
102	01100110	f	103	01100111	g	104	01101000	h
105	01101001	i	106	01101010	j	107	01101011	k
108	01101100	l	109	01101101	m	110	01101110	n
111	01101111	o	112	01110000	p	113	01110001	q
114	01110010	r	115	01110011	s	116	01110100	t
117	01110101	u	118	01110110	v	119	01110111	w
120	01111000	x	121	01111001	y	122	01111010	z
123	01111011	{	124	01111100	|	125	01111101	}
126	01111110	~	127	01111111	[DEL]			

Codificación de sonido

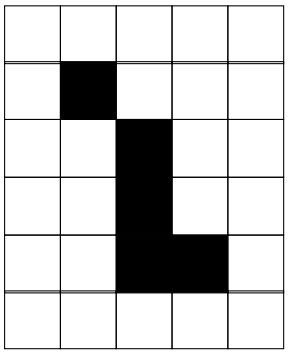
- El sonido es analógico
  - Hay que empezar por discretizarlo: muestreo (sampling)
  - Cada muestra se codifica (por ejemplo como un entero con un byte)
- Se pierde información
  - La calidad del sonido reconstruido dependerá de:
    - Número de muestras por segundo
    - Número de bits usados para codificar cada muestra



Codificación básica de Imágenes

-También es necesario muestrearla.

- División de la imagen en una matriz de píxeles
- A cada píxel se le asocia un valor (1 o 0, para blanco y negro, varios bits por píxel para imagen en color)

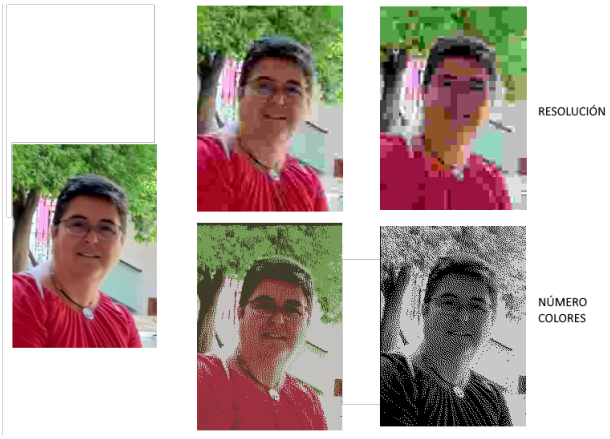


$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{matrix}$$

- Hay otras formas mejores (con compresión)

-La calidad de una imagen dependerá de:

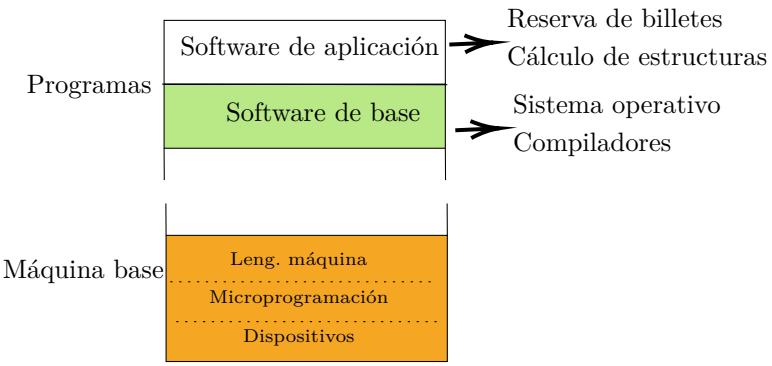
- Resolución: número de píxeles por unidad de superficie
- Número de bits por píxel (8,16,24,32)
    - 8 bits por píxel permite representar 256 colores diferentes

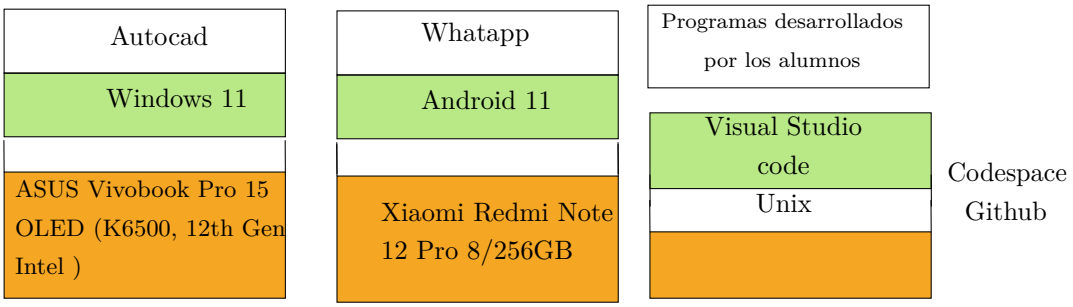


Algoritmos y lenguajes de programación

Veamos una versión extendida de la arquitectura de una computadora pero desde el punto de vista del software.

El Software es conjunto de programas asociados a una computadora que definen su comportamiento.





En esta imagen se destacan no los dispositivos físicos, puesto que los programas se almacenan en en la memoria (primaria y/o secundaria), sino que destacamos los distintos tipos de software que se pueden encontrar, y que se organizan por niveles.

- Software de base o programas del sistema: programas que gestionan el funcionamiento de la computadora:
    - *Sistema Operativo* - Pertenece a los componentes fijos de un máquina dada. Porque si bien es un elemento lógico, es necesario para que la máquina funcione. Un móvil o un portátil sin sistema operativo no es más que un bonito pisapapeles.
    - *Utilidades para construcción/ejecución de aplicaciones*. Destacamos aquí el concepto de **lenguaje máquina**; puesto que cada modelo de computadora física tiene una forma de entender los programas que depende de cuál sea su arquitectura y sus componentes, el lenguaje máquina es el sistema de codificación que tiene cada computadora/procesador para almacenar los programas.
- Software de aplicación, programas que se cargan sobre la memoria para ejecutar una tarea.

Un programa es una lista de instrucciones máquina que al ejecutarse producen un resultado concreto. Deben estar expresados en un lenguaje que entienda la máquina, o bien debemos escribirlos en algún lenguaje que pueda ser traducido a lenguaje máquina utilizando algún tipo de software de base como son los compiladores o los interpretes.

Todo programa comienza con idea, algo que se quiere hacer, generalmente ese algo resulta como solución a un problema específico, la solución de un problema requiere el diseño de un **algoritmo**.

Un "ejemplo" es el algoritmo para llenar un vaso de agua

O de como hacer un sandwich de mantequilla de cacahuete y mermelada de merienda.

Los algoritmos son soluciones abstractas a problemas, que generalmente son codificados en un lenguaje de programación y luego traducidos al lenguaje máquina que es el que una computadora puede ejecutar, para entonces con sus resultados solucionar el problema real para el que se concibieron. Los algoritmos son independientes del lenguaje de programación y de la máquina que lo ejecute, una analogía de la vida real nos la encontramos cuando un cocinero con vista cansada quiere hacer una receta:

La receta de un plato de cocina (algoritmo) puede expresarse en inglés, francés o español (lenguaje) e indicará la misma preparación independientemente del cocinero (máquina). Después el cocinero concreto que lea la receta, debe poder entender lo escrito y como el cocinero es corto de vista necesita sus gafas (compilador) para que le traduzca los garabatos a letras.

Un programa contiene la información codificada del comportamiento deseado o descrito en el algoritmo. Cada modelo de computadora podrá utilizar una forma particular de codificación dependiendo de sus dispositivos físicos, no es lo mismo un móvil que una computadora de sobremesa y que puede cambiar incluso con el modelo de procesador. La forma de codificar programas de una máquina particular se llama código máquina o lenguaje máquina. Los programas en lenguaje máquina son muy difíciles de leer, y se suelen llamar **programas objeto** (obj).

Los lenguajes de programación sirven para representar programas de forma simbólica en forma de texto. Un **lenguaje de programación** es un idioma artificial diseñado para que sea fácilmente entendible por un humano y traducible por una máquina (empleado una pieza de software de base). También son llamados lenguajes de alto nivel.

Se forman con símbolos tomados de un determinado repertorio (componentes **léxicos**). Se construyen siguiendo unas reglas precisas (**sintaxis**). Incorpora unas reglas de **semántica** que determinan el significado de cada construcción correcta.

Aun sin conocer como programar veamos un ejemplo:

```c
#include <stdio.h>

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

gcc compila.c -o compila

Tipos de lenguaje de programación

- Lenguajes de Marcado: No son considerados lenguajes de programación como tales se añade códigos en formato texto (marcas) que indican información para la visualización de la información o alguna procesamiento. Ejemplos: markdown, latex, HTML.
- Lenguajes de programación. Es un conjunto de reglas y símbolos que permiten a un programador **escribir instrucciones** que una computadora puede entender y ejecutar. Hay muchos tipos y de muchas categorías

  Se clasifican según su:

  - Nivel de abstracción:
    - Bajo Nivel. Se programa sobre el hardware, ejemplo ensamblador
    - Alto nivel. Tenemos una serie de símbolos y reglas que definen instrucciones mas complejas
  - Estilo de programación
    - Imperativa — Solo veremos esta. Hay que definir el detalle de lo que hay que hacer.
    - Funcional
    - Orientada a objetos
    - Declarativa
    - Guiada por eventos
    - Concurrente
  - Dominio de aplicación
    - Aplicaciones científico tecnológicas
    - Aplicaciones de gestión de la información

    - Aplicaciones de inteligencia Artificial
    - Aplicaciones de programación de sistemas
    - Aplicaciones para la web
- Ámbito de uso
    - Web
    - Móviles
    - Tradicional (equipos fijos y procesamiento local)
    - Hardware (Programación de sistemas, diseño de circuitos,...)
- Forma de traducción
    - Compiladores
    - Intérpretes

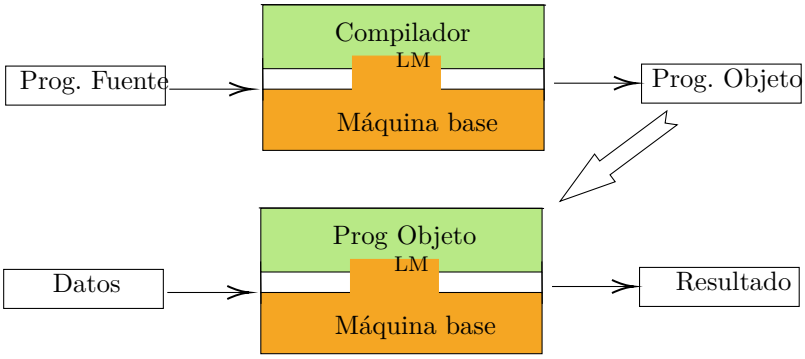
Existen diferentes estrategias para poder ejecutar/conseguir el lenguaje máquina escrito en un lenguaje de programación. Se necesita un software especial que procesará el lenguaje de alto nivel. Estos **procesadores de lenguajes o traductores** manipulan la descripción simbólica de una algoritmo escrito en un lenguaje de programación (programa) para obtener el código objeto que ya si es ejecutable en la máquina. Existen dos grandes categorías en los traductores: Compiladores e interpretes.

Traductores: Compiladores e interpretes

Los **procesadores de lenguajes o traductores** manipulan la descripción simbólica del algoritmo escrito en un lenguaje de programación (programa) para obtener el código objeto que ya si es ejecutable en la máquina. Las dos grandes categorías en los traductores:

- Compilador: una vez traducido el programa fuente, su ejecución (programa objeto) es independiente del compilador (se traduce una vez y se ejecuta múltiples veces).
- Intérprete: el programa fuente se traduce instrucción a instrucción cada vez que se ejecuta (no se crea el programa objeto).

La siguiente figura muestra las diferencia entre ambas estrategias.



Algunos lenguajes de programación

Python

- Fácil de aprender. Su filosofía hace hincapié en una sintaxis que favorezca un código legible.
- Multiparadigma (soporta orientación a objetos, programación imperativa y programación funcional).
- Multiplataforma
- Bibliotecas amplias y variadas
- Interpretado: Los archivos .py requieren tener python disponible para ser ejecutados.

Java

- Lenguaje portable, basado en C.
- Utilizado para aplicaciones en internet.
- A la vez compilado e interpretado. Con el compilador se convierte archivos .java, a un conjunto de instrucciones que recibe el nombre de bytecodes, en un archivo .class. Estas instrucciones son independientes del tipo de ordenador. El intérprete ejecuta estas instrucciones en un ordenador específico (máquina virtual java).

C

- Lenguaje básico de los sistemas UNIX, se inventó para escribir un sistema operativo llamado UNIX.
- Caracterizado por la economía de expresión.
- Utilizado para programación de sistemas.
- Revisiones del lenguaje: C99, C11 y C18.
- Gestión de memoria - Soporta la característica de asignación dinámica de memoria.
- Estándar ISO/IEC 9899:2018 (C18)
- Familia de lenguajes C: C++ y C#.
- Ampliamente utilizado en ingeniería por estar más cerca del dispositivo.
- Los programas escritos en C son eficaces y rápidos.
- C es básicamente una colección de funciones de la biblioteca C; también podemos crear nuestra propia función y añadirla a la biblioteca C.
- C es fácilmente extensible.
- C es un lenguaje robusto con un rico conjunto de funciones y operadores incorporados.
- Compilado. desde el archivo .c se genera un ejecutable .exe. Aunque suele ser trasparente desde los entornos de construcción de programas, la generación del ejecutable tiene dos fases:
    - Compilar: Crea los objetos (.o o .obj).

- Enlazar o Linkar: Une los objetos para crear ejecutables. Busca librerías y las añade si es necesario.

Tendencias en lenguajes de programación

[Evolución de los lenguajes de programación]https://www.youtube.com/watch?v=n6mVXfRCD0c)

El lenguaje de programación C

El lenguaje C fué desarrollado en la segunda mitad de los años 70 por Brian Kernighan y Dennis Ritchie. Es un lenguaje de alto nivel que está estrechamente vinculado con el sistema operativo Unix. No obstante, el lenguaje incluye bastantes facilidades propias de los lenguajes de bajo nivel. El lenguaje C nos permite estructurar los programas mediante acciones y funciones, y además, manipular de forma eficiente contenidos de información elemental como si se utilizase un lenguaje ensamblador.

El precio a pagar por tener un lenguaje orientado a la eficiencia es que su compilador tiene mecanismos de detección de errores más laxos que otros lenguajes de alto nivel. El programador en este lenguaje debe ser muy consciente de la codificación que produce puesto que, la detección de errores puede ser ardua y costosa y que es su responsabilidad llevar el control del programa, por ejemplo controlando no superar los límites de las colecciones de datos (arrays) como veremos más adelante.

El lenguaje C es un buen lenguaje para programar y un mal lenguaje para aprender a programar. No hay que alarmarse por esta afirmación; la realidad es la que es. Todo lenguaje está sujeto a una curva de aprendizaje que hay que superar. Lo importante es habituarse a que no siempre los conceptos teóricos se plasman tal cual en la práctica, y que hay que desarrollar hábitos y emplear metodologías adecuadas para elaborar programas correctos en cualquier lenguaje. En esta asignatura se introducen aspectos básicos del lenguaje C que se irán ampliando paulatinamente en otras asignaturas y probablemente con otros lenguajes de programación.

La resolución de problemas en C suele constar de varias partes: **un entorno** de desarrollo de programas (compilador y editor), **el lenguaje de programación** y **la biblioteca** estándar de C (funciones y/o utilidades preescritas y listas para ser usadas).

Los elementos que definen un **lenguaje de programación** son:

- **Alfabeto**, es decir, los caracteres que pueden usarse
- **Léxico**
  - Palabras y su significado
  - Elementos básicos con los que se componen los programas
- **Sintaxis** Reglas para combinar las palabras, de manera que tengan sentido.
- **Semántica**, significado, ligado a la teoría de la programación estructurada.

Alfabeto de C Símbolos que pueden aparecer en un programa en C

- Letras, exceptuando `ñ` y letras con tilde
- Números y operadores
- Caracteres especiales `( ) & " \` y el espacio
- El carácter `;` es importante en C porque indica el final de una instrucción. ¡¡¡Muy importante!!!

El compilador distingue MAYÚSCULAS y minúsculas.

Léxico de C Todo lenguaje de programación tiene un léxico = elementos básicos con los que se construyen los programas:

- **Palabras clave** o palabras reservadas
  - palabras que tienen un significado especial para el compilador
  - Siempre minúscula: `include` , `define` , `main` , `if` , etc.

En la tabla siguiente se listan todas las palabras clave reservadas por el lenguaje C. Cuando el lenguaje de programación actual es C o C++, estas palabras clave no se pueden abreviar, ni utilizarse como nombres de variable o utilizarse como cualquier otro tipo de identificador, tienen un significado definido en el propio lenguaje.

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

- **Separadores**
  - espacios en blanco, saltos de línea, tabuladores. Los espacios en blanco permiten identificar las palabras a "compilar". Los saltos de línea y tabuladores en **C** sirven para hacer más legible a los humanos los programas. Solo el carácter `;` es el indicador de fin de línea.
- **Operadores**
  - Representan operaciones como las aritméticas (`+`, `>`), lógicas (`&&`, `||`), de asignación (`=`), etc. También `#`.
- **Literales**
  - Especifican un valor concreto (Números, caracteres y cadenas de caracteres).
- **Identificadores**:

- Nombres de las variables , constantes y funciones definidas por el programador: “perimetro” , “PI”, “calcularRadio” (explicaremos más adelante).
- Las palabras clave no se pueden utilizar como identificadores.

La manera de hacer referencia a los distintos elementos que intervienen en un programa es darles un nombre particular a cada uno. En programación llamamos **identificadores** a los nombres usados para identificar un elemento dado en un programa.

Primer programa en C

Comenzamos con un sencillo programa en C que imprime una línea de texto.

```
1  /*
2  *  Descripción: Programa que saluda  al usuario con el mensaje Hola Mundo
3  *
4  */
5
6  #include <stdio.h>
7
8  // la función main inicia la ejecución del programa
9  int main(void)
10 {
11     printf("Hola mundo ");
12
13 } // fin de la función main
```

Comentarios En cualquier programa los programadores pueden incluir texto que será ignorado por el compilador si es le dice que es un comentario. En C hay dos formas de poner los comentarios.

`/*` para iniciar el comentario que finalizará con `*/`, para los comentarios multilínea. O bien `//` que indica que toda esa línea es un comentario.

En el ejemplo nos encontramos con los dos casos en las líneas de 1 a 3 y la línea 7 y 11.

Por estilo, las primeras líneas de un programa indican en lenguaje entendible por las personas la identificación de ese programa, su autor y la fecha de realización, de forma que habría que extender los comentarios de identificación de este programa de la forma:

```
/*
** Archivo: nombre_archivo.c
** Autor: Fulanito Menganito Blas
** Fecha: 21-02-24
**
** Descripción: Saludo al usuario
**
**
*/
```

Para los ejemplos que se incluyen en este documento y en la asignatura se utilizará otra versión distinta de los comentarios identificativos: se incluirá una breve descripción, el asunto que se cubre en ese código, como `funciones de math.h`, y la práctica o el trabajo individual al que hace referencia.

```
/*
** Descripción: <<texto ilustrativo de la funcionalidad>>
** Asunto: <<tema genérico por ejemplo condicionales, bucles,...>>
** Área: <<práctica o trabajo individual que resuelve>>
*/
```

Directiva del preprocesador `include` La línea

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

es una directiva del preprocesador de C. El preprocesador maneja las líneas que comienzan con `#` antes de la de la compilación.

Esta línea indica al preprocesador que incluya el contenido de la cabecera de entrada/salida estándar estándar (`<stdio.h>`). Este es un archivo que contiene información que el compilador utiliza para asegurarse de que utiliza correctamente las funciones de la librería de entrada/salida estándar como `printf` (línea 10). Esto se verá con detalle más adelante.

Líneas en blanco y espacios en blanco Hemos dejado la línea 8 en blanco. Se utilizan líneas en blanco, caracteres de espacio y caracteres de tabulación para facilitar la lectura de los programas. Juntos, se conocen como espacios en blanco y son generalmente ignorados por el compilador en C en otros lenguajes como Python son parte de su léxico.

Función principal - main La línea

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

forma parte de todos los programas en C. Los paréntesis después de main indican que `main` es un bloque llamado `function` (función). Los programas en C constan de funciones, una de las cuales debe ser `main` (programa principal).

Todo programa comienza a ejecutarse en la función main. Como buena práctica preceda cada función con un comentario indicando el propósito de la función.

Las funciones pueden devolver información. La palabra clave `int` a la izquierda de `main` indica que main "devuelve" un valor entero (número entero). En temas posteriores se profundizará en este apartado.

Por ahora, basta con incluir la palabra clave `int` a la izquierda de main en cada uno de los programas.

Las funciones también pueden recibir información cuando son llamadas a ejecutarse. El `void` entre paréntesis significa que main no recibe ninguna información.

Una llave izquierda, `{`, comienza el cuerpo de cada función (línea 9). La llave derecha `}`, termina el cuerpo de cada función (línea 11).

Cuando un programa alcanza el cierre de `main`, el programa termina. Las llaves y la parte del programa entre ellas forman un **bloque**. Llamamos **bloque de código** a las instrucciones comprendidas entre dos llaves.

Una instrucción de salida La línea

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

ordena a la computadora que realice una acción, a saber, mostrar en la pantalla la cadena de caracteres entre comillas. Una cadena se denomina a veces cadena de caracteres, un mensaje o un literal.

Se realiza la "llamada" a la función **printf** para que realice su tarea, el argumento de printf dentro de los paréntesis y el punto y coma (**;**) indica que se ha acabado la instrucción/orden al computador.

Toda instrucción debe terminar con un punto y coma. El "f" en printf significa "formateado".

Cuando se ejecuta, muestra el mensaje *Hola mundo* en la pantalla. Los caracteres normalmente se imprimen como aparecen entre las comillas dobles. Podrían haberse utilizado dos instrucciones:

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

Recursos adicionales

- Enlace para ver la notación en bits de un número real
- Enlace a Conversor en línea
- Enlace para el estudio de las formas de representación de información numérica
- Videos de Dr. Alberto Prieto sobre la arquitectura de una computadora

Fundamentos de la programación estructurada

Flujo de desarrollo de un programa

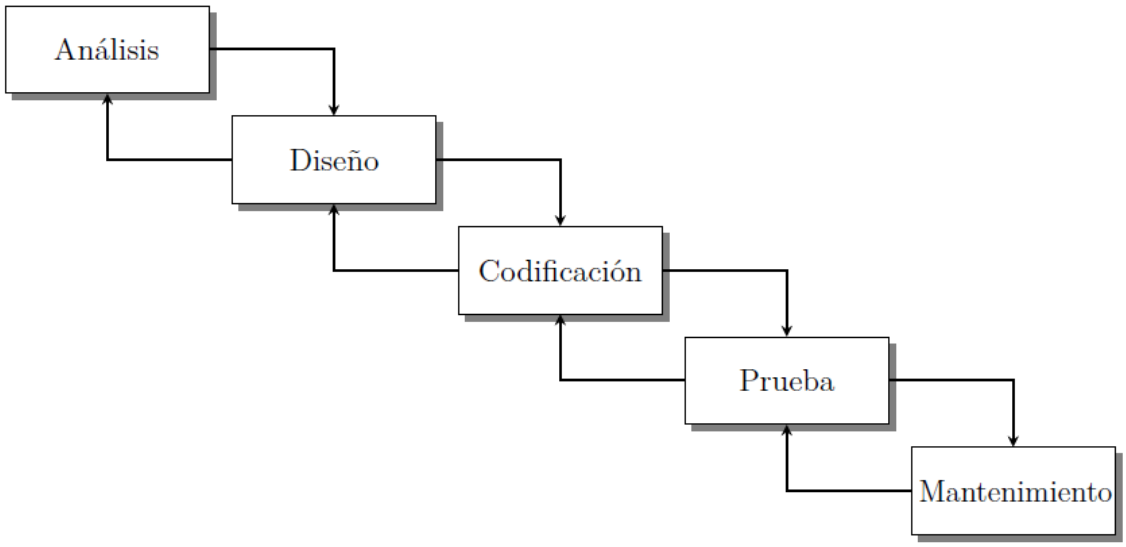
Crear un programa no es diferente de la resolución de problemas en general. Escribir un programa casi es el último paso del proceso de determinar, primero cuál es el problema y después el método que se usará para resolverlo, es decir el algoritmo.

La disciplina de ingeniería que se ocupa de la producción de software se denomina **ingeniería del software**.

La ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación, y mantenimiento del software.

El método usado por los profesionales que desarrollan software para entender el problema que se va a solucionar y para crear una solución de software efectiva y apropiada lo llamaremos procedimiento de desarrollo de software.

Este procedimiento, se ilustra en la figura, pero hemos de entender que se trata de una versión simplificada. Buscando un símil en ingeniería de la construcción, no es igual el método para construir un rascacielos que una casa unifamiliar, pero se fundamentan en los mismos principios, los ejemplos que se tratarán en esta asignatura utilizando este mismo símil son "casetas para mascotas".



El **análisis** representa el **Qué**. Conlleva la definición concreta del problema, la especificación de los datos de entrada y salida, la identificación de las tareas a realizar, así como la definición de los criterios de validación indicando cuáles son los casos para los que la solución debe resolver el problema.

Veámoslo sobre un ejemplo: queremos calcular el **área de una figura geométrica**. En el análisis estudiamos cuál es el problema exactamente y para que casos será válido el programa a construir, por ejemplo nuestras figuras serán triángulos y la información de la que disponemos es las coordenadas relativas de cada vértice del triángulo. Los valores de entrada son por tanto tres parejas de números y el resultado será un número real con la superficie del triángulo.

El **diseño** supone la elaboración de la solución, es decir, el **Cómo**. Conlleva la definición completa de los algoritmos y datos que se van a utilizar. Habitualmente se comienza con un diseño global o arquitectónico, donde se estudia la estructura y pasos a seguir sin utilizar ningún tipo de notación (incluso lenguaje natural - *preAlgoritmo*) para refinar el resultado mediante el **diseño detallado** de los algoritmos, utilizando una notación propia de la programación estructurada como puede ser el pseudocódigo o los diagramas de flujo de datos. Se trata de un proceso creativo que se dan en dos pasos de refinamiento, primero se hace el **diseño preliminar** donde se define la estructura de nuestro problema y la estrategia de resolución. Después durante el **diseño detallado** se define los pasos detallados que recogen el algoritmo.

Para el ejemplo del área del triángulo, el diseño arquitectónico sería decidir si se utiliza la formula clásica de dividir por dos el resultado de base por altura, o si bien la formula de Heron a partir del semiperímetro, que es la elegida en este caso. No obstante en ambos casos habrá que comprobar que es un triangulo propio (a < b+c).

El diseño detallado consiste en identificar los pasos correctos.

- Guardar coordenadas de tres vértices: v1, v2, v3.
- Calcular distancia a de v1 a v2.
- Calcular distancia b de v2 a v3.
- Calcular distancia c de v3 a v2.
- Buscar el lado más largo.
- Comprobar que su valor es más pequeño que la suma de los otros dos.
- Si se cumple la condición de Heron calcular el semiperímetro (a+b+c)/2
- area = raiz cuadrada (p(p-a)(p-b)(p-c)), siendo p el semiperímetro.

La **codificación** es la traducción a un lenguaje de programación de los algoritmos. Traducción del diseño a lenguaje de programación de alto nivel (programa fuente) que se traduce a lenguaje máquina (programa ejecutable), utilizando un compilador o un intérprete dependiendo del lenguaje.

La **prueba** o validación consiste en la ejecución del código escrito para comprobar que las salidas generadas se corresponden con la solución al problema según las entradas definidas para cada caso. Estos casos de prueba se construyen durante el análisis y representan las situaciones a las que se puede dar solución aplicando el algoritmo diseñado y codificado en el lenguaje de programación escogido.

El **mantenimiento** cambios en el programa debidos a:

- Corrección de nuevos errores.
- Adaptación a nuevos entornos (sistemas operativos, periféricos,...).
- Ampliaciones funcionales o de rendimiento.

Programación es la colección de actividades que rodean la descripción, el desarrollo y la implementación efectiva de soluciones algorítmicas a problemas bien especificados. En el ámbito de esta asignatura sobre el ciclo clásico de la ingeniería del software dejaríamos fuera el mantenimiento y habitualmente el trabajo de análisis esta simplificado puesto que se tratará de problemas bastante acotados y de complejidad mediana o pequeña. Con respecto a la fase de diseño, se hará enfocándose en la implementación final en lenguaje C.

Programación no es "codificar en un lenguaje de programación particular o para una arquitectura máquina particular", sino que consisten en el dominio de técnicas altamente estilizadas en un lenguaje de programación particular.

No obstante se deben cumplir unos criterios de calidad que definen lo que debe cumplir un programa:

- **Corrección**: la entrada definida produce los resultados requeridos (el programa se ajusta a la especificación).
- **Claridad**: prácticamente todos los programas se modificarán en el futuro. Su contenido debe ser entendible por personas distintas a su autor o por el mismo autor pasado un tiempo.
- **Eficiencia**: consumo óptimo de recursos de computación (tiempo de CPU, memoria,...).

A estos tres criterios básicos podemos añadir

- **Amigabilidad**: facilidad de uso para el usuario. En ciertos casos puede producir ineficiencia y se ha de priorizar.
- **Robustez**: entradas no definidas - mensajes de aviso sin generar bloqueo. Este criterio se relaja en ciertas condiciones.

Fases de construcción de un programa en C

La implementación de programas C suele pasar por seis fases para llegar a poder ser ejecutados: editar, preprocesar, compilar, enlazar, cargar y ejecutar, pero cuidado antes se ha de haber pasado por las etapas de análisis y diseño. Aunque en los entornos actuales esta división puede ser trasparente, es decir el usuario no ve de forma separada las fases y puede llegar a trasladarse a pulsar un botón en el entorno de desarrollo.

Fase 1: Creación de un programa La fase 1 consiste en editar un fichero en un programa editor: Los entornos de desarrollo de desarrollo C y C++ (IDEs) como Microsoft Visual Studio y Apple Xcode tienen editores integrados. Se escribe un programa C en el editor, se hacen correcciones si es necesario, luego se almacena el programa en un dispositivo de almacenamiento secundario como un disco duro, siendo necesario "guardar" un archivo. En C los archivos deben terminar con la extensión **.c**. También es posible trabajar sobre almacenamientos remotos o "codespaces" espacios de codificación donde el almacenamiento se hace remoto sobre herramientas tipo Git, como GitHub. Como recomendación si es posible se recomienda utilizar la opción de "autoguardado".

Fases 2 y 3: Preprocesamiento y compilación de un programa C Se da la orden para compilar el programa, entonces el compilador traduce el programa C a código de lenguaje de máquina (también denominado como código objeto). En un sistema C, el comando de compilación invoca un programa preprocesador antes de que comience la fase de traducción del compilador.

El preprocesador C obedece comandos especiales llamados directivas del preprocesador, que realizan manipulaciones de texto en los archivos de código fuente de un programa. Estas manipulaciones consisten en insertar el contenido de otros archivos y varios reemplazos de texto.

En la **Fase 3** el compilador traduce el programa C a código en lenguaje máquina:

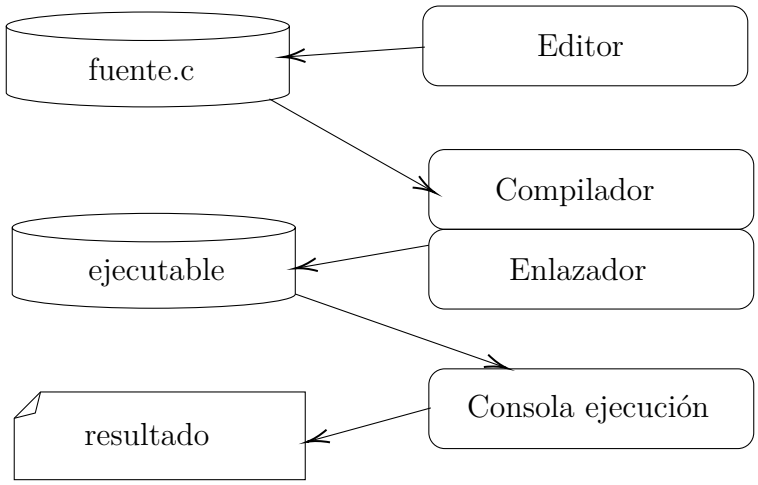
Un error de sintaxis ocurre cuando el compilador no puede reconocer una instrucción porque viola las reglas del lenguaje. El compilador emite un mensaje de error para ayudarle a localizar y corregir la instrucción incorrecta. El estándar C no especifica la redacción de los mensajes de error emitidos por el compilador, por lo que los mensajes pueden diferir entre los distintos sistemas. Los errores de sintaxis también se denominan errores de compilación o errores en tiempo de compilación.

Fase 4: de Enlazado o Linker Los programas en C suelen utilizar funciones definidas en otros lugares, como en las bibliotecas estándar bibliotecas de código abierto o bibliotecas privadas de un proyecto concreto. El código objeto producido por el compilador de C suele contener "agujeros" debidos a estas partes que faltan. A enlazador enlaza el código objeto de un programa con el código de las funciones que faltan para producir una imagen ejecutable (sin las piezas que faltan).

Fase 5 y 6: Carga o lectura y ejecución Antes de que un programa pueda ejecutarse, el sistema operativo debe cargarlo en memoria. El cargador toma la imagen ejecutable del disco y la transfiere a la memoria. Además, también se cargan los componentes adicionales de las bibliotecas compartidas que soportan el programa.

Finalmente, en la última fase, el ordenador, bajo el control de su CPU, ejecuta el programa instrucción a instrucción.

En los entornos actuales de codificación bastará con pulsar un único botón una vez editado el archivo y se lanzan todas las fases anteriores.



Veamos un ejemplo:

- Editamos con un editor local bloc de notas generando un archivo .c
- Subimos al codespace simplemente arrastrando el archivo `gcc hola.c -o hola.exe`
- Ejecutamos en la consola `./hola.exe`

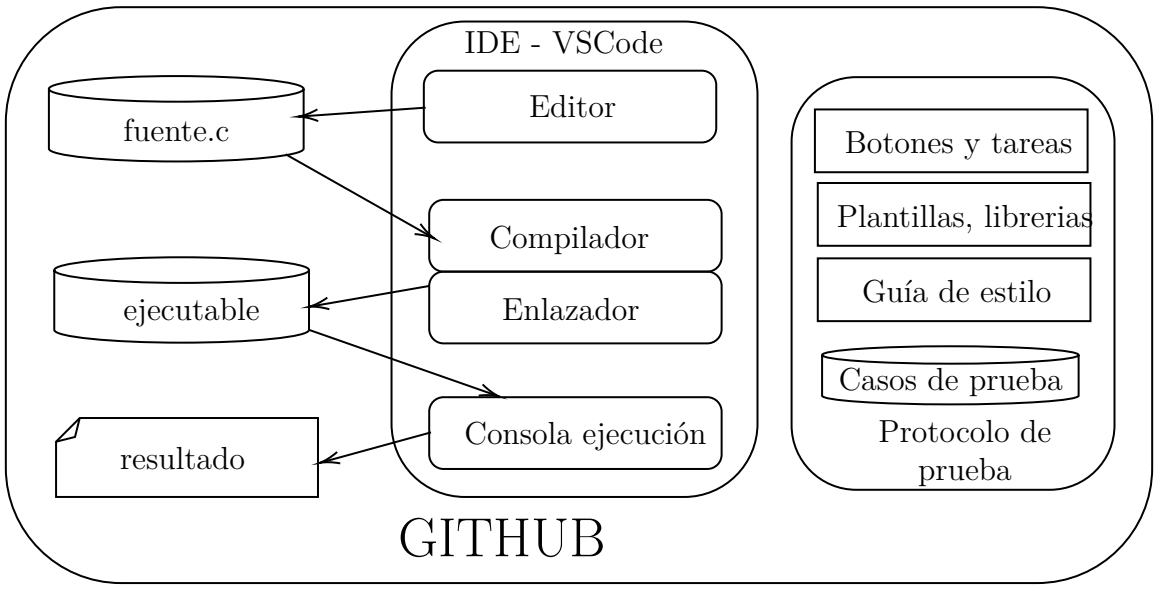
Ahora vamos a un entorno totalmente online - https://www.onlinegdb.com/online_c_compiler

O bien lo hacemos sobre VSCode en el codespace de Github.

Entorno de trabajo en la asignatura de programación

No obstante para trabajar más cómodamente se utilizan extensiones en VSCode que facilitan el trabajo, que validan parte del léxico de C o que facilitan la indentación del código.

Además hemos definido un entorno completo de ejecución, prueba y evaluación para la asignatura que presenta la siguiente estructura, pero sobre la que se irá profundizando poco a poco.



Pero no olvidemos: "¡Qué los árboles no nos impidan ver el bosque!". Las herramientas son lo de menos. Lo que nos interesa es aprender a programar. Y programar es diseñar y después escribir lo diseñado en un lenguaje de programación, como se compile o ejecute es lo de menos.

Tipos de datos simples

En este apartado se presentan los elementos mínimos de un lenguaje de **programación imperativo**. Este conjunto de elementos se particulariza para el lenguaje C.

Identificadores

En programación llamamos **identificadores** a los nombres usados para representar y utilizar un elemento dado en un programa (variables, constantes con nombre, o funciones...). En el ejemplo del "Hola mundo" "printf" es el identificador que los desarrolladores de C en la librería "stdio.h" le pusieron a la instrucción que permite mostrar algo en la pantalla. Este identificador no forma parte de las palabras reservadas de C, lo crearon quienes dieron forma a la librería "stdio.h". Aunque en la práctica pueda parecer parte del lenguaje C no lo es, sin incluir la librería el compilador no lo entiende.

Reglas generales para los identificadores:

- Utilizar solo letras, números y subrayados ('_').
- Sin espacios en blanco intermedios. El lenguaje utiliza los espacios para "entender" y compilar el código, al igual que el **;** da fin a la instrucción.
- El primer carácter debe ser una letra.
- No incluir la letra **'ñ'** ni las vocales acentuadas.
- No utilizar caracteres especiales.
- No usar palabras reservadas del lenguaje.
- Se distingue entre mayúsculas y minúsculas. **¡¡importante!!**

Recomendaciones:

- No utilizar identificadores que ya aparecen en las librerías habituales "printf", "sqrt".
- Utilizar siempre el mismo criterio en cuanto a la combinación de mayúsculas y minúsculas.
- Utilizar nombres significativos.
- Recordar que los acentos no son parte de los caracteres permitidos, ni la ñ.

Lo habitual es que los equipos de programadores definan una guía de estilo para dar una forma unificada a los programas que construyen donde se fija también como nombrar los identificadores. Existen diversas notaciones unificadas de formas de nombrar identificadores, al final de este tema se incluyen las normas a seguir en esta asignatura.

Clasificación de los tipos de datos

Recordemos que un computador es un máquina de tratamiento de información que manipula distintos tipos de datos. Un dato es elemento de información que es objeto de operación por parte de la computadora que tiene un tipo. Un tipo de dato es la propiedad abstracta asociada a la representación del dato en la computadora. Se caracteriza por:

- Implementación: representación interna.
- Dominio: rango de valores permitidos.
- Operaciones permitidas.

No obstante la información además de clasificarse por tu tipo, se distingue entre si puede o cambiar su valor entre varios posibles, en esta clasificación tenemos los **datos** y las **constantes**. Esta distinción es importante de cara a la representación sobre la máquina donde se ejecuten los algoritmos, los datos también se llaman **variables**, concepto retomaremos con detalle más adelante.

Clasificación de los tipos de datos:

- **Simples estándar** Simples o básicos (no estructurados):
    - Numéricos:
        - Entero

    - Real

  - Lógico o booleano
  - Carácter

- **Simples no estándar** o tipos enumerados. Por ejemplo las notas cualitativas de un estudiante.`{No presentado, Suspenso, Aprobado, Notable, Sobresaliente, Matricula}`
- **Compuestos o estructurados** o derivados:

  - Texto (cadena de caracteres)
  - Otros, incluye los definidos por el usuario (se estudiarán más adelante)

Las particularidades (implementación, dominio y operaciones) de cada tipo pueden variar entre lenguajes de programación. Pero se verán primero de forma general, para después particularizar su comportamiento en C.

Tipo numérico entero

La información se representa en los computadores utilizando un conjunto de celdas de memoria que almacenan dígitos binarios (0 o 1). La cantidad y el rango de los números enteros y reales en matemáticas es un conjunto infinito. El ordenador, por contra, dispone de un número finito de celdas en las que almacenar bits. La principal propiedad de un entero es que permite una **representación exacta**.

En lenguajes como C, de forma nativa, el tamaño máximo de un valor entero, tipo `int`, está determinado por la arquitectura del ordenador concreto que se utilice, típicamente 32 celdas binarias, bits, o sea 4 bytes (8 bits son 1 byte). Sin embargo, en Python, un lenguaje de más alto nivel, no hay un límite predeterminado para el valor absoluto del mayor entero representable. Este valor puede crecer ocupando más y más celdas hasta alcanzar los límites de la memoria disponible en el ordenador, a costa de la eficiencia.

- Implementación:

  - 2 bytes/4 bytes, complemento a 2

- Dominio

  - [-32768, 32767] (2 bytes) entero (**int**)
  - [-2147483648, 2147483647] (4 bytes) entero largo (**long**)

- Operaciones

  `+` Suma de enteros

  `-` Resta de enteros

  `*` Multiplicación de enteros

`/` DIV División de enteros (cociente)

Modulo `%` Resto de la división

`+` (unario) Identidad de enteros

`-` (unario) Cambio de signo

Tipo numérico real

Para los números reales, se utiliza una representación muy diferente a la de los enteros, denominada de punto (o coma) flotante, definida en el estándar IEEE 754. El tipo de dato se denomina **float** por ese motivo. El mayor y menor número real representable en valor absoluto, está en el rango aproximado [1e-323,1.7e308]. Esta es la consecuencia de utilizar un total de 64 bits, 8 bytes, para esta representación.

Se debe ser consciente del carácter limitado de la representación de los float. No solo no se puede expresar un número real tan grande como se quiera, sino que la precisión no es infinita y, por tanto, relativamente pocos números reales se pueden representar exactamente. Los cercanos a cero también tienen problemas.

Su aritmética es **aproximada** aunque permite un rango de valores más amplio.

- Implementación
  - IEEE-754, 4/8 bytes (simple/doble precisión)
- Dominio
  - -3.4E+38..-1.2E-38, 0, +1.2E-38..+3.4E+38 (**float**)
  - -1.7E+308..-2.3E-308, 0, +2.3E-308..+1.7E+308 (**double**)
- Operaciones

  `+` Suma de reales

  `-` Resta de reales

  `*` Multiplicación de reales

  `/` División de reales

  `+` (unario) Identidad de reales

  `-` (unario) Cambio de signo

Tipo lógico

Recoge la verdad o no de una situación. Este tipo de datos **no** está disponible en C. Debemos utilizar 2 bytes y emplear un entero para representar esta misma idea.

- Implementación -1 byte
- Dominio
  - Verdadero (true, 1, <>0)
  - Falso (false, 0)
- Operaciones `no` negación

  `y` conjunción

  `o` disyunción

Tipo de dato carácter

- Implementación:
  - 1 byte/ 2 bytes (**char**)
- Dominio:
  - Juego de caracteres disponibles en la computadora (ASCII, Unicode)
- Operaciones: Funciones que se verán más adelante.

Tipo de datos simples no estándar

Se definen mediante especificación con identificadores de los valores de su dominio (**enumeración**) ó mediante un subconjunto de un tipo ordinal (subrango). Son tipos de datos internos al programa no se pueden involucrar en operaciones de E/S y son isomorfos a un subconjunto de los enteros.

```
int entero = 10;
float decimal = 3.14;
double mayorPrecision = 3.1415926535;
char caracter = 'A';

enum Meses {ENERO, FEBRERO, MARZO, ABRIL, MAYO, JUNIO, JULIO,
            AGOSTO, SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE};
enum Meses miMes = ABRIL;
enum {FALSO = 0,    VERDADERO = 1}
```

Tipo texto

Los computadores son conocidos por su capacidad para trabajar con valores numéricos. De igual forma, cualquier usuario sabe que frecuentemente demandan otros tipos de datos, de carácter alfanumérico.

El manejo de textos es una tarea recurrente en las aplicaciones de los ordenadores (procesadores de texto, correos electrónicos, sistemas de mensajería, etc.). En todos estos casos, las cadenas de caracteres juegan un papel fundamental.

Una cadena de caracteres, tipo de dato **string** (no disponible como tal en C), representa precisamente eso: una ristra de caracteres alfanuméricos que puede ser tratada como una unidad.

En C se distingue entre caracteres (comillas simples 'a') y cadenas de caracteres (comillas dobles, "cadena").

Variables, operadores y expresiones

Constantes

Son valores que no cambian durante la ejecución del programa. Hay dos tipos de constantes (literales y simbólicas):

Constante literal Cualquier valor escrito directamente en una instrucción de un programa es una constante literal que puede ser de distintos tipos. Ej: 3.14159

Constantes Enteras: 10, -3987. Si no caben en un entero se almacenan en forma entero largo.

- Definición de constantes grandes directamente:
  - 123L, 123l
- Definición de constantes sin signo: - 45U, 45u
- Definición de constantes grandes sin signo:
  - 234UL (U antes de L)
- Enteros no decimales:
  - octales: 0123 (0 delante)
  - hexadecimales: 0X478 ó 0x478 (0X ó 0x delante).

Constantes Reales: punto fijo: 146.09, -234.3

Notación científica: 2.23E-15

Las constantes reales se almacenan en forma double.

Constantes de Caracteres: entre apóstrofes: 'A'. - en octal ASCII: '\101'.

Secuencias de escape (códigos de representación inconfundible: independientes del sistema de codificación). Permiten enviar caracteres de control no gráficos a un dispositivo de pantalla. Todas ellas comienzan con una barra invertida (\) seguida de otro carácter. En tiempo de ejecución, las secuencias de escape se sustituyen por los caracteres adecuados

`'\n'`	nueva línea
`'\t'`	tabulador horizontal
`'\v'`	tabulador vertical
`'\b'`	retroceso
`'\f'`	avance de página
`'\r'`	retorno de carro
`'\a'`	sonido
`'\0'`	nulo (marca de fin de cadena)
`'\?'`	signo de interrogación
`'\\'`	\ (barra invertida)
`'\''`	' (apóstrofe)
`'\"'`	" (comillas)

Constante simbólica (con nombre) Es un valor de dato constante que se referencia mediante un nombre (identificador).

```
PI
```

En C es posible definir constantes simbólicas utilizando dos alternativas, con la palabra reservada `const` o con la directiva del precompilador `#define`

```
const int NOMBRE = 10;
#define NOMBRECONSTANTE valor
```

La diferencia entre el uso de `const` y el uso de `#define` está en que mediante el primero se declara una constante que tiene un tratamiento semejante a una variable, pero sin poder cambiar su valor mientras que mediante define se indica que escribir el nombre especificado equivale a escribir el valor (de hecho se sustituye en pre-compilación por él), con una correspondencia directa.

Por ejemplo:

```
const int JUGADORES = 5;
#define JUGADORES 5
```

En la primera se indica que JUGADORES es una constante de tipo `int` (entero) mientras que en la segunda se indica que donde aparezca en el código la palabra JUGADORES deberá ser reemplazada por 5 directamente. En general, usar `#define` supone que la compilación sea más rápida. Por ello su uso resultará recomendable cuando existan ciertos valores numéricos que tengan un significado especial, valor constante y uso frecuente dentro del código. Las constantes definidas con **`#define`** se denominan constantes simbólicas, y algunas de ellas existen de forma predeterminada en el lenguaje.

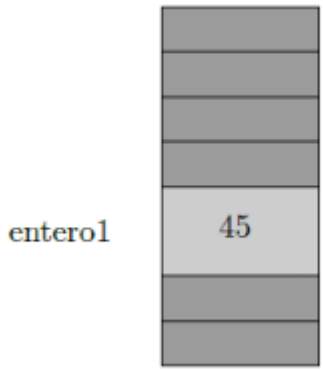
A continuación se incluye un ejemplo de uso de secuencias de escape y constantes simbólicas. La combinación `%s` permite formatear la salida, siendo un parámetro de la función `printf`.

```
#define NOMBRE "Computadora"

// la función main inicia la ejecución del programa
int main(void)
{
    printf("\n \n Hola mundo, me llamo  %s \n \n", NOMBRE);
    return 0;
} // fin de la función main
```

Variables

Se ha de distinguir entre dos conceptos, el de variable y el de identificador de la variable. Poniendo un ejemplo en un contexto distinto: Una cosa es la persona y otra el nombre que se utiliza para referenciarla. Isabel María del Águila es una persona, que se identifica como "profesora" o bien como "mamá", siendo la misma persona.

Variable es zona de memoria central que se referencia mediante un nombre o identificador, en lugar de por su dirección, donde se puede almacenar el valor de un dato que puede cambiar durante la ejecución del programa.




```
int main(void)
{

    int entero1 = 0; // guarda el primer numero introducido por el usuario
    int entero2 = 0; // guarda el segundo numero introducido por el usuario

    printf("Dame el primer entero: "); // mensaje en pantalla
    scanf("%d", &entero1);             // lee un entero

    printf("Dame el segundo entero: "); // mensaje en pantalla
    scanf("%d", &entero2);              // lee un entero

    int suma = 0;             // variable donde se guardará la suma
    suma = entero1 + entero2; // asigna el resultado de la operación a suma

    printf("La suma es %d\n", suma); // muestra el resultado

    return 0;
} // fin de main
```

Definición de variables Las líneas

```
int entero1 = 0; // contendrá el primer número que introduzca el usuario
int entero2 = 0; // contendrá el segundo número que introduzca el usuario
```

son definiciones. Los nombres `entero1` y `entero2` son variables-ubicaciones en memoria donde el programa puede almacenar valores para su uso posterior.

Estas definiciones indican que **entero1** y **entero2** son de tipo **int**. Esto significa que contendrán números enteros. enteros, como 7, -11, 0 y 31914 y que se reservará en memoria espacio (segun el tipo) para poderlos almacenar.

También se inicializan cada variable a 0 siguiendo el nombre de la variable con un `=` y un valor. Aunque no es necesario inicializar explícitamente cada variable, hacerlo ayudará a evitar muchos problemas si se usan sin haberlas inicializado.

Es necesario **definir** las variables antes de utilizarlas, a esto se le llama **declarar** una variable. Todas las variables deben definirse con un nombre o identificador y un tipo antes de que puedan utilizarse en un programa. Se puede colocar cada definición de variable en cualquier lugar de `main` antes de que la variable de esa variable en el código. En general, debe definir las variables cerca de su primer uso o bien al principio de la función que la utiliza.

Tal como hemos visto en el ejemplo, en el lenguaje C es necesario indicar el tipo que va a soportar una variable, para que así en tiempo de compilación se reserven en memoria las posiciones necesarias para manejarlo adecuadamente.

Un nombre de variable puede ser cualquier identificador válido. Cada identificador puede estar formado por letras, dígitos y guiones bajos (_), pero no puede empezar por un dígito. C distingue entre mayúsculas y minúsculas, por lo que `a1` y `A1` son identificadores diferentes.

Elegir nombres de variables significativos ayuda a que un programa esté autodocumentado, por lo que se necesitan menos comentarios.

Evite comenzar los identificadores con un guión bajo (_) para evitar conflictos con los identificadores generados por el compilador y los identificadores de la librerías estándar.

La función scanf y las entradas con formato Las dos linea mostradas a continuación sirven para que el usuario pueda introducir por teclado el valor de la variable con la que trabajar y se tratarán con detalle en el tema siguiente.

```
printf("Dame el primer entero: "); // mensaje en pantalla
scanf("%d", &entero1); // lee un entero
```

La primera sirve solo para mostrar un mensaje que ayude al usuario a dar un valor (`printf`). Se utiliza `scanf` para obtener un valor del usuario. La función lee de la entrada estándar, que normalmente es el teclado. La “f” en scanf significa “formateado”. Este scanf tiene dos argumentos- `%d` y `&entero1`. El primero es la cadena de control de formato. Indica el tipo de datos que el usuario debe introducir.

`%d` especifica que los datos deben ser un número entero. La `d` significa “entero decimal”. Un carácter `%` indica que especificación de conversión de formato.

El segundo argumento de `scanf` comienza con un ampersand (&) seguido del nombre de la variable. El `&` es el operador de dirección y, cuando se combina con el nombre de la variable, indica a scanf la ubicación (o dirección) en memoria de la variable entero1. scanf entonces almacena el valor que el usuario introduce en esa posición de memoria.

El uso del ampersand (&) es a menudo confuso para los programadores principiantes y para la gente que ha programado en otros lenguajes que no utilizan el ampersand, dirección. Por ahora bastará con preceder cada variable en cada llamada a scanf con un ampersand. Algunas variaciones a esta regla se tratan más adelante.

Solicitud e introducción del segundo número entero

```
printf("Dame el segundo entero: "); // mensaje en pantalla
scanf("%d", &entero2); // lee un entero
```

obtiene del usuario un valor para la variable **entero2**.

Declaración de la variable sum

```
int suma = 0; // variable donde se guardará la suma
```

define la variable `int sum` y la inicializa a `0` antes de que utilicemos `sum` en la línea siguiente.

Instrucción de Asignación

```
suma = entero1 + entero2; // asigna el resultado de la operación a suma
```

Calcula el total de la suma de las variables, luego asigna el resultado a la variable **suma** utilizando el operador de asignación (=).

Una vez que tenemos valores para `entero1` y `entero2`, la línea

```
suma = entero1 + entero2; // asigna el total a suma
```

suma estos valores y coloca el total en la variable suma, reemplazando su valor anterior. Aquí se combina una operación (+) con una operación de asignación (=).

Como se intercambian el contenido de dos variables en C:

```
// Intercambia el contenido de a y b utilizando una variable temporal
temp = a;
a = b;
```

Ciclo de vida de una variable Como resumen diremos que a una variable se le aplican cuatro acciones que definen los estados por los que pasa la variable y que no todos son necesarios dependiendo del tipo de variable o el lenguaje de programación que se utilice

- **Declaración**: Se indica el identificador asociado a la variable y su tipo. Es obligatorio en C, pero en python o en R no.
- **Inicialización**: Asignación de valor inicial a la variable. Consiste un dar un valor inicial fijo que prevenga de los problemas que podrían aparecer si se utiliza la variable antes de asignarle un valor. No es estrictamente necesaria, pero es muy recomendable y se puede incluir en la declaración

```
int i=0;
```

- **Utilización**: Durante la ejecución del programa una variable puede se utilizar en las instrucciones de dos formas básicas: dentro de expresiones o bien en operaciones de asignación. No se debe utilizar la variable en una expresión si no ha sido asignada o inicializada primero.
- **Destrucción**: Liberar la memoria asignada a esa variable. Esta operación es especialmente relevante cuando se utilizan estructuras de datos dinámicas referenciadas por punteros. En los ejemplos de esta asignatura no será necesario realizar destrucciones de variables. El programa recoge la basura, es decir libera la memoria que no se referenciada por un identificador.

Expresiones

Una **Expresión** es una combinación de constantes, variables, símbolos de operación y paréntesis que da como resultado un valor (de un tipo de datos determinado) también se le llama **operación**. Su formato es el clásico de las matemáticas.

Ejemplos:

$5/2 + 4 * 4$

$(x + y)/2$

$2 * \text{PI} * \text{radio}$

Elementos de una expresión:

- Operandos: valores constantes o variables.
- Operadores: manipuladores de los datos.

Los operandos de un operador deben de ser de un tipo de datos específico. Una operación/expresión genera un resultado de un tipo concreto.

Tipos de datos en C y como se declaran

Tipo	C	Rango	Tamaño
Enteros	short int	-32768 a 32767	2 bytes
	signed short int	-32768 a 32767	2 bytes
	unsigned short int	0 a 65535	2 bytes
	int	-2147483648 a 2147483647	4 bytes
	signed int	-2147483648 a 2147483647	4 bytes
	long int	-2147483648 a 2147483647	4 bytes
	signed long int	-2147483648 a 2147483647	4 bytes
	unsigned int	0 a 4294967295	4 bytes
	unsigned long int	0 a 4294967295	4 bytes
	long long int	-9223372036854775808 a 9223372036854775807	8 bytes
	unsigned long long int	0 a 18,446,744,073,709,551,615	8 bytes
Reales	float	$1{,}4 * 10^{-45}$ a $3{,}4 * 10^{38}$	4 byes
	double	$5{,}0 * 10^{-324}$ a $1{,}8 * 10^{308}$	8 bytes
Lógicos	int	0 (falso)	1 byte
	char	0 (falso)	1 byte
Carácter	char (tabla ASCII)		1 byte
	signed char	-128..127	1 byte
	unsigned char	0..255	1 byte
Cadenas	char [n]		n bytes

`signed` es el valor por defecto y no es necesario utilizarlo. Además, para los enteros se puede obviar `int`. Por tanto para definir un entero largo con signo se puede utilizar `signed long int` o solo `long`.

En el siguiente programa se muestran los límites y tamaños:

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

int main()
{
    printf("*\n");
    printf("Carácter con signo\n");
    printf("Mínimo: %d\n",CHAR_MIN);
    printf("Máximo: %d\n",CHAR_MAX);
    printf("*\n");
    printf("Carácter sin signo\n");
    printf("Mínimo: 0\n");
    printf("Máximo: %u\n",UCHAR_MAX);
    printf("*\n");

// Tamaños de tipos de datos primitivos
    printf("Tamaño de char: %zu bytes\n", sizeof(char));
    printf("Tamaño de short: %zu bytes\n", sizeof(short));
    printf("Tamaño de int: %zu bytes\n", sizeof(int));
    printf("Tamaño de long: %zu bytes\n", sizeof(long));
    printf("Tamaño de float: %zu bytes\n", sizeof(float));
    printf("Tamaño de double: %zu bytes\n", sizeof(double));
    printf("Tamaño de long long: %zu bytes\n", sizeof(long long));

    return 0;
}
```

Operadores en C Los operadores permiten definir operaciones entre variables y/o constantes. Existen muchos tipos de operadores distintos en función del tipo de resultado cuando se aplica o del número de constantes o variables necesarias para su utilización, es decir su modo de funcionamiento.

Según el modo de funcionamiento de los operadores pueden ser:

- Operadores unarios: solo se aplican a un operando

  `-1`

- Operadores binarios: Se aplican a dos operandos

  `1 + entero1`

Según tipo de operandos y de salida (resultado) obtenida los operadores pueden ser:

Tipo de operador	Tipo de operandos	Tipo de salida
Aritmético	numérico	numérico
Relacional	numérico/carácter/texto	lógico
Lógico	lógico	lógico
Carácter/texto	carácter/texto	carácter/texto
A nivel de bits	numérico/carácter	numérico/carácter
Condicional	lógicos	instrucción
Asignación	identificador/ expresión	-

- Aritméticos:
  - Binarios: `+` `-` `*` `/` `%` (resto)
  - Unarios: `-` (cambio de signo)
- Relacionales y lógicos:
  - Relacionales: `>=` `<` `<=`
  - De igualdad: `==` (igual a) `!=` (distinto de)
  - Conectivas lógicas: `&&` (“and” lógico) `||` (“or” lógico)
  - Negación lógica: `!` (“not” lógico)

p	q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

En C no hay tipos lógicos. En su lugar se utilizan valores numéricos: verdadero<>0 y falso=0.

- Incremento y decremento:

- Incrementa en uno una variable. ++
- Decrementa en uno una variable. --

Si se utilizan como prefijo (delante de la variable) producen el efecto antes de utilizar la variable en la expresión en que esté. Si se utilizan como sufijo, producen su efecto después de utilizar la variable en la expresión en que esté. Su utilización genera un código más eficiente.

Nota: no usarlas en variables que se empleen más de una vez en una expresión ó como argumentos de una función.

- Lógicos para bits:
  - Binarios: &Y lógico a nivel de bits (bit a bit).
    | O lógico a nivel de bits.
    ^ O lógico exclusivo a nivel de bits.
    << Rotación a la izq. (x<<y x*2y)
    >> Rotación a la der. (x<<y x/2y)
  - Unario: ~ Complemento a 1.

Los operadores a nivel de bit en C ofrecen ventajas principalmente en situaciones donde se necesita realizar manipulaciones directas sobre los bits de una variable, como en la programación de sistemas embebidos, manipulación de datos comprimidos, implementación de algoritmos de cifrado.

```
*/

#include <stdio.h>

int main() {
```

- Operador condicional:

```
expresión1 ? expresión2 : expresión3
```

Este operador (**?:**) evalúa la **expresión1**, si esta es verdadera, da el resultado de **expresión2**, y en caso contrario da el resultado de **expresión3**. Ejemplos:

```
 c=(a>b)?a:b;

printf("%s\n", calificacion >=70 ? "Aprobado" : "Suspenso");
```

- Operador de asignación

  =

Se considera como un operador con la mínima prioridad. Es decir lo último que se hace es la asignación.

Funciones internas Representan operaciones no elementales que se incorporan en las implementaciones de los lenguajes de programación equivalen a unos operadores especiales que tienen aspecto de funciones matemáticas. Ej:

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

Función	tipo	significado
acos(d)	double	Arco coseno (0-pi) del argumento (-1,+1)
asin(d)	double	Arco seno (-pi/2,+pi/2) del argumento (-1,+1)
atan(d)	double	Arco tangente (-pi/2,+pi/2) del argumento
atan2(d1,d2)	double	Devuelve el arco tangente de d1/d2
ceil(d)	double	Devuelve el entero mas pequeño mayor ó igual al argumento (redondeo hacia arriba).
cos(d)	double	Coseno del argumento expresado en radianes
cosh(d)	double	Coseno hiperbólico del argumento
exp(d)	double	Exponencial del argumento
fabs(d)	double	Valor absoluto de un número real
floor(d)	double	Redondeo hacia abajo (mayor entero menor ó igual al argumento)
fmod(d1,d2)	double	Devuelve el resto de d1/d2 con el mismo signo que d1
labs(l)	long int	Valor absoluto de un entero largo
log(d)	double	Logaritmo natural del argumento
log10(d)	double	Logaritmo decimal del argumento
pow(d1,d2)	double	Calcula d1 elevado a d2
sin(d)	double	Seno del argumento expresado en radianes
sinh(d)	double	Seno hiperbólico del argumento
sqrt(d)	double	Raíz cuadrada positiva del argumento
tan(d)	double	Tangente del argumento expresado en radianes
tanh(d)	double	Tangente hiperbólica del argumento

Nota: la segunda columna indica el tipo de dato devuelto por la función. En la primera columna, los argumentos que aparecen son: c: carácter, d: doble precisión, s: cadena.

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

int main()
{
   system("cls||clear");
   double x1, y1, x2, y2;

   printf("Ingrese las coordenadas del primer punto (x1 y1): ");
   scanf("%lf %lf", &x1, &y1);
   printf("Ingrese las coordenadas del segundo punto (x2 y2): ");
   scanf("%lf %lf", &x2, &y2);

   double distancia = sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));

   printf("La distancia euclidiana entre los dos puntos es: %.2lf\n", distancia);

   return 0;
}
```

Reglas de prioridad Precedencia y orden de evaluación:

Operador	Asociatividad
( ) [ ] -> .	I-D (Izquierda-Derecha)
! ~ ++ – - (tipo) * & sizeof	D-I
* / %	I-D
+ -	I-D
« »	I-D
< <= > >=	I-D
== !=	I-D
&	I-D
|	I-D
&&	I-D
||	I-D
?:	D-I
= += -= ...	D-I
,	I-D

Nota: hay que tener cuidado con la utilización de operandos dependientes del resultado de otros operandos.

Normas de estilo para la programación en lenguaje C. Variables

Un mismo programa se puede escribir de varias formas. Todas ellas pueden ser compiladas correctamente y obtener un ejecutable, pero algunas de ellas son más fáciles de entender por las personas que otras.

> "Cualquier tonto puede escribir código que una computadora pueda entender.
> Los buenos programadores escriben código que los humanos pueden entender".
> - Martin Fowler.

A continuación incluimos distintos aspectos que mejoran la legibilidad del código que consideraremos como *normas de estilo*.

Una *Normas de estilo* o *guía de estilo* es el documento que explica cómo debe escribirse código C en un determinado contexto. Este estilo cambia de una institución a otra, pero en entornos industriales se suele exigir un escrupuloso respeto a estas reglas. A continuación enumeramos las reglas que vamos a exigir en esta asignatura. y que serán *obligatorios* para los ejemplos desarrollados en la asignatura de Programación, en especial para los ejercicios de prácticos y trabajos evaluables.

Identificadores significativos

Al programar se trasladan palabras del mundo real o nombres de diferentes componentes al lenguaje que se este usando, representando dichos términos como identificadores de variables, funciones o clases. Normalmente los identificadores deben empezar por una letra, no pueden contener espacios (ni símbolos raros) y suelen tener una longitud máxima que puede variar, pero que no debería superar los 10-20 caracteres para evitar lecturas muy pesadas.

Por ejemplo si tenemos una variable que va a guardar el **área de un circulo** calculada por el programa que estamos escribiendo, se podría utilizar el nombre **areacirculo**, **acircu**, **area_circ**, **area_circulo**.

¿Cuál es el más legible?, ¿Somos consistentes en todos los identificadores?

- Los nombres de variables, funciones y ficheros deben ser cortos, descriptivos y concretos, y se debe ser *consistente* todos los programas.
- Existen diversas notaciones para crear los nombres combinando varias palabras:
  - Camel Case (areaCirculo)
  - Pascal Case (AreaCirculo)
  - Snake Case (area_circulo)
  - Kebab Case (area-circulo)
- Las macros y constantes deben escribirse en mayúsculas para distinguirlas de variables y funciones

```
#define PI 3.141592
```

- Es muy normal usar variables como **i**, **j** o **k** para nombres de índices de instrucciones de iteración, lo cual es aceptable siempre que la variable sirva sólo para el bucle y no tenga un significado especial. Se suelen llamar **variables triviales**. Incluimos **c** para variables de carácter por ejemplo para menús.
- Para las variables en general se utilizarán *camelCase*
- Uso de verbos en identificadores de función. Se suelen usar las formas de los verbos en infinitivo, seguido de algún sustantivo.

Por ejemplo, una función podría llamarse **escribirOpciones**, y sería más comprensible que si le hubiéramos llamado **escribir** o **escr_opc**. Si la función devuelve un valor, su nombre debe hacer referencia a este valor, para que sea más expresivo usar la función en algunas expresiones, como:

```
precioTotal = precioTotal + IVA(precioTotal,16) +
     gastosTransporte(destino);
```

- Para variables globales: **g_camelCase**
- Variables puntero. Se añade el prefijo **p_**
- Para **struct** se utiliza una notación similar a las variables, es decir, **camelCase**

```
struct complejo{
        float parteReal;
        float parteImaginaria;
};
```

- Variables miembro en una **struct**: camelCase
- Para tipos definidos por usuario se antepone el prefijo **tipo_** o bien se coloca el nombre en mayúsculas. Cabe decir que es habitual utilizar un `typedef` para una estructura.

```
typedef struct{
        float parteReal;
        float parteImaginaria;
}tipo_complejo;

typedef struct{
        float parteReal;
        float parteImaginaria;
}Complejo;

Complejo complejo;
```

- Las cadenas de caracteres son un elemento habitual se aconseja una notación que indique el tamaño de la cadena. Eliminado en este caso el prefijo **tipo_**

```
typedef char cadena20[21];
  Cadena20 nom;
```

- Con respecto a los vectores, cuando se definen directamente aplica lo mismo que a los identificadores para las variables, si se define un tipo se utiliza el prefijo.
- Para los nombres de archivo se utiliza **PascalCase**.

```
AceleracionNormal
HolaMundo
AreaTriangulo
```

- Para que el entorno de prueba funcione correctamente se debe ser estricto en el nombrado de archivos, para las versiones entregables de los ejercicios es necesario incluir "PR" a los nombres de archivo:

```
AceleracionNormalPR
HolaMundoPR
AreaTrianguloPR
```

Estructuras de control del programa

Programación es la colección de actividades que rodean la descripción, el desarrollo y la implementación efectiva de soluciones algorítmicas a problemas bien especificados en forma de **programa**, que no es mas que es una lista de instrucciones máquina que al ejecutarse producen un resultado concreto.

Nosotros nos centraremos en la definición de programas como un conjunto de **instrucciones**, lo que llamamos **programación estructurada**. Clasificamos las instrucciones en:

- Instrucciones de asignación, que utilizan el operador de asignación.
- Instrucciones de entrada salida
- Instrucciones de control del programa: las estructuras de control de programa son construcciones fundamentales en la programación que permiten controlar el flujo de ejecución de un programa. Estas estructuras determinan la secuencia en que se ejecutan las instrucciones, tomando decisiones y repitiendo bloques de código según sea necesario.

Un programa contiene la información codificada del comportamiento deseado o descrito en el algoritmo, que se puede representar de distintas formas antes de traducirlo aun lenguaje de programación.

Representaciones de algoritmos

El método para representar los cómputos/algoritmos, será del tipo procedimental estructurado, es decir vamos a describir los pasos para resolver un problema que luego se traducirá a un lenguaje de programación para obtener código.

Algoritmo estructurado: secuencia ordenada y finita de pasos/operaciones que conduce a la solución de un problema, pero puede ser expresada de distintas formas.

Requisitos de los algoritmos:

- Precisos: indican el orden de realización de cada paso.
- Definidos: siempre deben de dar el mismo resultado.
- Finitos: deben de terminar en algún momento

Ejemplos de algoritmos:

- Receta de cocina
- Instrucciones para cambiar la rueda de un coche
- Grabación de una película de vídeo
- Resolver una ecuación diferencial de primer orden

Programa: descripción interpretable por la máquina del algoritmo a ejecutar por esta.

El algoritmo es independiente del lenguaje de programación en el que se escribe el programa y de la computadora donde se ejecuta.

Vamos a ver las formas de representar algoritmos sobre un ejemplo

Problema: Construir un programa que calcule e imprima en pantalla la masa en kilogramos de una bola de hierro dado por teclado su diámetro en centímetros.

Análisis Como información necesaria :

- Diámetro en cm de la esfera (>0) - La da de **entrada** el usuario
- Densidad del Fe, que es fijo para el material.
- Masa en Kg - Se ofrece como **salida**

Diseño Aquí es donde podemos utilizar diferentes representaciones que van del lenguaje natural, pasando por el pseudocódigo a la utilización de notaciones gráficas.

Lenguaje natural

1. Obtener (por teclado) el valor del diámetro de la bola en cm.
2. Calcular el radio de la esfera, radio=diámetro/2.
3. Calcular el volumen de la esfera en cm3, volumen=4 * PI * radio3 / 3, con PI=3.141593.
4. Calcular la masa en Kg, masa=densidad*volumen, con densidad=7.86/1000 (Kg/cm^3).
5. Presentar (en pantalla) el resultado.

Notación algorítmica sencilla (lista informal de pasos), pero poco rigurosa y en ocasiones ambigua para especificar los detalles procedimentales de la solución (el lenguaje natural es ambiguo).

Otras notaciones más adecuadas para representar el diseño detallado o procedimental son:

- Gráficas: diagramas de flujo, diagramas de Nasi-Shneiderman (diagramas N-S o diagramas de Chapin)
- Tabulares: tablas de decisión
- Textuales: LDP (lenguaje de diseño de programas o pseudo-código)

Pseudocódigo

El pseudocódigo es un lenguaje artificial informal similar al lenguaje cotidiano que le ayuda a desarrollar algoritmos antes de convertirlos en programas en un lenguaje estructurados. El pseudocódigo es cómodo y fácil de usar. Ayuda a "pensar" un programa antes de escribirlo programa antes de escribirlo en un lenguaje de programación. Los ordenadores no ejecutan pseudocódigo. El pseudocódigo se compone únicamente de caracteres, por lo que puede escribirse en cualquier editor de texto. A menudo, convertir a C un pseudocódigo cuidadosamente preparado es tan sencillo como sustituir una sentencia de pseudocódigo por su equivalente en C, incluso hay aplicaciones software que lo hacen

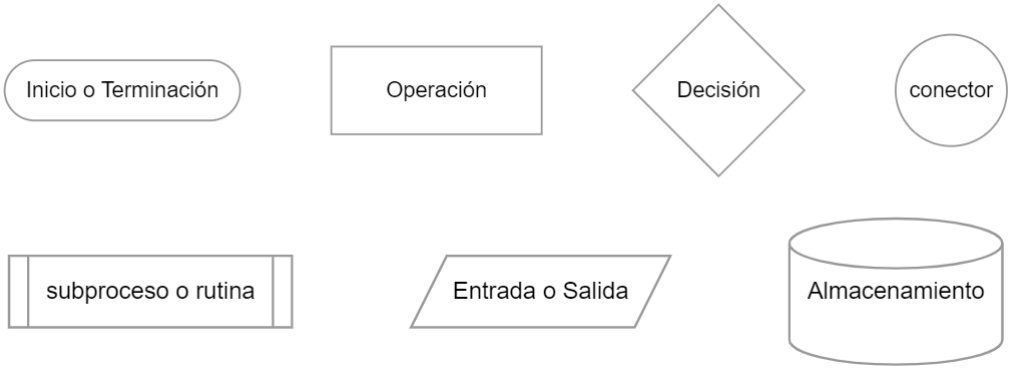
Se trata de una técnica textual que utiliza vocabulario de un lenguaje natural y la sintaxis de un lenguaje de programación.

Se emplea texto descriptivo incrustado dentro de las estructuras sintácticas de las instrucciones, centrándose en la lógica del problema más que en los detalles sintácticos.

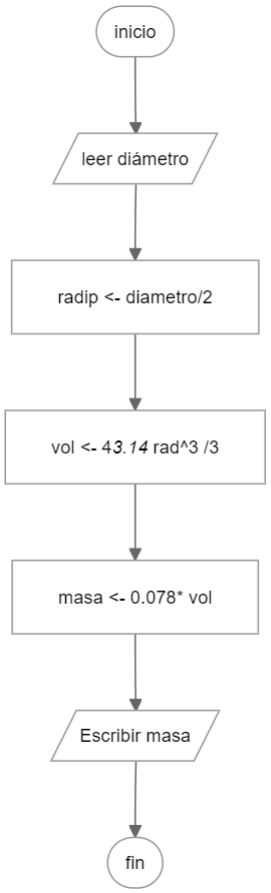
```
Algoritmo calcularMasaBolaHierro

        densidad <- 0.00786;

        Escribir "Introduzca el diámetro (cm): ";
        Leer diametro;
        radio <- diametro/2;
        volumen <- 4* PI * radio * radio * radio/3;
        masa <- 0.00786 * volumen;
        Escribir "Masa: ", masa, " Kg";

Finalgoritmo
```

Diagramas de flujo

Técnica gráfica de representación de algoritmos que utiliza símbolos estándar para representar las operaciones. La secuencia de operaciones viene indicada por flechas (líneas de flujo).Los elementos usuales de representación son:



Ejemplo



El pseudocódigo o los diagramas representan el **diseño detallado** que debe ser traducido a un lenguaje de programación.

```c
/*
** Archivo: MasaBolaHierro.c
** Fecha: 21-02-24
**
** Descripción: Calculo Masa bola de hierro
** Asunto: secuencial, ejemplo completo
*/

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

#define PI 3.1415
#define densidad 0.00786 /* Kg/cm3                    */

int main()
{
  char c;
  float diametro; /* diametro de la esfera    */
  float radio;    /* radio de la esfera (cm)  */
  float volumen;  /* volumen de la esfera     */
  float masa;     /* masa en kg               */

  printf("CÁLCULO DE LA MASA DE UNA BOLA DE HIERRO\n");
  printf("========================================\n\n");

  printf("Introduzca el diámetro (cm): ");
  scanf(" %f", &diametro);

  radio = diametro / 2;
  volumen = 4 * PI * radio * radio * radio / 3;
  masa = densidad * volumen;

  printf("\nMasa: %.2f Kg", masa);

  return 0;
}
```

```java
/*
** Descripción: Calculo Masa bola de hierro
*/

import java.util.*;

public class masaBolaHierro {
        public static void main(String[] args){
                Conio2.consola();
                final float PI = (float) 3.14159;
                final float DENSIDAD = (float) 0.00786;       /* Kg/cm3 */
                char c;
                float diametro;         /* diámetro de la esfera        */
                float radio;            /* radio de la esfera (cm)      */
                float volumen;          /* volumen de la esfera         */
                float masa;             /* masa en kg                   */
                Scanner teclado = new Scanner(System.in);
                teclado.useLocale(Locale.ENGLISH);
                do{ Conio2.clrscr();
                    System.out.printf("CÁLCULO DE LA MASA DE UNA BOLA DE HIERRO\n");
                    System.out.printf("========================================\n\n");
                    System.out.printf("Introduzca el diámetro (cm): ");
                    diametro=teclado.nextFloat();
                    radio=diametro/2;
                    volumen=4*PI*radio*radio*radio/3;
                    masa=DENSIDAD*volumen;
                    System.out.printf(Locale.ENGLISH,"\nMasa: %.2f Kg", masa);
                    System.out.printf("\n\nDesea efectuar una nueva operacion (s/n)? ");
                    c=Character.toUpperCase(Conio2.getch());
                }while (c!='N');
                teclado.close();
                System.exit(0);
        }
}
```

Anatomía de un programa en C

Como resumen de en que consiste una solución estructurada en forma de un programa en C se incluye una plantilla de como organizar un programa en este lenguaje:

```c
/* Includes de la aplicacion */
// #include "Ejemplos.h"

/* constantes */
#define MAXIMO 100
#define ERROR "Opcion no permitida"

/* tipos definidos por el usuario */

/* Prototipo de funciones locales */

int main(void)
{

    // Instrucciones, bloques de código,  que definen la funcionalidad implementada

    return 0;
}

/* Definiciones de funciones locales */
```

Concepto de bloque de código En C, un bloque de código se refiere a un conjunto de instrucciones delimitadas por llaves `{ }`. Estos bloques se utilizan para agrupar múltiples instrucciones juntas y pueden aparecer en lugares donde se espera una sola instrucción.

Un bloque de código en C es una secuencia de instrucciones delimitadas por llaves { } que se utilizan para agrupar instrucciones relacionadas. Pueden aparecer en varias situaciones, como en funciones, estructuras de control, inicialización de variables, entre otros.

Instrucción de Asignación

Una instrucción de asignación es una operación fundamental en programación que se utiliza para establecer o actualizar el valor de una variable. La estructura básica de una instrucción de asignación consiste en un **identificador** (nombre de la variable), un operador de asignación (generalmente =) y una **expresión** cuyo valor se asigna a la variable.

Componentes de una Instrucción de Asignación:

- Variable: El nombre de la variable a la que se le va a asignar el valor.
- Operador de Asignación: El símbolo `=`, que indica que el valor de la expresión a la derecha debe ser asignado a la variable a la izquierda. Es un operador con sus reglas de prioridad.
- Expresión: Puede ser un valor constante, otra variable, o una combinación de valores y variables a través de operadores aritméticos o de otro tipo.

```
variable=expresión;
```

Se pueden realizar asignaciones en tubería:

```
variable1=variable2=...=variableN=expresión;
```

El operador de asignación es asociativo de derecha a izquierda.

Acción que da el valor de una expresión a una variable

```
variable = expresión;
```

el valor resultante al evaluar la expresión de la parte derecha es almacenado en la posición de memoria que representa la variable de la izquierda.

Ejemplos:

```
x = 4;
y = 5;
z = (2 * x + y) % 3;
```

Características:

- Destructiva: se pierde el valor anterior de la variable.
- Distinta de la igualdad matemática: n = n+1
- El tipo de dato del resultado de la expresión ha de coincidir con el de la variable en la que se almacena. Nota: en muchos lenguajes de programación se relaja esta regla, controlando el compilador algunas conversiones de tipos de datos (por ejemplo almacenar un valor entero en una variable real).
- En las expresiones con operadores que admiten varios tipos de datos (normalmente numéricos), se hace la conversión del resultado y de los cálculos intermedios al tipo de dato con rango más amplio.

Errores posibles en instrucciones de asignación

- Alguna de las variables que aparecen en la expresión ESTÁN SIN INICIALIZAR

```
int e, f, g;
e=6;
g = e * f / 2 + 5 - (e - f); // Error ya que f NO SE HA INICIALIZADO
```

- El tipo de la variable y de la expresión NO COINCIDEN.

```
int e, f, g;
f=7.45; // Error ya que el tipo de f (int) no es un número real
```

Conversiones de tipo C no hace una comprobación fuerte del tipo de dato, y permite que tipos de datos diferentes se puedan mezclar en una misma expresión. Los char y los int se mezclan libremente.

Como norma general, cuando en una expresión aparecen tipos de datos diferentes, los tipos inferiores se convierten al tipo superior de dicha expresión y ese será el tipo del resultado:

En las asignaciones también se produce una conversión del tipo de dato de la expresión de la derecha al tipo de dato de la variable de la izquierda:

- de float a int por truncamiento.
- de double a float por redondeo.

- de long a int ó short por eliminación de bits de orden superior.
- de int a char por eliminación de los bits de orden superior.

Existe también la posibilidad de asignar un tipo a un dato (constante o variable) mediante un operador de conversión de tipo **("cast")**, de la siguiente manera:

```
(nombre_del_tipo)dato;
```

Ejemplo:

```
{
    printf("Valor de C: %f\n",1/3);
    printf("Valor de C: %f\n",1.0/3);
    printf("Valor de C: %f\n",(float)1/3);
    printf("Valor de C: %f\n",(float)(1/3));

{
    int a=3, b=2;
    float c;
    c=a/b;
```

- Operadores de acción sobre las variables:

```
+=   -=   *=   /=   %=   <<=   >>=   &=   ^=   |=
```

éstos últimos se utilizan en asignaciones de la forma:

```
variable += expresión
```

que equivale a:

```
variable = variable + expresión
```

Instrucciones de Entrada y Salida de información

C ofrece un conjunto de funciones para realizar operaciones de entrada y salida (E/S) con las que se puede leer y escribir cualquier tipo de archivo.

En C, un archivo se puede referir a un archivo de disco, a un terminal, a una impresora, un sensor, un motor, etc. Dicho de otro modo, un archivo representa un dispositivo concreto con el que puedes intercambiar información. Se trata el archivo como una serie de bytes (o caracteres) que es lo que realmente se transfiere entre un archivo y un programa, se le conoce como flujo (**stream** en inglés). Antes de poder trabajar con un archivo, se tiene que abrir ese archivo.

En C existen tres **streams** que ya están abiertos, disponibles para que se usen en cualquier programa:

- **stdin**: La entrada estándar de lectura. Generalmente va asociado al teclado.
- **stdout**: La salida estándar de escritura. Generalmente va asociado a la pantalla del terminal.
- **stderr**: La salida estándar de escritura para mensajes de error. Generalmente también va asociado a la pantalla del terminal.

Las funciones que permiten manejar la salida en C forman parte de la biblioteca estándar y no forman parte de la definición del C (C no tiene funciones predefinidas). Deben incluirse las librerías que las manejan

```
scanf("cadena de control",argumento1,argumento2,...);
printf("cadena de control",argumento1,argumento2,...);
```

La cadena de control va entre comillas y puede contener:

- Caracteres normales ASCII.
- Caracteres especiales (Secuencias de escape):

`\n` nueva línea `\t` tabulador ...

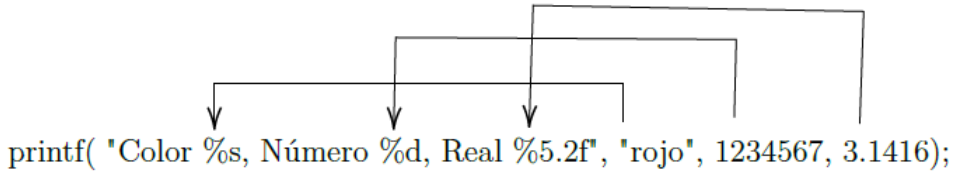
- Secuencias de salida/entrada: indican el tipo de dato del argumento a escribir ó leer (correspondencia con los argumentos)

printf:

carácter	tipo dato
`%d`	entero decimal
`%i`	entero con signo
`%x`	entero hexadecimal sin signo
`%o`	entero octal sin signo
`%u`	entero decimal sin signo
`%c`	un solo carácter
`%s`	una cadena de caracteres (\0 es el último carácter)
`%e`	real en notación exponencial
`%f`	real en notación decimal
`%g`	%e ó %f, según mas corto
`%%`	para escribir %

scanf:

carácter	tipo dato
`%d`	entero decimal
`%u`	entero decimal sin signo
`%o`	entero octal sin signo
`%x`	entero hexadecimal sin signo
`%i`	entero decimal, octal o hexadecimal
`%c`	un solo carácter
`%s`	una cadena (scanf lee palabras)
`%f`	real simple precisión
`%lf`	real doble precisión
`%e`	real
`%h`	entero short



Los argumentos de printf pueden ser constantes, variables y expresiones, y los de scanf han de ser variables. Al encontrar una secuencia de salida/entrada, se busca el siguiente argumento y se imprime/lee según se indica.

Ejemplos:

```
printf("v1=%d\nv2=%d\n",5+4,3*5);
scanf(" %d",&ve);    /* leer por teclado un entero */
scanf(" %c",&vc);    /* leer por teclado un carácter */
scanf(" %f",&vf);    /* leer por teclado un real s.p. */
scanf(" %lf",&vd);    /* leer por teclado un real d.p */
```

Observe que los argumentos de scanf han de ser variables. Cualquier variable de los tipos básicos ha de estar precedida por & (las cadenas no lo necesitan).

Modificadores de expresiones de conversión:

Formateadores de printf:

```
%-n1.n2f

%-n1.n2s
```

n1 anchura mínima del campo - ajustar a la izquierda del campo (defecto: derecha)

n2 reales: número de cifras decimales a la derecha del punto (6 por defecto) cadenas: número de caracteres a imprimir

Formateadores de scanf:

```
%*d
%nf
```

* no trata el correspondiente valor

n longitud máxima que puede tener la lectura

Ejemplos de utilización de scanf para leer cadenas de caracteres con espacios en blanco:

```
scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]s", linea);
```

Asigna a la variable de cadena linea los caracteres introducidos por el dispositivo estándar de entrada, y finalizará la lectura cuando se aparezca un carácter diferente de los encerrados entre corchetes.

No obstante existen otras funciones que permiten la entrada y salida de datos sobre los streams.

`sprintf ()` - Imprimir datos formateados en almacenamiento intermedio

`fprintf ()` - Escribir datos formateados en una stream especificado como argumento

fprintf

La función fprintf es una parte de la biblioteca estándar de C y se utiliza para escribir texto formateado a un archivo. Su funcionamiento es similar al de printf, pero mientras printf envía la salida a la consola (stdout), fprintf envía la salida a un archivo especificado.

```
int fprintf(FILE *stream, "<<cadena de formato>>", ...);
```

- `FILE *stream`: Un puntero a un objeto FILE que identifica el archivo donde se va a escribir. Este objeto se obtiene al abrir un archivo con la función fopen.
- `"<<cadena de formato>>"`: Una cadena de formato que sigue las mismas reglas que las cadenas de formato usadas en `printf`. Esta cadena puede contener caracteres literales y especificadores de formato que se reemplazarán con los valores de los argumentos adicionales.
- ...: Una lista de argumentos que se insertarán en la cadena de formato en las posiciones especificadas por los especificadores de formato.

```c
#include <stdio.h>

int main() {

    // Abrir un archivo en modo escritura
    FILE *file = fopen("salida.txt", "w");


    int edad = 25;
    float altura = 1.75;
    fprintf(file, "Edad: %d años\n", edad);
    fprintf(file, "Altura: %.2f metros\n", altura);

    // Cerrar el archivo
    fclose(file);

    return 0;
}
```

Se escribe en el archivo `salida.txt` texto formateado en el archivo. Los especificadores de formato (%d y %.2f) se reemplazan por los valores de edad y altura, respectivamente.

Teorema de la programación estructurada

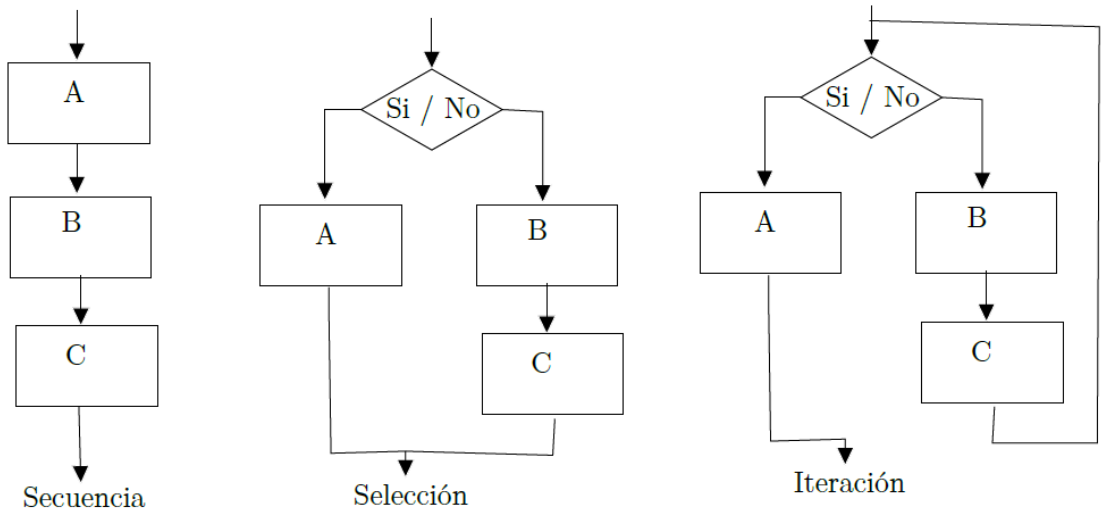
DijKstra propuso el uso de tres construcciones lógicas para diseñar cualquier programa formulando el **teorema de la programación estructurada**.

El teorema del programa estructurado es un resultado en la teoría de lenguajes de programación. Establece que toda función computable puede ser implementada en un lenguaje de programación que combine sólo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) específicamente son:

- **Secuencia**: ejecución de una instrucción tras otra, como todos los ejemplos que hemos visto hasta ahora. Implementa los pasos de procesamiento esenciales de cualquier algoritmo.

- **Selección**: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable o expresión lógica. Establece la posibilidad de seleccionar un conjunto de instrucciones u otro dependiendo de alguna ocurrencia lógica. (También llamado condición o condicionales)
- **Iteración**: ejecución de una instrucción (o conjunto) mientras una variable lógica/booleana sea *verdadera*. Esta estructura lógica también se conoce como ciclo o bucle. Proporciona la posibilidad de ejecutar repetidas veces un bloque de instrucciones. (También llamado repetición)

Dichas construcciones tienen una estructura de ejecución predecible con un único principio y un único final.



De esta forma conseguimos:

- Menor complejidad de los programas.
- Se facilita la legibilidad, la prueba y el mantenimiento de los programas.

Instrucción Secuencial

Las instrucciones del algoritmo se ejecutan una a continuación de otra, empezando por la primera y siguiendo el orden físico de escritura, en el caso de un programa en C se inicia por la primera instrucción de la función `main`. La salida de cada instrucción es la entrada de la siguiente, y así sucesivamente hasta que finalice el proceso.

PROBLEMA

Veremos un ejemplo del programa que calcula la nota de los estudiantes en función de las normas de la asignatura de **Programación**.

- Prácticas de laboratorio: Realización de las tareas correspondientes y entrega de la documentación y programas desarrollados en los plazos establecidos. La calificación de las prácticas para las actividades en equipo es común a todos los miembros del equipo. La nota de estas prácticas será (N_PR) será 0 o 1 (Apto o No apto).

- Trabajo evaluable (trabajo autónomo, no presencial): La nota de este apartado (N_TE) será 0 o 1 (Apto o No apto).

N_TE = 1 si se han entregado, defendido y superado el 80 % de los trabajos evaluables.

N_PR = 1 si se han entregado, defendido y superado las prácticas de laboratorio, dentro del periodo de prácticas.

Calificación Asignatura = N_PR * Examen Final *0.8+ N_TE + N_PR

Análisis

Se necesita leer por teclado las tres notas que permiten calcular la calificación y aplicar la formula dada.

Diseño

- Leer tres valores de variables
  - n_TE — entero
  - n_PR — entero
  - examen – real
  - calificacion – real (no estrictamente necesaria como variable, la mostramos)
- Calcular el resultado
- Mostrar el resultado en pantalla

Implementación

```
float n_TE = 0;
float n_PR = 0;
float examen = 0;

printf("¿Cuál es la nota de los trabajos evaluables (0-1)?: ");
scanf("%f", &n_TE);

printf("¿Cuál es la nota de las practicas (0-1)?:  ");
scanf("%f", &n_PR);

printf("¿Cuál es la nota obtenida en el examen (1-10)?:  ");
scanf("%f", &examen);

float calificacion=0;

calificacion= n_PR * (examen *0.8) + n_TE + n_PR;

printf("La calificación de alumno es %.2f\n", calificacion);
```

PROBLEMA

Una empresa de envasado automático de aceite dispone de diversos tipos de envases con capacidades de 50, 20, 10, 5, 2 y 1 litro, respectivamente. Construir un programa que dado por teclado un número entero de litros a envasar, determine el menor número de envases completos necesarios e indique el número de envases de cada tipo, presentándolos en pantalla.

Análisis

Se leerán los litros por teclado, y se calculan cuantas unidades de 50 litros caben, con lo que queda se van descendiendo en tamaño de envase hasta completar los litros introducidos.

Valores válidos:

- 55 litros – 1 envase de 50 y 1 de 5 VALIDO
- 55 litros – 2 envase de 20, 1 de 10 y 5 de 1 NO VALIDO

Diseño

- Dividir entre capacidad y el cociente da los envases.
- El resto lo que me queda.
- Para todos los tamaños de envase desde 50 a 1 litro.

Implementación

```
{include="../_Problemas/Secuencial/NumeroEnvases.c" .c startLine=10
endLine=42 . dedent=4}
```

Criterios de calidad de un programa

Una vez que ya somos capaces de escribir código debemos revisar los criterios de calidad de un programa:

- **Corrección**: la entrada definida produce los resultados requeridos.
- **Claridad**: el contenido debe ser entendible.
- **Eficiencia**: consumo óptimo de recursos de computación.
- **Robustez**: entradas no definidas - mensajes de aviso sin generar bloqueo. Este criterio se relaja en ciertas condiciones.
- **Amigabilidad**: facilidad de uso para el usuario.

Corrección

Para poder comprobar el correcto funcionamiento de una unidad de código se tienen que ejecutar una o varias pruebas que aseguren que cada unidad de codificación funciona adecuadamente. Un **caso de prueba** es el conjunto de datos de entrada y el valor esperado de los datos de salida que permiten realizar estas comprobaciones de funcionamiento. Es necesario escribir casos de prueba que cubran todas las posibilidades a comprobar durante el análisis.

Para el problema del número de envases. Por ejemplo si el número de litros es 316 litro, la solución n1=316 (n50=n20=n10=n5=n2=0) ¿Es correcta?, No. Debemos dar como solución n50=6 n20=0 n10=1 n5=1 n2=0 n1=1.

El conjunto de casos de prueba involucra las variables de entrada y en los entornos actuales esta prueba se puede automatizar, aunque en esta asignatura no lo haremos.

Datos de prueba

litros	envases 50	envases 20	envases 10	envases 5	envases 2	envases 1
316	6	0	1	1	0	1
200	4	0	0	0	0	0
7	0	0	0	1	1	0
513	10	0	1	0	1	1

Claridad y amigabilidad

Comparemos la solución del problema del número de envases con el siguiente código ahora con siguiente código:

```
#include <stdio.h>

intmain()
{
int n, m, k, j, h, u;
int v, resto;

scanf(" %d", &n);

m = n / 50;
resto = n % 50;
k = resto / 20;
resto = resto % 20;
j = resto / 10;
resto = resto % 10;
h = resto / 5;
resto = resto % 5;
u = resto / 2;
v = resto % 2;

printf("\nNumero de envases necesarios:\n");
printf(" %4d, %4d, %4d, %4d, %4d, %4d\n", m, k, j, h, u, v);

return 0;
}
```

Eficiencia

```
/*
**Descripción: Calculo número de envases
**Asunto: secuencial ejemplo de ineficiencia
*/

#include <stdio.h>

intmain()
{
float n;    /* numero de litros a envasar    */
float n50;  /* num. de envases de 50 litros  */
float n20;  /* num. de envases de 20 litros  */
float n10;  /* num. de envases de 10 litros  */
float n5;   /* num. de envases de 5 litros   */
float n2;   /* num. de envases de 2 litros   */
float n1;   /* num. de envases de 1 litros   */
float resto;

printf("CÁLCULO DEL NUMERO MÍNIMO DE ENVASES\n");
```

```
printf("====================================\n\n");

printf("Introduzca num. de litros a envasar: ");
scanf("%f", &n);
n50 = (int)(n / 50);
resto = n - n50*50;
n20 = (int)(resto / 20);
resto = n - n50*50 - n20*20;
n10 = (int)(resto / 10);
resto =n - n50*50 - n20*20 - n10*10;
n5 = (int)(resto / 5);
resto=  n - n50*50 - n20*20 - n10*10 - n5*5;
n2 = (int)(resto / 2);
resto = n - n50*50 - n20*20 - n10*10 - n5*5 - n2*2;
n1 = (int)(resto / 1);

printf("\nNumero de envases necesarios:\n");
printf("\tEnvases de 50 litros: %4.2f\n", n50);
printf("\tEnvases de 20 litros: %4.2f\n", n20);
printf("\tEnvases de 10 litros: %4.2f\n", n10);
printf("\tEnvases de  5 litros: %4.2f\n", n5);
printf("\tEnvases de  2 litros: %4.2f\n", n2);
printf("\tEnvases de  1 litro:  %4.2f\n", n1);
```

Si bien este programa funciona correctamente para todos los casos de prueba, se realizan muchas más de las operaciones necesarias y se utiliza un tipo de dato que consume mucho más espacio de memoria del necesitado.

Robustez

¿Qué pasa con una entrada no definida?

¿Y si el valor de litros que se introduce por teclado al ejecutar el programa es negativo?

Necesitamos construcciones adicionales para mejorar la robustez, en concreto que se pueda no ejecutar el programa si los datos de entrada no son correctos o seguir pidiendo valores hasta que los datos sean correctos para poder continuar.

Uso de los casos de prueba

La prueba automática en C es un proceso en el que se utilizan herramientas y marcos de trabajo (frameworks) específicos para escribir y ejecutar pruebas de manera automatizada, con el objetivo de verificar que el código funcione según lo esperado. Estas pruebas pueden incluir pruebas unitarias, pruebas de integración, pruebas de sistema, entre otras.

Nosotros nos centraremos en las pruebas unitarias que verifican el correcto funcionamiento de pequeñas unidades de código, como funciones o métodos individuales. Por ejemplo: Probar una función que suma dos números para asegurarse de que devuelve el resultado correcto.

Existen diversos frameworks de prueba en C: CUnit, Check, Unity, Google Test (gtest).

Pero nosotros vamos a encajar las pruebas dentro del marco que nos ofrece GitHub usando librerías de soporte a la prueba un diversas operaciones de inicialización y anotación de los casos de prueba.

A continuación se muestra como utilizar el framework de prueba que se utilizará en la asignatura, en especial en la parte práctica para validar el código.

Partamos del ejemplo ya estudiado de la suma de dos números enteros leidos de teclado.

```
int entero1 = 0; // guarda el primer numero introducido por el usuario
int entero2 = 0; // guarda el segundo numero introducido por el usuario

printf("Dame el primer entero: "); // mensaje en pantalla
scanf("%d", &entero1);             // lee un entero

printf("Dame el segundo entero: "); // mensaje en pantalla
scanf("%d", &entero2);              // lee un entero

int suma = 0;             // variable donde se guardará la suma
suma = entero1 + entero2; // asigna el resultado de la operación a suma

printf("La suma es %d\n", suma); // muestra el resultado
```

Sobre este ejemplo realizaremos la versión que permite la prueba automática de diversas opciones en entrada y salida. Los casos de prueba que son

primer entero	segundo entero	suma
0	0	0
1	1	2
1	3	4
-1	-1	-2
-3	0	-3
-1	1	0

Para esto se utilizarán los stream de salida donde se guardarán los datos a chequear incluyendo solo los resultados sin mensajes que hagan amigable el código. Se parte de dos archivo el fuente a probar `SumarEnteros.c` y el archivo que contiene los casos de prueba que por convenio para poder utilizar las tareas de prueba se llamará `SumarEnterosPRTest.txt`

```
0 0 |0 ||
1 1 | 2 ||
1 3 | 4 ||
-1 -1 | -2 ||
-3 0 | -3 ||
-1 1 | 0 ||
```

Es este archivo separados por una barra están en cada línea un caso de prueba del código.

El programa se modifica ligeramente incluyendo las librerías que facilitan la prueba y dos llamadas a inicio y fin del test. Finalmente si que es necesario indicar cuales los resultados guardar para ser verificados. Este nuevo programa lo llamaremos `SumarEnterosPR.c`.

```
#include <stdio.h>
#include "../.vscode/test/TEST.h"

// inicia ejecución en main
int main(void)
{
    INICIO_TEST();

    int entero1 = 0; // guarda el primer numero introducido por el usuario
    int entero2 = 0; // guarda el segundo numero introducido por el usuario
```

```
    printf("Dame el primer entero: "); // mensaje en pantalla
    scanf("%d", &entero1);             // lee un entero

    printf("Dame el segundo entero: "); // mensaje en pantalla
    scanf("%d", &entero2);              // lee un entero

    int suma = 0;             // variable donde se guardará la suma
    suma = entero1 + entero2; // asigna el resultado de la operación a suma

    printf("La suma es %d\n", suma); // muestra el resultado
    PRINT_TEST("%d",suma); // Guarda el resultado para verificar

    FIN_TEST();
    return 0;
} // fin de main
```

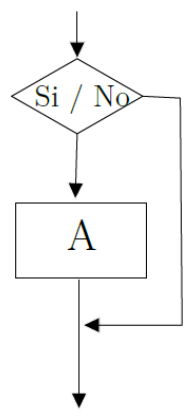
De esta forma el lanzar la tarea de prueba se obtiene un resumen de cuales ha sido los casos exitosos y para la ejecución de la prueba al encontrar un fallo

```
Inicio pruebas
---PASA: Salida Correcta para entrada '0 0'
---PASA: Salida Correcta para entrada '1 1'
---PASA: Salida Correcta para entrada '1 3'
---PASA: Salida Correcta para entrada '-1 -1'
---PASA: Salida Correcta para entrada '-3 0'
---PASA: Salida Correcta para entrada '-1 1'

Se pasan todas las pruebas.
```

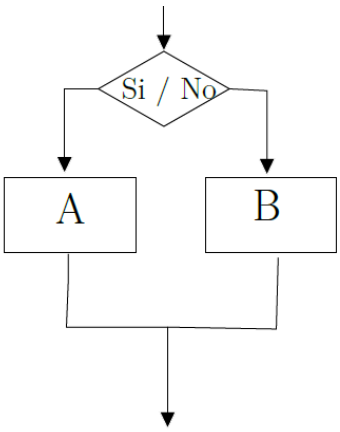
Instrucción selectiva o condicional

Estructura que ofrece la posibilidad de ejecutar las instrucciones en un orden lógico diferente del orden físico. Se evalúa una condición (expresión lógica) y en función del resultado de la misma se ejecuta una opción u otra.

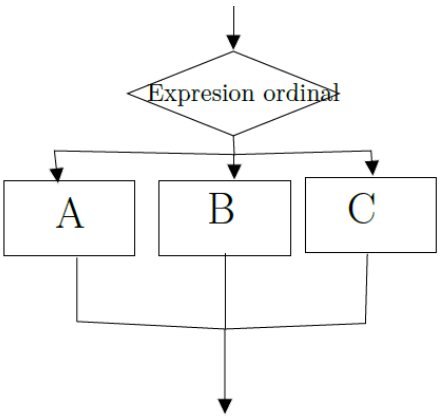
Podemos distinguir varios tipos de estructuras selectivas: *simple*, *doble* y *múltiple*.



Selección simple



Selección doble



Selección multiple

Selectiva simple: Se ejecuta una determinada acción cuando se cumple una condición. Se evalúa una condición; si esta es verdadera entonces ejecuta un bloque de instrucciones y si es falsa no se ejecuta nada.

Selectiva doble: Estructura que permite elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición. Primero se evalúa la condición y si esta es verdadera se ejecuta un bloque de instrucciones y en caso contrario (condición falsa) se ejecuta otro bloque diferente de instrucciones.

Selectiva múltiple: Estructura que evalúa una expresión que puede tomar n valores distintos (1, 2, 3,. . . , N) y según este `valor` el algoritmo seguirá un determinado camino entre n posibles. Se evalúa una expresión ordinal; si el valor de la misma es `valor1`, entonces se ejecutan las instrucciones asociadas al valor 1; si es `valor2`, se ejecutan las instrucciones asociadas al valor 2;. . . ; si el valor es `valorN`, entonces se ejecutan las instrucciones asociadas al valor N. Si no coincide con ninguno de los anteriores se ejecutaría las instrucciones asociadas a una alternativa adicional.

Las dos primeras se implementan en C sobre la instrucción `if-else`. Mientras que la segunda utiliza `switch`

```
if (expresion) instruccion
if (expresion) instruccion else instruccion

 // donde hay una instruccion se puede poner un bloque de código {}

 if (expresion) {instrucciones}
 if (expresion) {instrucciones} else {instrucciones}
```

PROBLEMA

Retomamos el programa para calcular la nota media del alumno en la asignatura.

La nota de estas prácticas será (N_PR) será 0 o 1 (Apto o No apto).

Trabajo Evaluable: La nota de este apartado (N_TE) será 0 o 1 (Apto o No apto).

N_TE = 1 si se han entregado, defendido y superado el 80 % de los trabajos individuales.
N_PR = 1 si se han entregado, defendido y superado las prácticas de laboratorio, dentro del periodo de prácticas.

Examen final: Para poder aprobar la asignatura, el alumno deberá obtener en el examen final una calificación mínima de 4 sobre 10 (3,2 sobre 8).

Calificación Asignatura = N_PR * Examen Final *0.8+ N_TE + N_PR

```
float n_TE = 0;
float n_PR = 0;
float examen = 0;

printf("¿Cuál es la nota de los trabajos evaluables (0-1)?: ");
scanf("%f", &n_TE);

printf("¿Cuál es la nota de las practicas (0-1)?:  ");
scanf("%f", &n_PR);

printf("¿Cuál es la nota obtenida en el examen (1-10)?:  ");
scanf("%f", &examen);
```

```
float calificacion = examen;

if (examen >= 4)
    calificacion = (n_PR * (examen * 0.8)) + n_TE + n_PR;

printf("La calificacion de alumno es %.2f\n", calificacion);
return 0;
```

PROBLEMA Construir un programa que lea por teclado dos valores de temperatura en grados centígrados y que determine e imprima en pantalla el mayor valor.

Análisis

Buscaremos el mayor de los dos datos de temperatura, admitiendo decimales.

Diseño

Se leen los dos valores `temp1`, `temp2` reales

Si `temp1` es mayor que `temp2` `tmax` es `temp1` sino `tmax` es `temp2`

Implementación

```
{
system("cls||clear");
float temp1, temp2, tmax;

printf("TEMPERATURA MÁXIMA\n");
printf("==================\n\n");
printf("Introduzca temperatura 1: ");
scanf(" %f", &temp1);
printf("Introduzca temperatura 2: ");
scanf(" %f", &temp2);

if (temp1 >= temp2)
    tmax = temp1;
else
    tmax = temp2;
```

Veamos la sintaxis de la instrucción selectiva múltiple en C.

```
switch (expresion)
{
    // declaraciones
    // ...
    case expresion_constante:
        // Intrucciones ejecutadas si la expresion es igual al
        // valor de expresion_contante
        break;
    default:
        // Intrucciones ejecutadas si la expresion no es igual a
        // ningun case de expresion_constante
}
```

PROBLEMA

Modificar el problema de la calificación en la asignatura para que se muestre un mensaje con la trasformación a texto de la nota.

```
printf("================================\n\n");

int n_TE = 0;
int n_PR = 0;
float examen = 0;

printf("¿Cúal es la nota de los trabajos evaluables (0-1):? ");
scanf("%d", &n_TE);

printf("¿Cúal es la nota de las practicas (0-1):  ?");
scanf("%d", &n_PR);

printf("¿Cúal es la nota obtenida en el examen (1-10):  ?  ");
scanf("%f", &examen);

float calificacion = examen;

if (examen >= 4)
    calificacion = n_PR * (examen * 0.8) + n_TE + n_PR;

int califEntera;
califEntera = (int)calificacion;

switch (califEntera)
{
case 0:
    printf("Calificación: Suspenso con un %.2f\n", calificacion);
    break;
case 1:
    printf("Calificación: Suspenso con un %.2f\n", calificacion);
    break;
case 2:
    printf("Calificación: Suspenso con un %.2f\n", calificacion);
    break;
case 3:
    printf("Calificación: Suspenso con un %.2f\n", calificacion);
    break;
case 4:
    printf("Calificación: Suspenso con un %.2f\n", calificacion);
    break;
case 5:
    printf("Calificación: Aprobado con un %.2f\n", calificacion);
    break;
case 6:
    printf("Calificación: Aprobado con un %.2f\n", calificacion);
    break;
case 7:
    printf("Calificación: Notable con un %.2f\n", calificacion);
    break;
case 8:
    printf("Calificación: Notable con un %.2f\n", calificacion);
    break;
case 9:
    printf("Calificación: Sobresaliente con un %.2f\n", calificacion);
    break;
case 10:
    printf("Calificación: Matrícula de Honor con un %.2f\n", calificacion);
    break;
default:
    printf("Calificación no válida. con un %.2f\n", calificacion);
```

Indicadores o centinelas

Un **indicador/interruptor/centinela/bandera** (del inglés `flag`) es una variable que puede tomar diversos valores a lo largo de la ejecución del programa y que permite comunicar información entre las diversas partes del programa. Al asociarse instrucciones

selectivas, permite variar la ejecución del programa. Normalmente toma 2 valores (0, 1 o bien verdadero, falso). En el caso de C al carecer de tipos de datos lógicos, habitualmente se definen los indicadores como números enteros.

```
    ....
int esPar;
    ....
if (numero % 2 == 0) {
    esPar = 1; // Establece el indicador en 1 si el número es par
} else {
    esPar = 0; // Establece el indicador en 0 si el número es impar
}
....

if (esPar) printf("Este número no puede ser primo");
    .....
```

Anidamiento

Estructuras condicionales anidadas:

Esta situación es la que aparece cuando una estructura selectiva incluye otra estructura selectiva y así sucesivamente.

Se utilizan para estructuras que contengan más de dos alternativas de ejecución, y/o estén controladas por expresiones no ordinales ó por varios datos lógicos.

Pueden aparecer ambigüedades sintácticas en el caso de tener varias instrucciones `if-else` anidadas, cuando alguna instrucción if no lleve asociada la parte else correspondiente (utilizar llaves como marca de bloque para evitar errores).

Se quiere escribir un programa que:

1. Pida por teclado la nota (real) de una asignatura.
2. Muestre por pantalla:
   - "APTO", en el caso de que la nota sea mayor o igual que 5 y menor o igual que 10.
   - "NO APTO", en el caso de que la nota sea mayor o igual que 0 y menor que 5.
   - "ERROR: Nota incorrecta.", en el caso de que la nota sea menor que 0 o mayor que 10.

```
/*
** Descripción: Cálculo nota
** Asunto: Condicional anidada
*/

#include <stdio.h>

int main()
{
   float nota;

   printf("\n   Introduzca nota (real): ");
   scanf(" %f", &nota);

   if ( nota >= 5 && nota <= 10 )
      printf("\n   APTO");
   else
```

```
      /* Inicio del anidamiento */
      if ( nota >= 0 && nota < 5 )
         printf("\n   NO APTO");
      else
         printf("\n   ERROR: Nota incorrecta.");
      /* Fin del anidamiento */

   return 0;
}
```

Aclaraciones sobre condicional múltiple

Las instrucciones `switch` son una fuente habitual de errores. La marca de bloque viene definida por la instrucción `break`. Veamos una modificación del problema de mostrar las calificaciones.

```
    if(examen>=4)
        calificacion= n_PR * (examen *0.8) + n_TI + n_PR;

    int califEntera;
    califEntera = (int) calificacion;

    switch (califEntera) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
                printf("Calificación: Suspenso con un %.2f\n", calificacion);
                break;
            case 5:
            case 6:
                printf("Calificación: Aprobado con un %.2f\n", calificacion);
                break;
            case 7:
                printf("Calificación: Notable con un %.2f\n", calificacion);
                break;
            case 8:
            case 9:
                printf("Calificación: Sobresaliente con un %.2f\n", calificacion);
                break;
            case 10:
                printf("Calificación: Matrícula de Honor con un %.2f\n", calificacion);
                break;
            default:
                printf("Calificación no válida. con un %.2f\n", calificacion);
                break;
        }
```

Alternativa a switch utilizada muy habitualmente

```
if (calificacion == 10)
{
    printf("Matricula de honor");
} // end if
else
{
    if (calificacion >= 9)
    {
        printf("Sobresaliente");
    } // end if
    else
    {
        if (calificacion >= 7)
        {
```

```
            printf("Notable");
        } // end if
        else
        {
            if (calificacion >= 5)
            {
                printf("Aprobado");
            } // end if
            else
            {
               printf("Suspenso");
            } // end else
        } // end else
    } // end else
} // end else
```

O bien con `else if`

```
if (calificacion == 10)
{
printf("Matricula de honor");
} // end if
else if (calificacion >= 9)
{
printf("Sobresaliente");
} // end else if
else if (calificacion >= 7)
{
printf("Notable");
} // end else if
else if (calificacion >= 5)
{
printf("Aprobado");
} // end else if
else {
printf("Suspenso");
} // end else
```

PROBLEMA

Construir un programa que calcule e imprima en pantalla el área de un triángulo dados por teclado las longitudes de sus tres lados.

Análisis

La información de la que disponemos es longitud de los lados del triángulo. Los valores de entrada son por tanto tres números y el resultado será un número real con la superficie del triángulo.

Diseño

Para el ejemplo el diseño arquitectónico sería decidir se utilizamos la formula clásica de dividir por dos el resultado de base por altura, habremos de calcular dichos valores, o si bien utilizamos la formula de Heron a partir del semiperímetro, que es la elegida en este caso. No obstante en ambos casos habrá que comprobar que es un triángulo propio ($a < b + c$).

El diseño detallado consiste en identificar los pasos correctos:

- Guardar valores de los lados l1, l2, l3
- Buscar el lado más largo.

- Comprobar que su valor es más pequeño que la suma de los otros dos
- Si se cumple la condición de Heron calcular el semiperímetro $(a+b+c)/2$
- Se calcula el área

$area = (\sqrt{s(s-l_1)(s-l_2)(s-l_3)}$

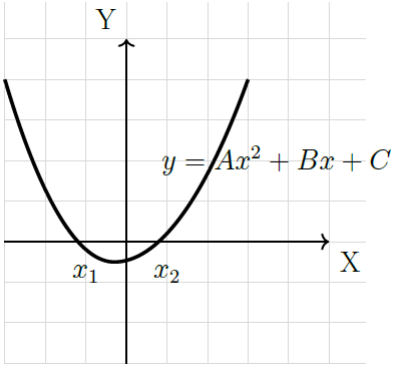
$semiperimetro = \frac{l_1+l_2+l_3}{2}$

Implementación

```
int main()
{
    float l1, l2, l3; /* longitudes de los lados */
    float s;          /* semiperimetro           */
    float area;       /* área del triángulo      */

    printf("\n CÁLCULO DEL ÁREA DE UN TRIÁNGULO\n");
    printf("==================================\n\n");
    printf("Introducir longitudes de lados:\n");
    printf("\tl1: ");
    scanf(" %f", &l1);
    printf("\tl2: ");
    scanf(" %f", &l2);
    printf("\tl3: ");
    scanf(" %f", &l3);
    if ((l1 <= 0) || (l2 <= 0) || (l3 <= 0))
        printf("Error en datos de entrada");
    else if ((l1 < (l2 + l3)) && (l2 < (l1 + l3)) && (l3 < (l1 + l2)))
    {
        s = (l1 + l2 + l3) / 2;
        area = sqrt(s * (s - l1) * (s - l2) * (s - l3));
        printf("\nÁrea= %.2f", area);
    }
    else
        printf("\nNo es un triángulo");
    return 0;
}
```

PROBLEMA

Construir un programa que calcule e imprima en pantalla las raíces de la ecuación de segundo grado: $Ax^2 + Bx + C = 0$, dados por teclado los coeficientes A, B y C.

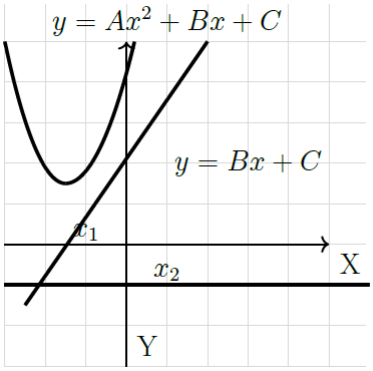


Análisis La entrada serán los coeficientes del polinomio: A, B y C, reales.

Como salida se darán las soluciones: x1 y x2, reales

$x_1 = \frac{-B+\sqrt{(B^2-4AC)}}{2A}$, $x_2 = \frac{-B-\sqrt{(B^2-4AC)}}{2A}$

No obstante, existen diversas situaciones que se tienen que contemplar:



Se debe contemplar el valor del *discriminante* y el valor de A:

- Si *discriminante* menor que cero no hay solución real
- Si A vale cero, tenemos una ecuación de primer grado con $x = -C/B$, que supone un problema si B es cero

Diseño Problema controlado por condiciones basadas en los valores de los coeficientes leídos.

- Se leen los coeficientes.
- Se estudia el valor de A para saber si estamos en una ecuación de segundo grado.
- Si es de segundo grado se calcula el discriminante.
- Según el valor del discriminante se define el resultado.
- Si no es una ecuación de segundo grado — No esta especificado pero se puede optar o no por dar una solución.

A	B	C	Resultado
<>0	-	-	Estudiar discriminante
0	-	-	No de segundo grado
0	0	0	No ecuación
0	0	<>0	Imposible

Implementación

```
printf("Introduzca coeficiente de x*x: ");
scanf(" %f", &a);
printf("Introduzca coeficiente de x: ");
scanf(" %f", &b);
printf("Introduzca termino independiente: ");
scanf(" %f", &c);
printf("\n\n");
if (a == 0)
    if (b == 0)
        if (c == 0)
            printf("Igualdad 0=0");
        else
            printf("Imposible %f=0", c);
    else
        printf("Sol. ec. de primer grado: x=%f", -c / b);
else
{
    d = b * b - 4 * a * c;
    if (d < 0)
        printf("Raíces imaginarias");
    else
    {
        x1 = (-b + sqrt(d)) / (2 * a);
        x2 = (-b - sqrt(d)) / (2 * a);
        printf("x1=%f\nx2=%f", x1, x2);
    }
}
```

PROBLEMA

Construir un programa que calcule e imprima la mayor de tres temperaturas (ºC) introducidas por teclado.

Análisis

La entrada son tres temperaturas: `t1, t2, t3` reales. Como salida se da la mayor de las temperaturas: `tmax` real

Diseño

Se pueden plantear diversas soluciones alternativas:

- Usando selectivas dobles anidadas controladas por datos lógicos complejos
- Usando selectivas dobles anidadas controladas por datos lógicos simples
- Utilizando cadena de selectivas

Implementación 1

```
if ((t1 >= t2) && (t1 >= t3))
    tmax = t1;
else if ((t2 >= t1) && (t2 >= t3))
    tmax = t2;
else
    tmax = t3;
printf("\nTemperatura maxima: %.1f", tmax);
```

Implementación 2

```
if (t1 >= t2)
    if (t1 >= t3)
        tmax = t1;
    else
        tmax = t3;
else if (t2 >= t3)
    tmax = t2;
```

```
    else
        tmax = t3;
    printf("\nTemperatura maxima: %.1f", tmax);
```

Y si ahora queremos cuatro temperaturas, ¿qué habría que modificar? Necesitamos sólo conocer el máximo por tanto podemos facilitar la implementación para hacerla más escalable.

Implementación 3

```
int main()
{

    float t;    /* temperaturas leídas */
    float tmax; /* temperatura máxima  */

    printf("MÁXIMA DE TRES TEMPERATURAS\n");
    printf("===========================\n\n");
    printf("Introduzca tres temperaturas:\n");
    printf("\tt1: ");
    scanf(" %f", &tmax);
    printf("\tt2: ");
    scanf(" %f", &t);

    if (t > tmax)
        tmax = t;
    printf("\tt3: ");

    scanf(" %f", &t);
    if (t > tmax)
        tmax = t;

    printf("\nTemperatura máxima: %.1f", tmax);
    return 0;
}
```

Variaciones sobre el problema anterior Si necesito saber cuál de las tres temperaturas es la mayor, o saber si se han introducido en orden de mayor a menor.

En este caso el **análisis** es distinto aunque se pueden utilizar estrategias similares de solución.

```
    if (t1 >= t2)
        if (t1 >= t3)
            printf("\nTemperatura maxima es la primera introducida: %.1f", t1);
        else
            printf("\nTemperatura maxima es la tercera introducida: %.1f", t3);
    else if (t2 >= t3)
        printf("\nTemperatura maxima es la segunda introducida: %.1f", t2);
    else
        printf("\nTemperatura maxima es la tercera introducida: %.1f", t3);
```

Pero sigue teniendo el problema de la escalabilidad. Como extender la solución de cadena de condicionales.

```
    if (t > tmax)
    {
        tmax = t;
        posicion = 2;
    }
    printf("\tt3: ");
    scanf(" %f", &t);
    if (t > tmax)
```

```
    {
        tmax = t;
        posicion = 3;
    }
    printf("\nTemperatura maxima: %.1f que ha sido la introducida en  %d lugar", tmax, posicion);
```

PROBLEMA

Programa para calcular el salario semanal de un trabajador, a partir de las horas trabajadas y el precio cobrado por hora. Ambos valores deben ser solicitados al usuario. La jornada normal es de 40 horas semanales. Las horas extra se pagan un cincuenta por ciento más caras que las normales, y pasan a pagarse al doble que las normales a partir de las 50 horas trabajadas.

Análisis

Existen unos valores constantes que son los límites que definen del cambio de coste por hora. Los datos de entrada son el **numHoras** y el **precioHora**. Definiendo distintos tramos

Horas trabajadas	Factor Multiplicador	Acumulado sueldo
0 a 40	1	numHoras * precioHora
41 a 49	1.5	(numHoras-40) * precioHora* 1.5
50 a –	2	(numHoras-50) * precioHora* 2

Diseño Condicional comprobando los límites para acumular el salario.

```
int main(void)
{
    float horas, salarioHora, salario;
    /* Leer datos */
    printf("¿Cuántas horas ha trabajado? ");
    scanf("%f", &horas);
    printf("¿Cual es el salario por hora? ");
    scanf("%f", &salarioHora);

    /* Calcular horas extra y salario */
    if (horas <= MAX_HORAS1)
    {
        salario = horas * salarioHora;
    }
    else if (horas <= MAX_HORAS2)
    {
        salario = MAX_HORAS1 * salarioHora + (horas - MAX_HORAS1) * salarioHora * FACTOR1;
    }
    else
    {
        salario = MAX_HORAS1 * salarioHora + (MAX_HORAS2 - MAX_HORAS1) *
            salarioHora * FACTOR1 + (horas - MAX_HORAS2) * salarioHora * FACTOR2;
    }

    /* Se muestran los resultados por pantalla */
```

En casos como este se pone de manifiesto un problema de **robustez** de los ejemplos vistos hasta ahora ¿qué pasa si se da como entrada un valor negativo para las horas extras?.

Aparece la necesidad de **validar los datos de entrada**, se debe comprobar si los datos de entrada son negativos, si es así no se realizarán los cálculos.

```c
#define MAX_HORAS1 40
#define MAX_HORAS2 50
#define FACTOR1 1.5
#define FACTOR2 2

int main(void)
{
    float horas, salarioHora, salario;

    printf("¿Cuántas horas ha trabajado? ");
    scanf("%f", &horas);
    printf("¿Cuál es el salario por hora? ");
    scanf("%f", &salarioHora);

    if (horas >= 0 && salarioHora >= 0)
    {
        if (horas <= MAX_HORAS1)
        {
            salario = horas * salarioHora;
        }
        else if (horas <= MAX_HORAS2)
        {
            salario = MAX_HORAS1 * salarioHora + (horas - MAX_HORAS1) * salarioHora * FACTOR1;
        }
        else
        {
            salario = MAX_HORAS1 * salarioHora + (MAX_HORAS2 - MAX_HORAS1) *
                 salarioHora * FACTOR1 + (horas - MAX_HORAS2) * salarioHora * FACTOR2;
        }

        printf("Le corresponde cobrar: %.2f \n", salario);
    }
    else
    {
        printf("Por favor, ingrese valores positivos para las horas y la tarifa por hora.\n");
    }
    return 0;
}
```

No obstante cuando se conozcan como funcionan los bucles se puede mejorar este proceso de validación.

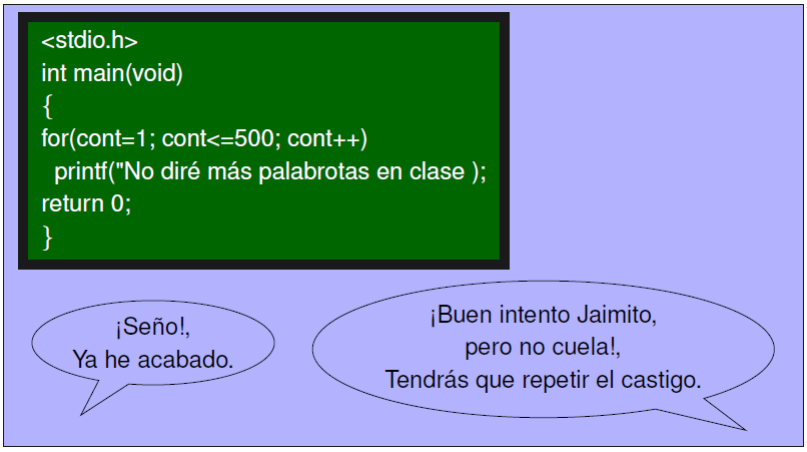
Instrucción iterativa o repetición

A menudo en la resolución de un problema es necesario ejecutar una instrucción o un bloque de instrucciones más de una vez. Por ejemplo *Implementar un programa que calcule la suma de N números leídos desde teclado*, o bien *¿Cuál es la mayor de* n *temperaturas leídas por teclado?*.

Se podría escribir un programa en el que apareciese repetido el código que deseamos, pero tenemos varios inconvenientes. El primero es que el programa resultante sería muy largo, el segundo que tendríamos mucho código duplicado lo que dificulta los futuros cambios en el código y el más importante, que una vez escrito el programa para un número determinado de repeticiones (p.ej. sumar matrices 3x3), el mismo programa no podríamos reutilizarlo si necesitásemos realizar un número distinto de operaciones (p.ej. matrices 4x4).

Las estructuras de control repetitivas o iterativas, también conocidas como **bucles**, nos permiten resolver este tipo de problemas. Algunas se pueden usar cuando el número de veces que deben repetirse las operaciones es conocido y otras permiten repetir un conjunto de operaciones mientras se cumpla una condición.

El diseño de un bucle consiste en identificar el llamado **cuerpo del bucle**, que es el bloque de instrucciones que se repite de forma controlada, bien controlado por un número de veces o bien por una condición. También se identifican las **variables que controlan** el bucle y los valores a los que se han de inicializar, así como la condición de salida en su caso. Este proceso se llama diseño detallado del bucle o definición del **esquema de iteración**.



Tipos de bucles según su sintaxis

Mientras En esta estructura el cuerpo del bucle se repite mientras se cumpla una determinada condición definida por una expresión.

Se evalúa la condición y si esta es verdadera se ejecuta el cuerpo del bucle y se vuelve a evaluar la condición. Este proceso se repite mientras la condición sea verdadera; cuando la condición sea **falsa** se sale de esta estructura y se pasa a la siguiente instrucción.

Un bucle de n iteraciones soporta:

- n ejecuciones cuerpo bucle
- n+1 evaluaciones de la condición

Casos especiales de bucles con la estructura Mientras:

- Bucle nulo: no se ejecuta ninguna vez el cuerpo del bucle (la primera evaluación de la condición del bucle es falsa).
- Bucle infinito: nunca termina (la condición del bucle es siempre verdadera).

Repetir (Hasta/Mientras) Estructura que ejecuta el cuerpo del bucle hasta que se cumpla una determinada condición que se comprueba al final de cada iteración.

No todos los lenguajes de programación la incluyen. Se ejecuta el cuerpo del bucle y se evalúa la condición del bucle; si esta es falsa se repite el cuerpo del bucle y si es verdadera el bucle termina y se pasa a ejecutar la siguiente instrucción

Diferencias entre *Mientras* y *Repetir Hasta* que:

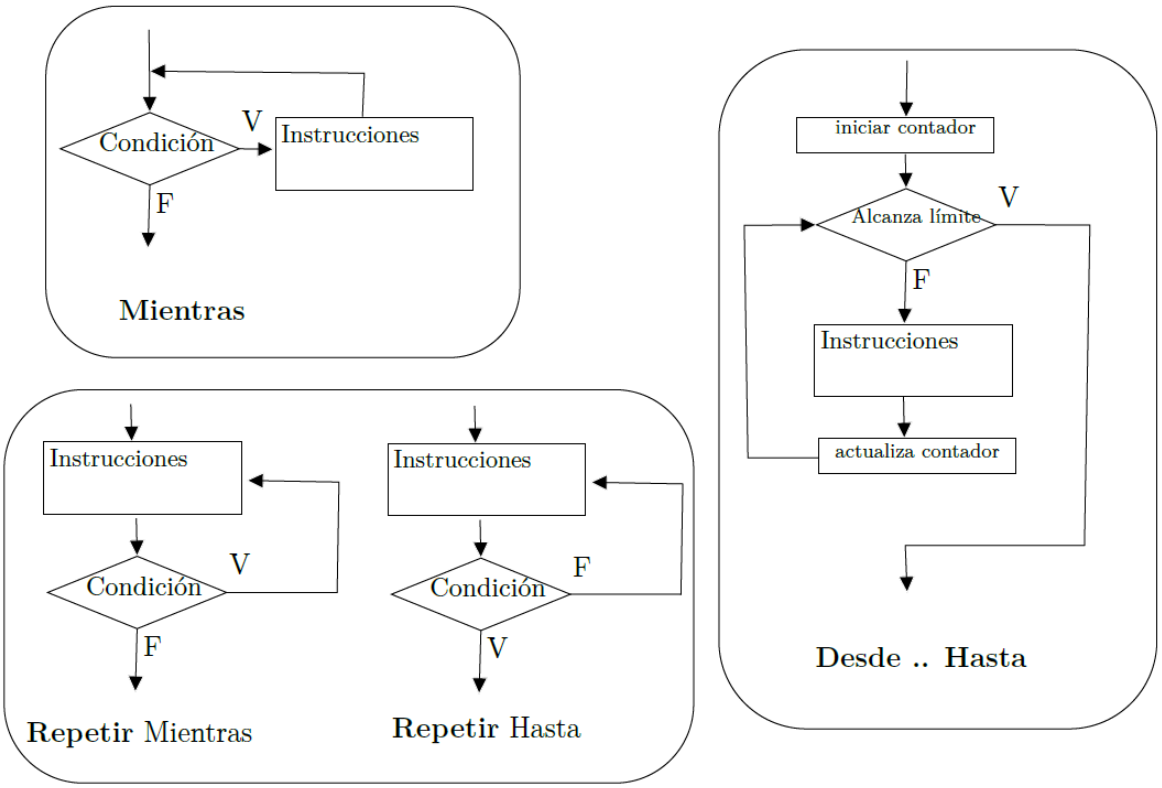
- Mientras termina cuando la condición es falsa y Repetir Hasta que termina cuando la condición es verdadera (aunque podemos encontrar la versión con condición falsa), llamada **Repetir Mientras**.
- La estructura Repetir Hasta que ejecuta el cuerpo del bucle por lo menos una vez; por el contrario la estructura Mientras es más general y permite la posibilidad de que el cuerpo del bucle no sea ejecutado.

Aplicaciones usuales de Repetir (mientras/hasta):

- Validación de datos de entrada (el dato debe cumplir determinadas restricciones antes de avanzar en el algoritmo).
- Presentación de menú de opciones para seleccionar entre diversas alternativas de ejecución.

Repetir según indica un contador o Bucle Desde .. Hasta Estructura que ejecuta las instrucciones del cuerpo del bucle un número determinado de veces y que de modo automático controla el número de iteraciones empleando una variable índice.

Se asigna el valor inicial a la variable índice. Se ejecuta el cuerpo del bucle tras comprobar que la variable índice es menor o igual al valor final (o mayor igual si el índice es decreciente), y se incrementa/decrementa la variable índice volviéndose a repetir el proceso hasta que la variable índice sea mayor que el valor final (o menor si es decreciente).



Sintaxis en C de los bucles

```
while(expresión)
     instruccion;

while(expresión)
{
     instrucciones;
}

do
{
    instrucciones;
}while(expresión);

do
  instruccion;
while(expresión); //no tiene mucho sentido
```

`while` comprueba la expresión, y si el resultado es verdadero ejecuta instrucción; se sale de la repetición cuando expresión sea falsa (en C: igual a 0).

`do-while` ejecuta primero instrucción y luego comprueba la expresión; se sale de la repetición cuando expresión sea falsa. ¡ojo! es un repetir-mientras no repetir-hasta, aunque bastaría con negar la condición para hacerlos equivalentes.

```
for(exp1;exp2;exp3)
    instruccion;

for(exp1;exp2;exp3)
{
    instrucciones;
}
```

Donde:

- exp1: inicialización. Puede haber más de una asignación separadas por comas.
- exp2: condición de comprobación para volver a ejecutar otra iteración del bucle.
- exp3: actualización. Puede haber más de una asignación separadas por comas.

Puede faltar cualquier parte del bucle, pero los separadores ";" no pueden faltar.

Ejemplo de bucles Partimos del problema de obtener al valor acumulado de 10 números enteros introducidos desde teclado. Debemos repetir 10 veces la lectura de número y el uso de una variable `suma` donde se vayan añadiendo los valores leídos, es decir tendremos que repetir 10 veces:

```
scanf("%d", &numero);
suma = suma + numero;
```

Para establecer un bucle con esas dos instrucciones como `cuerpo del bucle` podremos utilizar cualquiera de los tres tipos de bucles antes descritos

Uso de bucle mientras/while

```c
int contador = 0;   // Variable para llevar la cuenta de cuántos números se han leído
int suma = 0;       // Variable para almacenar la suma de los números
int numero;         // Variable para almacenar cada número introducido por el usuario

while (contador < 10)
{
    printf("Introduce un número entero: ");
    scanf("%d", &numero);

    suma = suma + numero;   // Suma el número al total
    contador++;       // Incrementa el contador
}

printf("La suma de los 10 números enteros es: %d\n", suma);
```

Uso de bucle repetir/do-while

```c
int contador = 0;   // Variable para llevar la cuenta de cuántos números se han leído
int suma = 0;       // Variable para almacenar la suma de los números
int numero;         // Variable para almacenar cada número introducido por el usuario

do
{
    printf("Introduce un número entero: ");
    scanf("%d", &numero);

    suma = suma + numero;   // Suma el número al total
    contador++;       // Incrementa el contador
} while (contador < 10);

printf("La suma de los 10 números enteros es: %d\n", suma);
```

Uso de bucle desde-hasta/for

```c
int suma = 0;       // Variable para almacenar la suma de los números
int numero;         // Variable para almacenar cada número introducido por el usuario

for (int contador = 0; contador < 10; contador++)
{
    printf("Introduce un número entero: ");
    scanf("%d", &numero);

    suma = suma + numero;   // Suma el número al total
}

printf("La suma de los 10 números enteros es: %d\n", suma);
```

Para definir el bucle en este caso se necesita:

- Respecto al cuerpo del bucle, lo que se repite:
  - Una variable donde leer el número
  - Una variable **acumulador** que guarde la suma -
- Respecto al control de bucle:
  - Necesitamos un **contador** que se vaya incrementando hasta 100
  - Una inicialización de estas variables antes de entrar en el bucle (`suma` y `contador`)
  - La actualización del contador (en `for` es implicita)
  - La condición de salida del bucle, definida por una expresión lógica.

Para este caso los tres bucles son equivalentes, pero no siempre es así. Por ejemplo si no sabemos cuantas iteraciones se van a producir el bucle `for` no se puede/debe utilizar. Si lo que queremos es:

> Acumular todos los enteros leídos por teclado hasta que se introduzca un 0, sólo son viables los bucles mientras,

```c
int numero = 1, suma = 0;

printf("Introduce números enteros (introduce 0 para salir):\n");

while (numero != 0) {  // El bucle se ejecutará mientras el número sea diferente de 0
    printf("Introduce un número: ");
    scanf("%d", &numero);

    suma = suma +numero;  // Acumular el número en la suma
}

printf("La suma de los números ingresados es: %d\n", suma);
```

o bien con un do-while

```c
int numero, suma = 0;

printf("Introduce números enteros (introduce 0 para salir):\n");

do {
    printf("Introduce un número: ");
    scanf("%d", &numero);

    suma += numero;  // Acumular el número en la suma

} while (numero != 0);

printf("La suma de los números ingresados es: %d\n", suma);
```

Las diferencias principales entre un bucle `while` y un bucle `do-while` en C (y en muchos otros lenguajes de programación) se basan en la **evaluación de la condición**, y el **uso típico**:

En el bucle while la condición se verifica antes de ejecutar el cuerpo del bucle. Si la condición es falsa desde el principio, el bloque de código nunca se ejecuta. Este bucle se utiliza cuando no se sabe si el cuerpo del bucle debe ejecutarse o no, y la condición se evalúa antes de entrar al bucle.

Sin embargo en el bucle do-while la condición se verifica después de ejecutar el cuerpo del bucle. Esto significa que el bloque de código se ejecuta al menos una vez, independientemente de si la condición es verdadera o falsa inicialmente. Este bucle se utiliza cuando se desea que el cuerpo del bucle se ejecute al menos una vez, sin importar si la condición es verdadera o falsa inicialmente. Luego, la condición se verifica después de ejecutar el bloque de código.

Por ejemplo, si se necesita solicitar datos al usuario y solo quieres hacerlo si cierta condición es verdadera, se usaría un `while`. Si se necesita solicitar datos al usuario y se desea garantizar que al menos se ingrese un valor válido, se usaría un `do-while`.

En esta asignatura preferiremos el uso de `while` restringiendo el uso `do-while` a la lectura de datos de entrada y la utilización de menús de opciones. Otro caso de uso es definir como cuerpo de bucle el programa a ejecutar, teniendo un bucle `do-while` controlado por

un valor pedido al usuario para indicar si se quiere repetir la ejecución. Como queremos al menos ejecutar el programa una vez se usa un `do-while`.

```
int main()
{
        char c;

        do
        {

                printf("\n");
                printf("======\n\n");

            // código que resuelve el problema

                printf("\n\n¿Desea efectuar una nueva operación (s/n)? ");
                scanf(" %c", &c);
        } while ((c != 'N') && (c != 'n'));
        return 0;
}
```

El otro uso habitual de los bucles `do-while` es la validación de los datos de entrada de un problema, por ejemplo si durante el análisis de define que los únicos valores válidos para un dato de entrada `numero` son los que están en el rango $[-5, 10]$ el bucle de validación de este dato de entrada sería:

```
do{ printf("\tIntroduzca el numero: ");
         scanf(" %d", &numero);
}while((numero<-5)||(numero>10));
```

Un ejemplo sobre el problema del cálculo del área del triángulo sería comprobar que los lados de triángulo son positivos puesto que son distancias:

```
        do{
            printf("Introducir longitudes de lados:\n");
            printf("\tl1: ");
            scanf(" %f", &l1);
            printf("\tl2: ");
            scanf(" %f", &l2);
            printf("\tl3: ");
            scanf(" %f", &l3);
        } while ((l1 <= 0) || (l2 <= 0 ) || (l3<=0));
```

Tipos de bucles según condición de terminación

El bucle más versátil y genérico es el bucle `while`, es decir que siempre se puede utilizar puesto los otros dos son tipos particulares de este bucle por tanto es el que permite expresar todos las condiciones posibles de terminación/operativa de un bucle. De hecho es el que se utilizará para mostrar los tipos de bucles en función de la condición de terminación.

Dependiendo de como se gestione la finalización o salida del bucle tenemos distintos tipos de bucles:

1. Bucle controlado por contador de iteraciones
2. Bucle controlado por centinela o indicador

Bucle controlado por contador de iteraciones Es aquel en el que se conoce antes de iniciar el bucle el número de veces que se repite, bien porque se conoce en tiempo de compilación o porque se obtiene o calcula en tiempo de ejecución. Hace uso de una variable entera para el control del bucle (contador del número de iteraciones).

En estos casos en cuando se podría utilizar tanto un bucle tipo `while` como un `for`. Pero puesto que el primero de ellos es más general lo utilizaremos en este caso.

PROBLEMA

Construir un programa que calcule e imprima la mayor de un conjunto de datos de temperaturas introducidas por teclado. El número de datos es conocido a priori (se lee por teclado).

Análisis

- Como entrada estará un valor entero para el número de temperaturas `n`.
- Como salida se dará el mayor de las `n` temperaturas leídas por teclado, `tmax`

Diseño

- Se lee `n`, que debe ser entero positivo.
- Un bucle controlado por contador de 1 a n donde
  - leer temperatura si es mayor que la máxima actual se cambia la máxima
- Se muestra el valor de la temperatura máxima

El cuerpo del bucle sería similar al caso ya descrito para la instrucción de selección:

```
scanf(" %f", &t);
if (t > tmax)
      tmax = t;
```

Implementación

```
    float t;    /* temperatura leída      */
    float tmax; /* temperatura máxima     */

    printf("TEMPERATURA MÁXIMA\n");
    printf("==================\n\n");
    printf("Introduzca número de datos: ");
    scanf(" %d", &n);
    if (n <= 0)
        printf("Sin datos a leer");
    else
    {
        tmax = -1E38;
        i = 1;
        while (i <= n)
        {
            printf("Introduzca temperatura: ");
            scanf(" %f", &t);
            if (t > tmax)
                tmax = t;
            i++;
        }
        printf("\nTemperatura máxima: %.0f", tmax);
    }
}
```

La lectura del valor de `n` podría haberse sustituido por un bloque `do-while` eliminando el bloque if. En cualquier caso lo importante es la validación del dato.

```
do
{
printf("\tIntroduzca el numero: ");
scanf(" %d", n);
} while(n<0);
```

La inicialización antes del bucle `tmax=-1E38` se hace para asegurarse de que partimos de un valor muy negativo (puede haber temperaturas negativas). En el caso de la suma de n números se inicializaba `suma=0`.

Bucle controlado por centinela Bucle controlado por centinela: finaliza cuando sucede algo en el interior del bucle que indica que se debe salir del bucle. Métodos típicos para terminar un bucle:

- Agotar los datos de entrada comprobando simplemente que no quedan más datos de entrada. Ejemplo: comprobación del final de un archivo, de una lista o un vector, se verá más adelante.
- Preguntar al finalizar cada iteración si se desea ejecutar el cuerpo del bucle una vez más. Como ejemplo el caso para la repetición de la ejecución de programas. Habitualmente se utiliza un instrucción `do-while`.
- Utilizar un interruptor o indicador que registre si se ha producido o no el suceso que controla la salida del bucle. Para salir del bucle, en alguna iteración se debe asignar *verdadero/falso* a la variable de indicador. Este es menos habitual en C puesto que no existen el tipo lógico. Se utiliza el valor de una variable numérica o carácter, lo que se corresponde con la siguiente condición de terminación.
- Utilizar una variable centinela, esto es, un dato (o combinación de datos) que no se procesa y que se utiliza para indicar el final de una lista de datos de entrada o una situación a identificar. Como ejemplo tenemos el caso anterior de calcular la suma de enteros hasta introducir un 0.
- Generalizando un bucle controlado por contador es un bucle controlado por centinela en el que el centinela es el contador.

PROBLEMA

Construir un programa que calcule e imprima la mayor de un conjunto de datos de temperaturas introducidas por teclado. El número de datos no es conocido a priori, finalizándose la introducción de datos con el valor –999.

En este caso, el diseño se basa en el uso de un valor **centinela** de forma que cuando se lee el valor del centinela se sale del bucle de lectura de datos de temperaturas.

Implementación

```
    {
        printf("Introduzca Temperatura (Fin=%.0f):", TFIN);
        scanf(" %f", &t);
        if (t == TFIN)
            fin = 1;
        else if (t > tmax)
            tmax = t;
    }

    // necesario para cuando el primer valor es el centinela
    if (tmax != -1E38)
        printf("\nTemperatura máxima: %.1f", tmax);
    else
        printf("\nSin datos válidos");

    return 0;
```

Lectura adelantada y lectura a demanda Al definir un bucle se pueden plantear distintas estrategias de diseño. En este sentido veamos en que consiste la **lectura adelantada**, para lo que vamos a trabajar sobre otro ejemplo.

Sea un programa para sumar todos los números introducidos por teclado hasta que aparezca un número negativo:

```
int numero = 1, suma = 0;
printf("Introduce números enteros (introduce un negativo para salir):\n");

while (numero > 0) {  // El bucle se ejecutará mientras el número sea diferente de un negativo
    printf("Introduce un número: ");
    scanf("%d", &numero);
    suma = suma + numero;  // Acumular el número en la suma
}
printf("La suma de los números ingresados es: %d\n", suma);
```

Pero ¿qué pasa si el primer número es un número negativo? la suma sería incorrecta. Para resolver esto se puede:

- Colocar un condicional (`if`) al finalizar el bucle que capture el valor incorrecto

```
if (numero < 0) suma=suma - numero;
```

En este caso como la salida es siempre con numero negativo se puede no poner bajo un `if`

```
 suma = suma - numero;
```

- O realizar la lectura adelantada. Leyendo el primer valor fuera del bucle y cambiando el orden de las instrucciones en el cuerpo del bucle, primero se procesa y luego se lee (el dato a tratar en la siguiente iteración). En la iteración n se lee lo que se va a procesar en la iteración n+1.

```
int numero, suma = 0;  // numero ya no tiene que se inicializado
printf("Introduce números enteros (introduce un negativo para salir):\n");
scanf("%d", &numero);
while (numero > 0)
{  // El bucle se ejecutará mientras el número sea diferente de un negativo
    suma = suma + numero;  // Acumular el número en la suma it n
    printf("Introduce un número: ");
    scanf("%d", &numero);   // lee número de siguiente iteración
}
printf("La suma de los números ingresados es: %d\n", suma);
```

Ambas soluciones son válidas y se tendrán que adaptar a cada problema estudiando los casos concretos de la primera iteración, una iteración general y la condición de salida, a esto se llama **diseño del bucle**.

Lo contrario a la **lectura adelantada** es la **lectura a demanda**.

Diseño de bucles

Con bucles complejos se ha de decidir la estructura del bucle definiendo correctamente, si se uso o no lectura adelantada, las condiciones de inicialización, exactamente que se incluye en el cuerpo del bucle y la condición de control de salida del bucle. No existe una "mejor" aproximación, según se decida cada elemento afectará a los demás. Depende de la especificación del problema una alternativa será mejor o no en términos de legibilidad y mantenibilidad que otro.

En líneas generales el proceso sería:

- Identificar las **acciones útiles a repetir** y las **variables** necesarias. Se debe tener claro que representan estas variables al principio y al final de la iteración.
- Identificar como **actualizar la información**, al pasar de una iteración a la siguiente, puede ser necesario utilizar variables adicionales.
- Identificar las **condiciones de terminación**. Puede ser necesario nuevas variables o acciones para mantenerlas actualizadas.
- Identificar los **valores iniciales** de las variables y si es necesaria alguna acción para asignar antes del inicio del bucle.
- Comprobar si son necesarias **acciones adicionales** o finales como la lectura adelantada que se consideran parte de bucle.
- Finalmente, **validar** el diseño comprobando el comportamiento en la primera iteración, la "normal" (entendiendo por esta el caso que no es el primero ni el último) y la última.

Para ilustrar el diseño trabajaremos con el ejemplo de la sucesión de Fibonacci.

PROBLEMA

Construir el programa que imprime los términos de la sucesión de Fibonacci tantos como sea posible, en el rango de los enteros.

Análisis

Un nuevo término en la sucesión de Fibonacci es el resultado de la suma de los dos anteriores

La sucesión comienza con los números 0 y 1. A partir de estos, "cada término es la suma de los dos anteriores".

Como son enteros debo tener en cuenta el límite de los enteros `INT_MAX`

Diseño

- Acciones a repetir

Calcular el término y escribir el número.

```
termino = termino + anterior;
printf( "%d\n", termino);
```

- Actualizar las variables. ¡Ojo! que para no perder el valor de usa una variable auxiliar.

```
aux= termino + anterior;
anterior = termino;
termino = aux;
```

- Condición de terminación. Cuando el término nuevo a calcular supere el límite de los enteros.

```
INT_MAX - termino >= anterior
```

Se pone como resta porque si bien la condición matemática es (`termino + anterior <= INT_MAX`), pero la operación superaría el límite de los enteros, para evitarlo se emplea la resta.

- Valores iniciales de cada variable En este caso el convenio dado a dos primeros elementos de la sucesión de Fibonacci.

Implementación

```
    while (INT_MAX - termino >= anterior)
    {
        aux = termino + anterior;
        anterior = termino;
        termino = aux;
        printf("%10d\n", termino);
    }

    return 0;
}
```

PROBLEMA

Construir el programa que pase un número entero positivo entre 0 y 1024 a binario empleando el método de las divisiones sucesivas.

Análisis El proceso es:

$$
\begin{array}{r|l}
26 & 2 \\
\hline
06 & 13 \\
\cline{2-2}
0 & 1 \\
\cline{1-1}
\end{array}
\qquad
\begin{array}{r|l}
6 & 2 \\
\hline
0 & 3 \\
\cline{2-2}
 & 1 \\
\cline{1-1}
\end{array}
\qquad
\begin{array}{r|l}
2 & 2 \\
\hline
1 & 1 \\
\cline{2-2}
\end{array}
$$

- Comenzamos con el número decimal que queremos convertir a binario, en este caso, 26.
- Dividimos 26 por 2. El cociente es 13 y el resto es 0.

- Anotamos el resto (0), que es el bit menos significativo del número binario. En la posición 10^0
- Dividimos 13 por 2. El cociente es 6 y el resto es 1.
- Anotamos el nuevo resto (1) como el siguiente bit del número binario. El la posición 10^1
- Continuamos dividiendo el cociente resultante por 2 y anotando los restos hasta que el cociente sea 0. En cada paso, anotamos el resto obtenido como el siguiente bit del número binario. (10^{n-1})

Diseño

- Acciones a repetir

Calcular la división y el resto. Acumular el resto sobre el número binario en función del factor (potencias de 10)

```
    resto = cociente % 2;
    cociente = cociente / 2;
    binario = binario + resto * factor;
    factor = factor * 10;
```

- Actualizar las variables. El bucle esta controlado por el valor del cociente, cuando este valga 0 se finaliza el bucle.

```
    cociente = cociente / 2;
```

- Condición de terminación. Cuando el cociente valga 0 (cociente==0). Si empleamos un bucle `while` la condicion es:

```
(cociente != 0)
```

- Valores iniciales de cada variable

```
factor = 1; // es 10^0
cociente = decimal;  // valor leido de teclado
```

Implementación

```
    int decimal, cociente, resto, binario = 0, factor = 1;

    // Solicitar al usuario ingresar el número decimal dentro del rango
    do {
        printf("Ingrese un número decimal entre 0 y 1024: ");
        scanf("%d", &decimal);
    } while (decimal < 0 || decimal > 1024);

    cociente = decimal;

    // Convertir a binario utilizando restas sucesivas
    while (cociente != 0) {
        resto = cociente % 2;
        cociente = cociente / 2;
        binario = binario + resto * factor;
        factor = factor * 10;
    }

    // Imprimir el número binario
    printf("El número binario equivalente es: %d\n", binario);
```

Consideraciones sobre el bucle `for`

Cuando tenemos bucles controlados por contador, es decir, sabemos cuantas iteraciones tenemos o al menos el rango en el que se producen las iteraciones el tipo de bucle candidato a ser utilizado es un **bucle desde**. No obstante, si nos vamos al lenguaje C. Este tipo de bucle `for` puede ser tan versátil como el `while` debido a la posibilidad de notación incompleta en su sintaxis, aunque esto no tiene porque poder extrapolarse a otros lenguajes de programación.

Un bucle controlado por contador requiere:

- el nombre de una **variable** de control,
- el valor **inicial** de la variable de control,
- el **incremento** (o decremento) por el que se modifica la variable de control en cada iteración, y
- la **condición de continuación** del bucle que comprueba el valor final de la variable de control para determinar si el bucle debe continuar.

Si se utiliza `while` el incremento se debe expresar explícitamente en una instrucción mientras que en `for` este incremento se hace automáticamente y se hace **siempre**. En un bucle `while` el incremento puede estar condicionado, en un bucle `for` se hace siempre.

```
for(exp1;exp2;exp3)
    instrucciones;

for(exp1;exp2;exp3){
    instrucciones;}
```

Donde:

- exp1: inicialización. Puede haber más de una asignación separadas por comas.
- exp2: condición de comprobación para volver a ejecutar otra iteración del bucle.
- exp3: actualización. Puede haber más de una asignación separadas por comas.

Puede faltar cualquier parte del bucle, pero los separadores ";" no pueden faltar.

```
for (int counter = 1; counter <= 5; ++counter)
```

Una de las particularidades que tiene el bucle `for` en C es que las tres expresiones de una cabecera for son opcionales. Si omite la condición `exp2` la condición es siempre verdadera, creando así un bucle infinito. Puede omitir la expresión de inicialización si el programa inicializa la variable de control antes del bucle. También se puede omitir la expresión de incremento si el programa calcula el incremento en el cuerpo del bucle o si no se necesita el incremento.

1. La variable de control varía de 1 a 100 en incrementos de 1.

```
for (int i = 1; i <= 100; ++i)
```

2. La variable de control varía de 100 a 1 en incrementos de -1 (es decir, decrementos de 1).

```
for (int i = 100; i >= 1; --i)
```

3. La variable de control varía de 7 a 77 en incrementos de 7.

```
for (int i = 7; i <= 77; i += 7)
```

4. La variable de control varía de de 20 a 2 en incrementos de -2.

```
for (int i = 20; i >= 2; i -= 2)
```

5. La variable de control varía sobre los valores 2, 5, 8, 11, 14 y 17.

```
for (int j = 2; j <= 17; j += 3)
```

6. La variable de control varía sobre la siguiente secuencia de valores: 44, 33, 22, 11, 0.

```
for (int j = 44; j >= 0; j -= 11)
```

Un caso extremo es no poner ni variable de control ni incremento y utilizar la condición con las variables y/o datos que se manejan en el bucle. Cuidado con los bucles infinitos:

```
    int anterior = 0;
    int termino = 1;
    int aux;

    for (; termino <= INT_MAX - anterior;) {
        aux = termino + anterior;
        anterior = termino;
        termino = aux;

        printf("%10d\n", termino);
    }
```

Se pueden definir diversos contadores para el bucle for por ejemplo:

```
#include <stdio.h>

int main() {
    int i, j;

    for (i = 0, j = 0; i < 5; i++, j+=2) {
        printf("i: %d, j: %d\n", i, j);
    }

    return 0;
}
```

En este ejemplo, el bucle for tiene dos contadores (i y j) inicializados en la misma línea, separados por una coma. La condición de terminación del bucle es i $<$ 5, y después de cada iteración, tanto i como j se incrementan (i++ e j+=2, respectivamente)

Otra particularidad de los bucles `for` en C, en Java o en Python es la posibilidad de interrumpir el bucle con la instrucción `break`, pero otro lenguajes con MATLAB no tienen esta opción, obligando a la utilización de un bucle `while`.

```
for (int i = 0; i < 100; ++i)
{
    int num;

    printf("Introduce un entero positivo (o un número negativo para salir): ");
    scanf("%d", &num);

    if (num < 0)
    {
        printf("Se introdujo un número negativo. Saliendo del bucle.\n");
        break;
```

```
    }

        // Resto del código dentro del bucle
        printf("Número introducido: %d\n", num);
}
```

Para esta asignatura consideraremos incorrecto el uso de `break` y su contrario `continue` en los bucles o condicionales puesto que se infringe el teorema de la programación estructurada. La única excepción son las instrucciones de selección múltiple `switch`.

De forma similar, la instrucción `continue` se puede emplear para indicar que no se hace nada y algunos malos programadores la usan en las instrucciones `if`. En esta asignatura se considerará un error.

Ejemplos de diseño de programas de aplicación en la ingeniería

PROBLEMA

Construir un programa que calcule e imprima la resultante de un conjunto de fuerzas concurrentes. Las componentes espaciales de cada fuerza son introducidas por teclado de una en una, finalizándose la entrada de datos con una fuerza nula, indicando también cuantas fuerzas se han acumulado.

Análisis

- Información de E:
  - Compon. de cada fuerza: F=(fx,fy,fz) deben ser reales
  - Fuerza nula (centinela): 0=(0.0,0.0,0.0)
- Información de S:
  - Compon. de resultante: R=(rx,ry,rz) también real
  - Número de fuerzas introducidas.

Se leerán por teclado las tres componentes de cada fuerza que se sumaran componente a componente hasta que se introduzca la fuerza nula.

Diseño

- Bucle controlado por **centinela**, la fuerza nula
- Se necesita acumular el valor de la fuerza resultante componente a componente: `rx, ry, rx`
- No es necesaria la lectura adelantada porque el centinela es cero y no acumula nada aunque se sume en la última iteración, se realiza la suma pero no afecta al resultado. Utilizaremos lectura a demanda.

Implementación

```
rx=ry=rz=0;
n=0;
while((fx!=0)||(fy!=0)||(fz!=0)){
    printf("Introducir componentes de Fuerza(%d):\n",n);
    printf("\tFx: ");
    scanf(" %f", &fx);
    printf("\tFy: ");
    scanf(" %f", &fy);
    printf("\tFz: ");
    scanf(" %f", &fz);
    rx=rx+fx;
    ry=ry+fy;
    rz=rz+fz;
    n=n+1;
}
printf("\nResultante de %d fuerzas:",n-1);
printf(" (%.1f,%.1f,%.1f)",rx,ry,rz);
```

Aunque también se puede hacer con lectura adelantada evitando así la última suma.

```
n = 0;
printf("Introducir componentes de Fuerza(%d):\n", n);
printf("\tFx: ");
scanf(" %f", &fx);
printf("\tFy: ");
scanf(" %f", &fy);
printf("\tFz: ");
scanf(" %f", &fz);
rx = 0;
ry = 0;
rz = 0;
while ((fx != 0) || (fy != 0) || (fz != 0))
{
    rx = rx + fx;
    ry = ry + fy;
    rz = rz + fz;
    n = n + 1;
    printf("Introducir componentes de Fuerza(%d):\n", n);
    printf("\tFx: ");
    scanf(" %f", &fx);
    printf("\tFy: ");
    scanf(" %f", &fy);
    printf("\tFz: ");
    scanf(" %f", &fz);
}
printf("\nResultante de %d fuerzas:", n);
printf(" (%.1f,%.1f,%.1f)", rx, ry, rz);
return 0;
```

PROBLEMA

Construir un programa que calcule e imprima en pantalla el máximo común divisor de dos números enteros positivos introducidos por teclado.

Análisis

Trabajaremos con tres variables enteras, `n1, n2` para los dos números de entrada y `mcd` para el resultado.

Todos conocemos el algoritmo se Euclides para el cálculo, descomponiendo en factores primos y cogiendo los comunes elevados al mayor exponente. Pero estamos programando y aún no sabemos hacer cosas tan complejas.

¿Existe alguna manera aunque sea más lenta de calcular el "divisor más grande" de ambos números?.

- Método de prueba y error: buscar el mayor de los divisores comunes:
  - Inicializar `mcd` a cualquiera de los dos números (al menor).
  - Dividir `n1` y `n2` por `mcd`, y si el resto de ambas divisiones es cero, ese es el mayor de los divisores comunes, y de lo contrario probar con el valor del divisor inmediatamente inferior y repetir el proceso que sabemos que acabará seguro con el 1.

Diseño

- Tenemos un bucle controlado por la condición

  `(n1% mcd == 0) && (n2% mcd == 0)`

- Cuerpo del bucle

  `mcd=mcd-1`

- Inicialización

`mcd= el más pequeño de ambos números`

Si se quieren minimizar las iteraciones.

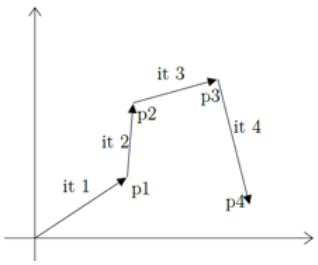
Implementación

```
{
    printf("Introduzca un número positivo: ");
    scanf(" %d", &n1);
} while (n1 <= 0);

do
{
    printf("Introduzca otro número positivo: ");
    scanf(" %d", &n2);
} while (n2 <= 0);

mcd = n1;

while ((n1 % mcd) || (n2 % mcd))
    mcd--;
```

PROBLEMA

Suponiendo un sistema de coordenadas en dos dimensiones donde se utiliza la distancia euclídea. Construir un programa en C que calcule la distancia recorrida comenzando desde el origen de coordenadas. Se irán pidiendo por teclado las dos coordenadas de los puntos de paso y finaliza el trayecto cuando se introduce el punto 0,0.

Análisis



Se leerán un número no conocido de puntos y el último sera 0,0 en el dibujo el `p5`.

Diseño

En cada iteración del bucle se lee un nuevo punto y se acumula la distancia, si el punto leído no es 0,0. Después se coloca como punto anterior el punto leído. El primer punto anterior es el origen de coordenadas.

Se puede partir del ejemplo de dos puntos descrito en páginas anteriores.

Implementación

```
printf("Coordenada X: ");
scanf("%lf", &x1);
printf("Coordenada Y: ");
scanf("%lf", &y1);

while (!((x1 == 0) && (y1 == 0)))
{
    double distancia = sqrt((x1 - x0) * (x1 - x0) + (y1 - y0) * (y1 - y0));
         // Calcular la distancia euclidiana
    distanciaTotal += distancia;      // Sumar la distancia al total
    x0 = x1;
    y0 = y1;
    printf("Coordenada X: ");
    scanf("%lf", &x1);
    printf("Coordenada Y: ");
    scanf("%lf", &y1);
}

printf("La distancia total recorrida es: %.2lf\n", distanciaTotal);

return 0;
```

Estructuras de repetición anidadas

Estructura **de bucle** que contiene otra estructura repetitiva y así sucesivamente.

- Reglas de las estructuras repetitivas anidadas
  - El bucle interno debe estar incluido totalmente dentro del bucle externo.
  - Para cada iteración del bucle externo se ejecuta totalmente el bucle interno
  - Diseño de bucles anidados:
    - Empezar diseñando el bucle más externo.
    - Diseño del proceso que se repite - diseñar el bucle interno (anidado).
    - Aunque algunas veces se puede hacer del interior al exterior. Pero siempre en orden.

PROBLEMA

Construir un programa que imprima en pantalla la tabla de multiplicar de los 9 primeros números naturales.

Análisis

- Información de E:
  - Primer número de la tabla: 1 (cte. entera)

- Último número de la tabla: 9 (cte. entera)

- Información de S:
  - Tabla de multiplicar:

*	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2		4	6	8	10	12	14	16	18
3			9	12	15	18	21	24	27
4				16	20	24	28	32	36
5					25	30	35	40	45
6						36	42	48	54
7							49	56	63
8								64	72
9									81

Diseño

- Cabecera de la tabla: Un bucle de 1 a 9
- Tabla:
  - Un bucle exterior que cuenta de 1 a 9
  - Para cada fila:
    - Huecos en blanco: de 1 a la fila actual
    - Para números desde la fila actual, según la columna actual calcular el producto.

Implementación

```
int main()
{
        int i, j;

        printf("TABLA DE MULTIPLICAR\n");
        printf("====================\n\n");
        printf("%-3c |", '*');
        i = 1;
        while (i <= 9)
        {
                printf("%3d", i);
                ++i;
        }
        printf("\n");
        i = 1;
        while (i <= 32)
        {
                printf("-");
                ++i;
        }
        printf("\n");
        i = 1;
        while (i <= 9)
        {
```

```
                printf("%3d |", i);
                j = 1;
                while (j < i)
                {
                        printf("%3c", ' ');
                        ++j;
                }
                j = i;
                while (j <= 9)
                {
                        printf("%3d", i * j);
                        ++j;
                }
                printf("\n");
                ++i;
        }
        return 0;
}
```

PROBLEMA

Construir un programa que imprima en pantalla mediante asteriscos las aristas de un rectángulo, dados por teclado las dimensiones del mismo, donde la base debe estar comprendida entre 3 y 40, y la altura entre 3 y 20. Ejemplos:

a=5, h=5

```
* * * * *
*       *
*       *
*       *
* * * * *
```

Análisis - Entrada: - Número de filas `f` - Número de columnas `c`

Diseño ¿Hay anidamiento? Si en el bloque central de asteriscos

- Primer bucle: Escribir c asterisco
- Segundo bucle: Repetir f-2 veces:
  - Escribir , *repetir c-2 espacios en blanco, Escribir*
- Otro bucle como el primero

Implementación

```
        int a;
        int h;
        int i, j;

        printf("RECTANGULO DE ASTERISCOS\n");
        printf("========================\n\n");
        do
        {
                printf("Introduzca ancho [3-40]: ");
                scanf(" %d", &a);
        } while ((a < 3) || (a > 40));
        do
        {
                printf("Introduzca altura [3-20]: ");
                scanf(" %d", &h);
```

```
} while ((h < 3) || (h > 40));
/* Arista superior */
j = 1;
while (j <= a)
{
        printf(" *");
        ++j;
}
printf("\n");
/* Aristas laterales */
i = 2;
while (i < h)
{
        printf(" *");
        j = 2;
        while (j < a)
        {
                printf("  ");
                ++j;
        }
        printf(" *\n");
        ++i;
}
/* Arista inferior */
j = 1;
while (j <= a)
{
        printf(" *");
        ++j;
}
```

PROBLEMA

Análisis

Triángulo de números. Se trata de escribir un triangulo de números del 1 al 9 donde la entrada sea la altura del triángulo.

- Entrada:
  - Número de filas `h`

```
        1
       121
      12321
     1234321
    123454321
   12345654321
  1234567654321
 123456787654321
12345678987654321
```

Diseño

- El bucle exterior cuenta de 1 a h
- Cada fila i:
  - n-i - Espacios en blanco
  - de 1 a i

- de i a 1

Implementación

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

int main()
{
        system("cls||clear");
    int altura, espacios, numeros, i, j;

    do{
        printf("Ingrese la altura del triángulo: ");
        scanf("%d", &altura);
    }while (altura<1 || altura >9);


    for (i = 1; i <= altura; i++) {
        for (espacios = 1; espacios <= altura - i; espacios++)
            printf(" ");
        for (numeros = 1; numeros <= i; numeros++)
            printf("%d", numeros);
        for (numeros = i - 1; numeros >= 1; numeros--)
            printf("%d", numeros);
        printf("\n");
    }

    return 0;
}
```

PROBLEMA

Escribe un programa en C que para un valor de una tirada de 3 dados por teclado, indique cuantas de las posibles tiradas de tres datos ganan a la tirada actual.

Análisis

Como entrada el valor de la tirada actual, como salida cuantas superan el valor dado.

Diseño

- Necesitamos calcular todas las tiradas posibles y acumular aquellas que superen el valor.
- Como hay tres dados necesitamos tres bucles uno para emular cada dado.

Implementación

```
#include <stdio.h>

int main()
{
    int tirada_actual;
    int contador_ganadoras = 0;
    do
    {
        printf("Ingrese el valor de la tirada de 3 dados (de 3 a 18): ");
        scanf("%d", &tirada_actual);
    } while (tirada_actual < 3 || tirada_actual > 18);

    for (int dado1 = 1; dado1 <= 6; dado1++)
    {
        for (int dado2 = 1; dado2 <= 6; dado2++)
        {
            for (int dado3 = 1; dado3 <= 6; dado3++)
            {
```

```
                int suma_tirada = dado1 + dado2 + dado3;
                if (suma_tirada > tirada_actual)
                {
                    contador_ganadoras++;
                }
            }
        }
    }

    printf("Numero de tiradas ganadoras: %d\n", contador_ganadoras);

    return 0;
}
```

Números aleatorios

Número aleatorio: su valor no es conocido a priori de forma determinista (es impredecible su valor).

Aplicaciones de los números aleatorios, por ejemplo:

- Juegos de azar (dados, lotería, bingo, ...).
- Técnicas de cálculo numérico (método de Monte Carlo).
- Simulaciones (generación de datos de entrada y valores de forma automática)

Los generadores de números pseudo-aleatorios son funciones de biblioteca usuales en todos los lenguajes de programación.

- La función `rand()` Genera y devuelve a través del identificador de la función un número aleatorio entero en el intervalo `[0,RAND_MAX]`. La constante `RAND_MAX` está definida en el archivo `stdlib.h` como `2E15-1`. Ejemplo:

```
#include <stdlib.h>

int x;
. . .
x = rand();
```

Para generar números aleatorios enteros en un intervalo $[0, n-1]$:

```
x= rand() % n     /* n entera positiva */
```

o bien:

```
x = random(n);
```

Generar números aleatorios enteros comprendidos entre 1 y n (ambos incluidos):

```
x = rand() %n +1;
```

o bien:

```
x= random(n) + 1;
```

Generar números aleatorios enteros entre a y b ($a < b$):

```
random(b-a+1)+a;
```

Generar números aleatorios reales en un intervalo $[a, b]$:

```
a+(b-a)*1.0*rand()/RAND_MAX
```

- función `srand()`

Inicializa el generador de números aleatorios con un valor aleatorio obtenido del reloj del sistema.

Se ejecuta solo una vez al principio del programa, consiguiendo de esta forma que en sucesivas ejecuciones del mismo programa no se genere la misma secuencia de números aleatorios.

```
#include <stdlib.h>
#include <time.h>
time_t t;
. . .
srand((unsigned) time(&t));
```

PROBLEMA

Construir un programa en C que genere un número de la lotería de Navidad, pero preguntando al usuario si quiere un número específico para una posición.

Análisis

En el sorteo de la Lotería de Navidad hay 100.000 números distintos. Estos van del 00000 al 99999

Diseño

Se necesitan generar 5 números aleatorios entre 0 y 9, y antes de generarlo preguntar si se fija un valor específico para esa posición.

Un bucle donde se pregunte/genere el número y vaya acumulando el resultado multiplicado por 10.

Implementación

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

int main()
{

    srand(time(NULL)); // Inicializar la semilla del generador de números aleatorios

    int numeroLoteria = 0;

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

        printf("Ingrese el digito %d (o ingrese -1 para un valor aleatorio): ", i + 1);
        scanf("%d", &digito);

        if (digito == -1)
        {
            digito = rand() % 9;
        }
        numeroLoteria = numeroLoteria * 10 + digito;
    }

    printf("El número de la lotería de Navidad es: %d\n", numeroLoteria);
```

```
    return 0;
}
```

PROBLEMA

Construir un programa en C emule el juego en n tandas de dos jugadores, de forma que cada jugada de un jugador consista en adivinar un número aleatorio entre 1 y 100, recibiendo como pista si es mayor o menor que un número introducido por teclado. El número máximo de intentos en cada jugada de cada jugador es de 10. Para saber que jugador gana se suman los intentos de todas las tandas de cada jugador. No obstante, en cada jugada se puede ajustar la puntuación de forma que si no se adivina el número se sumen 2 intentos más y si se emplean menos intentos que el jugador de la jugada anterior se resta 1 intento.

Análisis

- Cada tanda son dos jugadas una para cada jugador.
- Se ejecutan n tandas y se van sumando los intentos gastados de cada jugador, gana quien tenga menos intentos.
- Hay 10 intentos máximo
- Al final de cada jugada se ajusta el valor de los intentos, por una parte si no se ha adivinado se suman 5 y si adivina en menos intentos que la jugada justo anterior se restan 1 al total de intentos.
- Gana el jugador que emplea menos intentos

Diseño

- Bucles anidados
- Bucle exterior que cuenta las tantas y acumula los intentos de cada jugada sobre los totales de intentos de los jugadores
- Bucle de jugada (el mas interno) que esta controlado por dos variables centinela:
    - Si se ha acertado
    - Si se ha llegado a 10 intentos.
- Bucle intermedio que cuenta dos bucles jugada para cada jugador.

Detalle de los bucles - Bucle exterior

```
for (int tanda = 1; tanda <= cuantasTandas; tanda++) {
// jugada de jugador 1 y jugada de jugador 2

}
// Mostrar ganador según el total de jugadas de cada uno
```

No requiere de actualizaciones o inicializaciones adicionales solo cuenta el número de tandas

- Diseño del bucle intermedio.

Cada tanda consiste en dos jugadas, una de cada jugador. Son siempre dos

```
for (int jugador = 1; jugador <= 2; jugador++)
// Adivinación
// Ajuste de intentos según se haya adivinado o no
// Ajuste comparando con jugada anterior
```

La actualización y ajuste en cada iteración. También se debe reservar el valor de los intentos para la siguiente jugada. Esta valor no se actualiza entre tandas.

```
if (!acertado && intentos >= 10)
{
   printf("Has excedido el limite de intentos. El numero secreto era %d.\n", numero_secreto);
   intentos += 2;
}

if (intentos < intentos_anteriores)
{
        intentos -= 1;
        intentos_anteriores = intentos;
}
 else {  intentos_anteriores = intentos;}

 if (jugador == 1)
     total_intentos_jugador1 += intentos;
 else
     total_intentos_jugador2 += intentos;
```

.

- Diseño de bucle interior. Condición de terminación.

  ```
  (!acertado && intentos < MAX_INTENTOS)
  ```

  Acciones a repetir:

  - Leer número
  - Si coincide activar flag/indicador acertado sino dar pista de mayor o menor
  - Actualizar los intentos

Implementación

```
/*
Descripción: Juego adivinar un número para dos jugadores, con correctión de puntuación
Área:  Números aleatorios y tres bucles anidados.
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_INTENTOS 10

int main() {
    int cuantasTandas;
    int total_intentos_jugador1 = 0, total_intentos_jugador2 = 0;
    int intentos_anteriores=0;
    srand(time(0));

    do {
        printf("¿Cuantas tandas quieres jugar? ");
        scanf("%d", &cuantasTandas);
    } while (cuantasTandas < 1);


    for (int tanda = 1; tanda <= cuantasTandas; tanda++) {
        printf("\nTanda %d\n", tanda);
```

```
        for (int jugador = 1; jugador <= 2; jugador++) {
            int intentos = 0, leido;
            int numero_secreto = rand() % 100 + 1;
            int acertado = 0;

            printf("Turno del Jugador %d\n", jugador);

            while (!acertado && intentos < MAX_INTENTOS) {
                do {
                    printf("Introduce un numero entre 1 y 100: ");
                    scanf("%d", &leido);
                } while (leido < 1 || leido > 100);

                if (leido > numero_secreto) {
                    printf("El numero es menor.\n");
                } else if (leido < numero_secreto) {
                    printf("El numero es mayor.\n");
                } else {
                    printf("¡Felicidades! Adivinaste el numero secreto en %d intentos.\n", intentos + 1);
                    acertado = 1;
                }
                intentos++;
            }
            if (!acertado && intentos >= 10)
            {
               printf("Has excedido el limite de intentos. El numero secreto era %d.\n", numero_secreto);
               intentos += 2;
            }


           if (intentos < intentos_anteriores)
           {
                   intentos -= 1;
                   intentos_anteriores = intentos;
           }
           else {intentos_anteriores = intentos;}
            if (jugador == 1)
                total_intentos_jugador1 += intentos;
            else
                total_intentos_jugador2 += intentos;

        }
    }

    if (total_intentos_jugador1 == total_intentos_jugador2)
        printf("\n¡Empate!\n");
    else if (total_intentos_jugador1 < total_intentos_jugador2)
        printf("\n¡Gana el Jugador 1!\n");
    else
        printf("\n¡Gana el Jugador 2!\n");

    return 0;
}
```

Normas de estilo. Bloques de código

El archivo de código fuente

Existe una estructura global que se debe respetar:

- Los comentarios de presentación. Todos los archivos deben comenzar con un comentario con el autor, fecha y descripción. Aunque hay muchas propuestas, buscamos una sencilla.

- La inclusión de librerías de cabecera del sistema, los ficheros .h con el **#include** y entre ángulos (<...>) el nombre del fichero.

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

- Las librerías propias de la aplicación. Normalmente, en grandes aplicaciones, se suelen realizar varias librerías con funciones, separadas por su semántica. Nosotros no vamos a utilizarlas.

```
#include "rata.h"  /* Rutinas para control del ratón        */
#include "cola.h"  /* Primitivas para el manejo de una cola */
```

- Constantes simbólicas y definiciones de macros, con **#define**.
- Definición de tipos, con **typedef**.
- Declaración de funciones: Se escribirá sólo el prototipo de la función, no su implementación. De esta forma, el programa (y el programador) sabrá el número y el tipo de cada parámetro y cada función.
- Declaración de variables globales.
- Implementación de funciones: Aquí se programarán las acciones de cada función, incluida la función principal, función **main()**, que se suele poner la primera.

La función main() se pone la primera, aunque a veces se pone al final y **nunca** en medio. El resto de funciones, se suelen ordenar siguiendo algún criterio, por ejemplo el orden de aparición en la función principal y poner juntas las funciones que son llamadas desde otras. Deben aparecer en el mismo orden que sus prototipos, ya que así puede ser más fácil localizarlas. Si no hay criterio se ordenan al menos alfabéticamente.

```
/*
** Descripción: Arquitectura de un programa en C
** Asunto: Introducción
** Área: PRx, TEx
*/

/* Includes del sistema */
#include <stdio.h>

/* Includes de la aplicacion */
// #include "Ejemplos.h"

/* constantes */
#define MAXIMO 100
#define ERROR "Opcion no permitida"

/* tipos definidos por el usuario */

/* Prototipo de funciones locales */

int main(void)
{

    // Instrucciones, bloques de código,  que definen la funcionalidad implementada

    return 0;
}

/* Definiciones de funciones locales */
```

Uso correcto de los comentarios

- Escribir y luego mantener comentarios es un gasto.
- El compilador no verifica los comentarios, por lo que no hay forma de determinar si los comentarios son correctos.
- Por otro lado, tampoco se tiene la garantía de que la computadora está haciendo exactamente lo que su código le indica.
- Se **eliminarán las tildes** para evitar problemas de compatibilidad, tanto en los comentarios como en los textos.

En este apartado cabe decir que se quieren incluir tildes en `Windows` se debe incluir una librería específica `#include <window.h>` e incluir dos llamadas a funciones al principio del programa.

```
#include <windows.h>
....
system("cls||clear"); // Limpia el terminal valido para Window y linux
SetConsoleOutputCP(CP_UTF8);
SetConsoleCP(CP_UTF8);
```

- Los comentarios no deben duplicar el código. No todo lo que decimos los profesores es cierto. Evitar comentarios como el que se muestra a continuación.

```
if (x > 3)
  {
  ...
  } // if
```

- Los buenos comentarios no justifican el código poco claro. Lo que sea necesario para el código debe aparecer en él no en el comentario. Veamos un ejemplo

```
private static Node obtenerMejorNodoHijo(Node node)
{
    Node n; // candidato a mejor nodo
    for (Node node : node.obtenerHijo())
    {
        // modificar n si el estado actual es mejor
        if (n == null || utilidad(node) > utilidad(n))
        {
            n = node;
        }
    }
    return n;
}
```

La necesidad de comentarios podría eliminarse con una mejor denominación de variables:

```
private static Node obtenerMejorNodoHijo(Node node)
{
    Node mejorNodo;
    for (Node actualNodo : node.obtenerHijo())
    {
        if (mejorNodo == null || utilidad(actualNodo) > utilidad(mejorNodo))
        {
            mejorNodo = actualNodo;
        }
    }
    return mejorNodo;
}
```

- Si no se puede escribir un comentario claro, puede haber un problema con el código.
- Los comentarios deben disipar la confusión, no causarla.

Sangrado y separaciones

Al desarrollar es importante que se use una correcta separación entre líneas, algo que será de gran utilidad para identificar de forma rápida bloques (o conjuntos) de código. Un programa debe ser claro, estar bien organizado y que sea fácil de leer y entender. Casi todos los lenguajes de programación son de formato libre, de manera que los espacios no importan, y podemos organizar el código del programa como más nos interese.

- Uso correcto de tabulaciones. Las tabulaciones deben ser uniformes en todo el código, de tal forma que si en el editor o el IDE usado para escribir el código está configurado para que las tabulaciones sean, por ejemplo, cuatro espacios, se debe mantener de esta forma en todo el código.

CASO CORRECTO

```
if (correcto)
  printf("no hubo problemas");
else
  printf("SI hubo problemas");
```

CASO INCORRECTO

```
if (correcto) printf("no hubo problemas");
else printf("SI hubo problemas");
```

- Separar bloques de código por líneas en blanco.
- Las llaves deben colocarse según el estándar Allman, también conocido como BSD, es decir, en la línea siguiente a un if, o a un while.
- Debe haber un espacio en blanco antes y después de los operadores de comparación, asignación, etc. También debe haber un espacio entre las palabras clave (for, while, if, return, etc.) y las expresiones que le siguen.

CASO CORRECTO

```
int db_sync(void)
{
    int i, retval = 0, result = 0;

    for (i = 0; i < P_SIZE; i++)
    {
        if (param_info[i].dirty && param_info[i].sync_cb)
        {
            retval = param_info[i].sync_cb(i, param_db[i]);
            result |= retval;
            if (retval == 0)
                param_info[i].dirty = false;
        }
        else
        {
            LOG_WARNING("No callback for param %d", i);
        }
    }
    return result;
}
```

CASO INCORRECTO

```
int db_sync(void)
{
  int i, retval = 0, result = 0;

  for (i = 0; i < P_SIZE; i++){
  if (param_info[i].dirty && param_info[i].sync_cb) {
    retval = param_info[i].sync_cb(i, param_db[i]);
    result |= retval;
    if (retval == 0)
    param_info[i].dirty = false;
  } else {
  LOG_WARNING("No callback for param %d", i);
  }
  }
  return result;}
```

- Las condiciones deben clarificarse para facilitar la lectura

CASO CORRECTO

```
 while (cond1 != "error"  && cond2 != TRUE &&
        cond3 >= 7.5 && cond4 <= 10 &&
        cond5 != "esto es verdad")
```

CASO INCORRECTO

```
 while ( cond1 != "error"  && cond2 !=TRUE &&   cond3 >=
 7.5 && cond4 <=10 && cond5 !="esto es verdad");
```

- En una estructura o unión los elementos deben ir cada uno en una línea y tabulado, tanto en la definición directa como el tipo usando **typedef**

CASO CORRECTO

```
struct barco
{
    int flotacion;   /* en metros */
    int tipo;       /* según clasif. internacional */
    float precio;    /* en dólares */
};
```

CASO CORRECTO

```
typedef struct{
    int flotacion;   /* en metros */
    int tipo;     .  /* según clasif. internacional */
    float precio;    /* en dólares */
}tipo_barco;
```

Indentación típica en las estructuras de control

```
/* Instruccion if-else
*/

     if (condicion)
     {
        Instruccion1;
        Instruccion2;
        ...
     }
     else
     {
```

```
        Instruccion1;
        Instruccion2;
        ...
     }

/*Instruccion swith
*/

     switch (expresion)
     {
       case expresion1: Instruccion1;
                        Instruccion2;
                        ...
                        break;
       case expresion2: Instruccion1;
                        Instruccion2;
                        ...
                        break;
       .
       .
       .
       case default   : Instruccion1;
                        Instruccion2;
                        ...
     }

/* Instruccion for
*/

     for (exp1;exp2;exp3)
     {
        Instruccion1;
        Instruccion2;
        ...
     }

/* Instruccion while
*/
     while (condicion)
     {
        Instruccion1;
        Instruccion2;
        ...
     }

/* Instruccion do-while
*/

     do
     {
        Instruccion1;
        Instruccion2;
        ...
     } while (condicion);
```

Aunque estos formatos no son en absoluto fijos, lo que es muy importante es que quede bien claro las instrucciones que pertenecen a cada bloque, o lo que es lo mismo, donde empieza y termina cada bloque. En bloques con muchas lineas de código y/o con muchos anidamientos, se recomienda añadir un comentario al final de cada llave de cierre del bloque, indicando a qué instrucción cierra. Por ejemplo:

```
     for (exp1;exp2;exp3)
     {
        instruccion1;
        Instruccion2;
        ...

        while (condicion_1)
```

```
    {
       instruccion1;
       instruccion2;
       ...
       if (condicion)
       {
          instruccion1;
          instruccion2;
          ...

          while (condicion_2)
          {
            instruccion1;
            instruccion2;
            ...
          } /*while (condicion_2)*/

       } /*if*/
       else
       {
          instruccion1;
          instruccion2;
          ...

          if (condicion)
          {
             instruccion1;
             instruccion2;
             ...
          }
          else
          {
             instruccion1;
             instruccion2;
             ...
          }

       } /*else*/
    } /*while (condicion_1)*/
} /*for*/
```

Resumen

Elemento	Notación	Ejemplo	
Constantes simbólicas	Mayúsculas	PI, MAX_LECTORES	1
variables	camelCase	areaCirculo	2
variables triviales:	bucles, continuar	i, j, k	3
	carácter	c	4
Variable globales	Con prefijo g_	g_numLectores	5
Variables Puntero	Con prefijo p_	p_enteroLista	6
estructuras	camelCase	numeroComplejo	7
typedef	incluir palabra **tipo**	tipo_numeroComplejo	8
typedef (alternativo)	Colocar el nombre en mayúscula	NumeroComplejo	
cadenas (tipo)	indicar tamaño	cadena20	9
funciones	camelCase	calcularIVA	10
Archivos	PascalCase	AceleracionNormal.c	11

```c
#include <stdio.h>

// Ejemplo 1
#define MAXIMO 100
#define ERROR "Opción no permitida"

int g_numLectores = 10; // Ejemplo 5

int main(void)
{
   ....
   float areaCirculo,      // Ejemplo 2
   int *p_areaCirculo;     // Ejemplo 6

struct numeroComplejo1{
   float parteReal;
   float parteImaginaria;
};

typedef struct{
   float parteReal;
   float parteImaginaria;
}tipo_Complejo;

tipo_Complejo numeroComplejo2;

}

// Ejemplo 9
typedef char cadena20[21];
   cadena20 nom;
```

Funciones. Diseño Modular

Introducción

La programación estructurada se basa en que:

- Todo programa tiene un único punto de inicio y un único punto de fin,
- Uso de un número limitado de estructuras de control: secuenciales, alternativas y repetitivas,
- Prohibidos los saltos de una instrucción a otra.

Pero cuando el problema es complejo y requiere de muchas líneas de código, el resultado son programas muy largos y muy difíciles de mantener.

La **programación modular** es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable, puesto que cada módulos que se puede analizar, programar y depurar independientemente.

Un módulo es un conjunto de instrucciones con **propósito único e identificable** y/o proporciona unos determinados resultados, y que puede ser llamado (invocado) desde el programa principal o desde otros módulos.

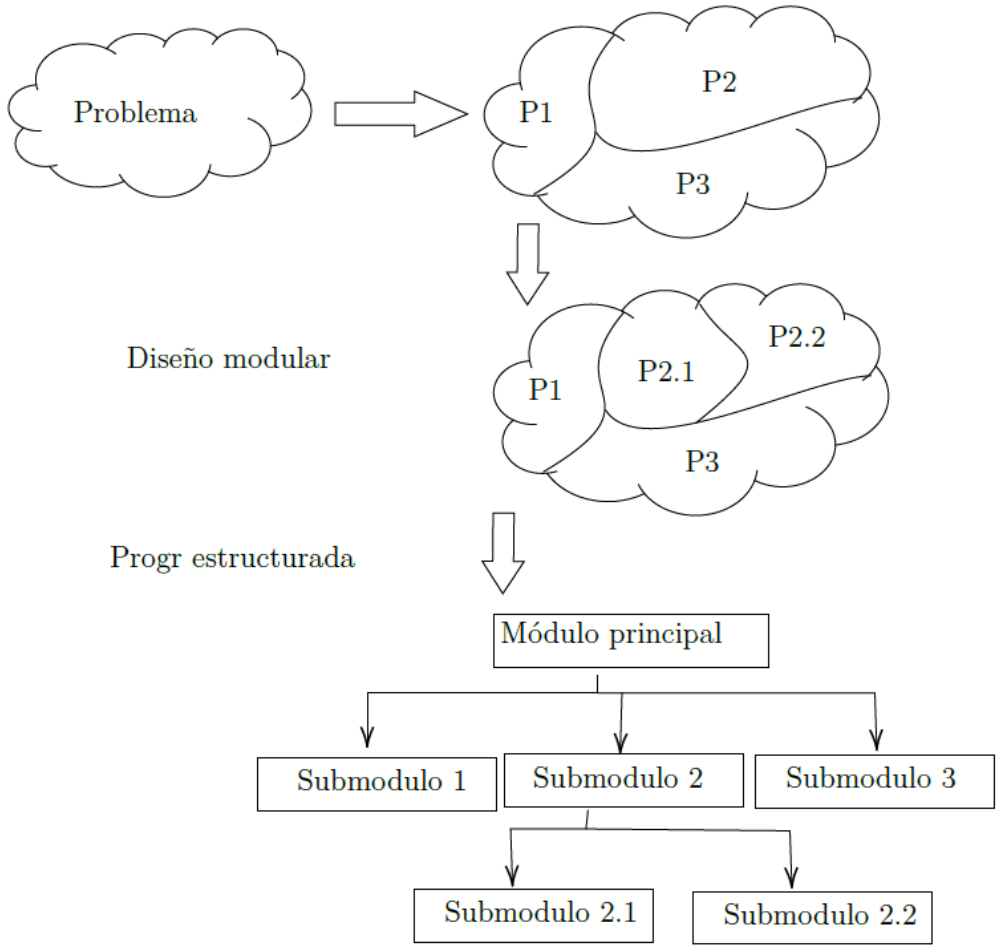
Módulo, subprograma o función son sinónimos en C, se llaman funciones. Todo programa en C tiene al menos un módulo `main()`.

De esta forma el diseño modular: Es una técnica que permite diseñar la solución de un problema con base en la modularización o segmentación, dado un enfoque de arriba hacia abajo (top-down).

Se descompone el problema en unidades independientes y que se resuelven de forma separada para después integrar estas soluciones parciales. Se trata de enfoque aceptado en todas las disciplinas de ingeniería. Las etapas del la construcción de un programa modular son por tanto:

1. Descomponer el problema en sub-problemas. Si alguno de estos sub-problemas sigue siendo complicado, se sub-divide en nuevos sub-sub-problemas, y así sucesivamente hasta que los sub-problemas sean tan pequeños que se puedan resolver con programación estructurada: (secuencia, selectiva, iteración).
2. Resolver cada sub-problema por separado.
3. Combinar (integrar) todas las soluciones parciales para obtener la solución al problema original

De esta forma lo que realmente tenemos al finalizar es un organigrama jerárquico de problemas



Normalmente se tienen entre dos o tres niveles de en la jerarquía si bien el nivel intermedio podría multiplicarse para problemas complejos.

- Nivel 0: La solución se establece en términos amplios, usando el lenguaje del dominio del problema.
- Niveles intermedios: Orientación más procedimental. La terminología orientada al problema incorpora más detalles que se acompañan con una terminología orientada a la implementación en un esfuerzo para establecer la solución.
- Nivel inferior: La solución se establece de manera que pueda implementarse directamente (usando el lenguaje de implementación)

Sobre la figura, el nivel 0 es el *problema*, que se descompone en tres subproblemas, dos de ellos no necesitan descomposición (por tanto son del nivel inferior y se puede diseñar una solución) y el segundo al estudiarlo se define como un subproblema con dos aspectos diferenciados que se tienen que tratar en un nivel inferior de solución.

Sea por ejemplo el problema de calcular la nota media de los alumnos en la asignatura de Programación con este enfoque, pero cuidado es un problema simple que no tiene porque necesitar el uso de módulos, pero que ayudará a entender el diseño estructurado, es decir tiene que entenderse como un ejemplo didáctico.

El nivel 0 es **Calcular la nota media**. Para un alumno dado tenemos que saber la nota en cada una de las tres partes evaluables de la asignatura y si se cumplen las condiciones de mínimos se podrá acumular la nota sino se queda con el valor de la parte que no alcance el mínimo.

El nivel intermedio es la descripción del procedimiento. En este caso se utiliza en patrón *Entrada-Proceso-Salida* que hemos seguido hasta ahora. Por tanto tendremos los módulos:

- **Obtener notas**
- **Calcular notas**
- **Imprimir acta**

Obtener notas consiste en recoger los datos de cada una de las partes. Puede ser tan simple como cargarlas desde teclado, leerlas de un archivo, o bien conectarse a una base de datos (por ejemplo: de Github o del aula virtual) para obtener las calificaciones de cada parte, pudiendo conllevar tareas complejas como utilizar GitHub para consultar el número de commits y sus fechas, para obtener las notas.

Imprimir acta igual que el módulo de lectura puede ser un `printf` en pantalla o un proceso complejo de conexión con la API del sistema de gestión de datos de los alumnos para enviar los datos al libro de calificaciones.

Para este caso los programadores optan por utilizar la versión más sencilla y se leerá desde teclado y se imprimirá en pantalla, por lo que no es necesario seguir descomponiendo estos módulos de entrada/salida, siendo por tanto posible dar una solución de implementación directa.

```
float n_TE = 0;
float n_PR = 0;
float examen = 0;

printf("Cual es la nota de los trabajos evaluables (0-1): ");
scanf("%f", &n_TE);
```

```
    printf("Cual es la nota de las practicas (0-1):  ");
    scanf("%f", &n_PR);

    printf("Cual es la nota obtenida en el examen (1-10):  ");
    scanf("%f", &examen)
```

y para imprimir el acta:

```
    printf("La calificación de alumno es %.2f\n", calificacion);
```

El módulo **Calcular notas**, tiene dos tareas diferenciadas:

- Comprobar si el alumno cumple las condiciones para superar la asignatura, es decir, superar las prácticas y haber obtenido más de un cuatro en el examen.
- Realizar el cálculo propiamente dicho.

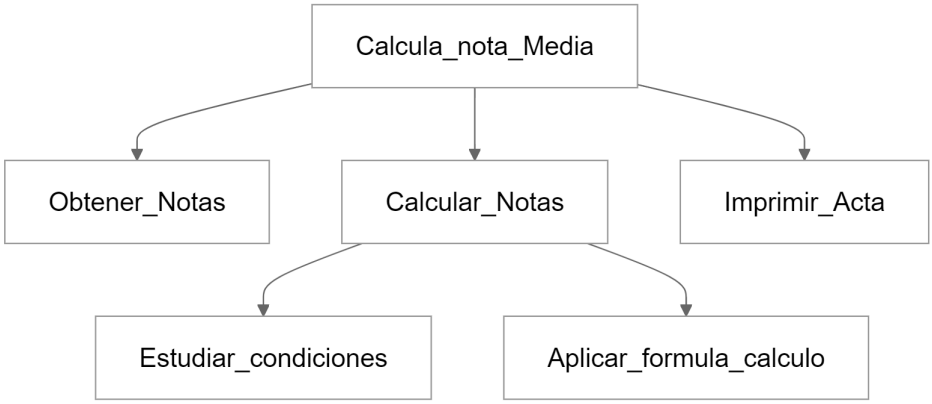
Estos dos módulos no requieren más descomposición.

Aplicar formula calculo sería:

```
calificacion= n_PR * (examen *0.8) + n_TE + n_PR;
```

y el módulo **Estudiar condiciones**

```
if(examen>=4)
  aplicar_formula_calculo;  // llamada al otro módulo
```



Ventajas del diseño modular:

1. El esfuerzo de resolver un problema complejo es menor cuando éste se descompone en piezas manejables:

P=P1+P2+P3 – E(P1+P2+P3)>=E(P1)+E(P2)+E(P3)

2. Mayor claridad lógica se facilita la legibilidad y la verificabilidad del programa.
3. Menor tiempo de desarrollo: módulos independientes desarrollo en paralelo (equipo de programadores).
4. Mayor facilidad de mantenimiento - se puede modificar o sustituir un módulo sin afectar al resto.

5. Programas más cortos y simples, debido a la división del problema complejo en partes
6. Re-utilización de los módulos:
   - Dentro del mismo programa - los módulos se escriben una sola vez pero se pueden referenciar múltiples veces, evitándose la duplicación innecesaria del código - instrucciones complejas (macro-instrucciones)
   - En otros programas:
     - del mismo programador.
     - de otros programadores - bibliotecas de módulos (componentes, piezas).

También tiene sus inconvenientes que iremos estudiando, la mayoría relativos a los mecanismos de comunicación entre los módulos creados.

En C un programa está compuesto, además de por las directivas del preprocesador, por un conjunto de bloques de código. Uno de los bloques debe ser obligatoriamente `main` que es por donde empieza la ejecución. Pero en problemas complejos podemos tener más de un bloque, siendo cada uno de ellos un **módulo** con una tarea específica. En los ejemplos que hemos visto hasta ahora sólo tenemos un bloque de código:

```
int main(void)
{
        ....
   return 0;
}
```

que devuelve 0 cuando acaba correctamente. `int` indica que el tipo del valor que devuelve la función/módulo `main` es un entero, por eso `return 0`, el contenido del módulo está definido por las llaves `{...}` que marcan el inicio y fin del módulo/bloque. Para poder ejecutarse `main` no recibe ningún parámetro `void`.

A modo de ejemplo se van a utilizar dos bloques para escribir en mensaje “hola mundo”, para lo que se definirá una función `saludar` responsable de mostrar el mensaje. Se construirán dos bloques (funciones/módulos) de código cada uno con un nombre, el modulo/función `saludar` y el modulo/función `main` entre llaves `{...}`

```
#include <stdio.h>

void saludo(void){
    printf("Hola Mundo");
}

int  main(void)
{
    saludo();
    return 0;
```

Como se puede observar, la función debe aparecer definida antes de que se utilice, es decir para que `saludar` sea comprendido por la función que lo utiliza, `main`, debe aparecer su definición antes del primer módulo que la use.

Para clarificar la lectura del código, es preferible que `main` sea la primera función de nuestro código, pero si tenemos otras funciones, estas tienen que ir antes. Para ello se utilizan los **prototipos de las funciones**, que evitan errores de compilación:

```c
#include <stdio.h>

void saludo(void);

int  main(void)
{
    saludo();
    return 0;
}

void saludo(void){
    printf("Hola Mundo");
}
```

Estamos en la situación más simple posible donde la función realiza una tarea, pero ni recibe ni devuelve ningún valor.

Sin embargo, como se verá más adelante, el potencial de un módulo aparece cuando se utilizan **parámetros** o argumentos, datos que comunican a la función con y hasta el punto de llamada. En el siguiente ejemplo se puede *parametrizar* la entrada, dependiendo de la cadena de caracteres que se pase a la función se muestra un mensaje u otro (recuérdese que las cadenas en C son colecciones de caracteres `char cadena[]`, que se estudiarán después ya que el tipo `string` no está definido en C).

```c
#include <stdio.h>

// Declaración de la función saludo
void saludo(const char mensaje[]);

int main() {

    // Llamada a la función saludo
    saludo("Hola Mundo");
    saludo(" buenos dias");

    return 0;
}

// Definición de la función saludo
void saludo(const char mensaje[]) {
    printf("%s\n", mensaje);
}
```

Veamos por ejemplo el caso de querer tener una función que nos dé la distancia de un punto en el plano al origen de coordenadas.

En este caso la función devuelve un real de doble precisión que representa el valor de distancia.

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

double distanciaAlOrigen(double x, double y);

int main() {
    double x = 3.0;
    double y = 4.0;
    double distancia = distanciaAlOrigen(x, y);
    printf("La distancia entre el punto (%.2lf, %.2lf) y el origen es: %.2lf\n", x, y, distancia);

    return 0;
}

double distanciaAlOrigen(double x, double y) {
```

```
    return sqrt(x * x + y * y);
}
```

La sintaxis genérica de cualquier función/bloque/módulo de código en C es:

```
tipo_devuelto nombre_funcion (tipo parametro_1, tipo parametro_2,... tipo parametro_n)
{
    ....
}
```

Descomposición modular

Probablemente una de las situaciones de toma de decisiones clave en el diseño de programas es cómo se definen los módulos, cuántos y por qué. Es decir cómo se realiza la descomposición modular.

La adecuada división de un programa en subprogramas constituye un aspecto fundamental en el desarrollo de cualquier código.

Desde un punto de vista práctico, la necesidad de la utilización de módulos o subprogramas aparece ligada a dos hechos específicos que se producen en programación:

- Reutilizar porciones de código ligado al uso de datos y/o variables deferentes. (Este suele ser el más fácil de entender).
- Necesidad de manejar problemas complejos, que ha de resolverse con la estrategia divide y vencerás.

Supongamos que necesitamos escribir un programa que calcule un número combinatorio $\binom{m}{n}$, que se calcula con la formula

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

donde $n!$ es el producto de los números desde 1 a n:

$$n! = \prod_{i=1}^{n} i$$

Sabemos como se puede construir un programa en C para calcular el factorial:

```
// Calcular el factorial de un número
#include <stdio.h>
void main(){
        int a, b, fact = 1;
        printf("Escribe un numero para calcular su factorial: \n");
        scanf("%d", &a);

    // Cálculo del factorial
    for (b = a; b > 1; b--){
            fact = fact * b;
    }
    // Resultado de salida
    printf("El factorial de %d es: %d\n", a, fact);
    return 0;
}
```

El programa que resuelve el número combinatorio sería:

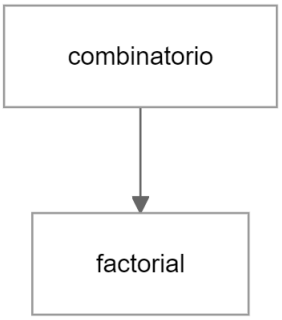
```
            printf("Introduzca numero de objetos:\n");;
            scanf(" %d",&m);
         }while(m<=0);

         do
         {
            printf("Introduzca n:\n");;
            scanf(" %d",&n);
         }while(n<=0);

         comb=1;
         fact=1;

         for (i =m; i > 1; i--)
         {
                 fact = fact * i;
         }

         comb=comb*fact;
         fact=1;

         for (i = (m-n); i > 1; i--)
         {
                 fact = fact * i;
         }
;
        comb=comb/fact;
        fact=1;
        for (i = n; i > 1; i--)
         {
                 fact = fact * i;
         }

         comb=comb/fact;

         //Escribir resultados
         printf("\nCombinaciones de %d elementos tomados de %d en %d= %.0lf",n,m,m,comb);
```

Se puede ver que los bucles `for` son iguales salvo la inicialización de la variable contador. Necesitamos un mecanismo que dado el valor de la inicialización funcione de forma parecida en todos los casos.

Necesitamos una "máquina" que calcule el factorial y nos lo ofrezca como salida, para así poder hacer algo como ese valor.

```
factorial(m)/(factorial (m-n)*factorial(n))
```

Por lo tanto, necesitamos una **función factorial**.




```
double factorial(int numero);

int main()
```

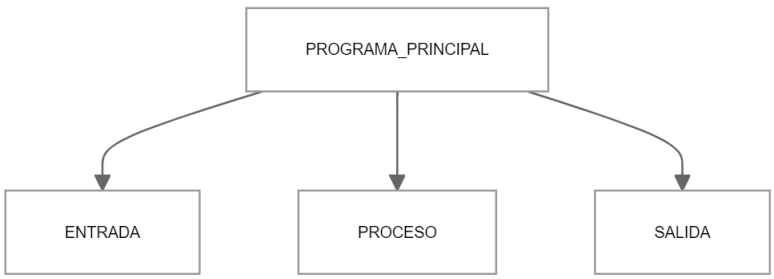
```
{
   char c;
   int n, m;
   double comb;

   do
   {

      printf("COMBINACIONES DE m OBJETOS TOMADOS DE n EN n\n");
      printf("=========================================\n\n");

      do
      {
         printf("Introduzca numero de objetos:\n");
         ;
         scanf(" %d", &m);
      } while (m <= 0);

      do
      {
         printf("Introduzca n:\n");
         ;
         scanf(" %d", &n);
      } while (n <= 0);

      comb = factorial(m) / (factorial(m - n) * factorial(n));

      // Escribir resultados
      printf("\nCombinaciones de %d elementos tomados de %d en %d= %.0lf", n, m, m, comb);

      printf("\n\nDesea efectuar una nueva operacion (s/n)? ");
      scanf(" %c", &c);

   } while ((c != 'N') && (c != 'n'));
   return 0;
}

double factorial(int numero)
{
   int fact = 1;
   for (int i = numero; i > 1; i--)
      fact = fact * i;
   return fact;
}
```

A continuación se verá un ejemplo de como utilizar las funciones y los módulos para manejar la complejidad de un problema.

Tradicionalmente la programación estructurada sigue un enfoque en tres etapas:

ENTRADA — PROCESO/CÁLCULOS — SALIDA

Lo que se traduce en un patrón de estructura modular inicial para cualquier problema:



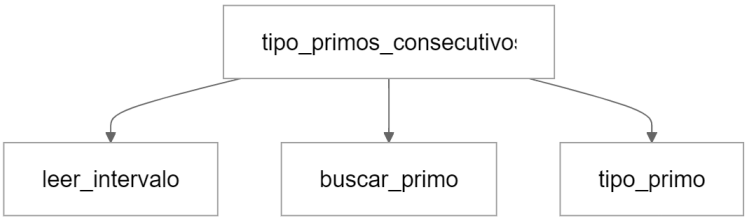
Esta estructura se refinará hasta que se definan módulos suficientemente simples y comprensibles. El objetivo principal de esta separación es restringir las operaciones de entrada y salida a los módulos correspondientes. De esta forma si cambia el mecanismo de entrada (por ejemplo de sensores o a través de una interfaz más compleja) los módulos de "cálculo" que suelen ser los más complejos no se verían alterados. No obstante estas estrategias o modelos de separación de la interfaz están fuera del alcance de la programación básica que se desarrolla en esta asignatura. El ejemplo del cálculo de la nota media es un ejemplo de este patrón.

Veamos otro ejemplo concreto:

> Dado un intervalo de números enteros identificar para todos los pares de primos la relación entre ellos, a saber:

- Primos Gemelos: Dos números primos se denominan primos gemelos si su diferencia es igual a 2, es decir, una pareja de la forma (p,p+2) siendo p un número primo. Por ejemplo las parejas (3,5) y (17,19) son dos parejas de primos gemelos.
- Primos Primos: Dos números primos se denominan primos primos (del inglés cousin prime) si su diferencia es igual a 4, es decir, una pareja de la forma (p,p+4) siendo p un número primo. Por ejemplo las parejas (3,7) y (7,11) son dos parejas de números primos primos.
- Primos Sexys: Dos números primos se denominan primos sexys (del inglés sexy prime) si su diferencia es igual a 6, es decir, una pareja de la forma (p,p+6) siendo p un número primo. Se llaman así porque la palabra latina para el número seis era sex. Por ejemplo (5,11) y (11,17) son dos parejas de números primos sexys.

Se comienza con el patrón entrada-proceso-salida



Estudiamos en que consiste cada módulo:

leer_intervalo

- Leer inferior > 0
- Leer superior > 0
- Validar inferior <= superior

buscar_primo

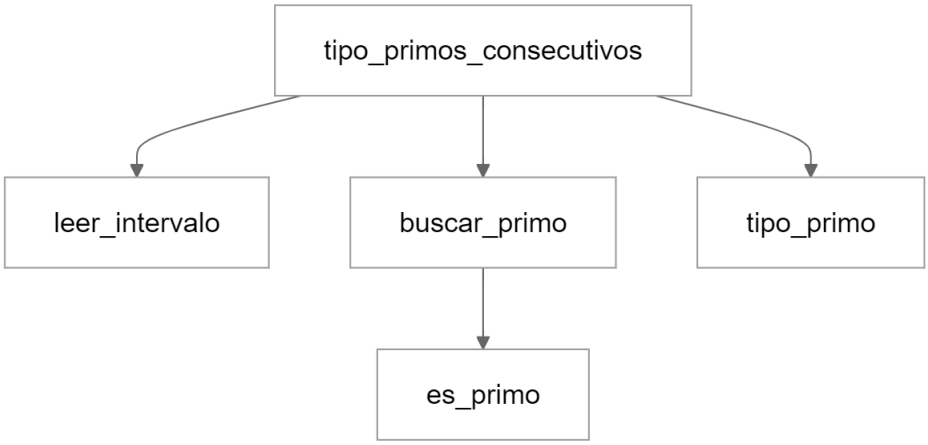
- Dado un número buscar el siguiente primo:
  - Sumar
  - ¿es primo?

tipo_primo

- Según sea diferencia escribir primo

tipo_primos_consecutivos - leer intervalo - Dentro del intervalo - buscar primer primo - buscar segundo primo - calcular diferencia ver tipo_primo

El módulo que decide si un número es primo se utiliza varias veces y buscar primo se usa en dos ocasiones.




```
int main()
{
  int inicio, fin, primo1, primo2;
  leerIntervalo(&inicio, &fin);

  primo1 = buscaPrimo(inicio);
  primo2 = primo1 + 1;

  while (primo2 <= fin)
  {
    primo2 = buscaPrimo(primo2);
    tipoPrimo(primo1, primo2);
    primo1 = primo2;
    primo2++;
  }
}

void leerIntervalo(int *inf, int *sup)
{
  do
  {
    printf("\nIntroduzca numero en para limite inferior: ");
    scanf(" %i", inf);
    printf("\nIntroduzca numero en para limite superior: ");
    scanf(" %i", sup);
  } while (*inf >= *sup || *inf <= 0 || *sup <= 0);
}

int esPrimo(int num)
{
  if (num <= 1)
  {
    return 0;
  }
  for (int i = 2; i * i <= num; i++)
  {
    if (num % i == 0)
```

```
    {
      return 0;
    }
  }
  return 1;
}

int buscaPrimo(int primo)
{
  int p = primo;
  while (esPrimo(p) == 0)
    p++;
  return p;
}

void tipoPrimo(int p1, int p2)
{
  switch (p2 - p1)
  {
  case 2:
    printf(" %i y %i son primos gemelos \n", p1, p2);
    break;
  case 4:
    printf(" %i y %i son primos primos \n", p1, p2);
    break;
  case 6:
    printf(" %i y %i son primos sexis\n", p1, p2);
    break;
  default:
    printf(" %i y %i son primos sin una relación especial \n", p1, p2);
  }
}
```

Pero que pasa si queremos no solo que estudie los primos consecutivos sino cualquier par de primos en el intervalo. Solo será preciso cambiar el módulo principal. Esta es la potencia de la descomposición modular.

```
  while (primo1<=fin){
    primo2=primo1+1;
    while (primo2<=fin){
       if (esPrimo(primo2))
          tipoPrimo(primo1,primo2);
      primo2=buscaPrimo(++primo2);
     }
     primo1=buscaPrimo(++primo1);
  }
```

Tipos de módulos

Ligados al concepto de módulo aparecen distintos términos con significados parecidos pero no exactamente iguales aunque en todos los casos identifican un conjunto de instrucciones con **propósito único e identificable** y/o proporciona unos determinados resultados, y que puede ser llamado (invocado) desde el programa principal o desde otros módulos.

Hablamos de **subrutinas**, **funciones** o **procedimientos**. A una subrutina no se le pueden enviar parámetros ("valores variables con los que ejecutar su código"), las funciones obligatoriamente devuelven un valor y los procedimientos no. No obstante debemos de considerarlos equivalentes, tal como ocurrió con los bucles. La realidad es que la sintaxis de la mayor parte de los lenguajes de programación permite que se unifique el concepto, por ejemplo en C sólo hay funciones, pero dependiendo del comportamiento que codifique

puede recoger la idea de subrutina o procedimiento, basta con no pasar parámetros y/o devolver `void`.

Las clasificaciones más utilizadas para los módulos se establecen según tres criterios:

- La actividad que desempeñan
- Según donde se implementa
- Según los valores que devuelven

Clasificación de los módulos según la actividad funcional que desempeñan:

- Módulos de adquisición (entrada) de datos.
- Módulos de cálculo/transformación de datos.
- Módulos de transmisión (salida) de datos.
- Módulos de control (integración de sub-problemas).
- Módulos combinación de los anteriores.

Clasificación de los módulos según donde se implementan:

- Módulos internos o intrínsecos al lenguaje de programación (forman parte de su sintaxis). Ej: leer por teclado, escribir en pantalla, seno, coseno,... `printf` es un módulo interno implementado en `stdio`, que requieren la incorporación de las librerías apropiadas.
- Módulos externos o definidos por el programador, bien dentro de propio código o bien en librerías propias en archivos `.h`.

Clasificación de los módulos según el número de resultados o valores que devuelven:

- Funciones: devuelven un valor especial a través del nombre del identificador del módulo. Ej: `sin(x)`.
- Procedimientos: pueden generar como salida 0, 1 o más datos, pero ninguno se vincula con el identificador del módulo.
- Subrutinas: no devuelven ni reciben valor (es una herencia del lenguaje ensamblador)

Comunicación entre módulos

Un módulo puede transferir temporalmente el control a otro (bifurcación o llamada), devolviendo este último el control al que originariamente se lo dio.

Los resultados producidos por un módulo pueden ser utilizados por cualquier otro módulo cuando se transfiere el control entre ellos.

Los mecanismos para la comunicación de información entre módulos:

- Variables globales (o externas) posiciones de memoria compartidas por los diferentes módulos del programa.
- Valor especial devuelto por una función a través de su nombre de identificador.
- Lista de parámetros o argumentos, paso de parámetros.

El uso de variables globales está desaconsejado. En C se declaran fuera de todos los bloques de código, incluido main. A continuación vamos a formalizar los conceptos de valor devuelto por una función y los parámetros (datos que necesita una función para realizar su tarea).

Si bien puede haber funciones sin parámetros, en general la utilidad de cualquier función se incrementa si se parametriza. Recoge la idea de una función matemática.

Concepto matemático: aplicación de un conjunto de datos (dominio) en otro conjunto de datos (imagen):

$f : A \to B$

$x \to y = f(x)$

Ejemplo:

$f : R^2 \to R$ (campo escalar)

Definición de la función explícita (regla de cálculo):

$f(x, y) = y/(1 + x^2)$

f: nombre de la función

x,y: argumentos (parámetros formales)

(variables independientes)

Para evaluar la función hay que dar un valor real o actual a sus argumentos:

$x = 3, y = 3 \quad \to \quad f(3, 3) = 3/(1 + 3^2) = 0{,}3$

Es el caso de las funciones internas o intrínsecas como: `sin(x)`, `sqrt(x)`,...

Las funciones se referencian dentro de las expresiones utilizando su nombre y una lista de argumentos que deben de coincidir en cantidad, tipo y orden con los de la función, en C, en otros lenguajes se relaja esta norma.

Desde el punto del código en donde se llama a la función se "envían" unos datos que son procesados y se inserta el valor de salida en el punto de llamada.

Función que recibe un argumento de entrada y devuelve un `double` como salida

```
// Un parámetro de entrada n, y un valor de salida
double factorial(int numero){
    int i;
    double fac;
    fac=1;
    i=1;
    while(i<=numero){
        fac=fac*i;
        i++;
    }
    return fac;
}
```

Pero podemos tener funciones que no devuelven nada, recibiendo o no parámetros de entrada.

```c
// no tiene parámetros y devuelve nada `void`
void saludo(){
    printf("Hola Mundo");
}
```

Pueden recibir valores de entrada y no devolver al punto de llamada ningun valor de salida.

```c
// Utiliza el valor del parámetro "mensaje" para escribirlo en la pantalla
void saludo(const char mensaje[]) {
    printf("%s\n", mensaje);
}
```

Cada función de C solo devuelve `una cosa`, un único dato. Cuando es necesario que se devuelvan dos resultados, por ejemplo una función que calcule la media y la desviación típica de un conjunto de valores leídos por teclado, se deben utilizar otras estrategias. Si no es posible separar en dos funciones, se debe hacer que se puedan devolver dos valores reales. Se puede hacer de diversas formas que se detallarán más adelante pero que introducen en este punto:

```c
// la media es devuelta por la función y se define un parámetro como de salida (se usa su dirección)
double estadistica(...., double *desviacion)

// Ambos valores son parámetros  de salida (se usa su dirección)
void estadistica(....,double *media double *desviacion)

// tipo_nuevo definido por el programador que compacta dos valores reales
tipo_nuevo estadistica(....)
```

Tipos de parámetros

Cuando se define una función se incluyen unas variables que representan los parámetros que utiliza la función. Estos son los **parámetros formales**. Son las variables , locales a la función, que se utilizan para la definición de la misma, cuando se escribe su código. Se llaman formales, porque no "existen" hasta que se usan en una llamada.

Los parámetros formales se "llenan" de contenido cuando lo reciben al realizar la llamada a la función con los llamados **parámetros reales**, que no son más que las expresiones que se utilizan en la llamada de la función, sus valores se copiarán en los parámetros formales.

Cuando hacemos una llamada a una función se relaciona el parámetro real o actual (en la llamada) con el parámetro formal (en la función). El parámetro real (en la llamada) pasa el valor al parámetro formal (parámetro de la función) y si éste se modifica en la función, el parámetro real no se ve afectado, salvo que se indique en la definición como de salida usando * aunque esto se explicará después. A este proceso se le llama **paso de parámetros** a la función. Se trasfiere el control al código de la función con los parámetros usados en la llamada.

Sobre el ejemplo del número factorial

```
double factorial(int numero)
{
   int fact = 1;
   for (int i = numero; i > 1; i--)
      fact = fact * i;
   return fact;
}
```

`numero` es el parámetro formal que usa en la codificación de la función.

Cuando se utiliza `m, n, m-n` son los parámetros reales.

```
comb = factorial(m)/(factorial(m-n)*factorial(n));
```

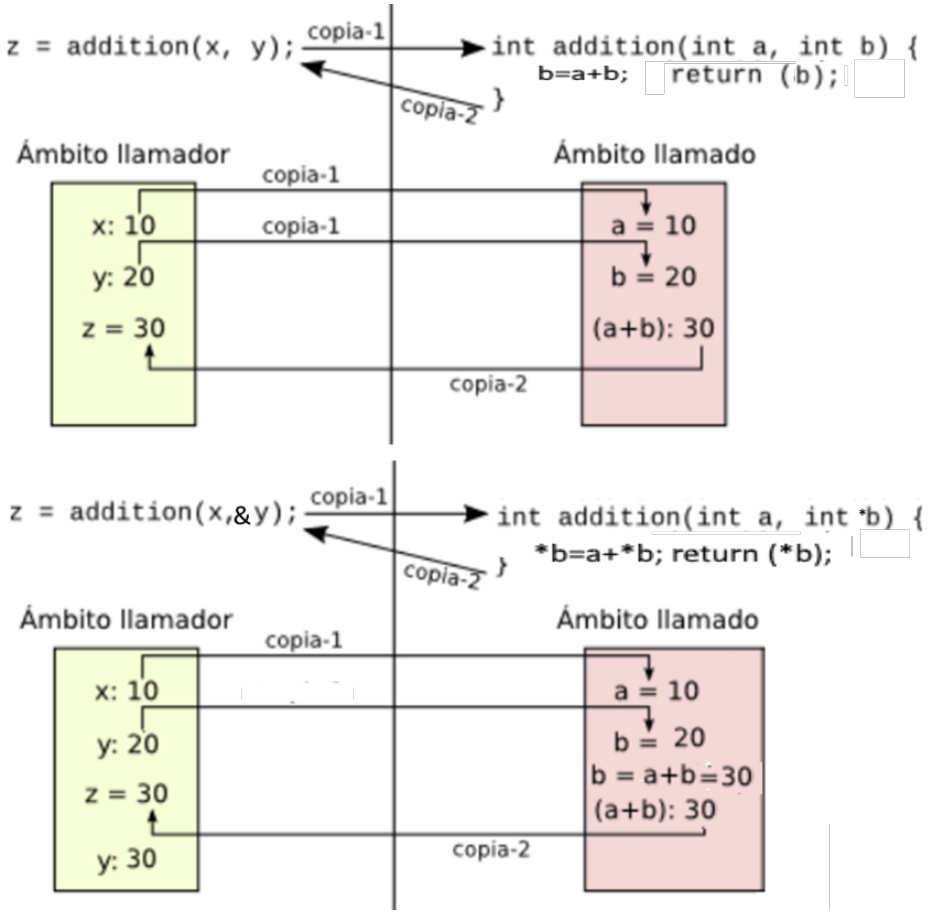
para `factorial(m)` el parámetro real es `m` y se copia su valor en `número` para realizar la operación.

Como ya se ha dicho las funciones no cambian los valores de los parámetros si son de tipos primitivos, se hace una copia, se están realizando un **paso por valor**. Es decir si dentro de la función se produce un cambio en el valor, cuando se devuelve el control punto de llamada no se mantiene el cambio sobre le parámetro real.

Pero ¿cómo funcionan los parámetros y las variables definidas en las funciones?. ¿Qué pasa si queremos que el efecto de una función sobre un parámetro perdure fuera del ámbito de una función?, o bien queremos que de una función se obtenga más de una cosa como resultado. En estos casos debemos usar **parámetros pasados por referencia** o **parámetros de salida**, es decir no se hará copia de los parámetros localmente a la función, sino que se utilizará la zona de memoria donde se almacena la función, por eso pasamos la dirección en la llamada `& *`. Como ocurre con `scanf`.

```
#include <stdio.h>

void sumar_referencia(int *numero); /* prototipo de la función */

int main(void)
{

    int numero = 57; /* definimos numero con valor de 57*/
    printf("\nValor de numero dentro de main() es: %d ", numero);
    sumar_referencia(&numero); /*enviamos número a la función*/
    return 0;

}

void sumar_referencia(int *numero)
{

    *numero += 1; /* le sumamos 1 al numero */
    printf("\nValor de numero dentro de sumar_referencia() es: %d", *numero);

}
```

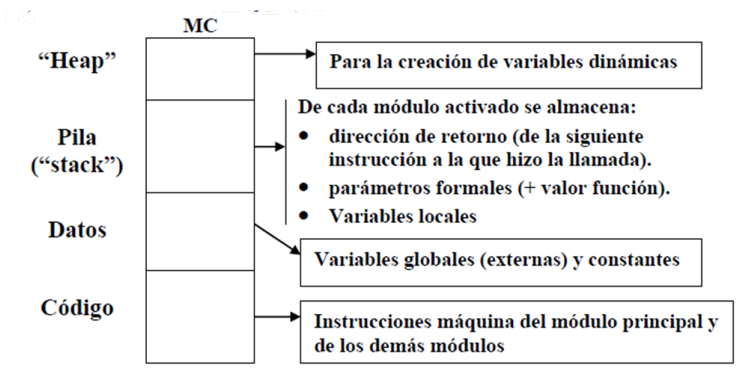
Este proceso se muestra en la figura. Aunque cada lenguaje de programación puede tener normas distintas en este aspecto y hay lenguajes en que todos los parámetros se pasan por referencia.



Paso de parámetros

- Los módulos se activan al ser llamados (invocados) desde el módulo llamador.
- Al activarse un módulo, el control de la CPU pasa al módulo, ejecutándose todas las instrucciones del mismo.
- Automáticamente se establece una correspondencia posicional entre parámetros formales y reales, deben de coincidir en cantidad, tipo y orden.
- Al finalizar la ejecución del módulo, el control de la CPU pasa a la siguiente instrucción que hizo la llamada.

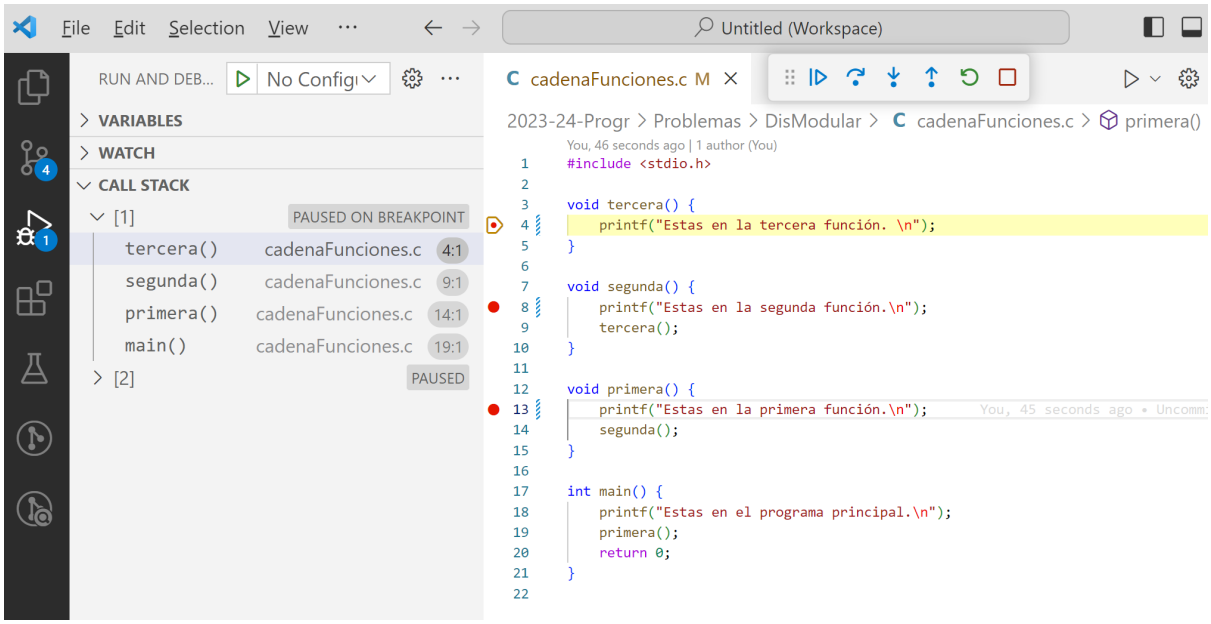
Pero ¿cómo se sabe donde y con que valores volver?. Para eso hay que recordar la estructura de la memoria de una computadora y los registros que de la CPU que intervienen.



- IR (registro de instrucción): almacena la instrucción que se está ejecutando.
- PC (contador de programa): almacena la dirección de memoria que contiene la siguiente instrucción a ejecutar.
- SP (puntero de pila): contiene la dirección de la cima de la pila de llamadas a módulos. La pila está inicialmente vacía.
- En la pila se registra el valor actual del registro PC (dirección de MC de la siguiente instrucción que hizo la llamada) y se despliegan las variables locales y formales del módulo (+posición adicional func.), incrementándose el puntero a la cima de la pila SP el valor correspondiente.
- Se modifica el registro PC, almacenándose en el mismo la dirección de memoria que contiene la primera instrucción del módulo (nombre módulo=dirección 1ª instrucción del módulo).
- Tras la ejecución de la instrucción actual, pasa a ejecutarse la primera instrucción del módulo.
- Se modifica el registro PC, almacenándose en el mismo la dirección de memoria de la siguiente instrucción a la que hizo la llamada (dicha dirección está registrada en la pila en la zona correspondiente al módulo actual).
- La pila se repliega disminuyéndose el puntero a la cima de la pila SP el valor correspondiente.
- Tras la ejecución de la instrucción actual, pasa a ejecutarse la siguiente instrucción del módulo llamador a la que hizo la llamada.

La mejor manera de ver como funciona este proceso es utilizar el depurador par ver la evolución de las variables sobre un ejemplo.

```c
#include <stdio.h>

void tercera() {
    printf("Estas en la tercera función. \n");
}

void segunda() {
    printf("Estas en la segunda función.\n");
    tercera();
}

void primera() {
    printf("Estas en la primera función.\n");
    segunda();
}

int main() {
    printf("Estas en el programa principal.\n");
    primera();
    return 0;
}
```



A continuación sobre programas muy similares se puede observar la diferencia entre el **paso por valor** (entrada) y el **paso por referencia** (entrada/salida). No obstante estamos estudiado el comportamiento de las funciones en C, cada lenguaje puede adaptar el paso de parámetros de forma diferente.

```c
#include <stdio.h>

void tercera(int contador) {
    contador++;
    printf("Estas en la tercera función. Con valor de %d\n", contador);
}

void segunda(int contador) {
    contador++;
    printf("Estás en la segunda función. Con valor de %d\n", contador);
    tercera(contador);
    printf("Estás en la segunda función después de llamar a tercera. Contador: %d\n", contador);
}

int primera() {
    int contador=1;
    segunda(contador);
    printf("Estás en la primera función. Con valor de %d\n", contador);
    return(contador);
}

int main() {
    printf("Estás en el programa principal. Resultado de llamada %d \n",primera());
    return 0;
}
```

Cuyo resultado en pantalla es

```
Estás en la segunda función. Con valor de 2
Estás en la tercera función. Con valor de 3
Estás en la segunda función después de llamar a tercera. Contador: 2
Estás en la primera función. Con valor de 1
Estás en el programa principal. Resultado de llamada 1
```

```c
#include <stdio.h>

void tercera(int *contador) {
    (*contador)++;
    printf("Estás en la tercera función. Contador: %d\n", *contador);
}

void segunda(int *contador) {
    (*contador)++;
    printf("Estás en la segunda función. Contador: %d\n", *contador);
    tercera(contador);
    printf("Estás en la segunda función después de llamar a tercera. Contador: %d\n", *contador);
}

int primera() {
    int contador = 1;
    printf("Estás en la primera función. Contador: %d\n", contador);
    segunda(&contador);
    return(contador);
}

int main() {
    printf("Estas en el programa principal. Resultado de llamada %d \n",primera());
    return 0;
}
```

Cuyo resultado en pantalla es

```
Estás en la primera función. Contador: 1
Estás en la segunda función. Contador: 2
Estás en la tercera función. Contador: 3
Estás en la segunda función después de llamar a tercera. Contador: 3
Estas en el programa principal. Resultado de llamada 3
```

PROBLEMA

Construir una función en C que dados tres enteros positivos, determine si forman un triángulo válido y con estas longitudes de lados, en su caso devuelva el área y del triángulo aplicando el semiperímetro.

Diseño

Podemos establecer diversas formas de comunicación entre módulos y también distintas aproximaciones al reparto de responsabilidades entre módulos.

Optamos por hacer que la función sea responsable de estudiar las propiedades del triángulo, responsabilizándose de la comprobación de la desigualdad de Herón, o bien hacer que sea la función la que compruebe la validez de las longitudes, con lo que la función haría más cosas. Pero supondrá que se le ofrecen datos válidos para el cálculo, es decir la comprobación de si los lados leídos son positivos radicará en el programa que llama a la función, es decir `main`.

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

double calcularAreaTriangulo(double a, double b, double c);
```

```c
int main()
{
    char c;
    double l1, l2, l3; /* longitudes de los lados */
    double area;       /* área del triángulo      */

    do
    {
        printf("CÁLCULO DEL ÁREA DE UN TRIÁNGULO\n");
        printf("================================\n\n");

        do{
            printf("Introducir longitudes de lados:\n");
            printf("\tl1: ");
            scanf(" %lf", &l1);
            printf("\tl2: ");
            scanf(" %lf", &l2);
            printf("\tl3: ");
            scanf(" %lf", &l3);

        } while ((l1 <= 0) || (l2 <= 0 ) || (l3<=0));


     // Calcular el área del triángulo
        area = calcularAreaTriangulo(l1, l2, l3);

    // Imprimir el resultado
        if (area >= 0) {
            printf("El área del triángulo es: %.2lf\n", area);
        }
        else
             printf("Los lados no forman un triángulo válido.\n");

        printf("\n\n¿Desea efectuar una nueva operación (s/n)? ");
        scanf(" %c", &c);
    } while ((c != 'N') && (c != 'n'));
    return 0;
}

double calcularAreaTriangulo(double a, double b, double c) {
    double semiperimetro = (a + b + c) / 2.0;
    double area;

    // Verificar si los lados forman un triángulo válido
    if (a <= 0 || b <= 0 || c <= 0 || a + b <= c || a + c <= b || b + c <= a) {
           area =-1; // Retornar un valor negativo para indicar error
    }
    else
        area = sqrt(semiperimetro * (semiperimetro - a) * (semiperimetro - b) * (semiperimetro - c));
    return area;
}
```

En este punto se define el concepto de **puntero** y **variable puntero**. En C, una variable puntero es una variable que almacena la dirección de memoria de otra variable. En otras palabras, en lugar de contener un valor directamente, contiene la dirección de memoria donde se encuentra ese valor en la computadora. Los punteros en C son especialmente útiles para manipular estructuras de datos dinámicas, como vectores, matrices, listas enlazadas y estructuras.

Por ejemplo, supongamos que tenemos una variable x que contiene el valor 10. Podemos crear un puntero que apunte a x de la siguiente manera:

```c
int x = 10;
int *ptr;
ptr = &x;
```

Aquí, `int *ptr;` declara un puntero llamado `ptr` que apunta a un entero. Luego, `ptr = &x;` asigna la dirección de memoria de `x` al puntero `ptr`.

Una vez que tenemos un puntero apuntando a una variable, podemos acceder al valor de la variable a través del puntero utilizando el operador de indirección `*`:

```
printf("El valor de x es: %d\n", *ptr); // Imprimirá: El valor de x es: 10
```

También podemos modificar el valor de la variable a través del puntero:

```
*ptr = 20; // Cambia el valor de x a 20
printf("El nuevo valor de x es: %d\n", x); // Imprimirá: El nuevo valor de x es: 20
```

Los punteros en C son una característica poderosa, pero también pueden ser propensos a errores si no se utilizan correctamente, ya que permiten un acceso directo a la memoria.

Ámbito de las variables: variables globales y locales. Efectos laterales

El ámbito o visibilidad de una variable (u identificador), es la parte del programa en que se define una variable y es accesible su valor:

- Variables locales su ámbito es módulo donde se definen.
- Variables globales su ámbito es valido para todos los módulos del programa.

Las funciones permiten al programador modularizar un programa. Todas las variables declaradas en las definiciones de función son variables locales que son conocidas solo en la función en la cual están definidas (ninguna otra función tiene acceso a ellas).

La mayor parte de las funciones tienen una lista de parámetros. Los parámetros de una función también son variables locales. Las variables locales comienzan su existencia cuando la función es llamada y desaparecen cuando la función termina su ejecución y no retienen sus valores de una llamada a otra. Recordemos que `main` es una función.

Es posible definir variables que son externas a todas las funciones, esto es *variables globales* que pueden ser accedidas por cualquier función. Pueden ser utilizadas en el lugar de las listas de parámetros para comunicar información entre funciones. Las variables externas existen permanentemente y retienen su valores aun después de que las funciones que las utilicen han terminado su ejecución. Las variables externas deben ser definidas exactamente una vez, fuera de todas las funciones, asignándoles almacenamiento.

Variable local:

- Variable que está declarada y definida dentro de un módulo.
- Es distinta de las variables con el mismo nombre declaradas en cualquier otra parte del programa.
- Su significado se confina al módulo en que se declara y solo tiene sentido al activarse el módulo.
- Su valor solo es accesible en el módulo en que se define (no es accesible directamente por parte de otros módulos).

Variable global:

- Es declarada para todos los módulos del programa.
- Su valor es accesible desde cualquier módulo del programa.
- Permite compartir información entre diferentes módulos sin una entrada correspondiente en la lista de parámetros.

Sin embargo el uso de variables globales tiene **efectos colaterales**.

Modificaciones que realiza un módulo de elementos situados fuera del mismo y que no son parte de su interfaz.

- Problemas: propagación de errores en el programa aparece un error en una parte del programa pero la causa del error se encuentra en otro sitio distinto del programa (se dificulta la depuración del programa).
- Regla: no usar (en general) variables globales. Las únicas variables que maneja un módulo están declaradas o bien localmente o bien en su interfaz, con esto se mejora la reutilización.

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

int varGlob = 9;

void ejemplo() {
    int varLoc = 5;
    printf("varLoc es una variable local y vale %d\n", varLoc);
    printf("varGlob*varLoc vale %d\n", varGlob*varLoc);
}

int main() {
    ejemplo();
    printf("varGlob es una variable global y vale %d\n", varGlob);
    //Genera error printf("varLoc es una variable local y vale%d\n", i);
    return 0;
}
```

Cada variable mantiene el valor según el alcance/bloque en el que está definida

```c
/* Un ejemplo de alcance */
#include <stdio.h>
void a(void);
void b(void);
void c(void);

int x = 1; /* variable global */
int main ()
{
    int x=5;
    printf("x local en alcance exterior de main es %d\n",x);
    { /* nuevo alcance */
        int x=7;
        printf("x local en alcance interior de main es %d\n",x);
    }

    printf("x local en alcance exterior de main es %d\n",x);
    a();
    b();
    c();
    a();
    b();
    c();
    printf("x local en main es %d\n",x);
return 0;
}
```

```
void a(void)
{
    int x = 25;
    printf("\nx local en a es %d al  entrar a a\n",x);
    ++x;
    printf("x local en a es %d antes de salir de a\n",x);
}

void b(void)
{
    static int x=50;
    printf("\nx local static en b es %d al entrar a b\n",x);
    ++x;
    printf("x local static en b es %d antes de salir de b\n",x);
}
void c(void)
{
    printf("\n x global es %d al entrar a c\n",x);
    x*=10;
    printf("x global es %d antes de salir de c\n",x);
}
```

Ejercicio

Supongamos que tenemos definido en un programa C una variable a de tipo real (simple precisión) y otra p de tipo puntero a real. Indicar el valor de las expresiones indicadas, suponiendo que se han ejecutado previamente las dos siguientes instrucciones:

```
float a, *p;
a=-4.55;
p=&a;
```

Indicar el resultado de las siguientes expresiones

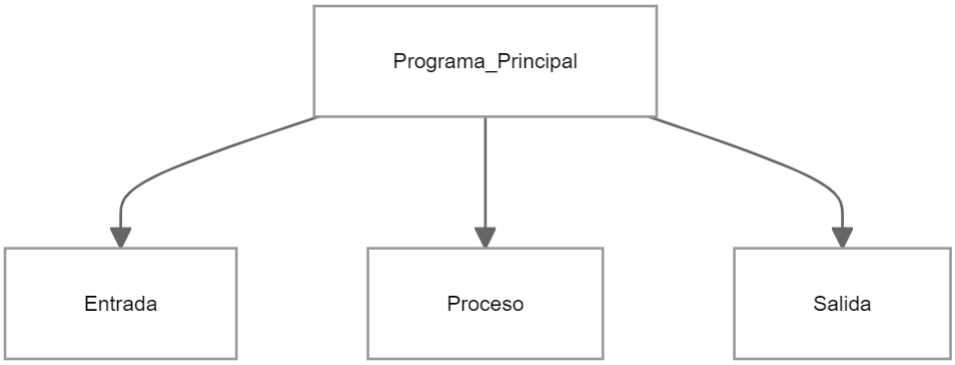
Expr						
a	p	p	&a	&*p	&p	*a

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

int main(){

   float a, *p;
   a=-4.55;
   p=&a;

   printf(" a, %f\n",a);
   printf(" p, %p\n",p);
   printf(" *p, %f\n",*p);
   printf(" &a, %p\n",&a);
   printf(" &*p, %p\n",&*p);
   printf(" &p, %p\n",&p);
//   printf(" *a,%p\n",*a);

        return 0;
}
```

PROBLEMA

Escribir un programa en C que imprima en pantalla los números primos comprendidos en un intervalo

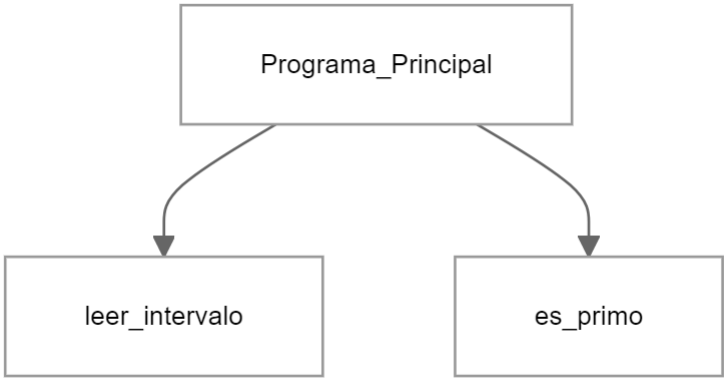
Diseño Patrón E-P-S



Entrada consiste en validar que se trata de un intervalo correcto de enteros

Proceso Bucle que itera desde el `limiteInferior` hasta `limiteSuperior` y comprueba si el número leido `es_primo`. Este proceso se colocará dentro de `main`.

Salida En este caso decidimos no tener un módulo separado, porque se imprimirá en número si es primo conforme se recorre el intervalo



Implementación

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

int es_primo(int num);
void leer_intervalo(int *l1, int *l2);


int main(){
   int limiteInferior,limiteSuperior,i,flag;

   leer_intervalo(&limiteInferior,&limiteSuperior);

   printf("Loa numeros primos entre  %d y %d son: ", limiteInferior, limiteSuperior);

   for(i=limiteInferior;i<= limiteSuperior;++i)
   {
      flag=es_primo(i);
      if(flag==0)
         printf("%d ",i);
   }
   return 0;
}
```

```
int es_primo(int num) {
   int j,flag=0;
   for(j=2;j<=num/2;++j){
        if(num%j==0){
            flag=1;
            break;
        }
   }
   return flag;
}


void leer_intervalo(int *l1, int *l2)
{
   int aux;
   do{
      printf("Introducir los numeros del intervalo: ");
      scanf("%d %d",l1, l2);
   }while (*l1<0 || *l2<0);
   if (*l2<*l1){
    aux=*l1;
    *l1=*l2;
    *l2=aux;
   }
}
```

Como vemos el patrón E-P-S para problemas simples se puede simplificar, dejando en `main` la parte del *proceso*, la *entrada* o la *salida* para minimizar la comunicación y el paso de mensajes (incremento de la velocidad).

PROBLEMA

Construir un programa que calcule e imprima en pantalla el resultado de la siguiente expresión, donde los componentes de los vectores A, B y C se introducen por teclado:

D = A ^ (B ^ C)

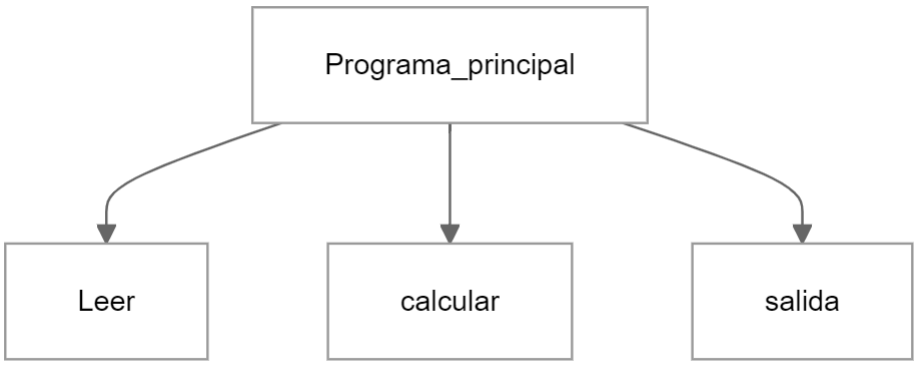
(^: producto vectorial)

Análisis

- Tenemos que representar los vectores en el espacio, cada uno tendrá tres componentes: x,y,z
- Como entrada tendremos pues tres ternas de reales.
- Como salida tres reales que representarán las componentes x,y,z resultado de la operación
- El producto vectorial de dos vectores M y N es:
    - $resultado_x = M_y N_z - M_z N_y$
    - $resultado_y = M_z N_x - M_x N_z$
    - $resultado_z = M_x N_y - M_y * N_x$

Diseño

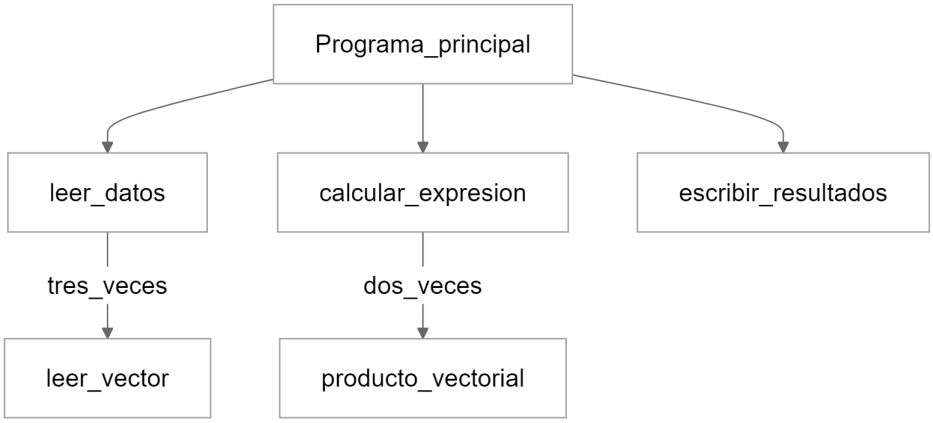
Se empieza con el patrón E-P-S



Leer tiene que leer los tres vectores, por lo que requiere de un módulo **leer_vector**

Calcular tiene que aplicar el producto vectorial dos veces con lo que se requiere **producto_vetorial**

La parte de mostrar los resultados implica escribir los datos del vector resultado.



Implementación

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

void leerDatos(float *ax, float *ay, float *az,
               float *bx, float *by, float *bz,
               float *cx, float *cy, float *cz);
void leerVector(float *vx, float *vy, float *vz);
void calcularExpresion(float ax, float ay, float az,
               float bx, float by, float bz,
               float cx, float cy, float cz,
               float *dx, float *dy, float *dz);
void prodVectorial(float ax, float ay, float az,
               float bx, float by, float bz,
               float *cx, float *cy, float *cz);
void escribirResultados(float vx, float vy, float vz);

int main(){
        char c;
        float ax,ay,az;
        float bx,by,bz;
        float cx,cy,cz;
        float dx,dy,dz;

        do{
```

```
                printf("CALCULO EXPRESION D=A^(B^C)\n");
                printf("==========================\n\n");
                leerDatos(&ax,&ay,&az,&bx,&by,&bz,&cx,&cy,&cz);
                calcularExpresion(ax,ay,az,bx,by,bz,cx,cy,cz,&dx,&dy,&dz);
                escribirResultados(dx,dy,dz);
                printf("\n\nDesea efectuar una nueva operacion (s/n)? ");
                scanf(" %c",&c);
        }while ((c!='N') && (c!='n'));
        return 0;
}

void leerDatos(float *ax, float *ay, float *az,
                float *bx, float *by, float *bz,
                float *cx, float *cy, float *cz)
{
        printf("\nIntroduzca vector A:\n");
        leerVector(ax,ay,az);
        printf("\nIntroduzca vector B:\n");
        leerVector(bx,by,bz);
        printf("\nIntroduzca vector C:\n");
        leerVector(cx,cy,cz);
}

void leerVector(float *vx, float *vy, float *vz)
{
        printf("\tx: ");
        scanf(" %f", vx);
        printf("\ty: ");
        scanf(" %f", vy);
        printf("\tz: ");
        scanf(" %f", vz);
}

void calcularExpresion(float ax, float ay, float az,
                float bx, float by, float bz,
                float cx, float cy, float cz,
                float *dx, float *dy, float *dz)
{
        float ex,ey,ez;

        prodVectorial(bx,by,bz,cx,cy,cz,&ex,&ey,&ez);
        prodVectorial(ax,ay,az,ex,ey,ez,dx,dy,dz);
}

void prodVectorial(float ax, float ay, float az,
                float bx, float by, float bz,
                float *cx, float *cy, float *cz)
{
        *cx=ay*bz-az*by;
        *cy=az*bx-ax*bz;
        *cz=ax*by-ay*bx;
}

void escribirResultados(float vx, float vy, float vz)
{
        printf("\nComponentes del vector D:\n");
        printf("\tx= %.1f\n",vx);
        printf("\ty= %.1f\n",vy);
        printf("\tz= %.1f\n",vz);
}
```

Plantear como ejemplo tener también una función **escribirVector**. Ventajas de la separación del proceso y la interfaz, por ejemplo si la lectura de datos se produce desde sensores o un archivo. En este caso como sólo se ha usado para un vector no supone una ventaja.

Recursividad

Un **concepto** se dice **recursivo** si se define en versiones más pequeñas de si mismo. Por ejemplo

$n! = n * (n-1)!$

En la defínción se utiliza el término definido.

o para la potencia de un número $x^n = x * x^{(n-1)}$

Pero estas definiciones no están completas, porque se necesita saber cuando se "para", es decir cuando detiene el procesos o la también llamas *condición de parada*. Falta conocer el casos base que indican cuando acaba la recursividad.

$$1 \quad \text{si} \quad n = 0$$

$$n * (n-1)! \quad \text{si} \quad n > 0$$

La **recursividad** en términos de diseño de programas es:

- Método alternativo a la repetición
- Estrategia: resolver el mismo problema repetidamente pero sobre un conjunto de datos más pequeño (divide y vencerás) hasta que se satisfaga una condición de terminación
- Módulo recursivo incluye una o más instrucciones de llamada a sí mismo (lo definido forma parte de la definición), es decir una **función es recursiva** si realiza una o más llamadas a sí misma.

Ventajas:

- Solución sencilla a problemas complejos (siempre que el problema admita un planteamiento recursivo que incluya una condición de parada)

Inconvenientes:

- Solución no eficiente (en general) mayor tiempo de ejecución (a veces se repiten cálculos innecesarios) y mayor asignación de espacio en el segmento de pila. Si los datos de partida son grandes se puede desbordar.
- No todos los lenguajes de programación soportan la recursividad

PROBLEMA

Módulo que calcula y devuelve el factorial de un número entero positivo. Nota: n!=n*(n-1)*(n-2)...2*1=

Análisis

- Nombre del módulo: Factorial
- Actividad funcional: cálculo/transformación
- Interfaz:
  - Entrada : n (entero, >= 0)
  - Salida: factorial de n (entero)

Diseño del módulo (sub-algoritmo):

Definición recursiva del factorial de un número:

```
n! = n * (n-1)! n>0  (caso general o recursivo)

n! = 1  n=0  (caso base - condición de salida)
```

Implementación

```c
#include <stdio.h>

// Declaración de la función factorial
int factorial(int n);

int main() {
    int numero;

    // Solicitar al usuario que ingrese un número
    printf("Ingresa un número para calcular su factorial: ");
    scanf("%d", &numero);

    // Llamar a la función factorial y mostrar el resultado
    printf("El factorial de %d es: %d\n", numero, factorial(numero));

    return 0;
}

// Definición de la función factorial
int factorial(int n) {
    // Caso base: factorial de 0 es 1
    if (n == 0 || n == 1) {
        return 1;
    } else {
        // Caso recursivo: n! = n * (n-1)!
        return n * factorial(n - 1);
    }
}
```

PROBLEMA Crear una función recursiva para calcular el termino **n** de la sucesión de Fibonacci.

Diseño

Definición recursiva:

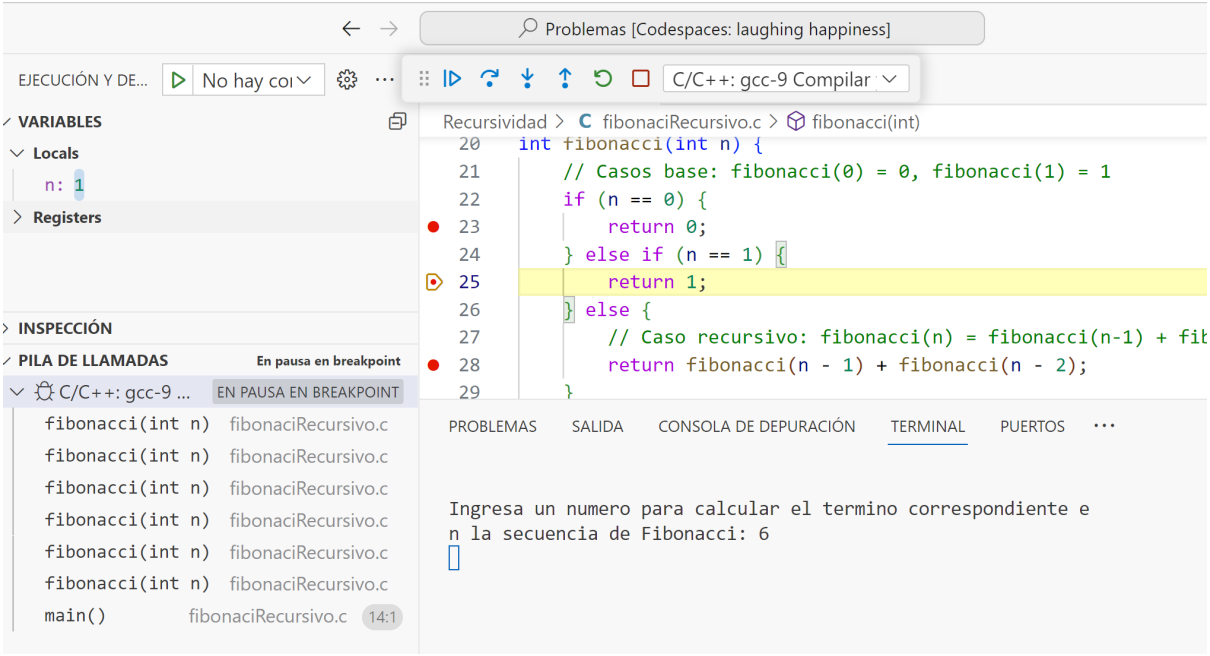
```
Fib(n)=1   n=1,2
Fib(n)=Fib(n-1)+Fib(n-2) n>2
```

Implementación

```c
int fibonacci(int n) {
    // Casos base: fibonacci(0) = 0, fibonacci(1) = 1
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        // Caso recursivo: fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Al igual que con la función factorial, la función Fibonacci se llama recursivamente hasta que alcanza los casos base (cuando n es 0 o 1), momento en el cual retorna 0 o 1, respectivamente. En otros casos, la función se llama a sí misma con los argumentos n-1 y n-2 y suma los resultados. Es importante tener en cuenta que este enfoque recursivo puede volverse ineficiente para valores grandes de n debido a la redundancia de cálculos.

En este caso vemos el gran inconveniente de las soluciones recursivas, que si una vez conocidas son más lógicas para el pensamiento humano, pueden saturar la pila.



PROBLEMA

Módulo que calcula y devuelve la potencia entera positiva de un número real positivo. Nota: $X^n = X * X * ... * X$

Diseño

Definición recursiva:

$$X^n = X * X^{n-1} \quad n > 1 \; X^n = X \quad n = 1$$

```
int potencia(int x, int n) {
    // Caso base: x^0 = 1
    if (n == 0) {
        return 1;
    } else {
        // Caso recursivo: x^n = x * x^(n-1)
        return x * potencia(x, n - 1);
    }
}
```

Muchas de las soluciones iterativas a un problema se pueden diseñar también de forma recursiva. Por ejemplo el máximo común divisor de dos números o contar las cifras de un número natural dado.

PROBLEMA Construya un módulo recursivo que devuelva el resultado de la multiplicación de dos números naturales mediante duplicación y mediación (método del campesino ruso). El método consiste en lo siguiente:

- Se escriben los dos números a multiplicar A y B en la parte superior de sendas columnas.
- Se divide A entre 2 sucesivamente, ignorando el resto, hasta llegar a la unidad, escribiendo los resultados en la columna A.
- Se multiplica B por 2 tantas veces como se ha dividido A entre 2, escribiendo los resultados sucesivos en la columna B.
- Se suman todos los números de la columna B que estén al lado de un número impar de la columna A, y ése es el resultado.

A	B	Sumandos	Resultado
27	82	82	
13	164	164	
6	328		
3	656	656	
1	1312	1312	
			2214

Análisis

Entrada:

```
 a (entero, >0)
 b (entero, >0)
```

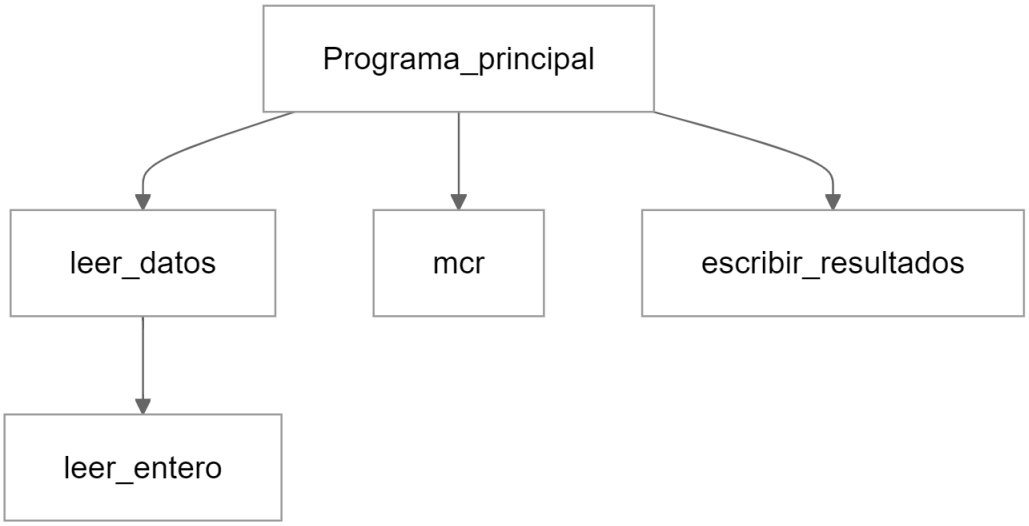
Salida:

```
producto a*b (entero)
```

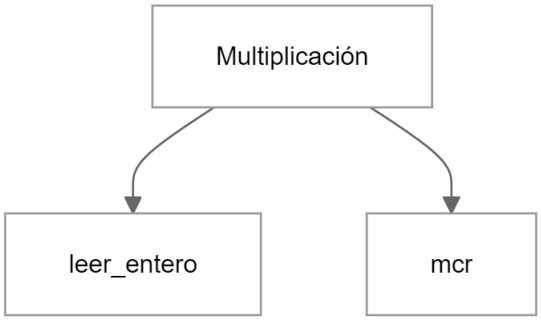
Diseño recursivo

```
a>1  (caso general o recursivo):
    mcr(a,b)= (a DIV 2)*(2*b)    si a par
    mcr(a,b)= (a DIV 2)*(2*b)+b   si a impar
a=1  (caso base condición de salida):
    mcr(a,b) = b
```

Diseño modular Para pensar en el problema completo debemos encajar el módulo recursivo `mcr` en un programa completo empezando por el patrón E-P-S



Versión simplificada de la arquitectura



Implementación

```
                printf("MéTODO DE MULTIPLICACIÒN DE NATURALES DEL CAMPESINO RUSO\n");
                printf("=======================================================\n\n");

                a = leerEntero();
                b = leerEntero();

                printf("\nEl producto vale: %d", mcr(a, b));

                printf("\n\nDesea efectuar una nueva operación (s/n)? ");
                scanf(" %c", &c);
        } while ((c != 'N') && (c != 'n'));
        return 0;
}

int leerEntero()
{
        int num;
        do
        {
                printf("Introduzca un numero: ");
                scanf(" %d", &num);
        } while (num <= 0);
        return (num);
```

```
}

int mcr(int a, int b)
{
        if (a == 1)
                return (b);
        else if (a % 2)
                return (b + mcr(a / 2, 2 * b));
        else
                return (mcr(a / 2, 2 * b));
}
```

Módulos como parámetros de otros módulos

Los punteros son uno de los aspectos más potentes de la programación en C, pero también uno de los más complejos de dominar. Los punteros permiten manipular la memoria del ordenador de forma eficiente. Dos conceptos son fundamentales para comprender el funcionamiento de los punteros:

- El tamaño de todas variables y su posición en memoria.
- Todo dato está almacenado a partir de una dirección de memoria. Esta dirección puede ser obtenida y manipulada también como un dato más.

Los punteros son también una de las fuentes de errores más frecuente. Dado que se manipula la memoria directamente y el compilador apenas realiza comprobaciones de tipos, el diseño de programas con punteros requiere una meticulosidad muy elevada que debe ir acompañada de una dosis idéntica de paciencia. Programar eficientemente usando punteros se adquiere tras escribir muchas líneas de código pero requiere una práctica sostenida que esta fuera del alcance de una asignatura inicial de programación.

No obstante, ya hemos utilizado los punteros de forma efectiva cuando al pasar parámetros por referencia a las funciones, porque como ya se ha indicado anteriormente, para que el efecto sobre una variable perdure fuera de la función se tiene que utilizar su posición en memoria, no su identificador. También como se tratará más adelante, tipos de datos compuestos/estructurados como los vectores o matrices también se tratan como direcciones.

Por la misma razón, un puntero puede apuntar a una función, es decir a un trozo de código en memoria. Esta propiedad permite generalizar aun más los diseños separando los algoritmos de cálculo (integración numérica, derivación numérica, cálculo de ceros, cálculo de máximos y mínimos, resolución de ecuaciones diferenciales ordinarias,...) de las funciones sobre las que se aplica el algoritmo. El método de cálculo es independiente de la función a la que se aplica.

Punteros a funciones

Un puntero a función es una variable que almacena la dirección de una función. Esta función puede ser llamada más tarde, a través del puntero. Este tipo de construcción es útil pues encapsula comportamiento, que puede ser llamado a través de un puntero.

Veamos cómo funciona mediante un ejemplo sencillo que crea un puntero a una función de imprimir y lo invoca:

```
#include <stdio.h>
void imprime()
{
  printf("Imprimiendo un mensaje\n");
}
int  main()
{
  void (*ptr_func)(void)=imprime;
  ptr_func(); //Llama a imprime
  return 0;
}
```

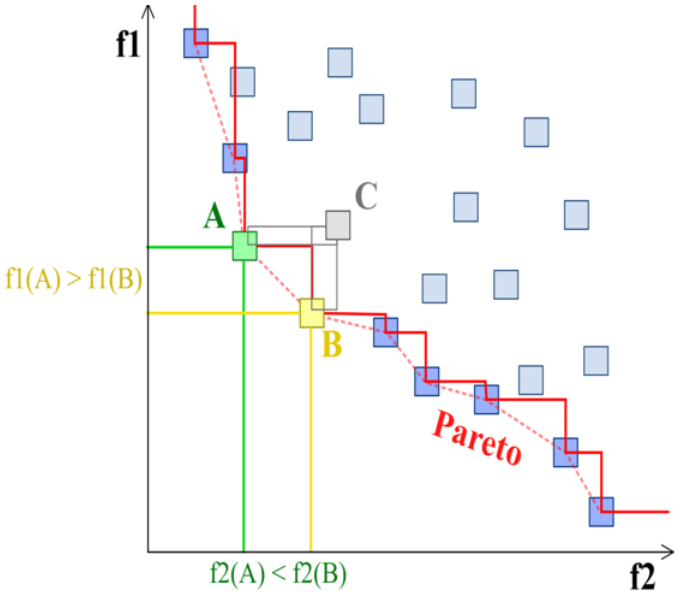
Observando este trozo de código no se ve que suponga un beneficio el uso de un puntero a función puesto que usamos dos líneas de código para lo que podríamos haber hecho en una sola línea `imprime();`. Pero la potencia radica en que `ptr_func()`, puede cambiar en tiempo de ejecución, esta ligado a un valor modificable.

PROBLEMA

Veamos un ejemplo en donde queremos una función `comparar` para dos enteros que solo durante la ejecución sabremos si es necesario obtener el máximo o el mínimo de dos números. Esta situación puede aparecer en problemas de optimización cuando a priori no se sabe si los criterios han de maximizarse o minimizarse, pero el método para saber si una solución domina a otra es genérico.

La definición de dominancia de Pareto (asumiendo minimización) para dos vectores de decisión x, y F (F se refiere a la región factible). En otras palabras, un vector domina a otro, cuando éste es menor o igual para todos los componentes y es estrictamente menor en al menos uno de ellos.

Pero también puede ser maximización con lo que en ese caso la dominancia depende del tipo de función de optimización aplicada.



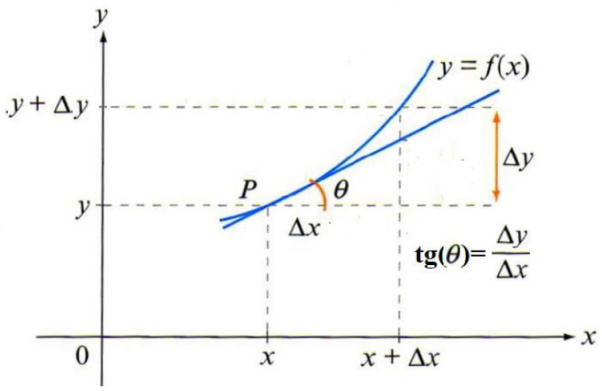
```c
#include <stdio.h>

// Prototipo de la función que toma dos enteros y un puntero a función
int operacion(int a, int b, int (*comparar)(int, int));


int maximo(int a, int b) {
    return (a > b) ? a : b;
}

int minimo(int a, int b) {
    return (a < b) ? a : b;
}
int dividir(int a, int b) {
    return (a < b) ? a : b;
}


int main() {
    int num1 = 10;
    int num2 = 20;

    // Utilizar la función para encontrar el máximo
    int resultado_max = operacion(num1, num2, maximo);
    printf("El máximo entre %d y %d es: %d\n", num1, num2, resultado_max);

    // Utilizar la función para encontrar el mínimo
    int resultado_min = operacion(num1, num2, dividir);
    printf("El mínimo entre %d y %d es: %d\n", num1, num2, resultado_min);

    return 0;
}


int operacion(int a, int b, int (*comparar)(int, int)) {
    return comparar(a, b);
}
```

PROBLEMA

Veamos un problema más complejo. Construir una función (por tanto reutilizable) que calcule el valor numérico de la derivada de una función f(x) **cualquiera** en un punto x_0 y con una precisión (h), representado h el incremento en torno al valor de x_0. El método es genérico independientemente de la función.

Análisis

La derivación numérica es una técnica de análisis numérico para calcular una aproximación a la derivada de una función en un punto utilizando los valores y propiedades de la misma.



$$\left.\frac{df(x)}{dx}\right|_{x=x_0} = \lim_{h\to 0}\frac{f(x_0+h)-f(x_0)}{h}$$

$$\left.\frac{df(x)}{dx}\right|_{x=x_0} = \lim_{h\to 0}\frac{f(x_0)-f(x_0-h)}{h}$$

$$\left.\frac{df(x)}{dx}\right|_{x=x_0} = \lim_{h\to 0}\frac{f(x_0+h/2)-f(x_0+h/2)}{h}$$

Por definición la derivada de una función f(x)

$f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$

Las aproximaciones numéricas que podamos hacer (para h > 0) serán

Diferencias hacia adelante:

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0)}{h}$$

Diferencias hacia atrás:

$$f'(x_0) \approx \frac{f(x_0) - f(x_0-h)}{h}$$

La aproximación de la derivada por este método entrega resultados aceptables con un determinado error. Para minimizar los errores se estima que el promedio de ambas entrega la mejor aproximación numérica al problema dado:

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0-h)}{2h}$$

Se puede incrementar la precisión utilizando más puntos, por ejemplo

$$f'(x_0) \approx \frac{f(x_0-2h) - 8 * f(x_0-h) + 8 * f(x_0+h) - f(x_0+2h)}{12h}$$

Para probar la función

Programa sencillo que calcula la derivada primera de las siguientes funciones para un punto introducido previamente por teclado y con precisiones de cálculo de 10^{-1} a 10^{-6}:

$f_1(x) = x^3 - 3 * x^2 + 5$

$f_2(x) = exp(-x^2)$

$f_3(x) = seno(x) * exp(-x)$

Diseño

Definido y conocido el método el problema es como establecer que $f(x)$ sea un parámetro de entrada a la función de derivación, en cada punto de los que se necesite calcular el valor de la función se puede variar en tiempo de ejecución.

Si la función a derivar es seno(c), `sin(x)`, el código de la función sería:

```
double derivadaseno(double x0, double h){
        return((8*(sin(x0+h)-sin(x0-h))-(sin(x0+2*h)-sin(x0-2*h)))/(12*h));
}
```

Si fuese $f(x) = x^2$ el código sería

```
double derivadacuadrado(double x0, double h){
        return((8*((x0+h)*(x0+h)-(x0-h)*(x0-h))-((x0+2*h)*(x0+2*h)-(x0-2*h)(x0-2*h)))/(12*h));
}
```

La solución es utilizar un puntero a función en el módulo `derivada`

Implementación

```
        char c;
        int i;
        double x0,h;


        do{
                        printf("DERIVACION NUMERICA\n");
                        printf("==================\n\n");
                        printf("Prueba derivacion numerica de funcion: x**3-3*x**2+5\n");
                        printf("Introduzca valor x0: ");
                        scanf(" %lf",&x0);
                        h=0.1;
                        for (i=1;i<7;++i){
                                printf("\ni=%2d h=%.6lf  derivada=%.6lf",i,h,derivada(f1,x0,h));
                                h=h/10;
                        }
                        printf("\n\nPrueba derivacion numerica de funcion: exp(-x**2)\n");
                        printf("Introduzca valor x0: ");
                        scanf(" %lf",&x0);
                        h=0.1;
                        for (i=1;i<7;++i){
                                printf("\ni=%2d h=%.6lf  derivada=%.6lf",i,h,derivada(f2,x0,h));
                                h=h/10;
                        }
                        printf("\n\nPrueba derivacion numerica de funcion: seno(x)*exp(-x)\n");
                        printf("Introduzca valor x0: ");
                        scanf(" %lf",&x0);
                        h=0.1;
                        for (i=1;i<7;++i){
                                printf("\ni=%2d h=%.6lf  derivada=%.6lf",i,h,derivada(f3,x0,h));
                                h=h/10;
                        }
                        printf("\n\nDesea efectuar una nueva operacion (s/n)? ");
        scanf(" %c",&c);
        }while ((c!='N') && (c!='n'));
        return 0;
}

double derivada(double (*f)(double x), double x0, double h){
        return((8*(f(x0+h)-f(x0-h))-(f(x0+2*h)-f(x0-2*h)))/(12*h));
}

double f1(double x){
        return (x*x*x-3*x*x+5);
}

double f2(double x){
        return (exp(-x*x));
}

double f3(double x){
        return(sin(x)*exp(-x));
}
```

Problemas de ingeniería

PROBLEMA

La relación entre el área y el perímetro de un triángulo puede influir en el diseño y la eficiencia de estructuras, la distribución de cargas, el flujo de fluidos y la planificación del terreno, entre otros aspectos. Utilizar esta relación de manera efectiva puede ayudar a los ingenieros a optimizar diseños y procesos para lograr resultados más eficientes y económicos. Por ejemplo el diseño de estructuras la distribución de cargas es fundamental. El área y el perímetro de un triángulo pueden influir en cómo se distribuyen las cargas a

través de una estructura, especialmente en el caso de puentes, marcos de edificios y otras estructuras que deben soportar cargas externas.

A partir de dos segmentos de cualquier longitud, es posible formar un triángulo añadiendo un tercero. De hecho podemos formar un número infinito de triángulos dependiendo del ángulo que formemos con los dos segmentos originales dentro del intervalo definido por $(l_1 - l_2, l_1 + l_2)$ con $l_2 > l_1$

Construir una función en C para determinar la pieza plana triangular con mayor relación área/perímetro, conocidas las longitudes de dos de sus lados en cm. La función devolverá la longitud del tercer lado de la pieza que maximiza dicha relación como parámetro de salida, y también devolverá el valor de dicha relación a través del identificador de la función. La longitud del tercer lado se determinará con una precisión de definida también como parámetro de la función. Considerar el siguiente prototipo:

```
double area_per(double a, double b, double prec, double *x);
```

Construir un programa en C que presente en pantalla una tabla con las longitudes del tercer lado que maximiza la relación área/perímetro, para longitudes de los otros lados comprendidas entre 5 cm y 50 cm, equi-espaciadas a intervalos de 5 cm, con una precisión en 1 mm.

Implementación

```
/*
 * @authors Equipo docente Programación
 * @project Creación de Materiales Didácticos en la Univer. de Almería (2021-2022)
 * Grados en Ingeniería Eléctrica, Electrónica Industrial, Mecánica y Química Industrial
 * @date 2021-02-06
 */



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

#define PREC 0.1        // Precisión del cálculo (cm)
#define AMIN 5.0        // Extremos a,b en tabla
#define AMAX 50.0
#define BMIN 6.0
#define BMAX 60.0
#define INC 2.0                 // Incremento valores a, b en tabla

/* Diseño Preliminar */
/* Diseño de Datos */
/* Nuevos tipos de datos */

/* Interfaces entre módulos */
/* Prototipos de funciones */
double triangulo(double a,double b, double prec, double *x);

/* Diseño Detallado */
/* Definiciones de funciones */
int main(){
    char c;
    double a,b,x,area;

    do{  system("cls||clear");
        printf("TRIANGULO DE AREA MAXIMA\n");
        printf("========================\n");
```

```
                printf("TABLA DE VALORES\n");
                printf("================\n\n");
                printf("%5s","");


                for(a=AMIN;a<=AMAX;a+=INC){
                        printf("%5.0f",a);
                }
                printf("\n");
                printf("%5s","");
                for(a=AMIN;a<=AMAX;a+=INC){
                        printf("-----");
                }

                printf("\n");
                for(b=BMIN;b<=BMAX;b+=INC){
                        printf("%4.0f|",b);
                        for(a=AMIN;a<=AMAX;a+=INC){
                                area=triangulo(a,b,PREC,&x);
                                printf("%5.1f",x);
                        }
                        printf("\n");
                }
        printf("\n\nDesea efectuar una nueva operacion (s/n)? ");
                scanf(" %c",&c);
        }while ((c!='N') && (c!='n'));
   return 0;
}

double triangulo(double a,double b, double prec, double *x){
        double y,ymax;
        double c;
        double s;

        ymax=0;
        c=fabs(b-a);
        while(c<=(b+a)){
                s=(a+b+c)/2.0;
                y=sqrt(s*(s-a)*(s-b)*(s-c))/(2*s);
                if(y>ymax){
                        ymax=y;
                        *x=c;
                }
                c+=prec;
        }
        return(ymax);
}
```

PROBLEMA

Comprobar si tres puntos en dos dimensiones son colineales. Es decir, están los tres en la misma línea. Existen diversos métodos.

Implementación

Comprobar si la pendiente definida por los dos segmentos de recta entre los puntos en la misma. Entonces están en la misma recta.

```
#include <stdio.h>

int colineales(double x1, double y1, double x2, double y2, double x3, double y3) {
// se basa en que las pendientes de los dos segmentos definidos
// por los puntos deben ser iguales.
    double pendiente1 = (y2 - y1) / (x2 - x1);
    double pendiente2 = (y3 - y2) / (x3 - x2);

    if (pendiente1 == pendiente2) {
```

```
        return 1;
    } else {
        return 0;
    }
}


int main() {
    double x1 = 1.0, y1 = 1.0;
    double x2 = 2.0, y2 = 2.0;
    double x3 = 3.0, y3 = 3.0;

    if (colineales(x1, y1, x2, y2, x3, y3)) {
        printf("Los puntos son colineales.\n");
    } else {
        printf("Los puntos no son colineales.\n");
    }

    return 0;
}
```

o bien calcular el área del triángulo definido por los tres puntos y si es cero (o casi cero porque son reales) son colineales.

Pero la función que tenemos utiliza tres longitudes no tres puntos, como también tenemos la función que devuelve la distancia euclidea dados dos puntos se reutilizar las funciones que ya hay implementadas.

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

int colineales(double x1, double y1, double x2, double y2, double x3, double y3);
double distancia (double x1, double y1, double x2, double y2);
double calcularAreaTriangulo(double a, double b, double c);


int main() {
    double x1 = 1, y1 = 1;
    double x2 = 2, y2 = 2;
    double x3 = 3, y3 = 3;


    if (colineales(x1, y1, x2, y2, x3, y3))
        printf("Los puntos son colineales.\n");
    else
        printf("Los puntos no son colineales.\n");

    return 0;
}
double distancia (double x1, double y1, double x2, double y2){
    return(sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2)));
}

double calcularAreaTriangulo(double a, double b, double c) {
    double semiperimetro = (a + b + c) / 2.0;
    double area;
    if (a <= 0 || b <= 0 || c <= 0 || a + b <= c || a + c <= b || b + c <= a) {
           area =-1; // Retorna un valor negativo para indicar error
    }
    else
        area = sqrt(semiperimetro * (semiperimetro - a) * (semiperimetro - b) * (semiperimetro - c));
    return area;
}

int colineales(double x1, double y1, double x2, double y2, double x3, double y3) {
    float l1=distancia(x1,y1,x2,y2);
    float l2=distancia(x2,y2,x3,y3);
    float l3=distancia(x1,y1,x3,y3);
```

```
    double area = calcularAreaTriangulo(l1,l2,l3);
    // no hay que capturar el valor negativo del area porque siempre será positiva
    return area < 1e-9;
}
```

Nota acerca de la visualización en consola Una de las tareas que se repite es la de limpiar la pantalla y configurar la entrada salida para windows, se puede definir una función que realice esta tarea.

Lo primer es limpiar la pantalla sin embargo es precisamente la entrada salida lo que una de las cuestiones que es estrechamente dependiente del sistema operativo. Por ejemplo para limpiar la pantalla en un sistema Windows debemos enviar la orden `cls` al sistema, mientras que en Linux se ha de usar `clear`

En un programa windows usaremos:

```
#include <stdlib.h>
...
system("cls"); // Limpia el terminal windows
```

mientras que en un programa para linux se usará:

```
#include <stdlib.h>
...
system("clear"); // Limpia el terminal linux
```

Otra de las cuestiones dependiente del sistema operativo es la utilización de los caracteres "diferentes", como las tildes, ¿ y la "ñ" que se llevan mal con la consola de entrada salida en especial en Windows. En los codespaces de C con Linux no hay problema y todos los acentos se incorporan sin alteración, pero si el mismo programa se ejecuta/compila en Windows existe un problema con la interpretación de estos caracteres. En este caso, la solución más simple es incorporar la librería `windows.h` e invocar dos operaciones/funciones que facilitan la interpretación de este juego de caracteres también al principio del programa, pero solo cuando se está en Windows :

```
        SetConsoleOutputCP(CP_UTF8);
        SetConsoleCP(CP_UTF8);
```

Para poder plantear que el programa sea visualizable tanto en windows como en linux se puede tomar partido de la condicional en tiempo de pre-compilación de esta forma se puede incluir el siguiente código.

```
#include <stdio.h>
// Asegúrate de incluir la siguiente directiva del preprocesador
#ifdef _WIN32
#include <windows.h>
#endif

void config_acentos(){
      // Configuraciones específicas para Windows
#ifdef _WIN32
    // Configura la consola de Windows para admitir UTF-8
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    system("cls");
#else
        system("clear");
```

```
#endif

}


void limpiapantalla(){
#ifdef _WIN32
    system("cls");
#else
        system("clear");
#endif

}

int main() {

    config_acentos();

    // Ahora puedes imprimir caracteres con acentos
    printf("¡Hola, mundo! ÁéíóÚ ñ Ñ\n");
    getchar();
    limpiapantalla();

    return 0;
}
```

Pero lo lógico sería tener esta función aparte para poder se ejecutada en cualquier programa, es decir, usar la compilación separada.

Compilación separada

La compilación separada es un enfoque en el desarrollo de programas que implica dividir el código fuente en múltiples archivos, y luego compilar y enlazar estos archivos por separado para formar el programa final. Este enfoque ofrece varias ventajas en términos de organización del código, modularidad y mantenimiento del software.

Aquí hay una descripción paso a paso del proceso de compilación separada:

División del código fuente: El código fuente del programa se divide en varios archivos. Cada archivo generalmente contiene una o más funciones relacionadas o una unidad de código coherente.

Archivos de encabezado (.h): Se crean archivos de encabezado que contienen las declaraciones de funciones, definiciones de estructuras y constantes que otros archivos pueden necesitar para utilizar las funciones definidas en los archivos fuente (.c).

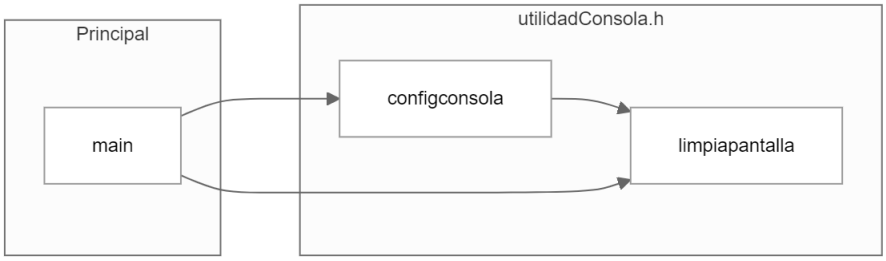
Compilación por separado: Cada archivo fuente (.c) se compila de forma independiente en un archivo objeto (.o o .obj). Durante la compilación, solo se necesita el código fuente y el archivo de encabezado correspondiente. El compilador genera un archivo objeto para cada archivo fuente.

Enlace: Finalmente, los archivos objeto se enlazan juntos para formar el programa ejecutable. Durante este proceso, se resuelven las referencias a funciones y variables definidas en diferentes archivos. El enlazador crea el programa ejecutable a partir de los archivos objeto.

Esta metodología facilita la gestión del código en proyectos grandes, ya que permite realizar cambios en una parte del programa sin tener que recompilar todo. Además, promueve la reutilización del código y facilita la colaboración entre diferentes desarrolladores o equipos, ya que cada uno puede trabajar en una parte específica del proyecto sin interferir con otros.

En lenguajes como C y C++, la compilación separada es una práctica común y es compatible con la creación de bibliotecas, donde los archivos objeto se pueden empaquetar para su reutilización en otros proyectos.

El objetivo de este curso básico de programación no incluye la generación de grandes proyectos o la necesidad de compilación separada pero puede ser interesante aprovechar la potencialidad de la directiva `#include` para simplificar la lectura de nuestro código.



El archivo `UtilidadConsola.h` incluirá las funciones que permiten incorporar los acentos tanto en linux como en Windows y que limpiará la pantalla

```
#ifdef _WIN32
#include <windows.h>
#endif

void config_acentos(){
      // Configuraciones específicas para Windows
#ifdef _WIN32
    // Configura la consola de Windows para admitir UTF-8
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    system("cls");
#else
        system("clear");
#endif
}

void limpiapantalla(){
#ifdef _WIN32
    system("cls");
#else
        system("clear");
#endif
}
```

Para utilizar ambas funciones bastará con incluir (debe estar accesible) el archivo en el código del programa

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

int main() {

    config_acentos();
```

```
    // Ahora puedes imprimir caracteres con acentos
    printf("¡Hola, mundo! Áéíóú ñ Ñ\n");
    getchar();
    limpiapantalla();

    return 0;
}
```

Estructuras de datos

Clasificación de las estructuras de datos

Recordando lo que se indicó en el primer apartado de este documento, los programas son instrucciones y datos. Los distintos tipos de datos que se utilizan en un programa son de diversa naturaleza y estructura dependiendo de:

- La organización:
  - Representación interna (implementación)
  - Rango de valores permitidos (dominio)
- Las operaciones permitidas

Los tipos de datos hacen referencia al tipo de información con la que se trabaja, donde la unidad mínima de almacenamiento es el dato. Hasta ahora se ha trabajado con los llamados tipos de datos simples, como son los enteros o los reales.

Ya es el momento de comenzar a pensar en situaciones que requieren tipos de datos de otra naturaleza. Por ejemplo, recordemos en caso de las operaciones con vectores en tres dimensiones cuando se implementó la operación D = A ^ (B ^ C), un vector son tres reales siempre. O el caso del conjunto de temperaturas leídas por teclado para calcular su media. Estas situaciones reflejan la necesidad de utilizar colecciones de datos (referenciados mediante un único identificador) que describe un objeto complejo de información, es decir **Tipos_abstractos de datos (TAD)** o lo también se suele llamar **tipos definidos por el usuario**.

Un tipo de dato abstracto es un modelo que define valores y las operaciones que se pueden realizan sobre ellos. Se denomina abstracto ya que la intención es que quien lo utiliza, no necesita conocer los detalles de la representación interna o bien el cómo están implementadas las operaciones, por ejemplo hablando del vector3D, sin saber si son tres reales, tres enteros o cualquier otra cosa. Se trata de, al igual que las funciones, separar la interfaz de la implementación, las funciones son abstracciones funcionales, recogen bajo un nombre un sistema complejo.

No obstante la mayoría de los lenguajes de programación parten de un conjunto de **tipos compuestos** o estructurados que permiten recoger la necesidad de utilizar datos complejos y que facilitan la definición de tipos específicos a los usuarios, y que serán el punto de partida de la definición tipos específicos para ese programa:

- **Texto (cadena de caracteres)**, Mientras que los caracteres individuales son datos simples, las cadenas de caracteres o *strings* son datos complejos, puesto que son una lista ordenada de datos simples que se acaba con un final de cadena.
- **Colecciones indexadas o vectores**, un grupo de datos del mismo tipo que se organizan por posición, desde el primero hasta el último del grupo
- **Colecciones estructuradas o registros**, un grupo de datos que recoge las propiedades a almacenar sobre un concepto o entidad, y que se identifican por la etiqueta de la propiedad. Por ejemplo, para una persona (entidad) se necesita saber su color de ojos, altura, DNI (etiquetas de propiedades).

Por tanto la **Clasificación de las estructuras de datos** podemos definirla ahora de forma completa:

- **Dato simple: (indivisible o atómico)**: No está compuesto de otros datos. Una variable de un tipo de datos simple representa a un solo dato.
  - **Datos simples estándar**: forma parte de la sintaxis del lenguaje de programación. Se suelen llamar también tipos de datos **primitivos** o **predefinidos**, disponibles en la mayoría de los lenguajes de programación.
    - Numéricos:
      - ◇ Entero
      - ◇ Real
    - Lógico o booleano (no disponible en C)
    - Carácter
    - Punteros, si bien no es un dato en si mismo sino una dirección, cuando "apunta" a un dato simple se puede considerar como simple, pero ¡ojo! si apunta a una colección de datos se puede categorizar como dato compuesto. En algunos lenguajes no se pueden manejar las direcciones como tales.
  - **Datos simples no estándar**: están definidos por el programador mediante especificación con identificadores de los valores de su dominio (**enumeración**) ó mediante un subconjunto de un tipo ordinal (subrango). Son Tipos de datos internos al programa no se pueden involucrar en operaciones de E/S que son isomorfos a un subconjunto de los enteros.

```
int entero = 10;
float decimal = 3.14;
double mayorPrecision = 3.1415926535;
char caracter = 'A';

enum Meses {ENERO, FEBRERO, MARZO, ABRIL, MAYO, JUNIO, JULIO, AGOSTO, SEPTIEMBRE,  OCTUBRE,
NOVIEMBRE, DICIEMBRE};
enum Meses miMes = ABRIL;
enum {FALSO = 0,    VERDADERO = 1}
```

- **Datos Compuestos o estructurados**. Es un tipo de dato que consta de varios datos de tipos primitivos. Es un constructor genérico de tipos de datos que el programador ha de completar. Una variable de un tipo de datos estructurado representa múltiples datos individuales, donde cada dato individual se puede referenciar de forma independiente a los demás. Las operaciones permitidas están relacionadas con almacenar y recuperar componentes individuales. Pueden ser *estructuras de datos estáticas* que tienen un tamaño fijo que no se puede modificar durante la ejecución del programa o bien *estructuras de datos dinámicas* que no tienen limitación ni restricciones en el tamaño (este puede cambiar durante la ejecución del programa).

    - **Texto** (cadena de caracteres)
    - **Colecciones indexadas** o vectores. Colección de elementos del mismo tipo.
    - **Colecciones estructuradas** o registros: Permite agrupar variables de diferentes tipos bajo un mismo nombre. En C hay estructuras y uniones.
    - **Punteros**, cuyo dominio es el de direcciones de memoria de variables de una tipología dada y con operaciones de comparación para igualdad y desigualdad, asignación/desasignación de memoria en tiempo de ejecución, acceso al contenido de la dirección apuntada (`&`). Es un caso bastante especial puesto que un puntero puede ser considerado como un tipo de datos simple que recoge una dirección de inicio al "trabajo" con la memoria.

```c
int *punteroEntero;
char miCadena[] = "Hola, mundo!";
char *miCadena = "Hola, mundo!";
int arrayEnteros[5] = {1, 2, 3, 4, 5};

struct Persona {
    char nombre[20];
    int edad;
};

union Datos {
    int entero;
    float decimal;
};
```

- **Definidos por el usuario**. Permite crear alias para tipos de datos existentes y ajustar las funciones y rango permitido para esos tipos, incluyendo tipos compuestos y/o primitivos. En C se utiliza '`typedef` para indicar que se está definiendo un tipo de dato. Es habitual que se utilice para "completar" las colecciones dotándolas de más contenido semántico.

```c
#include <stdio.h>

// Definición del tipo lógico con un enum
typedef enum {
    FALSO = 0,
    VERDADERO = 1
} Logico;


Logico logico_y(Logico a, Logico b) {   // Operación lógica "y"
    return a && b;
}

Logico logico_o(Logico a, Logico b) {    // Operación lógica "o"
    return a || b;
}
```

```
Logico logico_no(Logico a) { // Operación lógica "no"
    return !a;
}

typedef int fecha[3];

typedef struct {
    int dia;
    int mes;
    int anio;
} Fecha;

int main()
{
    // Ejemplos de uso
    Logico x = VERDADERO;
    Logico y = FALSO;

    printf("x AND y: %d\n", logico_y(x, y));
    printf("x OR y: %d\n", logico_o(x, y));
    printf("NOT x: %d\n", logico_no(x));

    Fecha hoy_fecha = {16, 11, 2023};
    printf("Hoy es: %02d/%02d/%d\n", hoy_fecha.dia, hoy_fecha.mes, hoy_fecha.anio);


    fecha hoy = {16, 11, 2023};
    printf("Hoy es: %02d/%02d/%d\n", hoy[0], hoy[1], hoy[2]);

    return 0;
}
```

Extendemos el ejemplo del cálculo de área de un triangulo dadas las coordenadas de sus lados, pero además debemos incorporar la información del tipo de triángulo manejando un tipo enumerado.

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

typedef enum {
    EQUILATERO,
    ISOSCELES,
    ESCALENO,
    NO_TRIANGULO
}tipo_triangulo;

double distancia (double x1, double y1, double x2, double y2);
double calcularAreadesDeLados(double a, double b, double c);
tipo_triangulo calcularAreaTriangulo(double x1, double y1, double x2, double y2,
double x3, double y3, double *area);

int main() {
    double x1 = 1, y1 = 1;
    double x2 = 2, y2 = 0;
    double x3 = 1, y3 = 0;
    double area;

    tipo_triangulo queTrianguloEs = calcularAreaTriangulo(x1, y1, x2, y2, x3, y3, &area);

    switch (queTrianguloEs) {
        case EQUILATERO:
            printf("El triángulo es equilátero\n");
            break;
        case ISOSCELES:
            printf("El triángulo es isósceles\n");
            break;
        case ESCALENO:
            printf("El triángulo es escaleno\n");
```

```
            break;
        case NO_TRIANGULO:
            printf("Los puntos no forman un triángulo válido\n");
            break;
    }
    printf("El área del triángulo es: %lf\n", area);
    return 0;
}


double distancia (double x1, double y1, double x2, double y2){
    return(sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2)));
}

double calcularAreadesDeLados(double a, double b, double c) {
    double semiperimetro = (a + b + c) / 2.0;
    double area;
    if (a <= 0 || b <= 0 || c <= 0 || a + b <= c || a + c <= b || b + c <= a) {
        area =-1; // Retorna un valor negativo para indicar error
    }
    else
        area = sqrt(semiperimetro * (semiperimetro - a) * (semiperimetro - b) * (semiperimetro - c));
    return area;
}


tipo_triangulo calcularAreaTriangulo(double x1, double y1, double x2, double y2,
             double x3, double y3, double *area) {
    double lado1, lado2, lado3;
    tipo_triangulo clase;

    lado1 = distancia(x1, y1, x2, y2);
    lado2 = distancia(x2, y2, x3, y3);
    lado3 = distancia(x3, y3, x1, y1);

    *area=calcularAreadesDeLados(lado1,lado2,lado3);

    // Calcular el tipo de triángulo
    if (*area==-1) {
        clase = NO_TRIANGULO; // No es un triángulo válido
    } else if (lado1 == lado2 && lado2 == lado3) {
        clase = EQUILATERO;
    } else if (lado1 == lado2 || lado1 == lado3 || lado2 == lado3) {
        clase = ISOSCELES;
    } else {
        clase = ESCALENO;
    }
    return clase;
}
```

Colecciones indexadas. Vectores y matrices/tablas

Se trata de una estructura de datos ordenada de elementos **homogéneos** cuyo orden viene definido por su **posición** (índice) no por su contenido. Los elementos se distribuyen en filas (1D), filas y columnas (2D), o incluso más dimensiones. Estas dimensiones son los **índices** que permiten acceder por posición a los elementos guardados en esta estructura de datos. La estructura de datos completa esta identificada representando un todo, es decir tienen un identificador.

En inglés el nombre que reciben estas estructuras de datos es **array**. Lo primero a destacar es la traducción que se ha hecho históricamente de este tipo de estructura de datos: formaciones, arreglos, vectores (si son de una dimensión), tablas y matrices (dos dimensiones). Si es multidimensional, matemáticamente hablando, parece que el nombre

vector es poco apropiado, para la programación la consideramos como la traducción más correcta y cuando se trate de vectores con más dimensiones se puede utilizar el término **vectores multidimensionales** o **matrices** solo si tienen dos dimensiones.

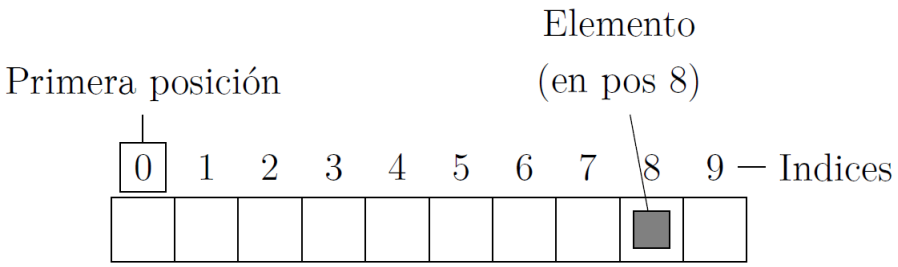
Desde el punto de vista de su organización sobre la memoria existen dos tipos de colecciones indexadas o vectores, las **estáticas** y las **dinámicas**. Para los primeros se fija su tamaño en tiempo de compilación mientras que los segundos puede ir aumentando de tamaño según demande el programa.

Si bien los vectores estáticos (**arrays**) son parte de la mayoría de los lenguajes de programación como constructores genéricos para la gestión de datos compuestos, no ocurre lo mismo con los arrays dinámicos, que están sujetos a las carácterísticas del propio lenguaje. En C supone el uso de punteros y memoria dinámica.

De forma general cuando se utiliza el término vector en programación estamos hablado de una estructura de datos **estática** que su tamaño se fija en el proceso de compilación, es **homogénea** ya que todos sus componentes son del mismo tipo de datos y está **ordenada** utilizando un **índice**, aunque puede utilizarse más de un indice, tratándose entonces de vectores multidimensionales. A esta estructura se le da un nombre común que identifica a la estructura globalmente, a modo de variable del programa.

Vectores unidimensionales

Un vector es estructura de datos que consta de una serie de elementos del **mismo tipo** de datos ordenados en secuencia única. Se utiliza cuando en un programa se necesita representar simultáneamente en un conjunto de datos de la misma tipología. Por convenio los arrays en C empiezan siempre con la posición 0, es decir el primer elemento de un vector en C está en la posición `[0]`.



Veamos un caso donde se tienen que tomar datos de 50 valores de temperatura. Se podría utilizar 50 identificadores diferentes, pero ¿que pasa si en vez de 50 son 45?.

Si no se conocen exactamente cuantos datos son no se pueden usar variables individuales, además la lectura y escritura no se podrían generalizar con un bucle. Se hace necesaria una estructura de datos que guarde `m` datos y pueda ser accedida (indexada) empleando un conteo, desde `0` hasta `n`, con $n = m - 1$.

Esta situación es la que se justifica el uso de vectores donde tengamos un vector `temperatura` que pueda accederse por posición de la 0 a la n.

`temp[0]` hasta `temp[n]`

Con un solo identificador `temp` estamos refiriéndonos a los 50 valores distintos que se invidualizan utilizando un *indice* o *posición*.

Ejemplos: modelos de información en los que aparecen bloques de datos simples de la misma tipología:

- 100 datos de temperatura medidos por un sensor a intervalos regulares de tiempo.
- Las notas obtenidas por los 50 alumnos de una asignatura en el último examen realizado.
- Las respuestas de un usuario a un cuestionario (test) de 100 preguntas con respuestas alternativas (V/F).
- Tres vectores tridimensionales: $v_1 = (x_1, y_1, z_1), v_2 = (x_2, y_2, z_2), v_3 = (x_3, y_3, z_3)$
- El nombre de una persona.
- Una lista de 50 números enteros generados aleatoriamente.
- Las abscisas y ordenadas de 50 puntos en el plano, donde cada punto esta representado por x, y

En estos ejemplos podemos ver que los elementos individuales, también llamados **tipo base** del vector pueden ser tipos de datos primitivos o datos compuestos. En este apartado se plantearán ejemplos sólo con vectores en los que el tipo base son tipos primitivos, cuando se estudien los registros (colecciones estructuradas) se extenderán estos ejemplos.

Declaración de vectores Los vectores son constructores genéricos de tipos de datos que el programador ha de completar indicando el tipo base sobre el que se sustentan, y también es posible definir un tipo propio, **tipo definido por el usuario** como un vector de este tipo básico específico. Esto da lugar a las dos formas de definir un vector en C:

- Directamente definiendo el tipo base del vector.

```
float temp[100];     /*vector de 100 reales de 0 a 99 */
int contadores[5];   /* vector de 5 enteros         */
char frase[50];      /* vector de 50 caracteres */
```

- Empleando un tipo definido y declarando variables de ese tipo.

```
typedef tipo_base tipo_vector[N];
tipo_vector nombre_variable;
```

Sobre los ejemplos:

```
/* tipos definidos por el usuario */

typedef float tipo_temperaturas[100];
typedef int tipo_contadores[5];
typedef char tipo_frase[50];

/* declaración de variables */
tipo_temperaturas temperaturas;
tipo_contadores contadores;
tipo_frase frase;
```

Encontramos numerosos ejemplos de estos dos métodos de declaración, cada uno tiene sus partidarios y sus detractores. No obstante, es interesante discriminar la organización (tipos de datos) de la aplicación (variables), es decir utilizar `typedef`, que será el método elegido para esta asignatura para el programa principal, sin embargo en las funciones buscaremos interfaces lo más genererales posibles, recurriendo a la notación sobre punteros al tipo base.

Nótese que el **tamaño del vector se declara al escribir el programa y que este no puede cambiar durante la ejecución**. Si se desea procesar una lista variable de elementos del mismo tipo de datos, se debe declarar un vector lo suficientemente grande y registrar el tamaño del sub-vector con datos en una variable entera, es decir, para manejar el vector usamos el vector en si y su tamaño representado por un valor entero.

Operaciones sobre los vectores Para trabajar con los **elementos individuales** de los vectores simplemente basta con tratarlo como si de una variable del tipo base se tratara, así se puede:

- Asignar

```
temp[5] = 17.4;
temp[i+1] = temp[i];
temp[2*i-j] = 3.4;
```

- Realizar la entrada salida

```
scanf(" %f" , &temp[8]);
printf(" %.1f", temp[10]);
```

- Utilizarlo en expresiones

```
temp[3]=temp[3]+1.0;
temp[11]=temp[10]+2.0;
(temp[5] > temp[3])
```

- Paso de parámetros

```
}

int main() {
    int miVector[] = {5, 2, 3, 4, 5};
```

- Acceso a los elementos del vector como una unidad o recorrido

Este tipo de operaciones pueden ser de entrada/salida/operación. La idea es que se tienen que "tratar todos" los elementos del vector, es decir **recorrer** la estructura de datos y realizar una acción con cada elemento de la estructura. (¡ojo! una acción puede se no hacer nada, o puede que solo se acceda a un subvector controlando las posiciones accedidas). El acceso a todos los elementos de uno en uno empezando por el primero y siguiendo el orden físico de almacenamiento, que es el caso más habitual, sobre el que se establecerán variaciones dependiendo del problema.

Por ejemplo para un vector de 100 elementos

```
#define max 100
typedef  tipo_base tipo_vector[max];

int i;
tipo_vector v;
. . . .
i=0;
while (i<max)
{
        /* procesar elemento v[i] */
        i=i+1;
}
```

Se necesita un bucle para acceder a cada elemento. En los vectores es muy habitual utilizar bucles tipo `for` puesto que evitan la necesidad del controlar el contador.

```
// Bucle para lectura de datos float con un mensaje
for(i=0; i<max; ++i)
{
    printf("v[%d]? ",i+1);
        scanf(" %f", &v[i]);
}
```

En este caso se pone de manifiesto que los vectores en C comienzan con el índice 0.

```
// Visualiza un array de max elementos float
for(i=0; i<max; ++i)
   printf("v[%d]= %.1f\n", i+1,v[i]);
```

- **Copia** de elementos del vector. Se debe tener en cuenta que conceptualmente los vectores son "direcciones" de inicio a una lista de objetos de un tipo, por lo tanto para hacer una copia de un vector no basta con asignar un identificador de un vector a otro. Se debe recorrer la estructura de datos y realizar la copia elemento a elemento, al menos en C.

```
tipo_vector v2;
for(i=0; i<max; ++i)
{
     v2[i]=v[i];
}
```

No se copian elementos con

```
v2=v;
```

- **Buscar** la primera ocurrencia de un valor en un vector (o búsqueda por contenido) es otra operación básica.

```
i=0;
enc=0;
while((i<max)&&(!enc))
{
    if(v[i]==x)         {
        enc=1;
    }
    else    {
              i=i+1;
   }
}

if (enc)  {
        printf("Encontrado");
}
else {
    printf("No encontrado");
}
```

Se ha utilizado un bucle `while` para hacer más eficiente el proceso, puesto que una vez encontrado no es necesario seguir recorriendo el vector hasta el final.

- **Paso parámetros** del vector completo

En C los vectores (y los "arrays" en general) se pasan por dirección (no existe el paso de vectores por valor).

- *Parámetro real* es el identicador del vector `v`

El nombre del vector representa la dirección del primer elemento (por lo tanto, no es necesario usar **&** delante del nombre de un vector cuando se pasa como argumento a una función).

- *Parámetro formal*: varias posibilidades de declaración (nótese que las dos últimas son compatibles con vectores de cualquier tamaño):
  - Usando el tipo predefinido `tipo_vector v`
  - Declarando directamente el parámetro formal como un vector, sin especificar el tamaño: `tipo_base v[]`.‘
  - Declarando el parámetro formal como un puntero al tipo base `tipo_base *v`

```
#include <stdio.h>

typedef tipo_base tipo_vector;

int longitud_vector_tipo_predefinido(tipo_vector v);
 //preferiremos estas dos siguientes
int longitud_vector_vector_sin_tamano(tipo_base v[]);
int longitud_vector_puntero_tipo_base(tipo_base *v);
```

En los dos primeros casos, el acceso dentro de la función llamada a un elemento del vector se realiza de la forma convencional `v[i]`

```
int longitud_vector_vector_sin_tamano(tipo_base v[]) {
    return v[0]; // Acceso al primer elemento del vector.
}

int longitud_vector_puntero_tipo_base(tipo_base *v) {
    return v[0]; // También acceso al primer elemento del vector.
}
```

En ambos ejemplos, aunque el parámetro formal v es diferente (una vez como v[] y otra como *v), se accede al primer elemento del vector de la misma manera (v[0]).

Nótese que dentro de la función llamada, el parámetro formal que representa al vector (que siempre se pasa por dirección) se manipula como si fuera un parámetro de entrada.

```
Copiar código
void modifica_vector(tipo_base v[]) {
    v[0] = 10; // Modifica el primer elemento del vector real que se pasó.
}

int main() {
    tipo_base mi_vector[3] = {1, 2, 3};
    modifica_vector(mi_vector);
    printf("%d\n", mi_vector[0]); // Imprime "10", ya que el vector ha sido modificado.
}
```

En este caso, aunque el vector v dentro de modifica_vector es un parámetro formal, puede modificar el contenido del vector original (mi_vector en main), ya que está pasando una referencia a los datos originales.

Si se quiere utilizar un subvector se deben dar el rango o límites del subvector

- Parámetros reales `v, i, j`
- Parámetros formales `tipo_vector v, int i, int j`

Veamos el siguiente ejemplo:

```
#include <stdio.h>


float calcularFuerzaTotal(int fuerzas[], int numSecciones) {
   float fuerzaTotal = 0.0;

   for (int i = 0; i < numSecciones; ++i) {
       fuerzaTotal += fuerzas[i];
   }

   return fuerzaTotal;
}
int main() {
    int numSecciones = 5;

    int fuerzas[] = {1000, 1500, 800, 1200, 2000};

    float fuerzaTotal = calcularFuerzaTotal(fuerzas, numSecciones);

    printf("Fuerzas en las secciones del puente:\n");
   for (int i = 0; i < numSecciones; ++i) {
       printf("Sección %d: %d Newtons\n", i + 1, fuerzas[i]);
   }

   printf("\nFuerza total en el puente: %.2f Newtons\n", fuerzaTotal);

   return 0;
}
```

En este ejemplo, tenemos un vector fuerzas que representa las fuerzas actuando en cada sección del puente. La función `calcularFuerzaTotal` toma este vector y calcula la fuerza total en el puente sumando todas las fuerzas individuales. La función principal luego muestra las fuerzas en cada sección y la fuerza total en el puente.

Es recomendable el uso de tipos definidos para la declaración de los vectores con lo que el problema anterior quedaría:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>

typedef float VectorFuerzas[100];

float calcularFuerzaTotal(int fuerzas[], int numSecciones);

int main() {

    int numSecciones = 6;
    VectorFuerzas fuerzas = {1000, 1500, 800, 1200, 2000, 22};
    float fuerzaTotal = calcularFuerzaTotal(fuerzas, numSecciones);

    printf("Fuerzas en las secciones del puente:\n");
```

```
    for (int i = 0; i < numSecciones; ++i) {
        printf("Sección %d: %d Newtons\n", i + 1, fuerzas[i]);
    }

    printf("\nFuerza total en el puente: %.2f Newtons\n", fuerzaTotal);

    return 0;
}

float calcularFuerzaTotal(int fuerzas[], int numSecciones) {
    float fuerzaTotal = 0.0;

    for (int i = 0; i < numSecciones; ++i) {
        fuerzaTotal += fuerzas[i];
    }
    return fuerzaTotal;
}
```

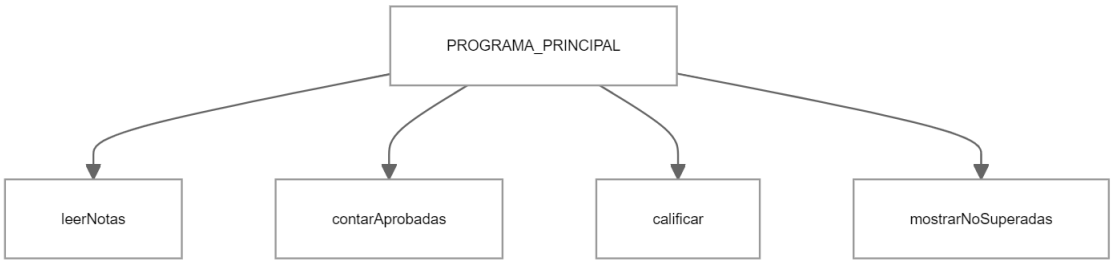
PROBLEMA Construir un programa que lea de teclado las notas de las prácticas de la asignatura y calcule cuantas de ellas se han superado para, pidiendo la nota del examen y trabajos evaluables, se calcule la nota definitiva del alumno. Cuando el alumno suspenda la asignatura debe indicarse cuales de las prácticas estaban no superadas.

Analisis Tengo que almacenar las notas de las prácticas para después revisarlas. Las notas de las prácticas se evalúan de 0 a 10 siendo 5 aprobado.

Diseño de datos

- Vector de 10 reales para mantener las notas de cada práctica.

Diseño arquitectónico



- `leerNotas` devolverá un vector
- `contarAprobadas` recibe un vector de entrada devuelve un int como salida.

Implemantación

```
#include <stdio.h>

#define TAM 10

typedef float tipo_notas[TAM];


void leerNotas(tipo_notas pr);
int contarAprobadas(float pr[]);
void mostrarNoSuperadas(float pr[]);
float calificar(float pr[], int ti, float ex);

int main() {
    tipo_notas notas_pr;
    int n_EI = 0;
```

```
    float examen = 0;
    float calificacion;

    leerNotas(notas_pr);
    printf("¿Cuál es la nota de los trabajos evaluables (0-1): ?");
    scanf(" %d", &n_E);
    printf("¿Cuál es la nota obtenida en el examen (1-10):  ?");
    scanf(" %f", &examen);

    calificacion =calificar(notas_pr, n_TE, examen);

    int califEntera;
    califEntera = (int) calificacion;

    switch (califEntera) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
                printf("Calificación: Suspenso con un %.2f\n", calificacion);
                mostrarNoSuperadas(notas_pr);
                break;
            case 5:
            case 6:
                printf("Calificación: Aprobado con un %.2f\n", calificacion);
                break;
            case 7:
                printf("Calificación: Notable con un %.2f\n", calificacion);
                break;
            case 8:
            case 9:
                printf("Calificación: Sobresaliente con un %.2f\n", calificacion);
                break;
            case 10:
                printf("Calificación: Matrícula de Honor con un %.2f\n", calificacion);
                break;
            default:
                printf("Calificación no válida. con un %.2f\n", calificacion);
                break;
        }
    return 0;
}

void leerNotas(float pr[]) {
    printf("Introduce las notas:\n");
    for (int i = 0; i < TAM; i++) {
        printf("PR-%d: ", i + 1);
        scanf("%f", &pr[i]);
    }
}

int contarAprobadas(float pr[]){
    int contador = 0;
    for (int i = 0; i < TAM; i++) {
        if (pr[i] >=5 ) {
            contador++;
        }
    }
    return contador;
}


float calificar(float pr[], int n_TE, float ex){
    float calificacion=0;
    int n=contarAprobadas(pr);
    int n_PR=0;
    if (n>7) n_PR=1;
    if(ex>=4)
        calificacion= n_PR * (ex *0.8) + n_TE + n_PR;
    return calificacion;
```

```
}

void mostrarNoSuperadas(float pr[]){
    printf("Introduce las notas:\n");
    for (int i = 0; i < TAM; i++) {
        if (pr[i]<5)    printf("PR-%d: no superada con calificacion %f\n ", i + 1, pr[i]);
    }
}
```

Para asegurarnos de no modificar el vector podemos usar `const`

```
int contarAprobadas(const tipo_notas notas);
```

Diferencia entre la capacidad y tamaño Un vector tiene una **capacidad** que se define en tiempo de compilación, que indica el tamaño máximo que puede tener un vector. También llamado dimensión.

La capacidad no puede ser alterada en la ejecución siendo una decisión de diseño:

- En ocasiones será fácil (días de la semana)
- Cuando pueda variar ha de estimarse un tamaño máximo.

Ni corto ni con mucho desperdicio (posiciones sin usar)

Esta capacidad se puede definir explicita o implícitamente, pero para poder manejar/recorrer el vector, debe conocerse antes de la ejecución.

- Definición explícita

```
#define MAX 100
typedef int tipo_vector[MAX];

...

tipo_vector vector;
....
for(int i=0;i<MAX; i++)
for(int i=0;i<=(MAX-1); i++)
```

De esta manera los recorridos pueden limitarse al tamaño del vector.

- Definición implícita

```
typedef int tipo_vector[MAX];

tipo_vector vector = {1000, 1500, 800, 1200, 2000};
    // Se crea en tiempo de compilación un vector de 5 posiciones
int capacidad= 5;
for(int i=0;i<capacidad; i++)
```

Para poder recorrer el vector debe conocer su capacidad, gestionada preferiblemente como una constante, aunque también puede ser una variable. Es **responsabilidad del programador** conocer los límites de los vectores.

No obstante a la capacidad máxima hay situaciones del problema que hace que no se necesiten todas las posiciones de un vector, pudiendo tener menos posiciones que el máximo.

Por ejemplo, queremos un programa que calcule los datos estadísticos (media, mediana, desviación estándar) de las notas de un estudiante, pero que pueda ser utilizado a lo largo de todo el curso cuando el estudiante aun no ha completado todas las actividades de evaluación. En este caso se necesita un vector que tenga una **capacidad** equivalente al máximo de notas que pueda obtener el alumno, pero que en el momento del uso del programa puedan ejecutarse los cálculos con de `[1,max_actividades]` del alumno.

En estas situaciones el **tamaño** del vector es distinto de su **capacidad** siendo necesario utilizar un contador posiciones ocupadas/válidas.

```
#define MAX 100
typedef int tipo_vector[MAX];

tipo_vector lista;
int tam = 0; // recoge los elementos válidos

lista[0]=1;
tam++;
```

Sólo ser recorre el vector entre las posiciones entre 0 y `tam`. Las demás posiciones no contienen información válida del programa.

```
for(int i=0;i<tam; i++)
.....
```

PROBLEMA

Construir un programa que lea por teclado entre 1 y 24 datos de temperatura de un sistema, medidos cada uno a intervalos de una hora en un día determinado, y que calcule e imprima en pantalla las diferencias de cada temperatura con respecto a la temperatura media de ese día.

Análisis:

La entrada son las 24 temperaturas de tipo real `float` la salida son 24 valores que muestran su diferencia frente a la media. Aunque puede haber menos de 24 Temperaturas por errores en los sensores. Se ha de preguntar cuantas temperaturas hay válidas.

Lo primero será calcular la media, pero para poder mostrar la diferencia debemos saber/almacenar hasta 24 temperaturas. En problemas anteriores, bastó con ir sumando los valores de entrada e incluso no había un máximo de valores. Aquí se deben leer y guardar las temperaturas.

Diseño preliminar

Necesitamos guardar las 24 temperaturas leídas, calcular la media y estudiar la diferencia de cada una de ellas respecto al valor calculado. Se necesita por tanto una estructura de datos tipo vector donde hay que controlar el tamaño del vector, es decir se utilizará una variable para registrar el número de datos válidos.

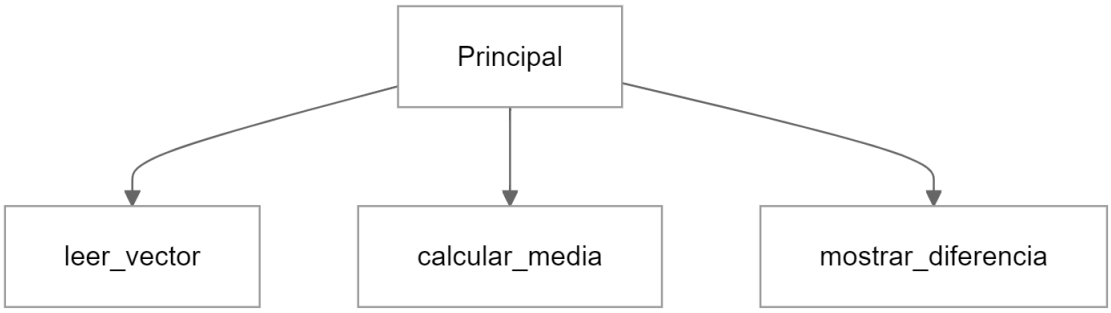
- Diseño de datos

Modelo de información: bloque de 24 datos reales t_i que hay que mantener en memoria para poder calcular las desviaciones con respecto a la media: $t_i - t_{med}$

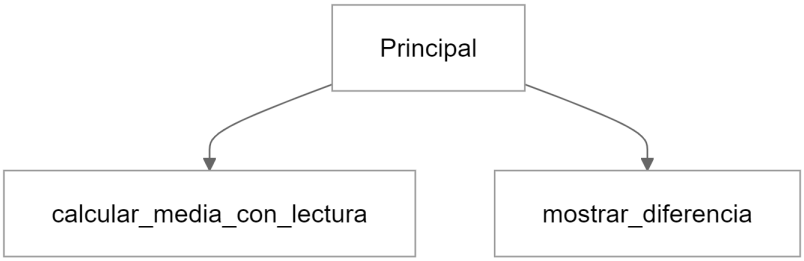
Bloque de datos estático, homogéneo y ordenado temporalmente en secuencia por tanto un vector unidimensional:

```
#define N 24

typedef float tipo_temp[N];
```

- Diseño de módulos

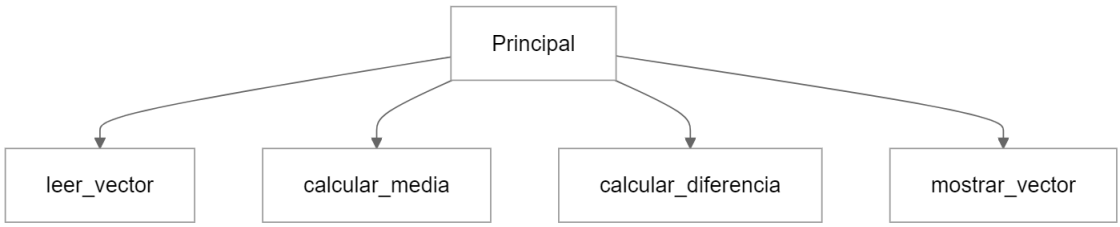


Pero podemos pensar que puesto que debemos recorrer el vector para leer los datos también ese módulo puede realizar la tarea de acumular los valores de la temperatura para calcular la media (puede ser devuelta por el módulo), entonces la arquitectura modular sería la siguiente:



Evidentemente al utilizar menos módulos se simplifica el problema de comunicación entre módulos, pero incluso el nombre llega a resultar confuso puesto que tenemos que utilizar dos verbos (leer y calcular) para definir los módulos.

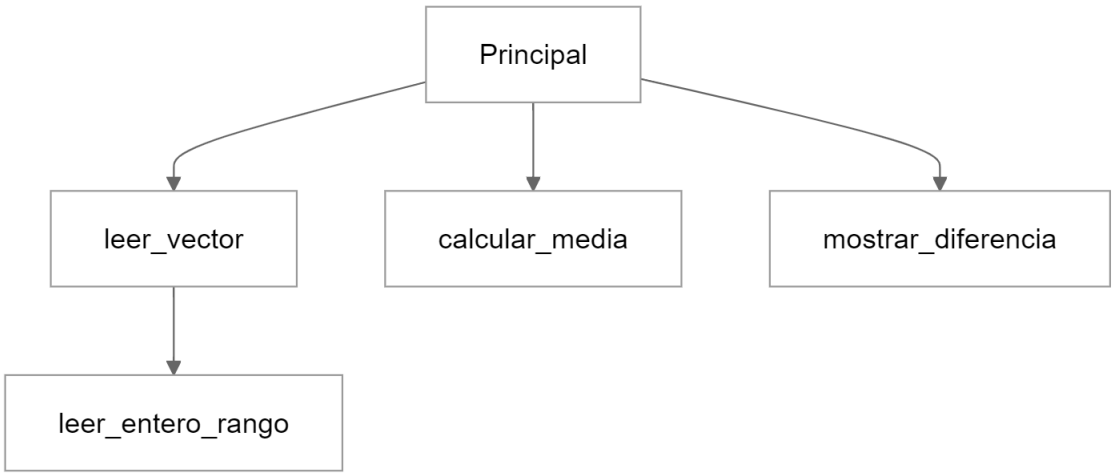
Otra alternativa es:



Esta situación es la que más claramente recoge el patrón E-P-S, separando la interfaz del proceso, pero necesitaría una estructura de datos vector adicional para sustentar las diferencias, siendo este el vector recorrido a la salida para mostrar los resultados.

La decisión del número de módulos y su estructura queda fuera de los conocimientos básicos de programación, pero se debe tener en cuenta como norma general, si aumenta el número de módulos disminuye la complejidad de cada módulo individual, pero se complican las tareas de comunicación entre ellos. Desde un punto de vista práctico, si están disponibles módulos ya implementados que puedan reutilizarse (para recorrido, búsqueda en vectores, lectura de datos con restricciones), se pueden incluir en el diseño de la arquitectura de la solución para reducir el tiempo de implementación.

Para este ejemplo se optará por el primero de los modelos, pero puesto que se ha de leer el tamaño del vector y se pueden reutilizar una función de `leer_entero_rango` para la lectura del valor del tamaño del vector que deberá estar entre 1 y 25.

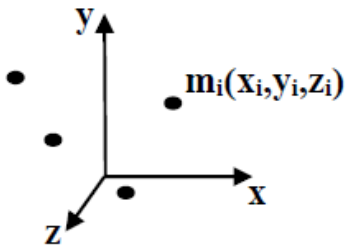



```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#define MAX 24

/* Nuevos tipos de datos */
typedef float tipo_vector[MAX];

/* Prototipos de funciones */
void leer_entero_rango(int a,int b,int *m);
void leer_vector(tipo_vector v, int *n);
float calcular_media(tipo_vector v, int n);
void escribir_inferior_media(float media, tipo_vector v, int n);

/* Definiciones de funciones */
int main(){
    char c;
    float media;
    tipo_vector v;
    int n;

    do{ system("cls");
        printf("TEMPERATURAS POR DEBAJO DE LA MEDIA\n");
        printf("==================================\n\n");
        leer_vector(v,&n);
        media=calcular_media(v,n);
        escribir_inferior_media(media,v,n);
        printf("\n\nDesea efectuar una nueva operación (s/n)? ");
        c=toupper(getch());
    }while(c!='N');
    return 0;
```

```
}

void leer_vector(tipo_vector v,int *n){
        int i;
        printf("Introduzca num. de datos:\n");
        leer_entero_rango(1,MAX,n);
        printf("Introduzca temperaturas: \n");
        for(i=0;i<*n;++i){
                printf("\t temperatura[%d]: ",i+1);
                scanf(" %f",&v[i]);
        }
}

void leer_entero_rango(int a,int b,int *m){
        int aux;
        if(a>b){
                aux=a;
                a=b;
                b=aux;
        }
        do{         printf("\tIntroduzca entero [%d,%d]: ",a,b);
                        scanf(" %d",m);
        }while((*m<a)||(*m>b));
}

float calcular_media(tipo_vector v,int n){
        int i;
        float suma;
        suma=0;
        for(i=0;i<n;++i)
                suma+=v[i];
        return(suma/n);
}

void escribir_inferior_media(float media, tipo_vector v, int n){
        int i;
        int cont;
        printf("\nListado de temperaturas inferiores a la media (%.2f)\n",
        media);
        cont=0;
        for(i=0;i<n;++i)
                if(v[i]<media){
                        printf("\tv[%d] = %10.2f\n",i+1,v[i]);
                        cont++;
                }
        if(cont==0)
                printf("No hay ninguna");
}
```

PROBLEMA: construir un programa interactivo para gestionar un sistema de partículas, que permita las siguientes opciones que se puedan seleccionar a través de un menú, con un máximo de 100 partículas:

1. Insertar una nueva partícula, leyendo por teclado su masa mi y su posición espacial (x_i, y_i, z_i). Dos partículas no pueden ocupar la misma posición espacial.
2. Escribir en pantalla un listado de todas las partículas del sistema, clasificadas por valores decrecientes de masa.
3. Eliminar una partícula, dando su número de orden en el sistema de partículas clasificadas por valores decrecientes de masa.
4. Escribir en pantalla la masa total del sistema y la posición del centro de masas.
5. Finalizar la ejecución del programa.



Análisis De cada partícula tenemos que conocer su **masa** y su **posición** en el espacio que viene definida por tres reales, en todo momento se debe saber las partículas **válidas**. Puesto que tenemos un menú habrá que saber la **opción** elegida por el usuario para realizar la operación oportuna. Como resultado de las operaciones se obtendrá el **centro de masas** y si es preciso los **mensajes** que indican que no hay partículas y que no caben.

Diseño preliminar

Modelo de datos

Debemos utilizar un conjunto de estructuras de datos para soportar la información en concreto **4 vectores** uno para las coordenada X, otro para Y, otro para la Z y otro vector para el centro de masas. Hay que definir un MAX de partículas permitidas. Todos los valores son reales.

Se debe llevar un *contador* de partículas válidas para por una parte conocer los límites de recorrido y por otra no permitir más partículas más allá de la capacidad del vector.

Por tanto tenemos un `tipo_vector` de reales con capacidad 100, valor acordado en análisis.

```
#define MAX 100
...

typedef float tipo_vector[MAX];
....
tipo_vector m;     /* masas de las partículas       */
tipo_vector x,y,z; /* coordenadas de sus posiciones */
int n;             /* Numero de partículas          */
...
```

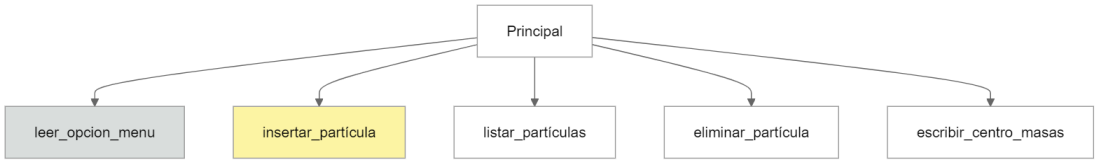
Por tanto el modelo de datos son cuatro vectores de reales y un entero que contendrá el número de partículas. Este modelo veremos más adelante como puede mejorarse utilizando vectores multidimensionales o registros.

Lo que si está claro como requisito del análisis es que estas partículas deben mostrarse "ordenadas" según su masa.

La decisión es ¿Se guardan los valores ordenados sobre el vector? ¿Se ordenan cuando haya que mostrarlos? Esta decisión condiciona el diseño de los módulos y la complejidad de cada uno de ellos.

Diseño arquitectónico

Se trata de un programa dirigido por menú cada opción del menú es una tarea diferente (primer nivel de estructuración) de módulos.



`leer_opcion_menu` es básicamente una operación de entrada/salida que muestra al usuario sus alternativas y lee y trasfiere al resto de módulos la decisión del usuario.

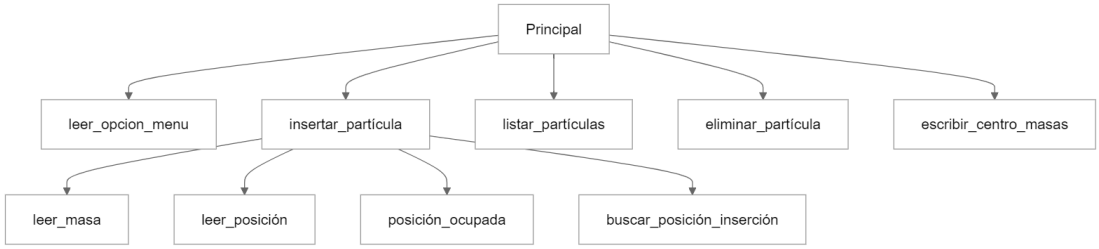
Si las partículas están ordenadas por masa, `listar_partículas` y `escribir_centro_masas` son módulos ligados al recorrido del vector y mostrar en pantalla o bien ir acumulando los valores de la masa considerando su posición.

Si no están ordenadas, `listar_partículas` implica una ordenación. En este caso se opta por hacer una **inserción ordenada** por masa, pero tenemos dos alternativas:

- Insertar al final y usar un algoritmo de clasificación interna (estos algoritmos se estudiarán más adelante).
- Como solo hay un criterio de clasificación, al introducir una nueva partícula, ésta no se añade al final, sino que se inserta en el lugar correspondiente de acuerdo con el valor de su masa.

Por tanto para la complejidad de la ordenación se quedaría en `insertar_partícula` que de forma general conlleva:

- Comprobar que hay espacio suficiente para insertarla (`n<max`).
- Leer datos nueva partícula: `nm`,`nx`,`ny`,`nz`
- Localizar la posición de inserción (`i`).
- Desplazar los elementos i..n una posición hacia delante, empezando por el último (¡¡en los 4 vectores!!).
- Copiar los datos del nuevo elemento en la posición i-ésima de cada vector.
- `n=n+1`



Diseño detallado y comunicación entre módulos

A continuación se muestra en forma de tabla como se comunican los módulos, teniendo en cuenta que no se usan variables globales, con lo que si se manipula (lee o escribe) sobre el sistema de partículas es necesario manejar el modelo de datos completo (4 vectores y tamaño del vector)

Nombre módulo	Tipo de parám.	Nombre parám.	Tipo datos
Módulo_principal			
leer_opción_menú	S	c	carácter
insertar_partícula	E/S	m,x,y,z	tipo_vector
	E/S	n	entero
leer_masa	S	m	real
leer_posición	S	x,y,z	real
posición ocupada	E/S	x,y,z	tipo_vector
	E	n	entero
	E	nx,ny,nz	real
	S		lógico
buscar_posición_inserción	E/S	m	tipo_vector
	E	n	entero
	E	nm	real
	S		entero
listar_partículas	E/S	m,x,y,z	tipo_vector
	E	n	entero
eliminar_partícula	E/S	m,x,y,z	tipo_vector
	E/S	n	entero
escribir_centro_masas	E/S	m,x,y,z	tipo_vector
	E	n	entero

Si se trata de parámetros de entrada/salida será necesario pasarlos por referencia, pero hay que recordar que todos los vectores en C se pasan por referencia.

```
void leer_opcion_menu(char *c);
void insertar_particula(tipo_vector m, tipo_vector x, tipo_vector y,tipo_vector z, int *n);
void leer_masa(float *m);
void leer_posicion(float *x, float *y, float *z);
int posicion_ocupada(tipo_vector x, tipo_vector y, tipo_vector z, int n, float nx, float ny, float nz);
int buscar_posicion_insercion(tipo_vector m, int n, float nm);
void eliminar_particula(tipo_vector m, tipo_vector x, tipo_vector y, tipo_vector z, int *n);
void listar_particulas(tipo_vector m, tipo_vector x, tipo_vector y, tipo_vector z, int n);
void escribir_centro_masas(tipo_vector m, tipo_vector x, tipo_vector y, tipo_vector z, int n);
```

Diseño detallado

`insertar_partícula`

Estrategia de inserción preservando el sistema de partículas clasificadas por valores decrecientes de masa: 1. Comprobar que hay sitio en la estructura de datos para registrar una nueva partícula (n<max). 2. Leer datos de nueva partícula (nm,nx,ny,nz), validando que la posición no está ocupada.

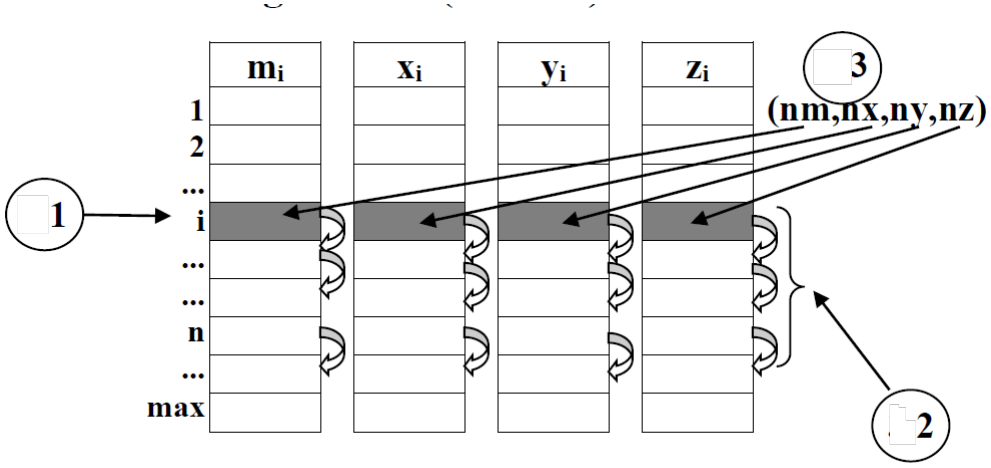
- Leer los datos de la – *leer_masa*
- Leer los datos de la posición, *leer_posición*
- Si la posición esta ocupada deshechar la posición y pedir otra (*posicion_ocupada(..)*)

3. Insertar nueva partícula en estructura de datos, implica buscar la posición según la masa

- Buscar la posición de inserción (i) , posición de la primera partícula de masa estrictamente menor. *buscar_posición_inserción*
- Desplazar todas las partículas desde la posición i hasta la n una posición hacia abajo, recorrido secuencial parcial en orden inverso (n=i).
    - Implica un bucle que va de n a i, con paso -1

```
m[j+1]=m[j]
x[j+1]=x[j]
y[j+1]=y[j]
z[j+1]=z[j]
```

- Copiar los datos de la nueva partícula en la posición i-ésima de los 4 vectores paralelos.
- Incrementar en una unidad el nº de partículas registradas (`n=n+1`).



`Posición_ocupada`

Comprobar que la posición de la nueva partícula a insertar (nx,ny,nz) no está ocupada por alguna de las n partículas registradas, conlleva recorrido secuencial paralelo de los vectores x, y, z, comparando en cada iteración la posición de cada partícula con la de la nueva partícula, repetición controlada por contador de iteraciones con dos condiciones de salida:

- El contador de iteraciones vinculado al índice de los vectores es mayor que el nº de partículas registradas.
- Se encuentra una partícula i que está en la misma posición espacial. Esta segunda condición de salida la implementaremos con una variable lógica inicializada a falso.

`Buscar_posición_inserción`

Localiza la posición de la primera partícula con masa estrictamente menor que la de la nueva partícula a insertar (nm), es decir, recorrido secuencial del vector de masas, comparando en cada iteración la masa de cada partícula con la de la nueva partícula, por tanto, repetición controlada por contador de iteraciones con dos condiciones de salida:

- El contador de iteraciones vinculado al índice del vector es mayor que el nº de partículas registradas. Esa será la posición de inserción.
- Se encuentra una partícula i con masa estrictamente menor que la masa de la nueva partícula. Esta segunda condición de salida la implementaremos con una variable lógica inicializada a falso.

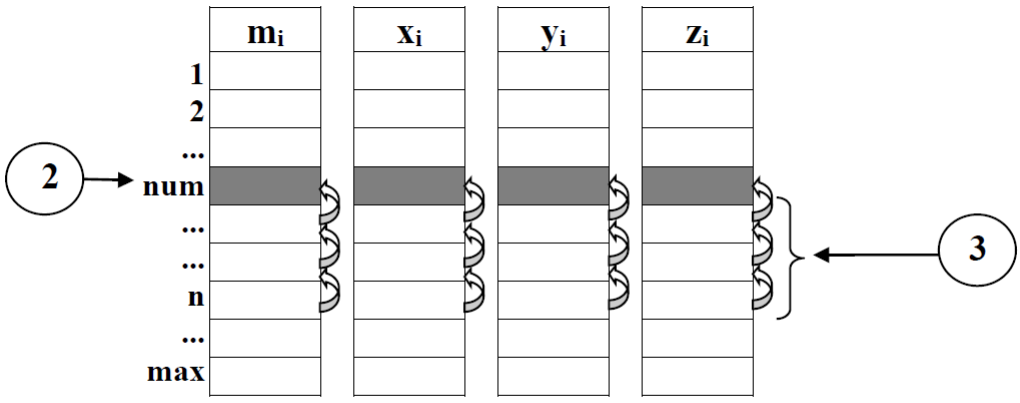
`Eliminar_partícula`

Estrategia de eliminación de una partícula en estructura de datos (preservando la clasificación por masa decreciente), se ha optado por el método de eliminación más simple, unicamente indicando el número de la partícula a eliminar, es decir indicado su posición sobre el vector:

1. Comprobar que hay partículas registradas ($n \neq 0$).
2. Leer nº de orden de la partícula a eliminar, comprobando que está en los límites válidos:
   - Leer `num` con ($num > 0$) y ($num \leq n$)
3. Desplazar todas las partículas desde la posición num+1 hasta la n una posición hacia arriba, se trata de un recorrido secuencial parcial, con un bucle desde `i=num+1` hasta `n`.

```
m[i-1]=m[i]
x[i-1]=x[i]
y[i-1]=y[i]
z[i-1]=z[i]
```

4. Decrementar en una unidad el nº de partículas registradas (n=n-1).



`listar_partículas`

Recorrido secuencial en paralelo de los cuatro vectores (repetición controlada por contador de iteraciones): en cada iteración i se escriben en pantalla los datos almacenados en la posición i de dichos vectores.

`escribir_centro_masas`

Recorrido secuencial en paralelo de los cuatro vectores (repetición controlada por contador de iteraciones): en cada iteración i se calculan cuatro acumuladores de suma correspondientes a la masa total y a los numeradores de las coordenadas del centro de masas.

Implementación

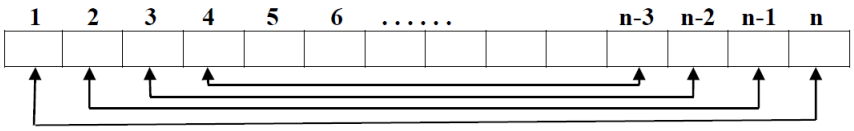
```
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#define MAX 10      /* Numero máximo de partículas ponemos 10 para facilitar la prueba
                    despues se cambia por el valor deseado 100*/

/* Nuevos tipos de datos */
typedef float tipo_vector[MAX];

/* Prototipos de funciones */
void leer_opcion_menu(char *c);
void insertar_particula(tipo_vector m, tipo_vector x, tipo_vector y,tipo_vector z, int *n);
void leer_masa(float *m);
void leer_posicion(float *x, float *y, float *z);
int posicion_ocupada(tipo_vector x, tipo_vector y, tipo_vector z, int n, float nx, float ny, float nz);
int buscar_posicion_insercion(tipo_vector m, int n, float nm);
void eliminar_particula(tipo_vector m, tipo_vector x, tipo_vector y, tipo_vector z, int *n);
void listar_particulas(tipo_vector m, tipo_vector x, tipo_vector y, tipo_vector z, int n);
void escribir_centro_masas(tipo_vector m, tipo_vector x, tipo_vector y, tipo_vector z, int n);

int main(){
    char c;             /* Seleccion opcion de menu      */
    tipo_vector m;      /* masas de las particulas       */
    tipo_vector x,y,z; /* coordenadas de sus posiciones */
    int n;              /* Numero de particulas          */

    n=0;
    do{ system("cls");
        leer_opcion_menu(&c);
        switch(c){
        case '1': insertar_particula(m,x,y,z,&n);
                  break;
        case '2': eliminar_particula(m,x,y,z,&n);
                  break;
        case '3': listar_particulas(m,x,y,z,n);
                  break;
        case '4': escribir_centro_masas(m,x,y,z,n);
                  break;
        case '5': printf("\n\nFIN EJECUCION\n");
                  getch();
                  break;
        default:  printf("\a");
                  break;
        }
    }while(c!='5');
    return 0;
}

void leer_opcion_menu(char *c){
        system("cls");
        printf("SISTEMA DE PARTICULAS\n");
        printf("=====================\n\n");
        printf("\t1.- Insertar nueva particula\n");
        printf("\t2.- Eliminar particula\n");
        printf("\t3.- Listar particulas\n");
        printf("\t4.- Calcular centro de masas\n");
        printf("\t5.- Terminar ejecucion\n");
        printf("\t\tIntroduzca opcion: ");
        *c=getch();
}

void insertar_particula(tipo_vector m, tipo_vector x,
        tipo_vector y, tipo_vector z, int *n){
```

```
/* Módulo para insertar partículas; tras insertar una partícula  */
/* pregunta si se desean introducir mas partículas.              */
/* Las particulas se insertan en orden de masa, de mayor a menor */
    int i;                          /* posición de inserción    */
    int j;                          /* contador de bucle        */
    float nm, nx, ny, nz;           /* datos de nueva partícula */
    char c;                         /* respuesta a pregunta     */

    do{ if(*n==MAX){
            printf("\n\nVector lleno");
            c='N';
            getch();
        }
        else{ printf("\n\nIntroduzca datos particula:");

            leer_masa(&nm);

            do{ leer_posicion(&nx,&ny,&nz);
            }while(posicion_ocupada(x,y,z,*n,nx,ny,nz));

            i=buscar_posicion_insercion(m,*n,nm);

            for(j=*n;j>i;--j){
                m[j]=m[j-1];
                x[j]=x[j-1];
                y[j]=y[j-1];
                z[j]=z[j-1];
            }
            m[i]=nm;
            x[i]=nx;
            y[i]=ny;
            z[i]=nz;
            (*n)++;
            printf("\nDesea introducir nueva particula (S/N): ");
            c=toupper(getch());
        }
    }while((c=='S')&&(*n<MAX));
}

void leer_masa(float *m){
    do{ printf("\n\tMasa: ");
        scanf(" %f", m);
    }while(*m<=0);
}

void leer_posicion(float *x, float *y, float *z){
        printf("\tPosicion:\n");
        printf("\t\tx: ");
        scanf(" %f", x);
        printf("\t\ty: ");
        scanf(" %f", y);
        printf("\t\tz: ");
        scanf(" %f", z);
}

int posicion_ocupada(tipo_vector x, tipo_vector y,
        tipo_vector z,int n, float nx, float ny, float nz){
/* Devuelve verdadero si en la nueva posición  ya hay  una partícula */
    int res;        /* resultado de la comprobación */
    int i;          /* contador de bucle            */

    res=0;
    i=0;
    while((i<n)&&(!res)){
        if((x[i]==nx)&&(y[i]==ny)&&(z[i]==nz))
            res=1;
        else i++;
    }
    return(res);
}
```

```
int buscar_posicion_insercion(tipo_vector m, int n, float nm){
/* Devuelve la posición del primer elemento de masa menor a nm        */
    int i;          /* contador de bucle   */
    int enc;        /* indicador de suceso */

    i=0;
    enc=0;
    while((i<n)&&(!enc)){
        if(m[i]<nm)
            enc=1;
        else ++i;
    }
    return(i);
}

void eliminar_particula(tipo_vector m, tipo_vector x,
        tipo_vector y, tipo_vector z, int *n){
    int num;        /* posicion de la particula a borrar */
    int i;          /* contador de bucle                 */
    char c;         /* respuesta a pregunta              */

    do{ if(*n==0){
        printf("\n\nNo hay ninguna partícula");
        c='N';
        getch();
        }
        else{
            do{ printf("\n\nIntrod. num. partícula(1-%d): ",*n);
                scanf(" %d", &num);
            }while((num<0)||(num>*n));
            for(i=num;i<*n;++i){
                m[i-1]=m[i];
                x[i-1]=x[i];
                y[i-1]=y[i];
                z[i-1]=z[i];
            }
            (*n)--;
            printf("Desea eliminar otra particula (S/N): ");
            c=toupper(getch());
        }
    }while((c=='S')&&(*n>0));
}

void listar_particulas(tipo_vector m, tipo_vector x,
        tipo_vector y, tipo_vector z, int n){
    int i;           /* contador de bucle */

    system("cls");
    printf("LISTADO DE PARTICULAS\n");
    printf("=====================\n\n");
    printf("Numero       masa          x          y          z\n");
    for(i=0;i<n;++i)
        printf("   %3d %10.2f %10.2f %10.2f %10.2f\n",
            i+1,m[i],x[i],y[i],z[i]);
    getch();
}

void escribir_centro_masas(tipo_vector m, tipo_vector x,
        tipo_vector y, tipo_vector z, int n){
    int i;          /* contador de bucle               */
    float cx,cy,cz; /* coordenadas del centro de masas  */
    float masa;     /* masa total sistema de partículas */

    cx=0;
    cy=0;
    cz=0;
    masa=0;
    if(n==0){
        printf("\n\nNinguna particula introducida");
        getch();
    }
```

```
    else
    {
        for(i=0;i<n;++i){
            cx+=m[i]*x[i];
            cy+=m[i]*y[i];
            cz+=m[i]*z[i];
            masa+=m[i];
        }
        cx=cx/masa;
        cy=cy/masa;
        cz=cz/masa;
        printf("\n\nCentro de masas: (%10.2f,%10.2f,%10.2f)",
            cx,cy,cz);
        printf("\nMasa: %10.2f",masa);
        getch();
    }
}
```

PROBLEMA: Invertir orden de vector



Diseño Recorrido secuencial del vector (repetición controlada por contador de iteraciones): en cada iteración se intercambian dos elementos:

`Intercambiar v[i] y v[n-i-1]`

con i variando de n las ultimas iteraciones intercambian lo primero de nuevo así que hay que llegar hasta la mitad. (MAX/2)

```
    int i, x;

    for (i = 0; i < n / 2; i++) {
        x = v[i];
        v[i] = v[n - i - 1];
        v[n - i - 1] = x;
    }
}

int main() {

    tipo_vector vector;
    for(int h=0;h<MAX;++h) vector[h]=h;

    printf("Vector original:\n");
    for (int i = 0; i < MAX; i++) {
        printf("%d ", vector[i]);
    }
    printf("\n");

    invertir(vector, MAX);

    printf("Vector invertido:\n");
    for (int i = 0; i < MAX; i++) {
        printf("%d ", vector[i]);
    }
```

```
    printf("\n");

    return 0;
}
```

Vectores multidimensionales

Representan grupos de datos del mismo tipo que se encuentran ordenados según un conjunto de índices.

Ejemplos:

- Distancias kilométricas entre una serie de ciudades:

Albacete	Alicante	Almería	. . . .	Zaragoza
Albacete				
Alicante				
Almería				
. . . .				
Zaragoza				

- Coeficientes de un sistema lineal de m ecuaciones con n incógnitas:

$a_{11} * x_1 + a_{12} * x_2 + ... + a_{1n} * x_n = c_1$
$a_{21} * x_1 + a_{22} * x_2 + ... + a_{2n} * x_n = c_2$
. . . .
$a_{m1} * x_1 + a_{m2} * x_2 + ... + a_{mn} * x_n = c_m$

- Un sistema de monitorización de la temperatura de invernaderos donde se registra en diferentes localizaciones (bandas, centro del invernadero) a lo largo del tiempo en varias alturas para cada ubicación relevante. Necesitamos tres dimensiones:
- La primera dimensión representa la ubicación.
- La segunda dimensión representa la altura de la medición.
- La tercera dimensión representa el tiempo.

Los **vectores bidimensionales** son cuadro de valores del mismo tipo de datos ordenados en filas y columnas (cada valor se referencia mediante dos índices: número de fila y número de columna).

	1	2	3	. . .	m
1					
2					
3					
. . .					
n					

Se suelen llamar:

- **matrices**. De ciencia y tecnología que es el más habitual.
- **tablas**. De economía y humanidades, aunque no se debe confundir con otro nuevo tipo estructurado que esta surgiendo con fuerza en lenguajes de programación que se utilizan en las ciencias de datos que de denomina **dataframe**, donde el tipo base puede variar entre columnas.

Los índices indican un orden entre criterios y el primero es la fila y el segundo la columna. Es decir `[1,4]` representa el elemento de la primera fila para la columna 4.

Esta estructura de datos se puede extender en número de dimensiones. Si tiene tres estamos pensado en una estructura espacial tipo cubo. En sí mismo C no impone un límite específico en el número de dimensiones para un array. Sin embargo, hay algunos límites prácticos y consideraciones que pueden afectar la capacidad de utilizar vectores de muchas dimensiones en un programa.

En primer lugar, el límite práctico está determinado por la memoria disponible en el sistema. Cada dimensión adicional aumenta significativamente la cantidad total de memoria necesaria para almacenar el vector. En segundo lugar, la legibilidad del código y la mantenibilidad pueden verse afectadas negativamente a medida que aumenta el número de dimensiones. El código puede volverse difícil de entender y trabajar con un número muy grande de índices.

Puesto que son la extensión a diversas dimensiones de un vector su declaración se hace:

- Declaración implícita

```
tipo_base nombre_variable[N1][N2]..[Ni];
```

N1, N2,..., Ni son los tamaños de las respectivas dimensiones del "array". En C la indexación empieza en 0 y acaba en N-1

```
float  m[10][20];        /* matriz 10*20 datos reales */
int cubo[5][10][8];        /* "array" 5*10*8 enteros    */
char pagina[25][80];        /* tabla 25*80 caracteres     */
```

- Declaración explícita

```
typedef tipo_base tipo_array[N1]...[Ni];
        tipo_array  nombre_variable;
```

Ejemplos:

```
/* tipos definidos por el usuario */
typedef float  tipo_matriz[10][20];
typedef int tipo_cubo[5][10][8];
typedef char tipo_pagina[25][80];

/* declaración de variables */
tipo_matriz m;
tipo_cubo cubo;
tipo_pagina pagina;
```

- También es posible construir un tipo matriz como un vector de vectores.

```
typedef float tipo_fila[20];
typedef tipo_fila tipo_matriz[10];
```

Las operaciones a realizar con los vectores multidimensionales son las mismas que con los de una dimensión, pero indicando todos los índices. A partir de aquí veremos ejemplo de estas operaciones, pero solo para matrices.

- Acceso a **elementos individuales** del "array": a través de sus índices (respetando siempre los límites de cada dimensión: 0..Ni-1):
  - Asignación:

```
m[0][1] = 1.5;
```

  - Entrada/salida:

```
scanf(" %f", &m[2][2]);
printf("%.0f",m[1][5]);
```

- Expresiones:

```
m[1][1]=m[1][1]+1.0;
(m[1][5] > m[1][4])
```

- **Acceso secuencial** (recorrido del vector): acceso a todos los elementos de 1 en 1 empezando por el primero y siguiendo el orden físico de almacenamiento, con una repetición para cada dimensión (estructuras repetitivas anidadas).

```
#define        nf 10
#define        nc 15
typedef        float tipo_matriz[nf][nc];
. . .
int i,j;
tipo_matriz m;
```

- Recorrido secuencial por filas: (C almacena la matriz en memoria por filas)

```
i=0;
while (i<nf)
{
        j=0;
        while (j<nc)
    {
                /* procesar elemento m[i][j] */
                j=j+1;
        }
        i=i+1;
}
```

- Recorrido secuencial por columnas:

```
j=0;
while (j<nc)
{
        i=0;
        while (i<nf)
    {
                /* procesar elemento m[i][j] */
                i=i+1;
        }
        j=j+1;
}
```

- Entrada/salida: E: tipo de base float

```
 for(i=0; i<nf; ++i){
        for (j=0; j<nc; ++j){
                        printf("m[%d,%d]? ", i+1, j+1);
                        scanf(" %f", &m[i][j]);
    }
}

for (i=0; i<nf; ++i){
        for (j=0; j<nc; ++j){
                  printf("m[%d,%d]= %.1f",i+1,j+1,m[i][j]);
        }
}
```

- Copia o duplicado: tipo_matriz `m2`;

```
        . . .
for (i=0; i<nf; ++i)
{
        for (j=0; j<nc; ++j)
    {
                m2[i][j]=m[i][j];
        }
}
```

- Búsqueda secuencial: (1ª ocurrencia `enc=0`; elemento: `x`)

```
i=0;
enc=0;
while ((i<nf)&&(! enc))
{
        j=0;
        while ((j<nc)&&(! enc))
    {
                if (m[i][j]==x)         enc=1;
                else            j=j+1;
    }
    if (! enc)          i=i+1;
}
if (enc)
        printf("Encontrado");
else        printf("No encontrado");
```

- Paso de como parámetros
  - De componentes individuales (parámetros reales):

    ```
    m[i][j]  // por valor
    &m[i][j] // por dirección
    ```

  - Del vector completo: en C los vectores se pasan por dirección (no existe el paso de vectores por valor).

Como **Parámetro real** `m`. l nombre del representa la dirección del primer elemento (no es necesario poner `&` delante del identificador del vector). El **parámetro formal**, al igual que con los unidimensionales hay diversas posibilidades de declaración:

```
tipo_matriz m               /* usando el tipo definido          */
tipo_base m[][nc]        /* declarando directamente */
                                    /* el parámetro   formal       */
                                    /* como una matriz. No es    */
                                    /* necesario declarar el        */
                                    /* tamaño del primer índice */
tipo_base (*m)[nc]        /* declarando el parámetro  */
                                    /* formal como un puntero a*/
                                    /* un vector de datos del tipo*/
                                    /* base (puntero a filas)             */
```

En los dos primeros casos, el acceso dentro de la función llamada a un elemento del vector se realiza de la forma convencional `m[i][j]`. Nótese que dentro de la función llamada, el parámetro formal que representa la matriz (que siempre se pasa por dirección) se manipula como si fuera un parámetro de entrada.

Usando la notación de punteros, el elemento `m[i][j]` es `*(*(m+i)+j)`

PROBLEMA

Un número silla en el contexto de una matriz se refiere a un elemento que es el valor mínimo en su fila y, simultáneamente, el valor máximo en su columna. Dicho de otra manera, un número silla es aquel que no es superado por ningún otro número en su fila ni en su columna en una matriz dada.

Consideremos una matriz genérica M con elementos $M_{i,j}$ donde i representa la fila y j representa la columna.

Un número silla $M_{i,j}$ cumple con las siguientes condiciones:

- $M_{i,j}$ es el valor mínimo en la fila i.
- $M_{i,j}$ es el valor máximo en la columna j.
- Aquí hay un ejemplo sencillo para ilustrar el concepto:

5	3	2
7	9	8
1	4	6

En esta matriz, el número 7 es un número silla porque es el valor máximo en la primera columna y el valor mínimo en la segunda fila. En términos matemáticos,

$M_{2,1} = 7$ es un número silla en esta matriz.

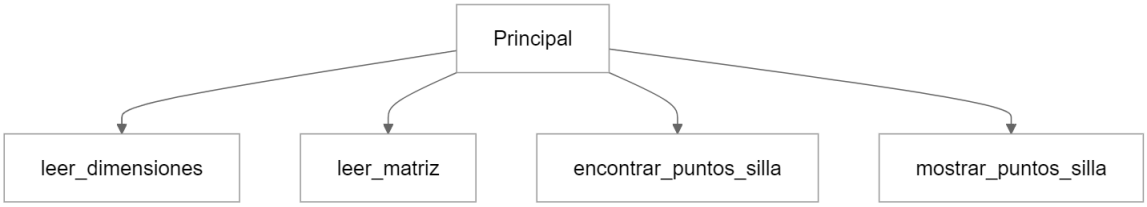
Es importante señalar que no todas las matrices tienen un número silla, y una matriz puede tener más de un número silla. El concepto de número silla es útil en problemas de optimización y análisis de matrices.

Construir una función en C que dada una matriz de NxM elementos, cuyos valores también son argumento de la función, devuelva la cantidad de números silla que tiene, así como la lista de sus coordenadas (índices de la matriz).

Diseño de datos

```
typedef int Coordenadas[MAXFILA*MAXCOL][2];  // Usamos un alias para representar las coordenadas
typedef int Matriz[MAXFILA][MAXCOL];  // Usamos un alias para la matriz
```

Diseño modular Se usa el patrón E-P-S




```c
#include <stdio.h>

#define MAXFILA 10
#define MAXCOL 10

typedef int tipo_coordenadas[MAXFILA*MAXCOL][2];
typedef int tipo_matriz[MAXFILA][MAXCOL];



int encontrar_puntos_silla(tipo_matriz m, int f, int c, tipo_coordenadas silla);
int leer_dimensiones(int *f, int *c);
void leer_matriz(tipo_matriz v, int f, int c);
void mostrar_puntos_silla(tipo_matriz m, tipo_coordenadas silla, int cuantos);



int main() {
    int totSilla, f, c, maxsilla;
    tipo_coordenadas puntos;

    tipo_matriz matriz = {
        {5, 3, 2},
        {7, 9, 8},
        {1, 4, 6}
    };
    f=c=3;
    maxsilla=9;

   // maxsilla=leer_dimensiones(&f,&c);
  //leer_matriz(matriz,f,c);

    totSilla= encontrar_puntos_silla(matriz, f, c, puntos);

    mostrar_puntos_silla(matriz, puntos, totSilla);

    return 0;
}

int leer_dimensiones(int *f, int *c){
        do{        printf("\tNumero de filas (1-%d): ",MAXFILA);
                   scanf(" %d", f);
        }while((*f<1)||(*f>MAXFILA));
        do{        printf("\tNumero de columnas (1-%d): ",MAXCOL);
                   scanf(" %d", c);
        }while((*c<1)||(*c>MAXCOL));
    return ((*c)*(*f));
}

void leer_matriz(tipo_matriz v, int f, int c){
        int i,j;
        for(i=0;i<f;++i)
                for(j=0;j<c;++j){
                        printf("\t\tv[%d,%d]: ",i+1,j+1);
                        scanf(" %d", &v[i][j]);
                }
}

int encontrar_puntos_silla(tipo_matriz m, int f, int c, tipo_coordenadas silla) {
```

```
    int k, numSilla = 0;

    for (int i = 0; i < f; ++i) {
        for (int j = 0; j < c; ++j) {
            int actual = m[i][j];
            int esSilla = 1;  // Asumimos que el elemento actual es un punto silla

            // Verificamos si es el mínimo en su fila
            k=0;
            while (k < c && esSilla) {
                if (actual > m[i][k]) {
                    esSilla = 0;  // No es el mínimo en su fila
                }
                k++;
            }
            k=0;
            while (k < f && esSilla) {
                if (actual < m[k][j]) {
                    esSilla = 0;  // No es el máximo en su columna
                }
                k++;
            }
            if (esSilla) {
                silla[numSilla][0] = i;
                silla[numSilla][1] = j;
                (numSilla)++;
            }
        }
    }
    return (numSilla);
}



void mostrar_puntos_silla(tipo_matriz m, tipo_coordenadas silla, int cuantos){

    printf("El número de puntos silla es: %d\n", cuantos);
    printf("Sus posiciones y su valor son:\n");
    for (int i = 0; i < cuantos; ++i) {
        printf("(%d, %d) = %d\n", silla[i][0], silla[i][1],
        m[silla[i][0]][silla[i][1]]);
    }


}
```

PROBLEMA

Construir un programa que resuelva un sistema lineal de 3 ecuaciones con 3 incógnitas mediante la regla de Cramer y presente en pantalla los resultados, dados por teclado los coeficientes de las incógnitas y los términos independientes de las ecuaciones respectivas

$a_{11}*x_1+a_{12}*x_2+a_{13}*x_3 = c_1$ $a_{21}*x_1+a_{22}*x_2+a_{23}*x_3 = c_2$ $a_{31}*x_1+a_{32}*x_2+a_{33}*x_3 = c_3$

Análisis La regla de Cramer es de un método muy rápido para resolver sistemas, sobre todo, para sistemas de dimensión 2x2 y 3x3. Para dimensiones mayores, los determinantes son bastante más engorrosos.

Un sistema de ecuaciones puede escribirse en forma matricial como:

$AX = B$

donde

- A es la matriz de coeficientes del sistema,

- X es la matriz con las incógnitas,
- B es la matriz con los términos independientes de las ecuaciones.

Para poder aplicar Cramer, la matriz A tiene que ser cuadrada y regular (determinante distinto de 0).

Expresado como determinantes

```
|a11 a12 a13|       |x1|       |c1|
|a21 a22 a23|   *   |x2|  =    |c2|
|a31 a32 a33|       |x3|       |c3|
```

La regla de Cramer establece que la incógnita x_k de la solución del sistema, cuyos coeficientes están en la columna k de A, es

$x_k = |A_k|/|A|$

De esta forma

```
          |c1 a12 a13|          |a11 c1 a13|          |a11 a12 c1|
          |c2 a22 a23|          |a21 c2 a23|          |a21 a22 c2|
          |c3 a32 a33|          |a31 c3 a33|          |a31 a32 c3|
x1  =  -------------    x2  =  -------------    x3  =  -------------
          |a11 a12 a13|          |a11 a12 a13|          |a11 a12 a13|
          |a21 a22 a23|          |a21 a22 a23|          |a21 a22 a23|
          |a31 a32 a33|          |a31 a32 a33|          |a31 a32 a33|
```

Diseño de datos

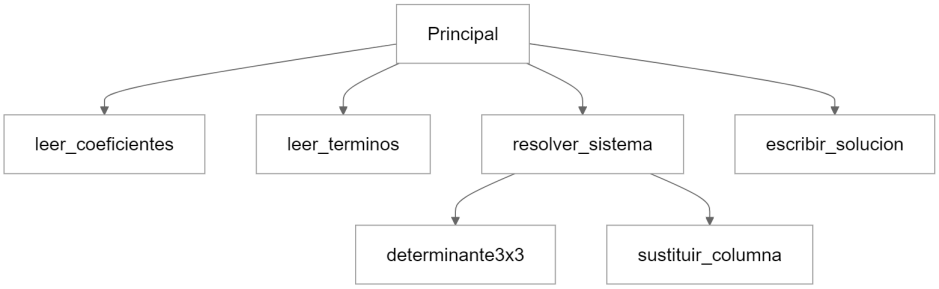
Necesitamos una matriz para los coeficientes y un vector para los resultados, todos con `n=3`

```
typedef float tipo_vector[n];
typedef float tipo_matriz[n][n];
```

Diseño de módulos

Partiendo del patrón E-P-S, vemos que el **Proceso** se basa en el calculo de un derminate 3x3. Por lo tanto es necesaria una función que dada una variable `tipo_matriz` devuelve su determinante. Se resolverán cuatro determinantes, el primero basado en los coeficientes de la matriz entrada y los otros tres sustituyendo secuencialmente la columna 1, 2 y 3 por los coeficientes independientes.

De esta forma la estructura modular será:



```
void leer_coeficientes(tipo_matriz a);
void leer_terminos(tipo_vector c);
int resolver_sistema(tipo_matriz a,tipo_vector c,tipo_vector x);
void sustituir_columna(tipo_matriz a,tipo_vector c,int i,
        tipo_matriz aux);
float determinante(tipo_matriz a);
void escribir_solucion(tipo_vector x);
```

Vectores de caracteres. Cadenas de caracteres: Operaciones con cadenas

En este apartado se presentan las funciones de la biblioteca estándar de C que le ayudan a procesar caracteres, cadenas, líneas de texto y bloques de memoria. El capítulo discute las técnicas utilizadas para desarrollar editores, procesadores de texto, software de diseño de páginas y otros tipos de software de procesamiento de texto. En especial el concepto de cadena de carácteres que se fundamenta en una colección de datos similar al vector.

Una **Cadena de caracteres** es estructura de datos formada por una secuencia de caracteres en un orden especificado. Un Carácter es un tipo de dato simple (representa un símbolo del juego de caracteres utilizado por la computadora: (ASCII, EBCDIC, Unicode,...))

Cadena es por tanto una colección estructurada/indexada (está formado por datos de tipos más simples) cuyo conjunto de caracteres que se almacenan en un área contigua de la memoria.

En el lenguaje de programación C, una cadena (*string* en inglés) es un conjunto de caracteres que termina con el carácter nulo ('\0'). Los caracteres dentro de una cadena pueden ser letras, números, caracteres especiales, y el carácter nulo actúa como el delimitador que indica el final de la cadena. Existen dos tipos de cadenas, las constantes cadena y los vectores de caracteres o variables cadena de caracteres (strings)

- **Constantes literales de cadena**, disponibles en prácticamente todos los lenguajes de programación que son literales encerrados entre **comillas**, para C comillas dobles.

Es una secuencia de 0 o más caracteres (normalmente del código ASCII extendido), escrita en una línea de programa y encerrada entre comillas dobles.

```
"Ejemplo de cadena"
""                                  /* Cadena nula          */
"\"¡Hola!\", soy yo"  /* Las comillas se representan         */
                      /* dentro de la cadena como \"         */
"¡Cuidado!\007"       /* caracteres especiales (de control de salida ó         */
"Linea 1\nLinea 2\n"  /* de representación inconfundible)        */
                      /* mediante su código octal, o su        secuencia de escape que es
                      /* independiente del sistema de codificación         */
```

- **Variables de cadena**, que tienen diversas implementaciones (a veces no existe como tipo predefinido y se implementa utilizando elementos más simples). Su implementación es dependiente del lenguaje de programación.

El conjunto de operaciones sobre cadenas es variado y en su mayoría están incorporados en las bibliotecas standar de los lenguajes, van desde buscar una subcadena en una cadena a obtener la longitud de la cadena. Pero podemos manejar estas estructuras de datos para generar nuestras propias operaciones necesarias para nuestro programa, por ejemplo comprobar si una cadena es "palíndroma".

La lista de operaciones habituales es: longitud de cadena, comparación de dos cadenas, concatenación, extracción de subcadena, búsqueda de subcadenas en una cadena. Pero suele haber muchas más.

En C, las cadenas de caracteres se representan mediante **vectores de caracteres**. Aquí hay un ejemplo de cómo se declara y se inicializa una cadena en C:

```c
#include <stdio.h>

int main() {
    // Declaración e inicialización de una cadena
    char miCadena[] = "Hola, mundo!";

    // Imprimir la cadena
    printf("La cadena es: %s\n", miCadena);

    return 0;
}
```

Se puede optar por la **declaración directa** `char nombre_variable[N+1]`, siendo N la longitud de la cadena:

```c
char  nombre[16];        /* cadena de hasta 15 caracteres        */
char direccion[31];        /* cadena de hasta 30 caracteres        */
char texto[2001];               /* cadena de hasta 2000 caracteres        */

/* Se necesita una posición adicional para el centinela fin de cadena
```

O bien se utiliza una **declaración previa** con `typedef`

```c
.... /* defincón de N como cte capacidad de la cadena */
typedef  char  tipo_cadena[N+1];

tipo_cadena  nombre_variable;
...
....
```

Por ejemplo:

```c
/* tipos definidos por el usuario */
typedef char cadena15[16];
typedef char cadena30[31];
typedef char cadena2000[2001];
....

/* declaración de variables        */
cadena15 nombre;
cadena30 direccion;
cadena2000 texto;
```

Si bien para las soluciones de ingeniería no es habitual trabajar con cadenas de caracteres, si que lo es trabajar con cadenas de números como carácter, para poder generar archivos legibles y que después será necesario trasformarlos en números para operar con ellos, basta con recordar los archivos `.csv` que contienen los datos generados por muchas aplicaciones

que se han convertido en un estándar de facto para el intercambio de información, donde los números son tratados como caracteres.

No obstante sólo nos vamos a centrar en listar y ver el funcionamiento con ejemplos sencillos de las librerías de C que tratan con caracteres y cadenas: `stdio.h ctype.h stdlib.h string.h`

Funciones que manejan caracteres

```
#include <ctype.h>
```

Función	tipo devuelto	Descripción
toascii(c)	int	Convierte el valor del argumento a ASCII.
toupper(c)	int	Convierte un carácter a mayúsculas.
tolower(c)	int	Convierte un carácter a minúsculas.
isalpha(c)	int	Devuelve un valor distinto de 0 (cumple la condición) si el argumento es un carácter alfabético.
isdigit(c)	int	Devuelve un valor distinto de 0 si el argumento es un dígito.
islower(c)	int	Devuelve un valor distinto de 0 si el argumento es una letra minúscula.
isspace(c)	int	Devuelve un valor distinto de 0 si el argumento es un espacio en blanco.
isupper(c)	int	Devuelve un valor distinto de 0 si el argumento es una letra mayúscula.
isalnum(c)	int	Devuelve un valor distinto de 0 si el argumento es un carácter alfanumérico.
isascii(c)	int	Devuelve un valor distinto de 0 si el argumento es un carácter ASCII (0-127).
iscntrl(c)	int	Devuelve un valor distinto de 0 si el argumento es un carácter de control.
ispunct(c)	int	Devuelve un valor distinto de 0 si el argumento es un signo de puntuación.
isgraph(c)	int	Devuelve un valor distinto de 0 si el argumento es un carácter ASCII gráfico (hex 0x21-0x7e; octal 041-176).
isodigit(c)	int	Devuelve un valor distinto de 0 si el argumento es un dígito octal.
isxdigit(c)	int	Devuelve un valor distinto de 0 si el argumento es un dígito hexadecimal.

```
  // uso funciones isdigit, isalpha, isalnum, and isxdigit

#include <ctype.h>
#include <stdio.h>

int main(void) {
    printf("%s\n%s%s\n%s%s\n\n", "Según isdigit: ",
    isdigit('8') ? "8 es un " : "8 no es  un ", "dígito",
    isdigit('#') ? "# es un " : "# no es un ", "dígito");

    printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n", "Según isalpha:",
```

```
    isalpha('A') ? "A es un " : "A no es un ", "letra",
    isalpha('b') ? "b es un " : "b no es un ", "letra",
    isalpha('&') ? "& es un " : "& no es un ", "letra",
    isalpha('4') ? "4 es un " : "4 no es un ", "letra");

    printf("%s\n%s%s\n%s%s\n%s%s\n\n", "Según isalnum:",
    isalnum('A') ? "A es un " : "A no es un ", "dígito o letra",
    isalnum('8') ? "8 es un " : "8 no es un ", "dígito o letra",
    isalnum('#') ? "# es un " : "# no es un ", "dígito o letra");

    printf("%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n", "Según isxdigit:",
    isxdigit('F') ? "F es un " : "F no es un ", "dígito hexadecimal",
    isxdigit('J') ? "J es un " : "J no es un ", "dígito hexadecimal",
    isxdigit('7') ? "7 es un " : "7 no es un ", "dígito hexadecimal",
    isxdigit('$') ? "$ es un " : "$ no es un ", "dígito hexadecimal",
    isxdigit('f') ? "f es un " : "f no es un ", "dígito hexadecimal");
}
```

Funciones de entrada/ salida

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

Función	tipo devuelto	Descripción
printf (. . .)	int	se usa la secuencia de salida `%-n1.n2s`, con `n1` anchura mínima del campo, `-` ajustar a la izquierda, `n2` número de caracteres a imprimir
puts (s)	int	Escribe en pantalla una cadena (hasta \0). Después de escribir salta de línea. Ejemplo: puts (cadena);
scanf(. . .)	int	Para leer del teclado una cadena se usa la secuencia de entrada %s. Nótese que por defecto scanf lee palabras.
fgets (s, n, stdin)	char*	Lee una cadena de hasta n-1 caracteres del teclado (entrada estándar), admite espacios en blanco y finaliza cuando se pulsa Intro. Coloca la marca \0 al final de la cadena. Devuelve un valor 0 ó NULL si hay problemas.
fflush (stdin)	int	Vacía el buffer de teclado (entrada estándar) eliminando todos los caracteres contenidos en el mismo. Devuelve EOF si ocurre un error, y 0 en caso contrario.
putchar (int c);	int	Imprime el carácter almacenado en c y lo devuelve como un entero.
puts (const char *s);	int	Imprime la cadena s seguida de un carácter de nueva línea. Devuelve un entero distinto de cero si tiene éxito, o EOF si se produce un error.

Función	tipo devuelto	Descripción
sprintf(char *s, const char* format, ...);	int	Equivalente a printf, pero la salida se almacena en la matriz `s` en lugar de imprimirse en la pantalla. Devuelve el número de caracteres escritos en s, o EOF si se produce un error
sscanf (char *s, const char* format, ...);	int	Equivalente a scanf, pero la entrada se lee de la matriz `s` en lugar del teclado. Devuelve el número de elementos leídos con éxito por la función, o EOF si se produce un error.
getchar (void);	int	Devuelve el siguiente carácter de la entrada estándar como un entero

Ejemplos de utilización de scanf para leer cadenas de caracteres con espacios en blanco (no va precedida por "&" por tratarse de un vector):

`scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]s", nombre);`

Asigna a la variable de cadena nombre los caracteres introducidos por el dispositivo estándar de entrada, y finalizará la lectura cuando aparezca un carácter diferente de los encerrados entre corchetes.

`scanf("%[^\n]s", apellidos);` Los caracteres de dentro de los corchetes se interpretan como los caracteres de finalización de la asignación por teclado a la variable apellidos.

```
#define SIZE 80

typedef char tipo_cadena[SIZE];

void reverse(const char * const sPtr);

int main(void) {
    tipo_cadena sentence= "";

    puts("Escribe una línea de texto:");
    fgets(sentence, SIZE, stdin); // read a line of text

    printf("\n%s", "En el espejo sería");
    reverse(sentence);
    puts("");
}

// recursively outputs characters in string in reverse order
void reverse(const char * const sPtr) {
    // if end of the string
    if ('\0' == sPtr[0]) { // base case
    return;
    }
    else { // if not end of the string
    reverse(&sPtr[1]); // recursion step
    putchar(sPtr[0]); // use putchar to display character
    }
}
```

```
#include <string.h>


int main() {
   char str[5][50], temp[50];
```

```
   printf("Dame 5 palabras: ");

   // Lectura de la cadena de entrada
   for (int i = 0; i < 5; ++i) {
      fgets(str[i], sizeof(str[i]), stdin);
   }

   // Ordenar
   for (int i = 0; i < 5; ++i) {
      for (int j = i + 1; j < 5; ++j) {

         // swapping strings if they are not in the lexicographical order
         if (strcmp(str[i], str[j]) > 0) {
            strcpy(temp, str[i]);
            strcpy(str[i], str[j]);
            strcpy(str[j], temp);
         }
      }
   }
printf("\nEn el orden correcto es: \n");
   for (int i = 0; i < 5; ++i) {
      fputs(str[i], stdout);
   }
   return 0;
}
```

Funciones de manejo de cadenas

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

Función	tipo devuelto	Descripción
strlen(s)	int	Devuelve la longitud dinámica de una cadena. Ejemplo: i=strlen(cadena);
strcat(s1,s2)	char*	Concatena dos cadenas (añade s2 a s1). Ejemplo: strcat(cadena1,cadena2);
strcmp(s1,s2)	int	Compara dos cadenas, y devuelve 0 si son iguales y distinto de cero si son diferentes (un valor negativo si s1<s2, y positivo si s1>s2). Ejemplo: i=strcmp(cadena1,cadena2);
strcmpi(s1,s2)	int	Ejemplo: i=strcmpi(cadena1,cadena2);
strcpy(s1,s2)	char*	Copia una cadena: pasa el contenido del segundo argumento al primero. Ejemplo: strcpy(cadena_d,cadena_f);
strset(s,c)	char*	Asigna a todos los caracteres de la cadena el carácter c (excluyendo el carácter de terminación \0). Ejemplo: strset(cadena,caracter);
strlwr(s)	char *	Convierte las letras mayúsculas de una cadena a minúsculas. Ejemplo: strlwr(cadena);
strupr(s)	char *	Convierte las letras minúsculas de una cadena a mayúsculas. Ejemplo: strupr(cadena);

Función	tipo devuelto	Descripción
strstr(s1,s2)	char *	Busca en la cadena s1 la 1ª ocurrencia de s2 y devuelve un puntero al primer carácter de dicha ocurrencia (NULL si no se encuentra). Ej: p=strstr("Hola esto es","es");

Funciones de conversión cadenas/números

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

Función	tipo devuelto	Descripción
atoi(s)	int	Devuelve el valor convertido a entero de la cadena de entrada y 0 en el caso de que esta no se pueda convertir. Ejemplo: i=atoi(cadena);
atol(s)	long int	Convierte una cadena en un entero largo. Ejemplo: n=atol(cadena);
atof(s)	double	Convierte una cadena en un número real. Reconoce además los caracteres punto decimal y la letra 'E' con y sin signo. Ejemplo: num_real=atof(cadena);
strtod(s,&p)	char *p double	Convierte una cadena a real en doble precisión. También reconoce +INF y –INF, y +NAN y –NAN. Ejemplo: num_real_doble = strtod(cadena, &p);

PROBLEMA La agencia de protección de datos ha dado unos criterios a seguir para anonimizar el DNI. Así, cuando sea necesario incluir el DNI o documento identificativo equivalente en publicaciones, se anonimizará de la siguiente manera:

Se publicarán los dígitos que en el formato del documento correspondiente ocupen la posición cuarta, quinta, sexta y séptima (numerados dichos dígitos de izquierda a derecha); y se evitarán los caracteres alfabéticos. Se sustituirán los caracteres alfabéticos y aquellos numéricos no seleccionados para su publicación por un asterisco (*) en cada posición. Pero al menos deben quedar visibles 4 posiciones (números o letras) .

Ejemplos:

- En el caso de un DNI 12345678X, se publicará como `***4567**`.
- En el caso de un NIE L1234567X, al evitarse los caracteres alfabéticos, se publicará como `****4567*`.
- Si se trata de un Pasaporte ABC123456, al evitarse los caracteres alfabéticos y contenerse solo seis dígitos, se publicará como `*****3456`.
- Cuando sea un Documento de identificación XY12345678AB, se publicará como `*****4567***`.
- Si es un Documento de identificación ABCD123XY, la publicación sería `*****23XY`.

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

typedef char cadena12[12+1];

void anonimizarDNI(cadena12 dni);

int main() {
    char dni1[] = "12345678X";
    char dni2[] = "L1234567X";
    char dni3[] = "ABC123456";
    char dni4[] = "XY12345678AB";
    char dni5[] = "ABCD123XY";

    anonimizarDNI(dni1);
    anonimizarDNI(dni2);
    anonimizarDNI(dni3);
    anonimizarDNI(dni4);
    anonimizarDNI(dni5);

    printf("%s\n", dni1); // Imprime ***4567**
    printf("%s\n", dni2); // Imprime ****4567*
    printf("%s\n", dni3); // Imprime *****3456
    printf("%s\n", dni4); // Imprime *****4567***
    printf("%s\n", dni5); // Imprime *****23XY

    return 0;
}
void anonimizarDNI(cadena12 dni) {
    int posicion, cuentanumeros = 0, cuentavisibles =0;

    cadena12 anonimo;
    for (posicion = 0; dni[posicion] != '\0'; posicion++) {
        if (isdigit(dni[posicion]) && cuentanumeros >= 3 && cuentanumeros <= 6) {
            anonimo[posicion] = dni[posicion];
            cuentanumeros++;
            cuentavisibles++;
        } else
            if (isdigit(dni[posicion])) {
                anonimo[posicion] = '*';
                cuentanumeros++;
            } else
                if (isalpha(dni[posicion])) {
                    anonimo[posicion] = '*';
        }
    }

    if (cuentavisibles<4){
        for (int j=1; j<=4;j++)
            anonimo[posicion-j]=dni[posicion-j];
    }

    anonimo[posicion] = '\0';

    strcpy(dni, anonimo);
}
```

PROBLEMA

Construir dos funciones que permitan la codificación y decodificación de un mensaje de hasta 50 caracteres utilizando el método de Julio César en base a una clave leida por teclado:

El cifrado César consiste en sustituir cada letra del abecedario por una letra desplazada un número determinado de posiciones (clave). Por ejemplo, si desplazamos 1 posición, reemplazaríamos la letra A con la B, la B con la C, y así sucesivamente hasta sustituir la Z por la A

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

typedef char cadena50[50+1];

void codificarCesar(cadena50 mensaje, int clave) {
    int i;
    for(i = 0; i < strlen(mensaje); i++) {
        if(isalpha(mensaje[i])) {
            char base = isupper(mensaje[i]) ? 'A' : 'a';
            mensaje[i] = ((mensaje[i] - base) + clave) % 26 + base;
        } else if(isdigit(mensaje[i])) {
            mensaje[i] = ((mensaje[i] - '0') + clave) % 10 + '0';
        }
    }
}

void decodificarCesar(cadena50 mensaje, int clave) {
    int i;
    for(i = 0; i < strlen(mensaje); i++) {
        if(isalpha(mensaje[i])) {
            char base = isupper(mensaje[i]) ? 'A' : 'a';
            mensaje[i] = ((mensaje[i] - base) - clave + 26) % 26 + base;
        } else if(isdigit(mensaje[i])) {
            mensaje[i] = ((mensaje[i] - '0') - clave + 10) % 10 + '0';
        }
    }
}

int main() {

    cadena50 mensaje = "Hola Mundo! 123";
    int clave = 3;

    printf("Mensaje original: %s\n", mensaje);

    codificarCesar(mensaje, clave);
    printf("Mensaje codificado: %s\n", mensaje);

    decodificarCesar(mensaje, clave);
    printf("Mensaje decodificado: %s\n", mensaje);

    return 0;
}
```

Separación entre interfaz (entrada/salida) e implementación

La utilización de la consola es el medio que hemos visto para la entrada salida, pero se pueden utilizar librerías gráficas que perminan desplegar ventanas o utilizar archivos de texto donde se vuelquen los resultados de las ejecuciones. Si bien todos los ejemplos que se han visto hasta ahora se han centrado en procesos interactivos donde se iteraban a demanda del usuario distintas ejecuciones del programa, esto no tiene porque ser lo habitual. Hay programas que no son interactivos y que leen entradas de sensores y generar acciones de salida no “mostrables” en la pantalla.

En este apartado se muestra la posibilidad de mejorar la mantenibilidad de un programa incluyendo la separación entre interfaz e implementación incorporando funciones que permiten hacer uso de la entrada y salida de forma independiente del procesamiento. Realmente es la aplicación del patrón MVC (modelo-vista-controlador), pero que tiene sentido en proyectos grandes, aquí se verá una versión simplificada pero que mejora la

calidad del software obtenido, aunque pudiera perder eficiencia en terminos de tiempo de ejecución y/o memoria utilizada, siempre en pro de la calidad y la generalidad.

Esta separación se plasmará en varias cuestiones básicas:

- En los módulos de procesamiento de la información **no puede** haber ninguna operación de entrada salida, es decir **scanf** o **printf**
- Los módulos de salida, **desconocerán** la estructura de datos sobre la que se soporta el problema. Se suele utilizar un módulo "toString".
- Los módulos de entrada **analizarán** la cadena de entrada para volcarlo sobre la estructura de datos del problema/modelo. Pero es algo más complejo y se optará por aproximaciones simples, pero manteniendo la separación.

El enfoque que vamos a utilizar es comunicar modelo e interfaz a través de cadenas de caracteres, el problema que hay en C que no en otros lenguajes es que las cadenas como cualquier vector debe acotarse en tiempo de ejecución.

```c
#include <math.h>

typedef char Cadena[50];


void leer(int *x, int *y);
void cartesianoAPolar(int x, int y, double *angulo, double *modulo);
void mostrar(double angulo, double modulo);

void toString(double angulo, double modulo, Cadena resultado);


int main() {
    int x, y;
    double angulo, modulo;

    leer(&x,&y);
    cartesianoAPolar(x, y, &angulo, &modulo);
    mostrar(angulo,modulo);

    return 0;
}


void leer(int *x, int *y)
{
    Cadena entrada;
    printf("Ingrese coordenadas cartesianas (x y): ");
    fgets(entrada, sizeof(entrada), stdin);
    sscanf(entrada, "%d %d", x, y);

}


void cartesianoAPolar(int x, int y, double *angulo, double *modulo) {
    *modulo = sqrt(x * x + y * y);
    *angulo = atan2(y, x) * 180.0 / M_PI;
}

void toString(double angulo, double modulo, Cadena resultado) {
    sprintf(resultado, "Coordenadas polares: (%.2f, %.2f)", angulo, modulo);
}

void mostrar(double angulo, double modulo){
    Cadena salida;
    toString(angulo, modulo, salida);
    printf("%s\n", salida);
}
```

Colecciones estructuradas. Registros

El último tipo de datos compuestos, **colecciones estructuradas** o registros, permite agrupar variables de diferentes tipos bajo un mismo nombre, al contrario que los vectores en los que todos los elementos de todas las dimensiones son del mismo tipo. En C hay dos tipos de registros: estructuras y uniones.

Los **registros** son útiles para organizar y agrupar datos relacionados en una sola unidad. Se utilizan comúnmente para representar entidades complejas en un programa, como usuarios, empleados, productos, etc.

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

// Definición de una estructura llamada 'Persona'
typedef struct {
    char nombre[50];  // Campo nombre -- cadena
    int edad;         // Campo edad -- entero
    float altura;     // Campo altura -- real
} Persona;

int main() {
    // Declaración de una variable de tipo 'Persona'
    Persona persona1;

    // Asignación de valores a los campos de la estructura
    strcpy(persona1.nombre, "Juan");
    persona1.edad = 25;
    persona1.altura = 1.75;

    // Acceso a los campos de la estructura y muestra de la información
    printf("Nombre: %s\n", persona1.nombre);
    printf("Edad: %d\n", persona1.edad);
    printf("Altura: %.2f\n", persona1.altura);

    return 0;
}
```

Un registro está formado por yuxtaposición de elementos que contienen información relativa a una misma entidad. Es una estructura **estática** puesto que consta de un número fijo de componentes y **heterogénea** ya que sus componentes pueden ser de tipos de datos diferentes.

Los elementos que componen el registro se les denomina **campos** (miembros, atributos, componentes) y el acceso a los componentes se realiza mediante un identificador (**nombre de campo**) en vez de por su posición relativa. En el ejemplo anterior tenemos un tipo de datos registro llamado *persona* que tiene tres campos *nombre*, *edad* y *altura*.

Declaración de registros

La declaración de los registros también se recomienda que se realice a través de la definición de un tipo `typedef`.

- Declaración de un registro (struct):

```
//Diseño del tipo de datos:
typedef struct{
        tipo1 campo1;
        tipo2 campo2;
        . . . .
        tipoN campoN;
        }tipo_reg;

// Declaración de variables:
tipo_reg r;
```

Veamos algunos ejemplos:

```
/* Diseño de nuevos tipos de datos */
// Persona empleada de empresa
typedef struct{
        char nombre[31];
        char apellidos[41];
        long int dni;
        char direccion[51];
        int edad;
        float sueldo;
}tipo_empleado;

//Punto en tres dimensiones
typedef struct{
        float x;
        float y;
        float z;
}tipo_punto3d;

// Cuenta en el banco
typedef struct{
        char iban[15];
        char tipo;
        long int dni;
        float saldo;
}tipo_cuenta;

//Alumno
typedef struct{
        char nombre[31];
        char apellidos[41];
        long int dni;
        char direccion[51];
        int edad;
        float nota_teoria;
        int practicas_aprobadas;
}tipo_ficha_alumno;


// Medida de un sensor
typedef struct{
        char codigo[6];
        int dia;
        float hora;
        char estado;
        float valor;
        char unidades[6];
}tipo_medida_sensor;
```

Para variables tipo registro

```
/* Declaración de variables */
tipo_empleado empleado;
tipo_punto3d p;
tipo_cuenta cuenta;
tipo_medida medida;
tipo_ficha_alumno ficha1, ficha2;
```

Operaciones con registros

- Acceso a componentes individuales:

```
variable.campo
```

El operador ., es el llamado operador selector de campo.

- Asignación:

```
strcpy(ficha1.nombre,"José R. ");
ficha2.edad=ficha1.edad;
ficha2.prácticas_aprobadas=1;
```

- Entrada/salida:

```
scanf(" %f", &ficha1.nota_teoría);
printf("%s %s\n",ficha1.nombre, ficha1.apellidos);
```

- Expresiones:

```
(ficha1.nota_teoría>=5.0)y(ficha1.prácticas_aprobadas)
```

- Acceso secuencial a todos los componentes: composición secuencial de instrucciones (1 instrucción para cada campo):

```
{ Procesamiento varReg.campo1 }
{ Procesamiento varReg.campo2 }
...
{ Procesamiento varReg.campoN }
```

- **Lectura** de un registro completo por teclado:

```
printf("Nombre: ");
fgets(ficha1.nombre,31,stdin);
fflush(stdin);
printf("Apellidos: ");
fgets(ficha1.apellidos,41,stdin);
fflush(stdin);
printf("DNI: ");
scanf(" %d", &ficha1.dni);
printf("Dirección: ");
fgets(ficha1.direccion,51,stdin);
fflush(stdin);
printf("Edad: ");
scanf(" %d", &ficha1.edad);
printf("Nota teoría: ");
scanf(" %f", &ficha1.nota_teoria);
printf("Prácticas aprobadas: ");
scanf(" %d", &ficha1.practicas_aprobadas);
```

- **Escritura** de un registro completo en pantalla:

```
printf("Nombre: %s\n",ficha1.nombre);
printf("Apellidos: %s\n",ficha1.apellidos);
printf("DNI: %d\n",ficha1.dni);
printf("Dirección: %s\n",ficha1.direccion);
printf("Edad: %d\n",ficha1.edad);
printf("Nota teoría: %.2f\n",ficha1.nota_teoria);
printf("Prácticas aprobadas: %d", ficha1.practicas_aprobadas);
```

- **Copia** de un registro completo a otra variable del mismo tipo de registro (definido con los campos en el mismo orden):

```
ficha2= ficha1;
```

Paso de parámetros

- De componentes **individuales** (campos):

Funciona como una variable. De esta forma, si se utiliza como parámetro de entrada se pone solo `var_reg.campo`, pero si se quiere que los efectos de la llamada perduren, es decir se utiliza como parámetro de entrada/salida, se ha de poner la dirección: `&var_reg.campo`.

- Del registro **completo**

	por valor (entrada)	por dirección (E/S)
parámetro real	`var_reg`	`&var_reg`
parámetro formal	`tipo_reg var_reg`	`tipo_reg *reg`
para acceso a un campo:	`(*reg).campo` o bien `reg->campo`	

Nótese que una función también puede devolver a través de su identificador de función un registro completo, con lo que podrían unificarse todas las salidas sobre un registro.

Acceso a campos con operador punto o flecha La diferencia entre usar el operador de punto (`.`) y el operador de flecha (`->`) para acceder a los campos de una estructura en C depende de cómo se esté accediendo a la estructura.

El operador de punto se utiliza cuando se accede a los campos de una estructura a través de una variable directa de esa estructura. Específicamente, se utiliza cuando la variable en sí es la estructura (no un puntero a la estructura).

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

// Definición de una estructura llamada 'Persona'
typedef struct {
    char nombre[50];
    int edad;
    float altura;
} Persona;

int main() {
    // Declaración de una variable de tipo 'Persona'
    Persona persona1;

    // Acceso a los campos de la estructura usando el operador de punto
    strcpy(persona1.nombre, "Juan");
    persona1.edad = 25;
    persona1.altura = 1.75;

    // Mostrar información
    printf("Nombre: %s\n", persona1.nombre);
    printf("Edad: %d\n", persona1.edad);
    printf("Altura: %.2f\n", persona1.altura);

    return 0;
}
```

El operador de flecha se utiliza cuando se accede a los campos de una estructura a través de un puntero a esa estructura. Específicamente, se utiliza cuando la variable es un puntero que apunta a una estructura.

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

// Definición de una estructura llamada 'Persona'
typedef struct {
    char nombre[50];
    int edad;
    float altura;
} Persona;

int main() {
    // Declaración de una variable de tipo 'Persona' mediante un puntero
    Persona persona1;
    Persona *punteroPersona = &persona1;

    // Acceso a los campos de la estructura usando el operador de flecha
    strcpy(punteroPersona->nombre, "Juan");
    punteroPersona->edad = 25;
    punteroPersona->altura = 1.75;

    // Mostrar información
    printf("Nombre: %s\n", punteroPersona->nombre);
    printf("Edad: %d\n", punteroPersona->edad);
    printf("Altura: %.2f\n", punteroPersona->altura);

    return 0;
}
```

En resumen:

Si se está trabajando directamente con la estructura (no con un puntero), utiliza el operador de punto (.). Si se está trabajando con un puntero a la estructura, utiliza el operador de flecha (->) para acceder a los campos de la estructura a través del puntero. Ambos operadores se utilizan para acceder a los miembros de una estructura y son simplemente una cuestión de sintaxis en función del tipo de variable que estás utilizando.

PROBLEMA Comprobemos ahora como la gestión de un vector con tamaño se puede hacer con un registro dotando al problema de una solución mas compacta. En este caso trabajaremos con un vector de reales sobre los que vamos a calcular media y mediana.

```c
/*
 * @authors Equipo docente Programación
 * @project Creación de Materiales Didácticos en la Univer. de Almería (2021-2022)
 * Grados en Ingeniería Eléctrica, Electrónica Industrial, Mecánica y Química Industrial
 * @date 2021-02-06
 */


#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include <stdlib.h> // Para la función rand()
#include <time.h>   // Para inicializar la semilla de rand()
#define N 20

/* Nuevos tipos de datos */
typedef double Vector[N];

typedef struct{
        int n;
```

```
        Vector v;
}Datos;



/* prototipos de funciones */
void inicializar_vector(Datos *d, int n);
void copiar_vector(Datos d, Datos *d2);
void ordenar_vector(Datos *d);
double calcular_media(Datos d);
double calcular_mediana(Datos d);
void escribir_vector(Datos d);

int main(){
        char c;
        Datos d,d2;

        int n;
        printf("CALCULO DE MEDIA ARITMETICA Y MEDIANA\n");
        printf("=====================================\n\n");
        do{        printf("\tIntroduzca numero de datos a introducir: ");
                        scanf(" %d", &n);
        }while((n<=0)||(n>N));
        inicializar_vector(&d,n);
        printf("\nVector original:      ");
        escribir_vector(d);
        copiar_vector(d,&d2);
        ordenar_vector(&d2);
        printf("\nVector ordenado:      ");
        escribir_vector(d2);
        printf("\n\tMedia aritmetica: %.2f",calcular_media(d));
        printf("\n\tMediana: %.2f",calcular_mediana(d));

        return 0;
}
void inicializar_vector(Datos *d, int n) {
    srand(time(NULL)); // Inicializar la semilla de rand() con el tiempo actual

    int i;

    for (i = 0; i < n; ++i) {
        d->v[i] = 1.0 * rand() / RAND_MAX * 100.0;
    }
        d->n=n;
}
void copiar_vector(Datos d, Datos *d2){
        int i;
        for(i=0;i<d.n;++i)
                d2->v[i]=d.v[i];
        d2->n=d.n;
}

void ordenar_vector(Datos *d){
        int i,j,k;
        double x;

        for(i=0;i<(d->n)-1;++i){
                k=i;
                for(j=i+1;j<d->n;++j)
                        if(d->v[j]<d->v[k])
                                k=j;
                x=d->v[i];
                d->v[i]=d->v[k];
                d->v[k]=x;
        }
}

void escribir_vector(Datos d){
        int i;
        for(i=0;i<d.n;++i){
                printf("%.2f  ",d.v[i]);
```

```
        }
}

double calcular_media(Datos d){
        int i;
        double suma;

        suma=0;
        for(i=0;i<d.n;++i)
                suma+=d.v[i];
        return(suma/d.n);
}

double calcular_mediana(Datos d){
        if(d.n%2)
                return d.v[d.n/2];
        else return(d.v[d.n/2]+d.v[d.n/2-1])/2.0;
}
```

PROBLEMA Construir un programa para que calcule y visualice los datos de las notas de un alumno de programación.

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

# define NUMPR 10

typedef char cadena20[21];
typedef float tipo_notasPR[NUMPR];
typedef struct{
        cadena20 nom;
    tipo_notasPR practicas;
    float PR;
    float TI;
    float EX;
    float calificacion;
}tipo_reg;

int contarAprobadas(tipo_notasPR pr);
tipo_reg leer();
void mostrar_estudiante(tipo_reg estudiante);
void calificacion(tipo_reg *estudiante);


int main() {

    tipo_reg yo,tu, el;
    yo=leer();
    leer2(&tu);

    calificacion(&yo);
    mostrar_estudiante(yo);
    return 0;
}



int contarAprobadas(tipo_notasPR pr){
    int contador = 0;
    for (int i = 0; i < NUMPR; i++) {
        if (pr[i] >=5 ) {
            contador++;
        }
    }
    return contador;
}


void leer2(tipo_reg *estudiante){
    printf("¿Cuál es la nota de los trabajos individuales (0-1)?: ");
```

```
    scanf("%f", estudiante->TI);

    printf("¿Cuál es la nota obtenida en el examen (1-10)?: ");
    scanf("%f", estudiante->EX);

    }

tipo_reg leer(){
    tipo_reg estudiante;

    printf("¿Cuál es la nota de los trabajos individuales (0-1)?: ");
    scanf("%f", &estudiante.TI);

    printf("¿Cuál es la nota obtenida en el examen (1-10)?: ");
    scanf("%f", &estudiante.EX);

    for (int i = 0; i < NUMPR; i++){
        printf("Nota práctica %d: ", i+1);
        scanf("%f", &estudiante.practicas[i]);
    }
    return estudiante;
}

void mostrar_estudiante(tipo_reg estudiante){
    printf("El alumno %s ha obtenido las calificaciones \n",estudiante.nom );
    printf("En trabajos individuales: %f \n",estudiante.TI);

    for (int i=0; i<NUMPR;i++){
        printf(" Nota práctica %d: %f\n", i+1,estudiante.practicas[i]);
    }
    printf("En examen: %f \n",estudiante.EX);
    printf("Calificación final: %f \n",estudiante.calificacion);
}


void calificacion(tipo_reg *estudiante){

    if (contarAprobadas(estudiante->practicas) > 6)
        estudiante->PR = 1;
    else
        estudiante->PR = 0;

    if (estudiante->EX >= 4)
        estudiante->calificacion = estudiante->PR * (estudiante->EX * 0.8) + estudiante->TI + estudiante->PR;
}
```

PROBLEMA

Construir un programa para realizar aritmética de números complejos. El programa deberá leer por teclado dos números complejos y presentará a continuación un menú en pantalla con las siguientes opciones:

- Sumar números complejos.
- Restar números complejos.
- Multiplicar números complejos.
- Dividir números complejos.
- Terminar programa.

Tras seleccionar por teclado una opción, se ejecutará la operación correspondiente presentándose en pantalla los resultados de la misma.

Análisis Se parte de dos números complejos a, b con parte real y parte imaginaria $a = (a_{re} + i * a_{im})$ y $b = (b_{re} + i * b_{im})$

Siendo
$a_{re}, a_{im}, b_{re}, b_{im}$ números reales que simplificamos con a_r, a_i, b_r, b_i

$i = \sqrt{(-1)}$

Las operaciones con complejos con $(a_r + i * a_i) + (b_r + i * b_i) = ((a_r + b_r) + i * (a_i + b_i))$

$(a_r + i * a_i) - (b_r + i * b_i) = ((a_r - b_r) + i * (a_i - b_i))$

$(a_r + i * a_i) * (b_r + i * b_i) = ((a_r * b_r - a_i * b_i) + i * (a_r * b_i + a_i * b_r))$

$(a_r + i * a_i)/(b_r + i * b_i) = \frac{(a_r * b_r + a_i * b_i) + i * (a_i * b_r - a_r * b_i)}{(b_r^2 + b_i^2)}$ con $b_r^2 + b_i^2$ distinto de cero.

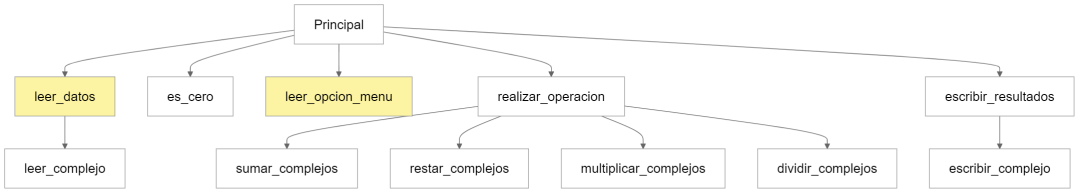
Diseño de datos

Un número complejo es un registro de dos reales

```
typedef struct{
        float re;
        float im;
}tipo_complejo;
```

Diseño arquitectónico

Mostraremos un menú de opciones donde se permitan los operaciones y antes debe estar precedido por la lectura de dos números complejos.




```
/* @authors Equipo docente Programación
 * @project Creación de Materiales Didácticos en la Univer. de Almería
 * Grados en Ingeniería Elécctrica, Electrónica Industrial, Mecénica y Química industrial
 * @date 2024-02-06
 */
/* Programa que realiza diversas operaciones */
/* con dos numeros complejos:                */
/* suma, resta, multiplicacion y division    */

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

/* Nuevos tipos de datos */
typedef struct{
        float re;
        float im;
}tipo_complejo;

/* Prototipos de funciones */
void leer_datos(tipo_complejo *a, tipo_complejo *b);
void leer_complejo(tipo_complejo *a);
void leer_opcion_menu(char *c);
void realizar_operacion(tipo_complejo a, tipo_complejo b,
        char op, tipo_complejo *r);
void sumar_complejos(tipo_complejo a, tipo_complejo b,
        tipo_complejo *r);
void restar_complejos(tipo_complejo a, tipo_complejo b,
        tipo_complejo *r);
```

```
void multiplicar_complejos(tipo_complejo a, tipo_complejo b,
        tipo_complejo *r);
void dividir_complejos(tipo_complejo a, tipo_complejo b,
        tipo_complejo *r);
void escribir_resultados(tipo_complejo a, tipo_complejo b,
        char op, tipo_complejo r);
void escribir_complejo(tipo_complejo a);
int es_cero(tipo_complejo a);

int main(){
    char c;
    char op;           /* opcion del menu (codigo de la operacion) */
    tipo_complejo a,b; /* Numeros complejos de entrada             */
    tipo_complejo r;   /* Resultado de la operacion                */

    printf("ARITMÉTICA DE COMPLEJOS\n");
    printf("=======================\n\n");
    leer_datos(&a,&b);
    do{ leer_opcion_menu(&op);
        if (op!='0')
            if((op=='/')&&(es_cero(b))){
                    printf("\nDivision por cero");
                    getch();
            }else{ realizar_operacion(a,b,op,&r);
                    escribir_resultados(a,b,op,r);
                    getch();
            }
        }while(op!='0');
    return 0;
}

void leer_datos(tipo_complejo *a, tipo_complejo *b){
    printf("Introduzca primer numero complejo:\n");
    leer_complejo(a);
    printf("Introduzca segundo numero complejo:\n");
    leer_complejo(b);
}

void leer_complejo(tipo_complejo *a){
    printf("\tIntroduzca parte real: ");
    scanf(" %f",&a->re);
    printf("\tIntroduzca parte imaginaria: ");
    scanf(" %f",&a->im);
}

void leer_opcion_menu(char *c){
    do{ system("cls");
        printf("MENU DE OPERACIONES:\n");
        printf("====================\n\n");
        printf("\t+  Suma de numeros complejos\n");
        printf("\t-  Resta de numeros complejos\n");
        printf("\t*  Multiplicacion de numeros complejos\n");
        printf("\t/  Division de numeros complejos\n");
        printf("\t0  Fin de operaciones\n");
        printf("\t\tIntroduzca operacion: ");
        *c=getch();
    }while((*c!='+')&&(*c!='-')&&(*c!='*')&&(*c!='/')&&(*c!='0'));
}

void realizar_operacion(tipo_complejo a, tipo_complejo b,
        char op, tipo_complejo *r){
    switch(op){
    case '+': sumar_complejos(a,b,r);
            break;
    case '-': restar_complejos(a,b,r);
            break;
    case '*': multiplicar_complejos(a,b,r);
            break;
    case '/': dividir_complejos(a,b,r);
            break;
    }
```

```
}

void sumar_complejos(tipo_complejo a, tipo_complejo b, tipo_complejo *r){
    r->re=a.re+b.re;
    r->im=a.im+b.im;
}

void restar_complejos(tipo_complejo a, tipo_complejo b, tipo_complejo *r){
    r->re=a.re-b.re;
    r->im=a.im-b.im;
}

void multiplicar_complejos(tipo_complejo a, tipo_complejo b, tipo_complejo *r){
    r->re=a.re*b.re-a.im*b.im;
    r->im=a.re*b.im+a.im*b.re;
}

void dividir_complejos(tipo_complejo a, tipo_complejo b, tipo_complejo *r){
    float den;        /* denominador de la expresion */
    den=b.re*b.re+b.im*b.im;
    r->re=(a.re*b.re+a.im*b.im)/den;
    r->im=(a.im*b.re-a.re*b.im)/den;
}

void escribir_resultados(tipo_complejo a, tipo_complejo b, char op, tipo_complejo r){
    printf("\n\n");
    escribir_complejo(a);
    printf(" %c ",op);
    escribir_complejo(b);
    printf(" = ");
    escribir_complejo(r);
}

void escribir_complejo(tipo_complejo a){
    if(a.im==0)
        if(a.re)
            printf("(%.1f)",a.re);
        else printf("(0)");
    else if(a.im>0)
            if(a.re)
                printf("(%.1f + i*%.1f)",a.re,a.im);
            else printf("(i*%.1f)",a.im);
        else if(a.re)
                printf("(%.1f - i*%.1f)",a.re,-a.im);
            else printf("(-i*%.1f)",-a.im);
}

int es_cero(tipo_complejo a){
    if((a.re==0)&&(a.im==0))
        return 1;
    else return 0;
}
```

Registros con parte variante

Los registros con parte variante (con discriminador o subtipos) se utilizan en muchos lenguajes que incorporan el tipo registro, permiten definir una parte fija (igual para todas las variables de ese tipo) y una parte variable con distintas opciones de campos.

Cada variante es una colección de campos, aplicables a un caso concreto. Todas las variantes ocupan el mismo bloque de memoria (el compilador reserva la memoria suficiente para la mayor de las variantes).

El campo discriminante permite definir (según su valor) cada uno de los casos. Es opcional, y en caso de no utilizarse el control del estado debe efectuarlo el programador.

Ventajas: una única tipología con diferentes sub-tipos pero con el mismo conjunto de operaciones para todos los sub-tipos.

Este concepto en C se recoge con las **uniones**. Una unión es una estructura de datos que define variables que comparten el almacenamiento de memoria. Puede contener, en distintos momentos, datos de tipos y tamaños diferentes. El compilador reserva suficiente espacio de memoria para acomodar el elemento mayor de la unión.

Al igual que las estructuras, se accede a un campo de una unión mediante el selector de campo (`.`), pero, a diferencia de las estructuras, los campos de la unión ocupan la misma posición de memoria (no pueden almacenar información de forma simultánea).

- Declaración de una **variable** tipo unión:

```
union etiqueta_union{
tipo1 nombre_campo1;
tipo2 nombre_campo2;
. . .
tipoN nombre_campoN;

}nombre_variable;
```

La etiqueta de la unión es opcional, y en caso de utilizarse permite la declaración futura de nuevas variables de unión con dicha estructura (funciona de forma parecida a los tipos de datos definidos mediante `typedef`).

Ejemplos:

```
union etiq_identif{        /* etiqueta de la union */
        long int dni;
        char pasaporte[20];
} identificacion;

...
union etiq_identif id2;
```

Aquí se define una unión llamada `etiq_identif` con una etiqueta. La unión tiene dos miembros.L uego, se declara una variable llamada `identificacion` del tipo de esta unión.

A continuación se incluye una declaración de la unión sin etiqueta.

```
union{                      /* sin etiqueta de union */
        float exp_real;
        int exp_entero;
}exponente1, exponente2;
```

En este caso, se define una unión anónima sin etiqueta. La unión tiene dos miembros. Luego, se declaran dos variables `exponente1` y `exponente2` del tipo de esta unión.

- Declaración de un **tipo unión**:

```
typedef  union{
tipo1 nombre_campo1;
tipo2 nombre_campo2;
. . .
tipoN nombre_campoN;
}tipo_union;
```

En este caso la declaración de variables sería:

```
tipo_union nombre_variable;
```

Ejemplos:

```
/* tipos definidos por el usuario */
typedef union{
        long int dni;
        char pasaporte[20];
} tipo_identificacion;

typedef union{
        float exp_real;
        int exp_entero;
}tipo_exponente;

                /* declaración de variables */
tipo_identificacion identificacion, id2;
tipo_exponente exponente1, exponente2;
```

En el siguiente ejemplo vemos como se puede utilizar una unión para guardar los datos del sensor.

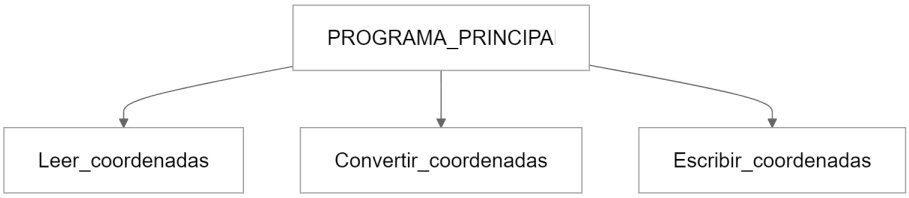
```
#include <stdio.h>

// Definición de la unión
union Medicion {
    int entero;
    float flotante;
    double doble;
};

int main() {
    union Medicion sensor;

    // Almacenando diferentes tipos de mediciones en la unión
    sensor.entero = 10;
    printf("Medición entera: %d\n", sensor.entero);

    sensor.flotante = 3.14;
    printf("Medición flotante: %.2f\n", sensor.flotante);

    sensor.doble = 123.456;
    printf("Medición doble: %.3lf\n", sensor.doble);

    return 0;
}
```

PROBLEMA Construir un programa que lea por teclado las coordenadas de un punto bidimensional, bien en coordenadas cartesianas o bien en coordenadas polares, las almacene en una variable estructurada e imprima a continuación en pantalla las coordenadas del punto en su representación actual. Seguidamente el programa transformará la representación de las coordenadas del punto a coordenadas cartesianas y finalmente volverá a imprimir las coordenadas del punto en su representación actual.

- Diseño de datos: Aunque los dos tipos de coordenadas se corresponden con dos valores de tipo real, los nombres de los campos deben ser distintos en una (`x` e `y`) y otra (`modulo` y `angulo`), con lo que la solución es la utilización de un registro con parte variante.

```
typedef struct{        /* Registro con parte variante         */
   int cartesianas;   /* Discriminante: 1-cartes 0-polares   */
   union{   /* sus campos comparten mismas posiciones de M.C.*/
        struct{        /* variante 1 */
            float x,y;
        };
        struct{        /* variante 2 */
            float modulo,angulo;
        };
    };
}tipo_coordenadas;
```

- Diseño arquitectónico En este caso tenemos una función para cada una de las tareas a realizar con el punto: Leer, escribir, convertir




```
/* Ejemplo de registro con parte variante:  Coordenadas bidimensionales de un punto: */
/* Cartesianas: (x,y) Polares: (modulo,angulo)                  */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#define PI atan(1.0)*4.0

/* Nuevos tipos de datos */
typedef struct{        /* Registro con parte variante         */
    int cartesianas;   /* Discriminante: 1-cartes 0-polares   */
    union{   /* sus campos comparten mismas posiciones de M.C.*/
        struct{        /* variante 1 */
            float x,y;
        };
        struct{        /* variante 2 */
            float modulo,angulo;
        };
    };
}tipo_coordenadas;

/* prototipos de funciones */
void leer_coordenadas(tipo_coordenadas *punto);
void c_p_c(tipo_coordenadas *punto);
void escribir_coordenadas(tipo_coordenadas punto);

int main(){
    char c;
    tipo_coordenadas punto;

    printf("EJEMPLO DE REGISTRO VARIANTE\n");
    printf("============================\n\n");
    printf("Introducir coordenadas 2D de un punto:\n");
    leer_coordenadas(&punto);
    printf("Punto introducido por teclado:\n");
    escribir_coordenadas(punto);
    c_p_c(&punto);
    printf("Punto en coordenadas cartesianas:\n");
    escribir_coordenadas(punto);

    return 0;
}
```

```
void leer_coordenadas(tipo_coordenadas *punto){
    do{ printf("\tTipo de coordenadas (1: cartesianas, 0: polares): ");
        scanf(" %d", &punto->cartesianas);
    }while((punto->cartesianas<0)||(punto->cartesianas>1));
    if(punto->cartesianas){
        printf("\t\tx: ");
        scanf(" %f",&punto->x);
        printf("\t\ty: ");
        scanf(" %f",&punto->y);
    }else{ do{ printf("\t\tr: ");
                scanf(" %f",&punto->modulo);
            }while(punto->modulo<0);
            printf("\t\tAngulo (grados): ");
            scanf(" %f",&punto->angulo);
    }
}

void c_p_c(tipo_coordenadas *punto){
/* Convierte coordenadas polares a cartesinas  */
    float x,y;
    if(!(punto->cartesianas)){
        x=punto->modulo*cos(punto->angulo*PI/180.0);
        y=punto->modulo*sin(punto->angulo*PI/180.0);
        punto->cartesianas=1;
        punto->x=x;
        punto->y=y;
    }
}

void escribir_coordenadas(tipo_coordenadas punto){
    if(punto.cartesianas){
        printf("\tx = %.2f\n",punto.x);
        printf("\ty = %.2f\n",punto.y);
    }else{ printf("\tr = %.2f\n",punto.modulo);
        printf("\tang (grados) = %.2f\n",punto.angulo);
    }
}
```

PROBLEMA

```
#include <stdio.h>

// Definición del tamaño máximo del array de mediciones
#define MAX_MEDICIONES 100

// Definición de la unión Medicion utilizando typedef
typedef union {
    int entero;
    float flotante;
    double doble;
} Medicion;

// Prototipos de funciones
double calcular_media(Medicion mediciones[], int num_mediciones, int tipo_medicion);
double calcular_mediana(Medicion mediciones[], int num_mediciones, int tipo_medicion);

int main() {
    // Declaración del array de mediciones
    Medicion mediciones[MAX_MEDICIONES];
    int num_mediciones;
    int tipo_medicion;

    // Solicitar al usuario el número de mediciones
    printf("Ingrese el número de mediciones (máximo %d): ", MAX_MEDICIONES);
    scanf("%d", &num_mediciones);

    // Solicitar al usuario el tipo de medición (0 para entero, 1 para flotante, 2 para doble)
    printf("Ingrese el tipo de medición (0 para entero, 1 para flotante, 2 para doble): ");
    scanf("%d", &tipo_medicion);
```

```c
    // Leer las mediciones y almacenarlas en la unión correspondiente
    switch (tipo_medicion) {
        case 0:
            for (int i = 0; i < num_mediciones; ++i) {
                printf("Ingrese la medición entera %d: ", i + 1);
                scanf("%d", &mediciones[i].entero);
            }
            break;
        case 1:
            for (int i = 0; i < num_mediciones; ++i) {
                printf("Ingrese la medición flotante %d: ", i + 1);
                scanf("%f", &mediciones[i].flotante);
            }
            break;
        case 2:
            for (int i = 0; i < num_mediciones; ++i) {
                printf("Ingrese la medición doble %d: ", i + 1);
                scanf("%lf", &mediciones[i].doble);
            }
            break;
        default:
            printf("Tipo de medición no válido.\n");
            return 1;
    }

    // Calcular la media aritmética y la mediana
    double media = calcular_media(mediciones, num_mediciones, tipo_medicion);
    double mediana = calcular_mediana(mediciones, num_mediciones, tipo_medicion);

    // Mostrar el resultado
    printf("La media aritmética de las mediciones es: %.2lf\n", media);
    printf("La mediana de las mediciones es: %.2lf\n", mediana);

    return 0;
}

// Función para calcular la media aritmética de las mediciones
double calcular_media(Medicion mediciones[], int num_mediciones, int tipo_medicion) {
    double suma = 0.0;

    switch (tipo_medicion) {
        case 0:
            for (int i = 0; i < num_mediciones; ++i) {
                suma += mediciones[i].entero;
            }
            break;
        case 1:
            for (int i = 0; i < num_mediciones; ++i) {
                suma += mediciones[i].flotante;
            }
            break;
        case 2:
            for (int i = 0; i < num_mediciones; ++i) {
                suma += mediciones[i].doble;
            }
            break;
        default:
            printf("Tipo de medición no válido.\n");
            return 0.0;
    }

    return suma / num_mediciones;
}

// Función para calcular la mediana de las mediciones
double calcular_mediana(Medicion mediciones[], int num_mediciones, int tipo_medicion) {
    // No se puede calcular la mediana para un número par de mediciones
    if (num_mediciones % 2 == 0) {
        printf("No se puede calcular la mediana para un número par de mediciones.\n");
        return 0.0;
```

```
    }

    // Ordenar las mediciones (se omiten aquí los detalles de ordenación)
    // ...

    // Devolver la mediana según el tipo de medición
    switch (tipo_medicion) {
        case 0:
            return mediciones[num_mediciones / 2].entero;
        case 1:
            return mediciones[num_mediciones / 2].flotante;
        case 2:
            return mediciones[num_mediciones / 2].doble;
        default:
            printf("Tipo de medición no válido.\n");
            return 0.0;
    }
}
```

Definición del modelo de datos

Si bien existen otro tipo de colecciones de datos más complejas como las pilas, las colas, los dataframes, en su origen tienen los vectores y los registros como punto de partida y lo que se hace es incorporar módulos que permiten gestionarlas adecuadamente. En cursos superiores, se trabajará con lenguajes como Matlab que es un entorno de programación y un lenguaje de programación diseñado específicamente para el cálculo numérico y la manipulación de matrices. Es extremadamente potente en términos de gestión de matrices debido a su sintaxis optimizada y las numerosas funciones incorporadas que facilitan las operaciones matriciales. En Matlab, las operaciones matriciales son simples de expresar y ejecutar haciéndolo muy eficiente para tareas científicas y de ingeniería.

Implementación de una pila

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

#define MAX_SIZE 100

typedef struct {
    int items[MAX_SIZE];
    int top; // índice de la posición del primer elemento de la pila
} Stack;

void initialize(Stack *s) {
    s->top = -1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

int isFull(Stack *s) {
    return s->top == MAX_SIZE - 1;
}

void push(Stack *s, int value) {
    if (isFull(s)) {
      //  printf("La pila está llena. No se puede insertar más elementos.\n");
        return;
    }
    s->items[++(s->top)] = value;
}
```

```c
int pop(Stack *s) {
    if (isEmpty(s)) {
       // printf("La pila está vacía. No se puede eliminar ningún elemento.\n");
        return -1;
    }
    return s->items[(s->top)--];
}

int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("La pila está vacía. No hay elementos en la cima.\n");
        return -1;
    }
    return s->items[s->top]; // Retorna el elemento en la cima de la pila
}

int main() {
    Stack stack;
    initialize(&stack);

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    printf("Elemento en la cima de la pila: %d\n", peek(&stack));

    printf("Elementos extraídos de la pila: %d, %d, %d\n", pop(&stack), pop(&stack), pop(&stack));

    printf("La pila está vacía: %s\n", isEmpty(&stack) ? "Sí" : "No");

    return 0;
}
```

De una forma muy simplificada, este lenguaje incorpora las Matrices como parte de su sintaxis, porque tiene definido un modelo de datos que las soporta.

Pero independientemente del lenguaje siempre hay una tarea difícil para ajustar el **dominio del problema**, es decir los datos que se manejan para resolver una tarea y el **dominio de la solución** que son las colecciones de datos y su combinación para resolver un problema. En el caso de sistema de partículas se optó por utilizar cuatro vectores con capacidad para resolver un problema, es decir debíamos mantener los cuatro vectores, mas un entero con el número de partículas.

Puesto que los cinco "datos" son dependientes, podríamos definirlos como un registro, porque así podremos tener distintos sistemas de partículas para un problema, por ejemplo para fusionarlos:

```c
typedef float tipo_vector[MAX];
typedef  struct{
    tipo_vector m;     /* masas de las partículas       */
    tipo_vector x,y,z; /* coordenadas de sus posiciones */
    int n;             /* Numero de partículas          */
}tipo_sistemaparticulas;

...

void main(){

tipo_sistemaparticulas sistema;

}
```

Pero podríamos pensar en otra alternativa, puesto que la realidad es que es la partícula la que tiene cuatro propiedades, su posición y su masa, asi:

```
typedef struct{
        float x, y, z, m;
}tipo_particula;

typedef tipo_particula tipo_vector[MAX];
typedef  struct{
    tipo_vector particulas;
    int n;
}tipo_sistemaparticulas;


void main(){

tipo_sistemaparticulas sistema;

}
```

Dependiendo de como se defina, así se tendrá que trabajar en los módulos que resuelvan el problema. Como norma se suele intentar mantener la misma relación que exista entre los datos en el dominio del problema. Por ejemplo: para los datos de una partícula, sus coordenadas y su masa se podría haber optado por utilizar un vector de cuatro posiciones de 0 a 3 donde los tres primeros (índices 0, 1 y 2) referencien las posiciones y el cuarto (índice 3) la masa y en forma de matriz definir el número de partículas

```
typedef float tipo_particula[4][MAX];
typedef  struct{
    tipo_particula particulas;
    int n;
}tipo_sistemaparticulas;
```

Esta representación en vector de los datos de una partícula no parece apropiada puesto que en el "dominio del problema", las coordenadas no están indexadas por posición, son simplemente `x,y,z,m`, ademas perdemos el concepto de partícula individual.

Esta compartimentalización de los datos se puede llevar un poco más lejos y podemos plantear que hay una diferencia entre `x,y,z` y el valor de la masa `m` y puesto que son `tipos` distintos plantear el modelo:

```
typedef struct{
        float x, y, z;
}tipo_coordenada;

typedef struct{
    tipo_coordenada posicion;
        float m;
}tipo_particula;
typedef tipo_particula tipo_vector[MAX];
typedef  struct{
    tipo_vector particulas;
    int n;
}tipo_sistemaparticulas;
```

Cuanto más compleja es la representación de los datos más difícil es manejarla, aunque pudiera ser más reutilizable. En este caso, no parece acertada puesto que para nuestro problema siempre se manejan coordenadas y masas a la vez. Esta representación tendría utilidad si se disponen de librerías de funciones que manejan sistemas de coordenadas, por ejemplo teniendo librerías que permitan rotar o desplazar coordenadas que ya estén implementadas.

Destacamos un tipo de estructura de datos que se repite en C cuando se manejan colecciones indexadas que permiten recorrer estas colecciones y convierten un vector y/o una matriz

en un registro de dos componentes, la colección propiamente dicha y el tamaño de cada una de las dimensiones de esa colección.

```
typedef tipo_base tipo_vector[MAX_DIM1][MAX_DIM2].. [MAX_DIMn];
typedef  struct{
    tipo_vector coleccion;
    int dim1;
    int dim2;
    ...
    int dimn;
}tipo_coleccion;
```

No obstante se recomienda ser cuidadoso con la combinación de estructuras de datos puesto que C no hace validación de tipos.

Otros ejemplos

Ejemplo: reacción química diseño descendente:

$2NaOH + H_2SO_3 - > Na_2SO_3 + 2H_20$

Cómo se representa: Pensemos que en el mundo real tenemos una reacción formada por dos partes: los reactivos y los productos. Ambos son un conjunto de compuestos en los que dada uno tiene un coeficiente estequiométrico (entero) y una fórmula. Cada fórmula son un conjunto de átomos (identificados por su símbolo) con su valencia (número de intervinientes).

```
// Constantes
#define MAX_R 5  // Nº máximo de reactivos en cada ecuación
#define MAX_P 5  // Nº máximo de productos en cada ecuación
#define MAX_E 5  // Nº máximo de elementos en cada fórmula
#define MAX_NOM 30  // Longitud máxima del nombre de la fórmula

// Definición de tipos
typedef struct {
    int ce;  // Coeficiente estequiométrico
    tipo_formula f;
} tipo_sustancia;

typedef struct {
    char nom[MAX_NOM];  // Nombre fórmula química
    int n;              // Nº elementos (1-MAX_E)
    tipo_v_ae v;        // Átomos de cada elemento
} tipo_formula;

typedef struct {
    int n;
    tipo_atomos v[MAX_E];
} tipo_v_ae;

typedef struct {
    char sim[3];  // Símbolo elemento
    int n;        // Número átomos en fórmula
} tipo_atomos;

typedef struct {
    int n;
    tipo_sustancia v[MAX_R];
} tipo_lista_reactivos;

typedef struct {
    int n;
    tipo_sustancia v[MAX_P];
```

```
} tipo_lista_productos;

typedef struct {
    tipo_lista_reactivos r;
    tipo_lista_productos p;
} tipo_ecuacion;
```

que puede usarse como

```
// Función para contar el nº total de átomos en los reactivos
int contar_atomos(tipo_ecuacion ec) {
    int i, j, n;
    n = 0;

    for (i = 0; i < ec.r.n; i++) {
        for (j = 0; j < ec.r.v[i].f.n; j++) {
            n += ec.r.v[i].ce * ec.r.v[i].f.v[j].n;
        }
    }

    return n;
}
```

PROBLEMA

Construir un programa en C lo más modular posible que calcule el área de un polígono convexo cualquiera conocidas las coordenadas de sus vértices. Considerando un máximo de 20 vértices.

Un polígono convexo es un polígono en el plano euclidiano donde, para cualquier par de puntos dentro del polígono, la línea que los une también está completamente dentro del polígono. De manera más formal, un polígono P es convexo si para cualquier par de puntos A y B en P, el segmento de línea AB también está completamente contenido en P.

Se ha de plantear el modelo de datos y el modelo arquitectónico.

Tenemos puntos y un polígono de n lados es una lista de n puntos. Por tanto la estructura de datos debe ser:

```
typedef struct{
        double x,y;
}tipo_punto2d;
typedef tipo_punto2d tipo_lista_vertices[N];
typedef struct{
        int n;
        tipo_lista_vertices v;
}tipo_poligono;
```

Con respecto a las funciones veremos cuáles son los módulos necesarios en base al enunciado del problema.

```
/*
 * @authors Equipo docente Programación
 * @project Creación de Materiales Didácticos en la Univer. de Almería (2021-2022)
 * Grados en Ingeniería Eléctrica, Electrónica Industrial, Mecánica y Química Industrial
 * @date 2021-02-06
 */


#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#define N 20
```

```c
/* Nuevos tipos de datos */
typedef struct{
        double x,y;
}tipo_punto2d;
typedef tipo_punto2d tipo_lista_vertices[N];
typedef struct{
        int n;
        tipo_lista_vertices v;
}tipo_poligono;

/* prototipos de funciones */
void leer_poligono(tipo_poligono *pol);
void leer_vertice(tipo_punto2d *p);
double superficie(tipo_poligono pol);
double area_triangulo(double l1, double l2, double l3);
double distancia(tipo_punto2d p1,tipo_punto2d p2);

int main(){

        tipo_poligono pol;

        printf("SUPERFICIE PIEZA PLANA POLIGONAL CONVEXA\n");
        printf("========================================\n\n");
        leer_poligono(&pol);
        if(pol.n<3)
                printf("\nNo es un poligono (menos de 3 lados)");
        else
                printf("\nSuperficie (triangulacion): %.2lf",superficie(pol));
        return 0;
}

void leer_poligono(tipo_poligono *pol) {
    printf("Introduzca los vértices del polígono:\n");
        //scanf(); Lectura de n
    for (int i = 0; i < pol->n; i++) {
        printf("\tVertice num. %d:\n", i + 1);
        leer_vertice(&pol->v[i]);
    }
        pol->n=n;
}

void leer_vertice(tipo_punto2d *p){
        printf("\t\tx: ");
        scanf(" %lf",&p->x);
        printf("\t\ty: ");
        scanf(" %lf",&p->y);
}

double superficie(tipo_poligono pol){
        int i;
        double sup;

        sup=0;
        for(i=1;i<pol.n-1;++i)
                sup+=area_triangulo(distancia(pol.v[0],pol.v[i]),
                distancia(pol.v[0],pol.v[i+1]),
                distancia(pol.v[i+1],pol.v[i]));
        return sup;
}

double area_triangulo(double l1, double l2, double l3){
    double semiperimetro = (l1 + l2 + l3) / 2.0;
    double area;
    if (l1 <= 0 || l2 <= 0 || l3 <= 0 || l1 + l2 <= l3 || l1 + l3 <= l2 || l2 + l3 <= l1) {
        area =-1; // Retorna un valor negativo para indicar error
    }
    else
        area = sqrt(semiperimetro * (semiperimetro - l1) * (semiperimetro - l2) * (semiperimetro - 3));
    return area;
}
```

```
double distancia(tipo_punto2d p1,tipo_punto2d p2){
        return(sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y)));
}
```

PROBLEMA

- Vamos a pensar en jugar a las cartas de forma que después podamos emplear las cartas en diversos juegos.
- Por ejemplo, se realizará el juego de la carta más alta.
    - A los 2 jugadores se les reparten las cartas al azar
    - En tandas sacarán una carta de su montón.
    - El mayor valor ganará la jugada
    - Se para cuando se acaban las cartas o a decisión del que va perdiendo.

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

#define NAIPES 40


typedef enum {
    OROS,
    COPAS,
    ESPADAS,
    BASTOS
} Palo;

typedef char Cadena[20];

// Definición de la estructura Carta
typedef struct {
    int bocarriba;
    int numero;
    Palo palo;
} Carta;


// Definición de la estructura PilaCarta
typedef Carta tipo_baraja[NAIPES];

// Definición de la estructura PilaCarta
typedef struct {
    tipo_baraja cartas;
    int cima;
} PilaCarta;


// Definición de la estructura Jugador
typedef struct {
    PilaCarta pila;
    int cantidadCartas;
} Jugador;

// Definición de la estructura Jugador
typedef struct {
    Jugador jugadores[8];
    int numjugadores;
} Juego;


// Función para inicializar la baraja española
void inicializarBaraja(PilaCarta *baraja, int numCartas) {
    int idx = 0;
```

```
    for (int palo = OROS; palo <= BASTOS; palo++) {
        for (int numero = 1; numero <= 10; numero++) {
            (*baraja).cartas[idx].bocarriba = 0;
            (*baraja).cartas[idx].numero = numero;
            (*baraja).cartas[idx].palo = palo;
            idx++;
        }
    }
    baraja->cima=numCartas;
}


// Función para mezclar las cartas en la baraja
void mezclarBaraja(PilaCarta *baraja, int numCartas) {
    for (int i = 0; i < numCartas; i++) {
        int j = rand() % numCartas;
        Carta temp = (*baraja).cartas[i];
        (*baraja).cartas[i] = (*baraja).cartas[j];
        (*baraja).cartas[j] = temp;
    }
}


// Función para repartir cartas a un jugador
void repartirCartas(Jugador *jugador, PilaCarta baraja, int inicio, int fin) {
    for (int i = inicio; i < fin; i++) {
        jugador->pila.cartas[i - inicio] = baraja.cartas[i];
        jugador->pila.cima++;
        jugador->cantidadCartas++;
    }
}


void toString(Carta carta, Cadena resultado) {
    sprintf(resultado,"%d%s", carta.numero,
            (carta.palo == OROS) ? " de Oros" :
            (carta.palo == COPAS) ? " de Copas" :
            (carta.palo == ESPADAS) ? " de Espadas" :
            " de Bastos");
}


// Función para realizar una jugada
void jugar(Jugador *jugador1, Jugador *jugador2, int *puntuacion1, int *puntuacion2) {

    if (jugador1->cantidadCartas > 0 && jugador2->cantidadCartas > 0) {
        Carta carta1 = jugador1->pila.cartas[--jugador1->pila.cima];
        Carta carta2 = jugador2->pila.cartas[--jugador2->pila.cima];
        Cadena r1,r2;
        toString(carta1,r1);
        toString(carta2,r2);
        printf("Jugador 1: %s vs Jugador 2: %s\n",r1,r2);

        if (carta1.numero > carta2.numero) {
            (*puntuacion1) += 2;
            printf("Jugador 1 gana la ronda.\n");
        } else if (carta1.numero < carta2.numero) {
            (*puntuacion2) += 2;
            printf("Jugador 2 gana la ronda.\n");
        } else {
            (*puntuacion1)++;
            (*puntuacion2)++;
            printf("Empate. Cada jugador se lleva su carta.\n");
        }

        jugador1->cantidadCartas--;
        jugador2->cantidadCartas--;
    }
}
```

```c
int main() {
    srand((unsigned int)time(NULL));
    PilaCarta baraja;
    inicializarBaraja(&baraja, NAIPES);
    mezclarBaraja(&baraja, NAIPES);


    // Inicializar jugadores
    int cuantos=2;

    Jugador jugador1, jugador2;
    jugador1.pila.cima = 0;
    jugador2.pila.cima = 0;
    jugador1.cantidadCartas = 0;
    jugador2.cantidadCartas = 0;

    // Repartir cartas a los jugadores
    repartirCartas(&jugador1, baraja, 0, 20);
    repartirCartas(&jugador2, baraja, 20, 40);

    // Inicializar puntuaciones
    int puntuacion1 = 0, puntuacion2 = 0;

    // Juego
    int opcion;
    do {
        printf("Presiona 1 para ver el estado, 0 para terminar el juego: ");
        scanf("%d", &opcion);

        if (opcion == 1) {
            jugar(&jugador1, &jugador2, &puntuacion1, &puntuacion2);
            printf("Puntuaciones: Jugador 1: %d, Jugador 2: %d\n", puntuacion1, puntuacion2);
        }
    } while (opcion == 1 && jugador1.cantidadCartas > 0 && jugador2.cantidadCartas > 0);

    // Determinar el ganador
    if (puntuacion1 > puntuacion2) {
        printf("¡Jugador 1 gana el juego!\n");
    } else if (puntuacion1 < puntuacion2) {
        printf("¡Jugador 2 gana el juego!\n");
    } else {
        printf("¡Empate!\n");
    }

    return 0;
}
```

PROBLEMA El problema del viajante de comercio es como un juego de encontrar la ruta más corta para visitar varios lugares sin pasar dos veces por el mismo sitio. Imagina que eres un repartidor y tienes que visitar diferentes clientes en distintas ciudades, pero quieres hacerlo de manera eficiente, recorriendo la menor distancia posible y sin repetir ningún lugar. Ahí es donde entra el problema del viajante de comercio. La dificultad radica en encontrar la ruta óptima que minimice la distancia total recorrida. Aunque suena simple con pocos destinos, cuando tienes muchas ciudades, encontrar la mejor ruta se vuelve muy complicado. ¡Es un dolor de cabeza para matemáticos y computadoras por igual!

Lo primero es pensar en cómo vamos a representar el grafo que define las conexiones entre las ciudades. Una matriz de m x m siendo m el número de ciudades.

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

```c
// Definición del número máximo de ciudades
#define MAX_CIUDADES 10

typedef int Distancias[MAX_CIUDADES][MAX_CIUDADES];
typedef int Camino[MAX_CIUDADES];


int distancia(int ciudad1, int ciudad2, Distancias distancias) {
    return distancias[ciudad1][ciudad2];
}


int distancia_total(Camino ruta, int num_ciudades, Distancias distancias) {
    int distancia_total = 0;
    for (int i = 0; i < num_ciudades - 1; i++) {
        distancia_total += distancia(ruta[i], ruta[i + 1], distancias);
    }
    // Sumar la distancia de regreso a la ciudad inicial
    distancia_total += distancia(ruta[num_ciudades - 1], ruta[0], distancias);
    return distancia_total;
}


void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}


void generar_rutas(Camino ruta, int inicio, int num_ciudades, Distancias distancias, int* mejor_distancia) {
    if (inicio == num_ciudades - 1) {
        int distancia_actual = distancia_total(ruta, num_ciudades, distancias);
        if (distancia_actual < *mejor_distancia) {
            *mejor_distancia = distancia_actual;
        }
        return;
    }
    for (int i = inicio; i < num_ciudades; i++) {
        swap(&ruta[inicio], &ruta[i]);
        generar_rutas(ruta, inicio + 1, num_ciudades, distancias, mejor_distancia);
        swap(&ruta[inicio], &ruta[i]);
    }
}

int main() {
    Distancias distancias = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int num_ciudades = 4;
    Camino ruta;
    for (int i = 0; i < num_ciudades; i++) {
        ruta[i] = i;
    }

    int mejor_distancia = INT_MAX;

    generar_rutas(ruta, 0, num_ciudades, distancias, &mejor_distancia);

    printf("La distancia más corta es: %d\n", mejor_distancia);

    return 0;
}
```

Clasificación y búsqueda

Clasificación de vectores

Ordenar los datos, es decir, ponerlos en orden ascendente o descendente, es una de las aplicaciones informáticas más importantes. La clasificación de una lista de datos, basados en un criterio es una operación frecuente en proceso de datos. La clasificación permite realizar operaciones de búsqueda con más facilidad (basta recordar el problema del sistema de partículas).

Un banco ordena los cheques por número de cuenta para preparar extractos bancarios individuales al final de cada mes. Las compañías telefónicas Las compañías telefónicas ordenan sus listados de cuentas por apellidos y, dentro de éstos, por nombre para facilitar la búsqueda de números de teléfono, facilitar la búsqueda de números de teléfono. Prácticamente todas las organizaciones deben clasificar algunos datos y, en muchos casos, cantidades ingentes de ellos. y, en muchos casos, cantidades ingentes. La clasificación de datos es un problema intrigante que ha atraído algunos de los esfuerzos de investigación más intensos en ciencias de la computación.

La Clasificación (ordenación) consiste en operación de re-organizar un conjunto de datos en una secuencia específica:

- Creciente/decreciente en datos numéricos.
- Alfabética directa/inversa en datos textuales.
- Cualquier otra relación de orden. Ej: por fecha.

Hay doc categorías de algoritmos de clasificación según la estructura de datos usada para representar la información:

- Clasificación interna, para vectores con datos ubicados en memoria central.
- Clasificación externa, para archivos secuenciales, cuando los datos ubicados en memoria secundaria. Estos se retomarán más adelante.

Clasificación interna

Dados n elementos $a_1, a_2, ..., a_n$ clasificar dichos elementos consiste en encontrar una permutación de los elementos: $a_{k1}, a_{k2}, ...a_{kn}$, de forma que dada una función de clasificación f se verifique que:

$f(a_{k1}) \leq f(a_{k2}) \leq ... \leq f(a_{kn})$

Normalmente, la función de clasificación no se evalúa según una regla de cálculo sino que se almacena como un componente (campo) explícito de cada elemento, cuyo valor se le denomina la clave del elemento.

La clasificación de datos es un problema intrigante que ha ha atraído algunos de los esfuerzos de investigación más intensos en informática. A menudo, muchos algoritmos funcionan mal, pero su virtud es que son fáciles de escribir, probar y depurar. Los

algoritmos más complejos para obtener el máximo rendimiento suelen ser difíciles de manejar y comprender.

Un Algoritmo de clasificación tiene dos partes:

- Operación de re-organización (alg. de clasificación).
- Función de clasificación de elementos (relación de orden ó criterio de clasificación).

La representación del conjunto de elementos: vector de registros; que en C tendría una notación similar a:

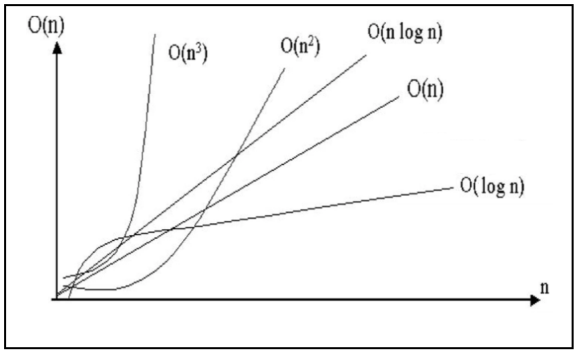
```
#define MAX ...
...

typedef struct{
        tipo_clave  clave;
        // Otros campos
}tipo_elemento;

typedef tipo_elemento tipo_vector[MAX];
```

Para la presentación de los algoritmos y de los ejemplos se utilizará una *clave entera* y se clasificarán los elementos en orden *ascendente* de clave (de menor a mayor valor). La clasificación se realizará "in situ" (las permutaciones de elementos utilizan el espacio ocupado por el vector).

Tabla resumen de los principales algoritmos de clasificación interna:

Métodos	Estrategia	Algoritmo	Caso peor	Caso promedio
Simples				
	Inserción	Inserción directa	$O(n^2)$	$O(n^2)$
	Selección	Selección directa	$O(n^2)$	$O(n^2)$
	Intercambio	Intercambio directo (burbuja)	$O(n^2)$	$O(n^2)$
Elaborados				
	Inserción	D.L.Shell	$O(n^{1,2})$	
	Selección	"Heapsort"	$O(n * log(n))$	$O(n * log(n))$
	Intercambio	"Quicksort"	$O(n^2)$	$O(n * log(n))$

Notación asintótica (orden de complejidad del algoritmo):



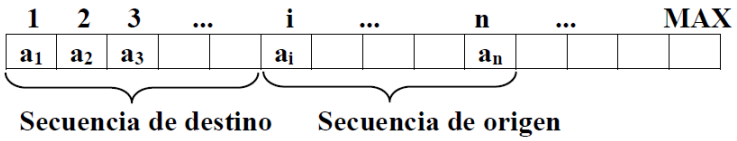
O(f(n)) es el tiempo de ejecución es proporcional a f(n)

Clasificación por inserción

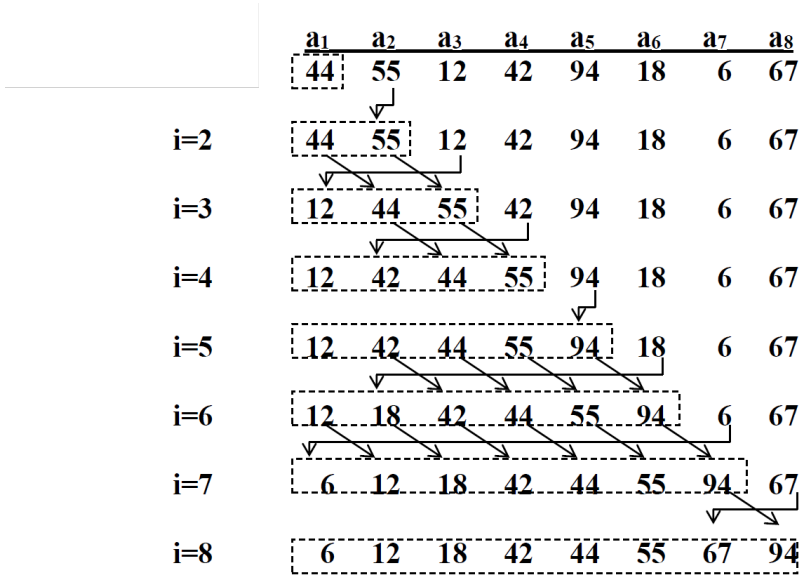
El algoritmo de ordenación por inserción es un algoritmo simple que funciona de la misma manera en que muchas personas ordenan cartas de juego en sus manos. Comienza con una mano desordenada y, a medida que se toma cada carta, se inserta en su lugar correcto entre las cartas ya ordenadas. Puede no ser la opción más eficiente para conjuntos de datos muy grandes. Sin embargo, es útil para conjuntos de datos pequeños o parcialmente ordenados.

La idea principal del algoritmo es "insertar" cada elemento en la posición correcta de una sublista ya ordenada.

- Los elementos se dividen conceptualmente en una secuencia de destino $(a_1, a_2, ..., a_{i-1})$ y una secuencia de origen $(a_i, a_{i+1}, ..., a_n)$.



- En cada paso, empezando con `i=2` e incrementando i de 1 en 1, se toma el primer elemento de la secuencia de origen (a_i) y se transfiere a la secuencia de destino insertándolo en el sitio apropiado.



Su implementación sobre un ejemplo

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Mover los elementos del subvector arr[0..i-1] que son mayores que key
           a una posición adelante de su posición actual */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n-1; i++) {
        // Encuentra el índice del elemento mínimo en el subvector no ordenado
        minIndex = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        // Intercambia el elemento mínimo con el primer elemento del subvector no ordenado
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

// Función para ordenar por burbuja de izquierda a derecha
void ordenarBurbujaIzquierdaDerecha(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Intercambia los elementos si están en el orden incorrecto
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

// Función para ordenar por burbuja de derecha a izquierda
void ordenarBurbujaDerechaIzquierda(int arr[], int n) {
    int i, j, temp;
    for (i = n-1; i > 0; i--) {
        for (j = n-1; j > n-i-1; j--) {
            if (arr[j] < arr[j-1]) {
                // Intercambia los elementos si están en el orden incorrecto
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }
        }
    }
}
```

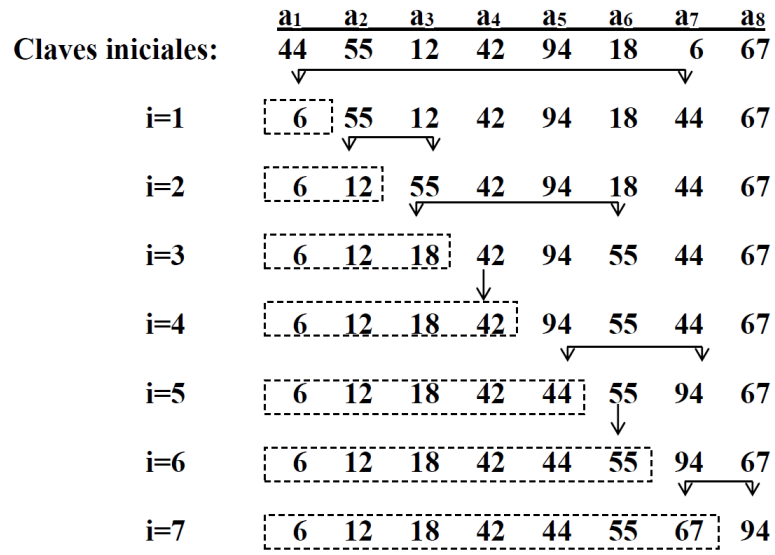
```c
int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = 5;

    printf("vector original: \n");
    printArray(arr, n);

   // insertionSort(arr, n);
    selectionSort(arr, n);

    printf("Vector ordenado por inserción: \n");
    printArray(arr, n);

    return 0;
}
```

Función `insertionSort`:

- Comienza con un bucle que recorre los elementos del vector desde el segundo elemento hasta el último.
- Para cada elemento, lo guarda en la variable `key`.
- Luego, se inicia un bucle while que recorre los elementos ya ordenados (en el subvector arr[0..i-1]).
- Mientras el elemento actual sea mayor que `key` y aún no se haya llegado al inicio del vector, se desplazan los elementos mayores una posición hacia adelante.
- Finalmente, se coloca `key` en la posición correcta.

Clasificación por selección directa

Funciona dividiendo la lista en dos partes: una parte ordenada y otra desordenada. El algoritmo selecciona repetidamente el elemento más pequeño de la parte desordenada y lo coloca al final de la parte ordenada.



La estrategia es:

- Seleccionar el elemento con clave mínima de la secuencia de origen.
- Intercambiarlo con el primer elemento.
- Repetir los 2 pasos anteriores con los n-1, n-2, ... elementos restantes hasta que quede un único elemento, el de clave mayor.

El diseño en pseudocódigo será:

- Desde i=1 Hasta n-1
  - "Asignar a k el índice correspondiente al elemento de clave mínima de la secuencia de origen"
  - "Intercambiar a[i] y a[k]"
- "Asignar a k el índice correspondiente al elemento de clave mínima de la secuencia de origen":

```
k=i
Desde j=i+1 Hasta n Hacer
    Si (a[j].clave<a[k].clave)
        Entonces k=j
```

- "Intercambiar a[i] y a[k]":

Su implementación en C será:

```c
void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n-1; i++) {
        // Encuentra el índice del elemento mínimo en el subvector no ordenado
        minIndex = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        // Intercambia el elemento mínimo con el primer elemento del subvector no ordenado
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

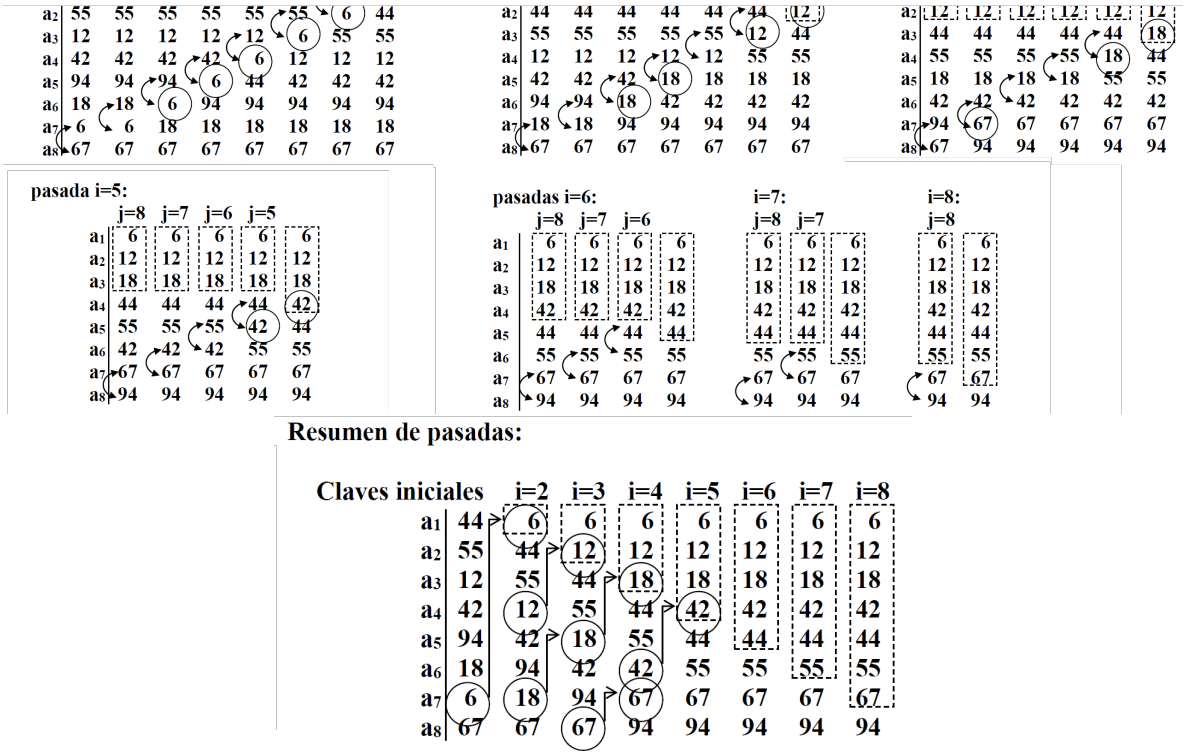
Clasificación por método de burbuja

Su característica dominante: comparar e intercambiar pares de elementos adyacentes hasta que estén todos los elementos clasificados:

```
Si (a[j].clave<a[j-1].clave) Entonces
 x=a[j]
 a[j]=a[j-1]
 a[j-1]=x
```

- Se hacen repetidas pasadas (recorridos secuenciales) sobre el vector ejecutando la instrucción selectiva anterior.

Hay varias versiones del algoritmo: - Pasadas de derecha a izquierda: en cada pasada se mueve el elemento de clave mínima hasta el extremo izquierdo del vector. - Pasadas de izquierda a derecha: en cada pasada se mueve el elemento de clave máxima hasta el extremo derecho del vector. - Método de sacudida: se alternan las pasadas de izquierda a derecha con las de derecha a izquierda.




```c
#include <stdio.h>

// Función para ordenar por burbuja de izquierda a derecha
void ordenarBurbujaIzquierdaDerecha(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Intercambia los elementos si están en el orden incorrecto
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

// Función para ordenar por burbuja de derecha a izquierda
void ordenarBurbujaDerechaIzquierda(int arr[], int n) {
    int i, j, temp;
    for (i = n-1; i > 0; i--) {
        for (j = n-1; j > n-i-1; j--) {
            if (arr[j] < arr[j-1]) {
                // Intercambia los elementos si están en el orden incorrecto
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }
        }
    }
}
```

Generalización para otros tipos de datos

A continuación vamos a generalizar el algoritmo de búsqueda para otros tipos de colecciones.

En este caso pensemos en el **PROBLEMA** de gestionar un conjunto de alumnos de los que tengo que conocer su nombre y su nota, que pueden ser de 0 a 100 alumnos y que deben poder ordenarse por nota. Así el **registro** de alumnos debe ser un `struct` con un vector de alumnos y un entero para controlar la capacidad. Siendo cada elemento del vector un tipo alumno, es decir un `struct` con dos campos, nota y nombre, que es una cadena de caracteres.

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

typedef char Cadena[20];
typedef float Real;

typedef struct {
    Cadena nombre;
    Real nota;
} Alumno;

// Typedef para el array de alumnos
typedef Alumno ArrayAlumnos[100];

typedef struct {
    ArrayAlumnos alumnos;
    int totalAlumnos;
} RegistroAlumnos;

// Prototipos de funciones
void incluirAlumno(RegistroAlumnos *registro);
void ordenarAlumnosPorNota(RegistroAlumnos *registro);
void mostrarAlumnos(RegistroAlumnos registro);

int main() {
    RegistroAlumnos registro;
    registro.totalAlumnos = 0;

    int opcion;

    do {
        printf("\n1. Incluir nuevo alumno\n");
        printf("2. Mostrar alumnos ordenados por nota\n");
        printf("3. Salir\n");
        printf("Seleccione una opción: ");
        scanf("%d", &opcion);

        switch (opcion) {
            case 1:
                incluirAlumno(&registro);
                break;
            case 2:
                ordenarAlumnosPorNota(&registro);
                mostrarAlumnos(registro);
                break;
            case 3:
                printf("Saliendo del programa. Adiós.\n");
                break;
            default:
                printf("Opción no válida. Inténtelo de nuevo.\n");
        }
    } while (opcion != 3);
```

```
    return 0;
}

void incluirAlumno(RegistroAlumnos *registro) {
    if (registro->totalAlumnos < 100) {
        printf("\nIngrese el nombre del alumno: ");
        scanf("%s", registro->alumnos[registro->totalAlumnos].nombre);

        printf("Ingrese la nota del alumno: ");
        scanf("%f", &registro->alumnos[registro->totalAlumnos].nota);

        registro->totalAlumnos++;
        printf("Alumno incluido con éxito.\n");
    } else {
        printf("No se pueden incluir más alumnos. Límite alcanzado.\n");
    }
}

void ordenarAlumnosPorNota(RegistroAlumnos *registro) {
    int i, j;
    Alumno temp;

    for (i = 0; i < registro->totalAlumnos - 1; i++) {
        for (j = 0; j < registro->totalAlumnos - i - 1; j++) {
            if (registro->alumnos[j].nota < registro->alumnos[j+1].nota) {
                // Intercambia los alumnos si están en el orden incorrecto
                temp = registro->alumnos[j];
                registro->alumnos[j] = registro->alumnos[j+1];
                registro->alumnos[j+1] = temp;
            }
        }
    }
}

void mostrarAlumnos(RegistroAlumnos registro) {
    printf("\nAlumnos ordenados por nota:\n");
    for (int i = 0; i < registro.totalAlumnos; i++) {
        printf("Nombre: %-20s Nota: %.2f\n", registro.alumnos[i].nombre, registro.alumnos[i].nota);
    }
}
```

Función `qsort`

Es una función estándar de la biblioteca estándar de C, que se encuentra en <stdlib.h>. La función qsort se utiliza para ordenar los elementos de un vector en C, y necesita una función de comparación como argumento para determinar el orden de los elementos.

```
 void
qsort(void *base, size_t nel, size_t size, int (*compar)(const void *, const void *));
```

Esta función ordenada un vector de `nel` elementos, donde `base` es un puntero al elemento inicial. El tamaño de cada elemento, en bytes, se especifica con el argumento `size`. Finalmente, `compar` es un puntero a una función que se le pasa como argumento dos punteros a elementos a comparar, y devuelve un entero menor, igual o mayor que cero, segun el primer elemento sea menor, igual o mayor que el segundo.

```
/* Implementación del algoritmo quick-sort para la ordenación de vectores
** y bsearch para buscar ciertos elementos
*/

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

#define ELEMENTOS 100
```

```c
int comparar(const void *arg1, const void *arg2){
    if(*(int *)arg1 < *(int *)arg2) return -1;
    else
        if(*(int *)arg1 > *(int *)arg2) return 1;
            else return 0;
}



int main()
{
    int i, num;
    int lista[ELEMENTOS];
    int *elementoPtr;

    for(i = 0; i < ELEMENTOS; i++) lista[i] = rand() % 100 + 1;

    qsort(lista, ELEMENTOS, sizeof(lista[0]), comparar);


    for(i = 0; i < ELEMENTOS; i++) printf("%d ", lista[i]);
    printf("\n");

    return 0;
}
```

Uso de varias claves

Puede que tengamos que utilizar distintas claves de ordenación para generalizar el proceso se usan punteros a funciones.

```c
#include <stdio.h>


typedef struct {
    int id;
    int coste;
    int valor;
} Articulo;


// Definición de un tipo de puntero a función de comparación como alias
// para mejorar la legibilidad

typedef int (*Comparador)(void *, void *);


void ordenar(Articulo *articulos, int n, Comparador comparador) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (comparador(&articulos[j], &articulos[j + 1]) > 0) {
                // Intercambiar los artículos
                Articulo temp = articulos[j];
                articulos[j] = articulos[j + 1];
                articulos[j + 1] = temp;
            }
        }
    }
}


int comparar_por_coste(Articulo *a, void *b) {
    Articulo *articulo1 = (Articulo *)a;
    Articulo *articulo2 = (Articulo *)b;
    return (articulo1->coste - articulo2->coste);
}
```

```c
int comparar_listo(Articulo *a, void *b) {
    Articulo *articulo1 = (Articulo *)a;
    Articulo *articulo2 = (Articulo *)b;
    return (articulo1->coste - articulo2->coste);
}

int comparar_por_valor(void *a, void *b) {
    Articulo *articulo1 = (Articulo *)a;
    Articulo *articulo2 = (Articulo *)b;
    return (articulo1->valor - articulo2->valor);
}


void imprimir_articulo(Articulo articulo) {
    printf("ID: %d, Coste: %d, Valor: %d\n", articulo.id, articulo.coste, articulo.valor);
}

int main() {

    Articulo articulos[] = {
        {1, 10, 50},
        {2, 20, 30},
        {3, 15, 40},
        {4, 25, 35},
        {5, 5, 60}
    };

    int num_articulos = sizeof(articulos) / sizeof(articulos[0]);

    // Ordenamos los artículos por coste
    ordenar(articulos, num_articulos, comparar_por_coste);

    printf("Artículos ordenados por coste:\n");
    for (int i = 0; i < num_articulos; i++) {
        imprimir_articulo(articulos[i]);
    }

    // Ordenamos los artículos por valor
    ordenar(articulos, num_articulos, comparar_por_valor);

    printf("Artículos ordenados por valor:\n");
    for (int i = 0; i < num_articulos; i++) {
        imprimir_articulo(articulos[i]);
    }

    return 0;
}
```

Búsqueda

La **Búsqueda** es la operación de encontrar la posición de un elemento x en un conjunto de elementos dados que consiste en determinar:

- si x pertenece al conjunto e indicar su posición.
- si x no pertenece al conjunto.

Es una operación frecuente en muchas aplicaciones informáticas: recuperación de la información para consulta o actualización, por ejemplo:

- Buscar el saldo de una cuenta bancaria.
- Buscar una palabra en un diccionario.
- Buscar una factura en una lista de facturas.

- Buscar la nota de un alumno en una lista de notas.
- Buscar la temperatura medida por un sensor en un día dado.

Como se observa esta operación puede asociarse a modelos de datos todo lo complejo que se desee en cuanto a la combinación de vectores y registros para un problema dado, lo único es que debe de existir un **elemento** de búsqueda. Este elemento en el caso de registros será un campo del registro llamado **clave** y en el de un vector, el valor de lo almacenado en el vector lo que actúa como clave para el caso de los tipos primitivos, pero si es un vector de registros, será la clave del registro. La única condición con para el campo clave es que pueda ser comparable con el dato buscado. Por ejemplo, para el caso de de registro de alumno de una asignatura podemos buscar por `nombre` de alumno o por `nota`, dentro de un vector de tipo `ArrayAlumnos`.

```
typedef char Cadena[20];

typedef struct {
    Cadena nombre;
    float nota;
} Alumno;

// Typedef para el array de alumnos
typedef Alumno ArrayAlumnos[100];
```

Un módulo de búsqueda debe tener como interfaz:

Entrada:

- vector donde buscar (tipo_vector)
- longitud_vector (entero)
- clave del elemento a buscar (tipo_clave)

Salida:

- si la búsqueda ha tenido éxito (lógico)
- posición de la 1ª ocurrencia de la clave (entero)

Otras alternativas para la interfaz:

- Buscar en un sub-vector delimitado por los índices de las posiciones izquierda y derecha.
- Devolver la posición de la primera ocurrencia de la clave a través del identificador del módulo (-1 si no se encuentra)

Simplificaremos viendo solo la búsqueda en **vectores de enteros**, pero sabiendo que puede extenderse a otros modelos de datos.

Búsqueda secuencial

La estrategia más simples es recorrer todos los elementos del vector de 1 en 1 empezando por el primero y siguiendo el orden físico de almacenamiento:

```
#define MAX 10
typedef int VectorEnteros[MAX];
....

// Función de búsqueda secuencial
int busquedaSecuencial(VectorEnteros vector, int n, int elemento) {
    for (int i = 0; i < n; i++) {
        if (vector[i] == elemento) {
            return i; // Devuelve la posición si se encuentra el elemento
        }
    }
    return -1; // Devuelve -1 si el elemento no está en el vector
}

....

int posicion = busquedaSecuencial(vector, MAX, elementoBuscado);

....
```

Si se trata de un vector ordenado se puede simplificar el proceso de búsqueda usando el contador de un bucle while como centinela.

```
int busquedaSecuencialOrdenado(VectorEnteros vector, int n, int clave) {
    int i = 0;
    while (i < n && vector[i] != clave) {
        i++;
    }
    return (i != n) ? i : -1;
}
```

¿Pero que ocurre cuando el vector tiene más de una ocurrencia?. En este caso se devuelve la posición de la primera ocurrencia. Si queremos añidir comportamiento diferente como devolver un vector de posiciones encontradas, debemos extender este módulo básico.

Búsqueda binaria

La búsqueda binaria es un algoritmo eficiente de búsqueda que se aplica a colecciones ordenadas. La idea fundamental de la búsqueda binaria es reducir a la mitad el espacio de búsqueda en cada paso. Funciona de la siguiente manera, para un vector `a` entre 1 a n:

- Compara el elemento buscado con el elemento central de la lista `a[med].clave`.
- Si son iguales, se ha encontrado el elemento y se devuelve su posición. `a[med].clave==clave`
- Si el elemento buscado es menor que el elemento central, se repite la búsqueda en la mitad inferior de la lista. `a[med].clave>clave`
- Si el elemento buscado es mayor que el elemento central, se repite la búsqueda en la mitad superior de la lista. `a[med].clave<clave`
- Este proceso se repite hasta que se encuentra el elemento o hasta que el espacio de búsqueda se reduce a cero.

```c
int busquedaBinaria(int vector[], int n, int clave) {
    int inicio = 0;
    int fin = n - 1;

    while (inicio <= fin) {
        int medio = inicio + (fin - inicio) / 2;

        // Si la clave es igual al elemento en la posición media
        if (vector[medio] == clave) {
            return medio; // Devuelve la posición
        }
        else if (vector[medio] > clave) {
            fin = medio - 1;
        }
        else {
            inicio = medio + 1;
        }
    }

    return -1; // La clave no se encuentra en el vector
}
```

Pero no necesariamente encuentra la primera ocurrencia, si queremos asegurarnos de que se encuentra se debe seguir buscando en la mitad inferior.

```c
int busquedaBinariaPrimeraOcurrencia(int vector[], int n, int clave) {
    int inicio = 0;
    int fin = n - 1;
    int primeraOcurrencia = -1; // Inicializamos a -1, indicando que aún no se ha encontrado

    while (inicio <= fin) {
        int medio = inicio + (fin - inicio) / 2;

        // Si la clave es igual al elemento en la posición media
        if (vector[medio] == clave) {
            primeraOcurrencia = medio; // Actualiza la posición de la primera ocurrencia

            // Busca en la mitad inferior para encontrar la primera ocurrencia
            fin = medio - 1;
        }
        // Si la clave es menor, busca en la mitad inferior
        else if (vector[medio] > clave) {
            fin = medio - 1;
        }
        // Si la clave es mayor, busca en la mitad superior
        else {
            inicio = medio + 1;
        }
    }

    return primeraOcurrencia; // Devuelve la posición de la primera ocurrencia o -1 si no se encuentra
}
```

Dado el siguiente vector, ¿cuántas iteraciones necesita el algoritmo de búsqueda binaria para encontrar el elemento 3?

0	1	2	3	4	5	6	7	8	9
1	3	5	7	9	11	13	15	17	19

Se necesitan varias iteraciones para encontrar un elemento específico. La búsqueda binaria funciona dividiendo repetidamente el espacio de búsqueda a la mitad, eliminando así la mitad de los elementos restantes en cada iteración.

El vector, que contiene los números impares del 1 al 19 (incluidos), ordenados de forma ascendente, la búsqueda binaria sería algo así:

- Inicialmente, tenemos el vector completo: 1 3 5 7 9 11 13 15 17 19.
- Se compara el elemento del medio (en este caso, el de la posición de índice 4, que es 9) con el valor buscado (3).
- Dado que 3 es menor que 9, nos quedamos con la mitad inferior del vector: 1 3 5 7.
- Se repite el proceso, ahora comparando el elemento medio (el de la posición de índice 1, que es 3) con el valor buscado (3).
- Como coinciden, terminaría la búsqueda en 2 iteraciones.

En este caso específico, se necesitan un total de 2 iteraciones para encontrar el 3.

Archivos y bases de datos

Concepto de persistencia de datos

Hasta el momento, toda la información (datos) que hemos sido capaces de gestionar, la hemos tomado de dos únicas fuentes: o eran datos del programa, o eran datos que introducía el usuario desde el teclado. Y hasta el momento, siempre que un programa ha obtenido un resultado, lo único que hemos hecho ha sido mostrarlo en pantalla.

Sería muy útil poder almacenar la información generada por un programa, de forma que esa información pudiera luego ser consultada por otro programa, o por el mismo u otro usuario; o bien que la información que un usuario va introduciendo por consola quedase almacenada para sucesivas ejecuciones del programa o para posibles manipulaciones de esa información.

Los archivos, en contraposición con las estructuras de datos vistas hasta ahora (variables simples, vectores, registros, etc.), son estructuras de datos almacenadas en memoria secundaria que consiguen ser persistentes, es decir se almacenan para poder ser retomados en otro momento. Pero su tratamiento no es igual que la asignación que hemos visto hasta el momento.

Entendemos por **tipo de dato con persistencia**, o **archivo**, o **fichero** aquel cuyo tiempo de vida no está ligado al de ejecución del programa que lo crea o lo maneja. Es decir, se trata de una estructura de datos externa al programa, que lo trasciende. Un archivo existe desde que un programa lo crea y mientras que no sea destruido por este u otro programa.

Un archivo está compuesto por registros homogéneos que llamamos **registros de archivo**. La información de cada registro viene recogida mediante **campos**.

El límite de tamaño de un archivo viene condicionado únicamente por el límite de los dispositivos físicos que lo albergan, no por el tamaño de la memoria. Los programas trabajan con datos que residen en la memoria principal del ordenador. Para que un programa manipule los datos almacenados en un archivo y, por tanto, en un dispositivo de memoria masiva, esos datos deben ser enviados desde esa memoria externa a la memoria

principal mediante un proceso de **extracción** también llamado **lectura**. Y de forma similar, cuando los datos que manipula un programa deben ser concatenados con los del archivo se utiliza el proceso de **grabación** o **escritura**.

En C, un archivo es un concepto lógico que puede aplicarse a muchas cosas desde archivos de disco hasta terminales o una impresora. Se asocia una secuencia con un archivo especifico realizando una operación de apertura. Una vez que el archivo está abierto, la información puede ser intercambiada entre este y el programa. Realmente se tiene una abstracción para la entrada y salida de datos, ya sea desde y hacia archivos, la consola o cualquier otro dispositivo que se llaman **streams**.

```c
#include <stdio.h>

int main() {
    int numero = 42;
    float decimal = 3.14;
    char cadena[] = "Hola, mundo!";

    // Utilizando fprintf para imprimir en la consola
    fprintf(stdout, "Número: %d\n", numero);
    fprintf(stdout, "Decimal: %.2f\n", decimal);
    fprintf(stdout, "Cadena: %s\n", cadena);

    printf( "Número: %d\n", numero);
    printf( "Decimal: %.2f\n", decimal);
    printf( "Cadena: %s\n", cadena);

    return 0;
}
```

De hecho, los archivos se conciben como estructuras que gozan de las siguientes características:

1. Capaces de contener grandes cantidades de información.
2. Capaces de sobrevivir a los procesos que lo generan y utilizan.
3. Capaces de ser accedidos desde diferentes procesos o programas.

Desde el punto de vista físico, o del hardware, un archivo tiene una dirección física: en el disco toda la información se guarda (grabación) o se lee (extracción) en bloques unidades de asignación o "clusters" referenciados por un nombre de unidad o disco, la superficie a la que se accede, la pista y el sector: todos estos elementos caracterizan la dirección física del archivo y de sus elementos.

Habitualmente, el sistema operativo simplifica mucho esos accesos al archivo, y el programador puede trabajar con un concepto simplificado de archivo o fichero: cadena de bytes consecutivos terminada por un carácter especial llamado **EOF** ("End Of File"); ese carácter especial (EOF) indica que no existen más bytes de información más allá de él.

El formato de declaración de un archivo es el siguiente:

```c
FILE *nom_var_fich;
```

Como definimos una variable puntero que mantendrá, cuando se establezca, la relación con el archivo físico en memoria secundaria. Una **variable de archivo** representa un bloque de memoria central que contiene dos punteros (un puntero a un buffer de memoria gestionado por el S.O. y el otro puntero a la posición actual del archivo), pero esta variable no es el archivo (está en el almacenamiento secundario).

En otros lenguajes la declaración del archivo determina el tipo de datos que se van a almacenar en él. En C la filosofía es distinta, todos los archivos almacenan bytes y es cuando se realiza la apertura y la escritura cuando se decide cómo y qué se almacena en el mismo; durante la declaración del archivo no se hace ninguna distinción sobre el tipo del mismo.

Se puede conseguir la entrada y la salida de datos a un archivo a través del uso de la biblioteca de funciones; C no tiene palabras claves que realicen las operaciones de E/S. La siguiente tabla da un breve resumen de las funciones que se pueden utilizar. Se debe incluir la librería `stdio.h`. Observe que la mayoría de las funciones comienzan con la letra "f", esto es un vestigio del estándar C de Unix.

Nombre	Función
fopen()	Abre un archivo
fclose()	Cierra un archivo
fgets()	Lee una cadena de un archivo
fputs()	Escribe una cadena en un archivo
fseek()	Busca un byte especifico en un archivo
fprintf()	Escribe una salida con formato en el archivo
fscanf()	Lee una entrada con formato desde el archivo
feof()	Devuelve cierto si se llega al final del archivo
ferror()	Devuelve cierto si se produce un error
rewind()	Coloca el localizador de posición del archivo al principio del mismo
remove()	Borra un archivo
fflush()	Vacía un archivo

Tipos de archivos y operaciones básicas

En C, los tipos de archivos se dividen principalmente en binarios y de texto, y además, los archivos pueden ser de acceso secuencial o aleatorio. A continuación, un resumen de estos conceptos:

Binarios y de Texto:

Binarios:

- Almacenan datos en su forma cruda (raw), sin realizar ninguna interpretación. Pueden contener cualquier tipo de datos, incluidas estructuras y vectores, y son más eficientes para la lectura y escritura de datos complejos.

De Texto:

- Almacenan datos de manera legible para los humanos. Los caracteres se almacenan como caracteres ASCII y se interpretan según el formato del texto. Son adecuados para archivos que contienen información legible y son más portables entre diferentes sistemas operativos. Es una secuencia de caracteres organizadas en líneas terminadas por un carácter de nueva línea. Los archivos de texto se caracterizan por ser planos,

es decir, todas las letras tienen el mismo formato y no hay palabras subrayadas, en negrita, o letras de distinto tamaño o ancho.

Acceso Secuencial y Aleatorio:

Acceso Secuencial:

- Los archivos de acceso secuencial leen y escriben datos en un orden secuencial, desde el principio hasta el final del archivo. Las funciones `fread` y `fwrite` son comunes para este tipo de archivos. No se pueden realizar operaciones de lectura/escritura en posiciones arbitrarias del archivo sin recorrer el archivo desde el principio.

Acceso Aleatorio:

- Los archivos de acceso aleatorio permiten el acceso directo a cualquier posición del archivo utilizando funciones como `fseek` y `ftell`. Esto significa que puedes leer o escribir datos en cualquier posición del archivo sin necesidad de recorrerlo secuencialmente.

En resumen, la distinción principal entre binarios y de texto se refiere a cómo se almacenan y leen los datos. Mientras que la distinción entre acceso secuencial y aleatorio se refiere a cómo puedes acceder y manipular la información dentro del archivo. Estas categorías no están estrictamente separadas, y puedes tener archivos binarios o de texto que sean de acceso secuencial o aleatorio dependiendo de las necesidades del problema y el modo en que se abran.

Apertura y cierre de archivos

Hasta ahora, para obtener y almacenar datos de una estructura de datos bastaba con realizar asignaciones a la misma. Para utilizar los archivos el procedimiento es distinto.

Antes de usar un archivo es necesario realizar una operación de **apertura** del mismo; posteriormente, si se desea almacenar datos en él hay que realizar una operación de **escritura** y si se quiere obtener datos de él es necesario hacer una operación de **lectura**. Cuando ya no se quiera utilizar el archivo se realiza una operación de **cierre** del mismo para liberar parte de la memoria principal que pueda estar ocupando (aunque el archivo en sí está almacenado en memoria secundaria, mientras está abierto ocupa también memoria principal).

La instrucción más habitual para abrir un archivo es :

```
FILE *archivo;
archivo = fopen (nombre_archivo, modo);
```

La función `fopen` devuelve un puntero a un archivo que se asigna a una variable de tipo archivo. Si existe algún tipo de error al realizar la operación, por ejemplo, porque se desee abrir para leerlo y éste no exista, devuelve el valor NULL. Supone la asociación de una variable de archivo a un archivo externo y apertura/creación:

El `nombre_archivo` será una cadena de caracteres que contenga el nombre (y en su caso la ruta de acceso) del archivo tal y como aparece para el sistema operativo.

El `modo` es una cadena de caracteres que indica el tipo del archivo (texto o binario) y el uso que se va ha hacer de él lectura, escritura, añadir datos al final, etc. Los modos disponibles son:

Valor	Descripción
r	abre un archivo para lectura. Si el archivo no existe devuelve error.
w	abre un archivo para escritura. Si el archivo no existe se crea, si el archivo existe se destruye y se crea uno nuevo.
a	abre un archivo para añadir datos al final del mismo. Si no existe se crea.
+	símbolo utilizado para abrir el archivo para lectura y escritura.
b	el archivo es de tipo binario.
t	el archivo es de tipo texto. Si no se pone ni b ni t el archivo es de texto. Los modos anteriores se combinan para conseguir abrir el archivo en el modo adecuado.

Por ejemplo, para abrir un archivo binario ya existente para lectura y escritura el modo será "`rb+`; si el archivo no existe, o aun existiendo se desea crear, el modo será `wb+`. Si deseamos añadir datos al final de un archivo de texto bastará con poner `a`, etc.

Para especificar que un archivo dado está siendo abierto o creado en modo texto se añade una `t` en el modo de apertura (ejemplos: `rt`, `w+t`), Análogamente, para especificar el modo binario se añade una `b` (ejemplos: `wb`, `a+b`). La función `fopen` permite que las letras `t` y `b` se inserten entre la letra y el carácter `+` (ejemplo: `rt+` es equivalente a `r+t`).

La forma habitual de utilizar la instrucción `fopen` es dentro de una instrucción condicional que permita conocer si se ha producido o no error en la apertura, por ejemplo:

```
FILE *arch;
if ((arch = fopen("nomfich.dat", "r")) == NULL)
    { /* control del error de apertura */
    printf ( " Error en la apertura. Es posible que el archivo no exista \n ");
    }
```

El resultado de `fopen` se almacena en la variable `arch` y después se compara `arch` con NULL para saber si se ha producido algún error. Toda la operación se puede realizar en la misma instrucción, tal y como aparece en el ejemplo.

Cuando se termine el tratamiento del archivo hay que cerrarlo; si la apertura se hizo con fopen el cierre se hará con `fclose(arch)`; Para utilizar las instrucciones de manejo de archivos que veremos en esta unidad es necesario incluir la librería `<stdio.h>`.

Lectura y escritura en archivos

Para almacenar datos en un archivo es necesario realizar una operación de escritura, de igual forma que para obtener datos hay que efectuar una operación de lectura. En C existen muchas y variadas operaciones para leer y escribir en un archivo; entre ellas tenemos: `fread -fwrite fgetc -fputc fgets -fputs fscanf -fprintf`. Depende del tipo de archivo y la forma de acceso.

Archivos de texto

Un archivo de texto es una secuencia de caracteres formateada en líneas, donde cada línea termina con una marca de fin de línea (CR + LF). Las funciones `fscanf` y `fprintf` permiten leer y escribir valores que no son de tipo `char`: dichos valores se trasladan automáticamente a y desde su representación binaria a su representación en forma de cadena de caracteres.

orden	descripción
#include <stdio.h>	necesario para gestión de operaciones
FILE *f;	Declaración de variables de archivo.
f=fopen(“nombre_físico”,“r”);	Apertura para leer.
f=fopen(“nombre_físico”,“w”);	Creación + apertura para escribir.
f=fopen(“nombre_físico”,“a”);	Apertura para escritura (+ creación).
fprintf(f, “cad_control”, lista expresiones);	Escritura en el archivo de texto.
fscanf(f, “cad_control”, lista variables);	Lectura del archivo de texto.
fclose(f);	Cierre del archivo.
feof(f);	Función que comprueba el final del archivo.

`fscanf(f, "%d",&i);` Lee del archivo asociado a la variable f una secuencia de dígitos, interpreta la secuencia como un dato entero decimal y lo almacena en la variable entera i.

`fprintf(f, "%5d", i);` Escribe en el archivo la representación en cadena del contenido de la variable i, formateando el dato a la derecha con una longitud de campo mínima de 5 caracteres.

Veamos un ejemplo sencillo de un programa para calcular las notas de los alumnos de programación donde el punto de partida es un archivo de texto con el nombre del alumno y sus notas del examen, prácticas y trabajos individuales, como el siguiente.

```
Juan_Nadie 4 1 1
Pepe_Perez  7 0 1
Jonh_Doe 4 1 0
Fulanito_de_Copas 4 1 1
```

Tras abrir el archivo se leen comprobando que se han leído cuatro items de datos para cargarlo sobre el registro de alumnos

```
#include <stdio.h>

#define MAX_ALUMNOS 100

typedef char cadena[50];
typedef struct {
    cadena nombre;
    float exa;
    float npr;
    float nti;
    float nota;
} Alumno;

typedef struct{
    Alumno alumnos[MAX_ALUMNOS];
```

```
    int num_alumnos;
}tipo_lista_alumnos;

int main() {
    FILE *archivo;
    tipo_lista_alumnos lista;
    lista.num_alumnos=0;

    // Abrir el archivo en modo de lectura
    archivo = fopen("alumno.txt", "rt");
    if (archivo == NULL) {
        perror("Error al abrir el archivo");
        return 1;
    }

    while (fscanf(archivo, "%s %f %f %f",
        lista.alumnos[lista.num_alumnos].nombre,
        &lista.alumnos[lista.num_alumnos].exa,
        &lista.alumnos[lista.num_alumnos].npr,
        &lista.alumnos[lista.num_alumnos].nti) == 4 && lista.num_alumnos < MAX_ALUMNOS)
    {

        // Calcular la nota resultado
        lista.alumnos[lista.num_alumnos].nota =
            lista.alumnos[lista.num_alumnos].exa * lista.alumnos[lista.num_alumnos].npr +
            lista.alumnos[lista.num_alumnos].npr + lista.alumnos[lista.num_alumnos].nti;

        lista.num_alumnos++;
    }
    // Cerrar el archivo
    fclose(archivo);

    for (int i = 0; i < lista.num_alumnos; i++)
        printf(" %s tiene la calificación: %.2f \n", lista.alumnos[i].nombre, lista.alumnos[i].nota);
    return 0;
}
```

El proceso de escritura es similar pero utilizando `fprintf`. Con este código se escribe la tabla del siete en un archivo

```
#include <stdio.h>
#define TABLA 7

void main()
{
    int i, m;

    FILE *f;
    f = fopen("data.txt", "w");

    fprintf(f, "Tabla de multiplicar del %i\n", TABLA);
    for(i = 1; i <= 10; i++)
    {
        m = TABLA * i;
        fprintf(f, "%i x %i = %i\n", TABLA, i, m);
    }

    fclose(f);

    printf("Tabla creada.");
}
```

que genera como resultado el archivo

```
Tabla de multiplicar del 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
```

```
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

Se pueden utilizar otras funciones para la lectura y escritura de los datos de un archivo, se recomienda utilizar pares de funciones por ejemplo si se usa `fprintf` se empareja con `fscanf`, si se usa `fgets` se empareja con `fputs`. Aunque para el caso de `fputs` sería necesario formatear la cadena y es más fácil el uso de `fprintf`

```
....
    char linea[100]; // Tamaño suficiente para una línea del archivo
    while (fgets(linea, sizeof(linea), archivo) != NULL && lista.num_alumnos < MAX_ALUMNOS) {
        sscanf(linea, "%s %f %f %f",
               lista.alumnos[lista.num_alumnos].nombre,
               &lista.alumnos[lista.num_alumnos].exa,
               &lista.alumnos[lista.num_alumnos].npr,
               &lista.alumnos[lista.num_alumnos].nti);
```

Lectura y escritura sobre archivos secuenciales binarios (fread – fwrite)

Para los archivos binarios la información se almacena en memoria secundaria en su representación interna binaria: ASCII para datos textuales, complemento a 2 para datos enteros en IEEE754 para datos reales. Si bien es mas eficiente, el gran inconveniente es que los formatos no son del todo estándar. Los enteros se representan en algunos sistemas con 2 bytes y en otros con 4 bytes,...

Para leer y escribir en archivos que no sean de texto las operaciones que a utilizar son `fread` y `fwrite`. Como hemos dicho es aconsejable utilizar funciones emparejadas; es decir, si se escribe con `fwrite` se debe leer con `fread`.

Para la utilización del archivo se deben conocer la estructura de los registros de archivo.

```
typedef struct{
      tipo1 campo1;
      tipo2 campo2;
       . . . .
       tipoN campoN;
   } tipo_componente;
/* En C  los archivos son tipo puntero a FILE, no se ha de hay que definir nuevas tipologías de datos */
```

Operaciones con archivos

- Declaración de variables de tipo archivo:

```
#include <stdio.h>

     FILE *var_archivo;
```

- Creación:

```
var_archivo=fopen("nombre_físico", "wb");
/*-----Apertura para lectura-----*/
```

- Apertura:

```
var_archivo=fopen("nombre_físico", "rb");
/*-----Apertura para escritura-----*/

var_archivo=fopen("nombre_físico","ab");
/* Si hay errores en la apertura fopen devuelve NULL */
```

- Operaciones sobre el registro actual

```
fwrite(&var_registro,sizeof(tipo_componente),1, var_archivo);

fread (&var_registro,sizeof(tipo_componente),1,
 var_archivo);
/* fread devuelve una cuenta pequeña (posiblemente 0) si la posición actual del archivo está sobre
la marca fin de archivo: antes de procesar los datos leídos por fread, comprobar previamente
que no es el fin del archivo */
```

- Fin de archivo

```
feof(var_archivo)
```

- Cierre

```
fclose(var_archivo);
```

- Borrado

```
remove("nombre_físico");
```

- Renombrar

```
rename("nombre_ant", "nombre_nuevo");
```

Si modificamos el ejemplo del calculo de notas el siguiente programa permite ingresar datos de alumnos hasta que se ingresa “-1” como nombre. Luego, guarda esos datos en un archivo binario. También hay una opción en el menú para leer los datos desde el archivo y mostrar las notas de los alumnos. El archivo se llama “alumnobinario.dat”.

```
#include <stdio.h>

#define MAX_ALUMNOS 100

typedef char cadena[50];
typedef struct {
    cadena nombre;
    float exa;
    float npr;
    float nti;
    float nota;
} Alumno;

typedef struct {
    Alumno alumnos[MAX_ALUMNOS];
    int num_alumnos;
} tipo_lista_alumnos;

void guardarDatosEnArchivo(tipo_lista_alumnos *lista) {
    FILE *archivo;

    // Abrir el archivo en modo de escritura binaria
    archivo = fopen("alumnobinario.dat", "wb");
    if (archivo == NULL) {
        perror("Error al abrir el archivo");
        return;
    }
```

```
    // Escribir los datos en el archivo
    fwrite(lista, sizeof(tipo_lista_alumnos), 1, archivo);

    // Cerrar el archivo
    fclose(archivo);
}

void leerDatosDesdeArchivo(tipo_lista_alumnos *lista) {
    FILE *archivo;

    // Abrir el archivo en modo de lectura binaria
    archivo = fopen("alumno.dat", "rb");
    if (archivo == NULL) {
        perror("Error al abrir el archivo");
        return;
    }

    // Leer los datos desde el archivo
    fread(lista, sizeof(tipo_lista_alumnos), 1, archivo);

    fclose(archivo);
}

int main() {
    tipo_lista_alumnos lista;
    lista.num_alumnos = 0;

    int opcion;

    do {
        printf("\nMenu:\n");
        printf("1. Ingresar datos de alumnos\n");
        printf("2. Mostrar notas de alumnos\n");
        printf("0. Salir\n");
        printf("Ingrese su opcion: ");
        scanf("%d", &opcion);

        switch (opcion) {
            case 1:
                // Ingresar datos de alumnos hasta que se ingrese -1 como nombre
                printf("Ingrese datos de los alumnos (ingrese -1 como nombre para salir):\n");
                do {
                    if (lista.num_alumnos >= MAX_ALUMNOS) {
                        printf("Se alcanzó el número máximo de alumnos.\n");
                        break;
                    }
                    printf("Nombre: ");
                    scanf("%s", lista.alumnos[lista.num_alumnos].nombre);

                    // Salir si se ingresa -1 como nombre
                    if (strcmp(lista.alumnos[lista.num_alumnos].nombre, "-1") == 0) {
                        break;
                    }
                    printf("Examen: ");
                    scanf("%f", &lista.alumnos[lista.num_alumnos].exa);
                    printf("NPR: ");
                    scanf("%f", &lista.alumnos[lista.num_alumnos].npr);
                    printf("NTI: ");
                    scanf("%f", &lista.alumnos[lista.num_alumnos].nti);

                    // Calcular la nota resultado
                    lista.alumnos[lista.num_alumnos].nota =
                        lista.alumnos[lista.num_alumnos].exa * lista.alumnos[lista.num_alumnos].npr +
                        lista.alumnos[lista.num_alumnos].npr + lista.alumnos[lista.num_alumnos].nti;

                    // Incrementar el contador de alumnos
                    lista.num_alumnos++;
                } while (1);

                // Guardar los datos en el archivo
                guardarDatosEnArchivo(&lista);
```

```
                break;

            case 2:
                // Leer datos desde el archivo y mostrar las notas de los alumnos
                leerDatosDesdeArchivo(&lista);

                // Imprimir los resultados
                for (int i = 0; i < lista.num_alumnos; i++)
                    printf("El alumno %s ha obtenido la calificación: %.2f\n", lista.alumnos[i].nombre,
                        lista.alumnos[i].nota);
                break;

            case 0:
                printf("Saliendo del programa.\n");
                break;

            default:
                printf("Opción no válida. Inténtelo de nuevo.\n");
        }
    } while (opcion != 0);

    return 0;
}
```

Lectura y escritura sobre archivos de acceso aleatorio (fseek)

Cuando se lee un dato de un fichero y después el que está a continuación de él, y así sucesivamente, se dice que se está realizando una lectura secuencial del mismo. Cuando se puede acceder a cualquier dato de un fichero sin tener que pasar por anteriores se está realizando un acceso directo a los datos.

La función que permite situarse en un determinado dato del fichero es:

```
int fseek(FILE *var_arch, long despl, int posición_ref);
```

Establece la posición actual del archivo a la posición especificada que dista `despl` bytes con respecto a la posición de referencia. Devuelve 0 si el puntero a la posición actual se mueve con éxito y distinto de cero si se produce algún fallo. En caso de error, la variable global `errno` se establece a alguno de los siguientes valores:

Valor	significado
EBADF	puntero a archivo defectuoso
EINVAL	argumento no válido
ESPIPE	búsqueda ilegal en dispositivo

Nota: para archivos de texto, despl solo puede ser 0 o un valor devuelto por la función `ftell`. La posición de referencia puede ser:

Posición de referencia	Constante simbólica	Localización en el archivo
0	SEEK_SET	Comienzo del archivo
1	SEEK_CUR	Posición actual del archivo
2	SEEK_END	Fin del archivo

Por ejemplo, en un archivo que almacene números enteros la instrucción `fseek (f, 0, SEEK-SET);` colocará el puntero al principio del archivo. `fseek (f, 3*sizeof(int), SEEK-CUR);` colocará el puntero 3 posiciones más allá de la posición actual del puntero . Para saber cuál es la posición en la que está el puntero del archivo C proporciona la función siguiente: `ftell (fich);` que devuelve la posición actual en bytes del puntero del archivo con respecto al principio del mismo.

Clasificación y busqueda externa

Los algoritmos de clasificación externa son algoritmos diseñados para ordenar grandes conjuntos de datos almacenados en archivos que no pueden caber completamente en la memoria principal (RAM) de una computadora. La clasificación externa es esencial cuando la cantidad de datos a ordenar supera la capacidad de la memoria disponible. Estos algoritmos trabajan leyendo y escribiendo bloques de datos desde y hacia el almacenamiento secundario (como un disco duro) de manera eficiente.

Algunos de los algoritmos de clasificación externa más comunes incluyen:

Merge Sort Externo (Mezcla):

- Este algoritmo utiliza una estrategia de división y fusión.
- Divide el archivo en bloques que caben en memoria.
- Cada bloque se ordena internamente.
- Luego, se fusionan los bloques ordenados para obtener el resultado final.

Polyphase Merge Sort (Mezcla en varias fases):

Similar al Merge Sort externo, pero con la capacidad de fusionar múltiples bloques al mismo tiempo, reduciendo el número de pasadas sobre los datos.

Replacement Selection (Selección con reemplazo):

- Utiliza un enfoque de selección de registros, seleccionando registros de entrada y escribiéndolos en un archivo de salida ordenado.
- La selección se realiza de manera que permita reemplazar registros en el archivo de salida sin perder el orden.

Distribution Sort (por distribución):

- Divide los datos en bloques y clasifica cada bloque en memoria.
- Luego, distribuye los bloques a diferentes archivos temporales según el rango de valores.
- Finalmente, fusiona los archivos temporales para obtener la salida ordenada.

QuickSort:

- Utiliza una versión modificada del algoritmo QuickSort que opera en bloques de datos en lugar de en la memoria principal completa.
- Realiza particiones de los bloques y luego realiza la ordenación recursiva.

Burbuja:

- Aplica el concepto básico del algoritmo de burbuja, pero opera en bloques de datos en lugar de en el conjunto completo.

Estos algoritmos son esenciales en situaciones donde la cantidad de datos es demasiado grande para caber en memoria y deben ser procesados de manera eficiente desde y hacia un medio de almacenamiento secundario. La eficiencia de estos algoritmos depende de factores como el tamaño de los bloques, la velocidad del disco y la capacidad de la memoria principal.

Veremos un ejemplo de como se diseñaría el caso de un algoritmo de partición y mezcla.

Partimos de F con `n=9`:

```
19 27 2 8 36 5 20 15 6 EOF
```

Los pasos para la ejecución del algoritmo son: