APUNTES PROGRAMACIÓN:

1) Tema 1: Datos en C.

- Para programar un programa siempre es necesario emplear una librería donde aparecen los programas con el comando #include <nombre de la librería> al inicio de cada programa. La primera usada es <stdio.h> la cual permite enseñar la pantalla y leer el teclado con las funciones (print f y scan f).
- Todo programa debe incluir una función main la cual es el punto de partida de todo programa. Esta función tiene la siguiente sintaxis:

```
#include <stdio.h>

int main()
{

   return 0;
}
```

Como se observa después del main siempre hay un () y el contenido de ella se encuentra entre llaves {} y finalmente necesita un valor que se le devuelva, en este caso se hace mediante el comando return. Siendo el ; esencial después de cada comando. *[Como a la función le hemos asignado que trabaja con datos del tipo enteros el return devuelve un entero]*

- Para comentar se emplea // antes del texto
- La función print f permite que se muestre texto en pantalla y tiene la siguiente sintaxis:

print f ("texto %insertar dato \comandos", asignación del valor del dato)

```
printf("Hoy es día %i\n", 6);
```

El siguiente comando mostrara tras el porcentaje el dato entero (i) que se le ha asignado el 6.

Comandos:

- Para añadir un párrafo tras el print f se emplea \n

Tabla de abreviaturas de los tipos de datos:

tipo	Tipo de datos	Salida
d	int	Enteros con signo en decimal
i	int	Enteros con signo en decimal
u	int	Enteros sin signo en decimal
o	int	Enteros sin signo en octal
x	int	Enteros sin signo en hexadecimal (abc..)
X	int	Enteros sin signo en hexadecimal (ABC..)
f	double	Valor con signo de la forma [-]m.dddddd
e	double	Valor con signo de la forma [-]m.ddddde[±]ddd
E	double	Valor con signo de la forma [-]m.dddddE[±]ddd
g	double	Valor con signo compacto en formato f o e
G	double	Valor con signo compacto en formato f o E
c	int	Carácter individual de un byte
s	cadena de caracteres	Escribe la cadena hasta el primer '\0'

Para ajustar los decimales de los enteros usamos %.2f (float con 2 decimales). Además, con este formato no aparece de forma exponencial

Notación científica

Automático

Para escribir enteros cona más resolución que un float (f) se emplean los doubles (lf). Con los enteros pasa igual, al añadir la L minúscula aumenta su resolución. Dos tipos de datos no incluidos son los caracteres (char – c) y los booleanos (0 y 1) [_Boal] empleado para operadores lógicos. %lu sirve para long int o unsigned long int

- Para ver cuanto ocupa una variable se emplea la función sizeof (tipo de dato) y para ver lo que abarca se emplean los comandos INT_MIN INT_MAX (ejemplo). Es importante para esto incluir la librería <limits.h>

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

int main() {
  printf("Un int ocupa %d bytes",
        sizeof(int));
  printf(" y abarca desde %d hasta %d.\n",
        INT_MIN, INT_MAX);
  printf("Un long int ocupa %d bytes",
        sizeof(long int));
  printf(" y abarca desde %ld hasta %ld.\n",
         LONG_MIN, LONG_MAX);
  printf("Un long long int ocupa %d bytes",
        sizeof(long long int));
  printf(" y abarca desde %lld hasta %lld.\n",
        LLONG_MIN, LLONG_MAX);
  return 0;
}
```

- Para declarar variables se pone el tipo de dato abreviado *(no una sola letra)*. Y es posible declarar más de una a la vez. Además, si se le añade la palabra cons antes del tipo de dato no puede cambiar su valor. Para asignar un valor a una variable se usa el igual. Por último, en una línea se pueden declarar varias constantes si se separan mediante comas.

```c
int main()
{
  // declara una variable con el identificador v1
  int v1;
  // declara una constante simbólica
  // con el identificador c1
  const int c1 = 4;
  // declara una variable v2,
  // y le asigna el valor 2 (una constante literal)
  int v2 = 2;
  // asigna el valor de la
  // constante c1 a la variable v1
  v1 = c1;
  // ídem con v2 (cambia su valor previo)
  v2 = c1;
  // error: c1 es una constante
  c1 = 3;
  return 0;
}
```

- Normativa para nombrar variables:
  - Primer carácter: letra o carácter de subrayado (_) (**nunca un número**).
  - Una o más letras (A-Z, a-z, *ñ excluida*), dígitos (0-9) o caracteres de subrayado.
  - Tienen que ser distintos de las palabras clave.
  - Las mayúsculas y las minúsculas son diferentes para el compilador.
  - Es aconsejable que los nombres sean representativos

Palabras clave:

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

- Las variables tipo char se pueden representar mediante su letra con el símbolo 'letra' o mediante su código en Ascii que es el valor numérico que tendrá dicha letra.

```c
#include <stdio.h>

int main()
{
  // Usamos %c para caracteres
  // Atención: para delimitar caracteres usamos '
  printf("La última letra del alfabeto es la %c\n",
         'z');
  //Usamos %i para enteros
  printf("Su valor en la tabla ASCII es %i\n",
         'z');
  //Y si usamos %c para un número?
  printf("El número %i es la letra %c\n",
         122, 122);
  return 0;
}
```

dec	ch	dec	ch	dec	ch	dec	ch
0	NUL (null)	32	(space)	64	@	96	‘
1	SOH (start of header)	33	!	65	A	97	a
2	STX (start of text)	34	“	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	’	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	HT (horizontal tab)	41	)	73	I	105	i
10	LF (line feed - new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (form feed - new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of transmission block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL (delete)

- Función scan f la función tiene la siguiente sintaxis: scan f(“%dato1 %dato2”, &variable1, &variable 2) Lo que realizara la función es leer el tipo de dato que le demos y almacenara el primero en la variable 1 y el segundo en la variable 2. Los datos se los damos escribiendo en la consola. También se pueden escribir letras si se usan tipos de dato char. SIEMPRE POR PRECUACIÓN ANTES DE UN CHAR UN ESPACIO.

```c
#include <stdio.h>

int main()
{
  int num;

  printf("Escribe un número\n");

  //Atención: con scanf el nombre de la
  //variable debe ir precedido de &
  scanf("%i", &num);

  printf("Has escrito el número %i\n", num);

  return 0;
}
```

- Errores comunes:
  - Escribir dentro de la cadena de control mensajes y secuencias de escape (p.ej. \n).
  - Olvidar poner el operador & delante de los argumentos cuando son variables de los tipos básicos (`int`, `float`, `double`, `char`)
  - Poner un especificador de formato no compatible con el tipo del argumento.

2) Tema 2: Operadores.

- Los operadores aritméticos son, también se puede hacer con letras:

```
x + y
x - y
x / y
x * y
x % y //módulo o resto de división de enteros
```

- Los operadores relacionales son, siendo == la pregunta ¿Es igual? y != ¿Es diferente?. Devolviendo 1 si es cierto y 0 si la pregunta es falsa.

```
x == y
x != y
x > y
x >= y
x < y
x <= y
```

```
#include <stdio.h>
int main()
{
    int x = 10, y = 3;

    printf("x igual a y = %d\n",
           (x == y));
    printf("x distinto a y = %d\n",
           (x != y));
    printf("x mayor que y = %d\n",
           (x > y));
    printf("x menor o igual a y = %d\n",
           (x <= y));
    printf("x mayor o igual que y = %d\n",
           (x >= y));
    return 0;
}
```

```
#include <stdio.h>

int main()
{
  float peso, altura, imc;

  printf("Indica tu peso (kg) y altura (m)\n");
  scanf("%f %f", &peso, &altura);

  imc = peso / (altura * altura);
  printf("Tu índice de masa corporal es %f\n", imc);

  return 0;
}
```

- Lógicos y condicionales

```
x && y //AND
x || y //OR
!x //NOT, operador unario
```

```
// expresión boleana ? valor si cierto : valor si falso
x > y ? "cierto" : "falso"
x == y ? "true" : "false"
```

- Asignación:

```
x = y
x += y // x = x + y
x -= y // x = x - y
x *= y // x = x * y
x /= y // x = x / y
x %= y // x = x % y
```

- Incrementales:

```
y = ++x // x = x + 1; y = x (preincremento)
y = x++ // y = x; x = x + 1 (postincremento)

y = --x // x = x - 1; y = x (predecremento)
y = x-- // y = x; x = x - 1 (postdecremento)
```

- Ejemplos:

```c
#include <stdio.h>
int main()
{
    int a = 3, b = 2, c = 4, d = 5;

    printf("resultado = %d\n",
          (a > b) && (c < d));

    printf("resultado = %d\n",
          (a < 10) || (d != 5));

    printf("resultado = %d\n",
          (a != b) && (2 * d < 8));

    return 0;
}
```

```c
#include <stdio.h>
int main()
{
 int x, resto;

 printf("Escribe un número entero: ");

 scanf("%d", &x);

 // Calcula el resto de dividir por 2
 resto = x % 2;

 // Si el resto es 0, x es par.
 printf("Es un número %s\n",
      (resto == 0) ? "par" : "impar");

 return 0;
}
```

- Conversión:
  - Implícita, a una variable de un tipo se le asigna otra de otro tipo de manera que cambia el tipo de dato:

```c
float f1 = 3.7, f2;
int i1 = 2, i2;
// Real a entero: pierde decimales
i2 = f1;
printf("Un real %f convertido a entero %d\n",
      f1, i2);
// Entero a real: no cambia valor
f2 = i1;
printf("Un entero %d convertido a real %f\n",
      i1, f2);
```

  - Explicita, se fuerza un tipo de expresión (se puede realizar para cambiar valores durante una operación si se antepone el comando)

```c
 float f1 = 3.7, f2;
 int i1 = 2, i2;

 f2 = (float) i1;
 printf("Un entero %d convertido a real %f\n",
       i1, f2);

 i2 = (int) f1;
 printf("Un real %f convertido a entero %d\n",
       f1, i2);
```

```c
#include <stdio.h>
int main() {
int x = 10, y = 3;
printf("El resultado de dividir 10 entre 3 es %f",(float)x/y);

return 0;
}
```

  - Expresiones, durante operaciones los operandos se convierten al tipo con mayor precisión

```c
#include <stdio.h>

int main()
{
  double f1 = 100;
  int i1 = 150, i2 = 100;
  printf("Un entero, %d, dividido por un real, %f,",
        i1, f1);
  printf(" produce un real, %f\n",
        i1 / f1);
  printf("Un entero, %d, por un entero, %d: %d\n",
        i1, i2, i1 / i2);
  return 0;
}
```

3) Tema 3: Sentencias condicionales.

- La función if comprueba una condición para ejecutar una tarea.

if (condición)

sentencia _A;

Sentencia_B;

Si condición == true se ejecuta A sino B

```
if (condicion)
   sentencia_A;
sentencia_B;
```

• Para ejecutar un conjunto de sentencias hay que agruparlas entre llaves.

```
if (condicion)
 {
   sentencia_A1;
   sentencia_A2;
   ...
 }
sentencia_B1;
sentencia_B2;
```

```
# include <stdio.h>
int main ()
{
  int n;
  printf("Escribe un número entero\n");
  scanf("%d", &n);
  if (n % 2 == 0) // Condición
    {// Uso de llaves
      printf("Se cumple la condición: ");
      printf("El número %d es par.\n", n);
    } // Fin de if
  printf("Gracias por participar.\n");
  return 0;
}
```

- Si a la función if se le añade el else, si la condición se cumple se ve sino otra. Al igual que con el if si se ejecuta más de un comando se emplean llaves.

```
if (condicion)
 {
   sentencia_A1;
   sentencia_A2;
   ...
 }
else
 {
   sentencia_B1;
   sentencia_B2;
   ...
 }
sentencia_C1;
sentencia_C2;
...
```

- Se pueden combinar varios if else para ir añadiendo condiciones

```
#include <stdio.h>

int main(){
  int x, signo;
  printf("Escribe un número: ");
  scanf("%i", &x);

  if (x < 0)
    // se cumple la primera condición
    printf("El número es negativo.\n");
  else if (x == 0)
    // se cumple la segunda
    printf("El número es 0.\n");
  else
    // no se cumple ninguna
    printf("El número es positivo.\n");
  return 0;
}
```

```
if (condicion_1)
    sentencia_A;
else
  if (condicion_2)
    sentencia_B;
  else
    sentencia_C;
sentencia_D;
```

- El comando switch-case permite para una pregunta devolver varios resultados dependiendo del valor de una expresión. Es importante escribir break; tras cada case pues sino se ejecutarán uno tras de otro de seguido (no ponerlo podría ser útil para mayúsculas y minúsculas). Además con case no se usan llaves, solo con switch

En este caso si a vale 1 saldrá A y si vale 2 saldrá B.

```
#include<stdio.h>
 main()
 {
     int a;
scanf("%i",&a);
switch (a)
     {
          case 1:
     printf("A");
     break;

     case 2:
     printf("B");
     break;
     }


 }
```

```
#include <stdio.h>
int main (){
  float v1, v2;
  char op;
  scanf("%f  %c  %f", &v1, &op, &v2);
  switch(op)
    {
    case '+':
      printf("Operación Suma:\n");
      printf("%.2f\n", v1 + v2);
      break;
    case '-':
      printf("Operación Resta:\n");
      printf("%.2f\n", v1 - v2);
      break;
    default:
      printf("No se hacer esa operación.\n");
      break;
    }
  return 0;
}
```

4) Tema 4: Sentencias repetitivas.

- La función for ejecuta una sentencia un determinado número de veces hasta que el resultado de una expresión sea falso.

for(expresión inicial, expresión final a comprobar, expresión de incremento)

{ }

```c
#include <stdio.h>

int main()
{
  // Todas las variables deben estar definidas
  // Asigno valor inicial 0 a suma
  int i, suma = 0, n = 10;

  for (i = 1; i <= n; i++)
    {
      suma += i;
    }
  printf("La suma de los %d primeros enteros es %d",
        n, suma);
  return 0;
}
```

```c
#include <stdio.h>

int main()
{
  int i, j;
  for (i = 1; i <= 10; i++)
    {//Atención al uso de las llaves
      printf("Tabla del %d\n", i);

      for (j = 1; j <= 10; j++)
        printf("%d x %d = %d\n",
              i, j, i * j);
    }
  return 0;
}
```

- El comando while ejecuta una sentencia en función del resultado de una expresión. Es decir, mientras una sentencia sea verdadera ejecuta todo el código en su interior. (Si la condición siempre es falsa nunca se ejecutará y si siempre es verdadera se ejecutara infinitamente).

  while(condición (mientras la condición sea verdadera))
  {
  Sentencias….
  }

```c
#include <stdio.h>
int main()
{
  int i;
  i = 5;
  while (i > 0)
    {
      printf("%d...", i);
      --i;
    }
  printf("Despegue!");
  return 0;
}
```

- Si se le añade un do al while realiza la expresión al menos una vez (si el while es falso), si al iniciarse el comando es verdadero se comporta igual que un while.

```c
#include <stdio.h>

int main()
{
  int num = 123456, cifra;

  do
    {
      cifra = num % 10;
      printf("%d", cifra);
      num = num / 10;
    }
  while (num > 0);
  printf("\n");
  return 0;
}
```

- Si se conoce el número de veces que debe ejecutarse la tarea es recomendable usar `for`.
- Si el número de veces es desconocido a priori:
  - Si debe realizarse al menos una vez se debe usar `do-while`.
  - Si no es imprescindible que se ejecute alguna vez, se puede usar `while`.

`break`

- Finaliza la ejecución de un bucle (si el bucle está anidado sólo finaliza él, pero no los bucles más externos).

`continue`

- Ejecuta la siguiente iteración del bucle.
- En un bucle `while` o `do-while` vuelve a `expresion`.
- En un bucle `for` ejecuta `expr_avance` y a continuación comprueba `expr_final`

5) Tema 5: Funciones:

- Una función es un bloque de código que realiza una tarea determinada a partir de unos datos, ventajas:
  - Permiten programación estructurada y abstracta, sin necesidad de conocer el detalle de la implementación de una tarea concreta
  - Mejoran la legibilidad
  - Facilitan el mantenimiento del programa
  - Permiten reutilizar código

- Declaración de una función

Prototipo de una función:

1. Tipo de valor que devuelve (`int`, `void`, ...)
2. Nombre de la función (debe ser un identificador válido y **útil**).
3. Lista de argumentos que emplea, por tipo y nombre (puede estar vacía).

```
tipo nombre_funcion(tipo1 arg1, tipo2 arg2, ...);
```

Ejemplos

```
void printHello(int veces);

float areaTriangulo(float b, float h);
```

- Definición de una función

```
// Definicion de la funcion printHello

// No devuelve nada (void)

// Necesita un argumento llamado veces,
// un entero (int), para funcionar.

void printHello(int veces)
{
  int i;
  for (i = 1; i <= veces; i++)
    printf("Hello World!\n");
}
```

```
// Definicion de la funcion areaTriangulo

// Devuelve un real (float)

// Necesita dos argumentos, b y h, reales.

float areaTriangulo(float b, float h)
{
  float area;

  area = b * h / 2.0;

  return area;
}
```

- Estructura de un programa

- *Puede* incluir directivas de inclusión (`include`).
- *Puede* incluir directivas de sustitución (`define`).
- Declaración de funciones (prototipo).
- Todos los programas tienen al menos una función: `main`.
- Definición de las funciones.

- Directivas de inclusión

```
//Librerias del sistema
#include <stdio.h>
#include <math.h>
//Librerias propias del desarrollador
#include "myHeader.h"
```

- El comando #define permite definir símbolos que al ponerlos en el código serán sustituidos por su valor y si se pone undef se elimina la definición del símbolo:

```
//Atención: SIN signo igual NI punto y coma
#define PI 3.141592
```

```
#undef PI
```

- Declaración de funciones, se pone primero las funciones que se van a incluir y se termina con (;) luego se puede incluir la función dentro de main para que haga su función, terminando con (;) y al final del todo se pone la definición de la función (que hace), sin el (;).

```c
#include <stdio.h>
// Prototipo de la función (termina en ;)
void printHello(int n);

// Función main
int main() {
  //Uso de la función en main
  printHello(3);
  return 0;
}
// Definición de la función
void printHello(int n)
{
  int i;
  for (i - 1; i <- n; i++)
    printf("Hello World!\n");
}
```

```c
#include <stdio.h>
// Prototipo de la función (termina en ;)
float areaTriangulo(float b, float h);
// Función main
int main(){
  float at;
  //Uso de la función en main
  at = areaTriangulo(1, 2);
  printf("%f", at);
  return 0;
}
// Definición de la función
float areaTriangulo(float b, float h)
{
  float area;
  area = b * h / 2.0;
  return area;
}
```

- Variables globales: variables declaradas fuera de una función, es decir, pueden funcionar fuera de una función y cualquier función la puede modificar (NO SON RECOMENDABLES), si se declara dentro de main no se consideran globales.

```c
#include <stdio.h>

int gVar = 3; //Variable global

void foo(void);

int main(){
  printf("main (1):\t gVar es %d.\n", gVar);
  foo();
  gVar *= 2;
  printf("main (2):\t gVar es %d.\n", gVar);
  return 0;
}

void foo(void){
  gVar = gVar + 1;
  printf("foo:\t gVar es %d.\n", gVar);
}
```

```
main (1):        gVar es 3.
foo:     gVar es 4.
main (2):        gVar es 8.
```

- Variables locales: variables declaradas dentro de una función y solo puede variar dentro de la función y las funciones auxiliares no la pueden modificar.

```c
#include <stdio.h>

void foo(void);

int main()
{
  int x = 1; // variable local en main
  printf("main (1):\t x es %d.\n", x);
  foo();
  printf("main (2):\t x es %d.\n", x);
  return 0;
}

void foo(void)
{
  int x = 2; // variable local en foo
  printf("foo:\t x es %d.\n", x);
}
```

```
main (1):        x es 1.
foo:     x es 2.
main (2):        x es 1.
```

- Funciones que llaman a otras funciones: se puede meter una función en otra y si se llama a si misma es recursiva

```c
#include <stdio.h>
#define PI 3.141592

float eleva3(float x);
float volEsfera(float r);

int main(){
  float radio, vol;
  scanf("%f", &radio);
  vol = volEsfera(radio);
  printf("El volumen es %f", vol);
  return 0;
}
float volEsfera(float r){ //Usa eleva3
  return 4.0/3.0 * PI * eleva3(r);
}
float eleva3(float x){
  return x * x * x;
}
```

```c
#include <stdio.h>

int fact(int n);

int main(){
  int x;
  printf("Indica un número:\n");
  scanf("%d", &x);
  printf("El factorial de %d es %d\n", x, fact(x));
  return 0;
}
int fact(int n){
  int res;
  if (n > 1) // Incluye llamada a si misma
    res = n * fact(n - 1);
  else
    res = 1;
  return res;
}
```

- Funciones en ficheros: se crean librerías para dar mejor legibilidad a un código y poder reutilizarlo en varios programas. Si el fichero esta en la misma carpeta que el programa principal basta simplemente con llamarlo "fichero.h". Si el fichero esta en otro directorio es necesario poner la dirección completa del fichero.

Uso

- Debe existir un (o varios) fichero(s) `.h` (cabecera) y un fichero `.c` (código fuente, implementación de las funciones).
- Se debe usar `#include "nombre_lib.h"` al comienzo del programa.
- Hay que compilar conjuntamente (*en un proyecto*).

Fichero `myLib.h` (cabecera)

```c
#define PI 3.141592

float eleva3(float x);
float volEsfera(float r);
```

Fichero `myLib.c` (código fuente)

```c
#include "myLib.h"

float volEsfera(float r){
  return 4.0/3.0 * PI * eleva3(r);
}
float eleva3(float x){
  return x * x * x;
}
```

Programa principal

```c
#include <stdio.h>
// Directiva para incluir la librería local
#include "myLib.h"

int main()
{
  float radio, vol;
  scanf("%f", &radio);
  vol = volEsfera(radio);
  printf("El volumen es %f", vol);
  return 0;
}
```

6) Tema 6: Vectores:

Definición

Conjunto de valores numéricos del mismo tipo

Código

```
tipo identificador[dimensión];
```

tipo Tipo de los elementos del vector (`int`, `float`, etc.).
identificador Nombre del vector.
dimensión Número de elementos del vector (literal entero).

```
// Declara un vector llamado miVector compuesto por
// tres elementos de tipo int.
int miVector[3];

// Declara un vector e inicializa todos sus elementos
int miVector[3] = {2, 23, 0};

// Declara un vector e inicializa el primer elemento
// (resto quedan a 0)
int miVector[3] = {2};

// Declara un vector sin dimensión.
// La dimensión queda determinada a partir
// del numero de elementos
int miVector[] = {2, 23, 24};
```

- Se referencian con el nombre del vector seguido de un subíndice entre corchetes.
- El subíndice representa la posición del elemento dentro del vector.
- La **primera posición** del vector tiene el **subíndice 0**.

```
#include <stdio.h>

int main(){
 int miVector[3];
 miVector[0] = 10;
 miVector[1] = 2 * miVector[0];
 miVector[2] = miVector[0] + miVector[1];

 printf("Posicion 0 = %d\n", miVector[0]);
 printf("Posicion 1 = %d\n", miVector[1]);
 printf("Posicion 2 = %d\n", miVector[2]);

 return 0;
}
```

- Acceso a datos de un vector:

```
#include <stdio.h>

int main()
{
 float temp[5] = {2.1, 4.9, 0.51, 4.3, 9.01};
 int i;
 // Es común el uso de bucles for para
 // recorrer un vector. Es importante
 // recordar que el primer elemento
 // tiene índice 0.
 for (i = 0; i < 5; i++)
   printf("El elemento %d es %f\n",
         i + 1, temp[i]);
 return 0;
}
```

- Operaciones con vectores:

Suma de dos vectores

```
#include <stdio.h>
int main(){
 float v1[5] = {1, 34, 32, 45, 34};
 float v2[5] = {12, -3, 34, 15, -5};
 float v3[5];
 int i;

 for(i = 0; i < 5; i++)
    v3[i] = v1[i] + v2[i];

 printf("Vector3: ");
 for(i = 0; i < 5; i++)
   printf("%f ", v3[i]);
 printf("\n");
 return 0;
}
```

Multiplicar un vector por una constante

```
#include <stdio.h>
int main(){
 float v1[5] = {1, 34, 32, 45, 34};
 float v2[5];
 float K = 3.0;
 int i;

 for(i = 0; i < 5; i++)
    v2[i] = K * v1[i];

 for(i = 0; i < 5; i++)
   printf("V1: %f\t V2: %f\n",
         v1[i], v2[i]);

 return 0;
}
```

- La dimensión de un vector es un valor constante y no puede usarse una variable para definirla:

```c
int main()
{
  int miVector[10];

  return 0;
}
```

```c
# define N 10
int main()
{
  // Correcto: el precompilador sustituye N
  // por el valor constante 10
  int miVector[N];

  return 0;
}
```

- Solución provisional si no se conoce el número de elementos a introducir:

```c
#include <stdio.h>
int main() {
 int i, n;
 // Definimos un vector de dimension muy grande
 int vect[100];

 printf("Nº datos? ");
 //El usuario debe teclear un n < 100
 scanf("%d", &n);
 //Utilizamos solo las n primeras
 for(i = 0; i < n; i++)
   {
    scanf("%d", &vect[i]);
   }
 return 0;
}
```

- Funciones con vectores: La función puede modificar el contenido de los elementos del vector ya que conoce la dirección de memoria donde están almacenados.

Paso por referencia

Cuando un vector se pasa como argumento a una función **no se pasa el vector completo** sino la **dirección de memoria del primer elemento**.

```c
void funcion (int vector[], int dimension);
```

```c
#include <stdio.h>

void imprime(int v[], int n);

int main()
{
  int v1[3] = {10, 20, 30};
  imprime(v1, 3);
  return 0;
}

void imprime(int v[], int n)
{
  int i;
  for(i = 0; i < n; i++)
    printf("%d\n",v[i]);
}
```

```c
#include <stdio.h>

void toy(int vector[]);

int main()
{
  int x[] = {1, 2, 3};
  printf("Antes: %d\n", x[0]);
  toy(x); // ¡Sin asignacion!
  printf("Después: %d\n", x[0]);
  return 0;
}

void toy(int vector[]){
  //Funcion simple que modifica el valor del primer elemento
  vector[0] = 100;
}
```

7) Tema 7: Cadenas de caracteres:

Definición

Conjunto de caracteres individuales (`char`)

Código

```
char identificador[dimensión];
```

tipo `char`

identificador Nombre de la cadena.

dimensión Número de elementos de la cadena (constante entero) **incluyendo el carácter de cierre** (\0).

- Definición e inicialización de cadenas:

```
// Declara una cadena de 10 caracteres
//(+1 para el cierre)
char cadena[11];

// Declara y asigna contenido
char cadena[5] = "Hola"; // 4 + 1

// Asigna por valores individuales
char cadena[5] = {'H', 'o', 'l', 'a', '\0'}; // 4 + 1

// Asigna por codigo ASCII
char cadena[5] = {72, 111, 108, 97, 0};
```

```
// Declara una cadena, *no* define dimension
// y asigna contenido
char cadena[] = "Hola";

// Asigna por elementos individuales
char cadena[] = {'H', 'o', 'l', 'a', '\0'}; // 4 + 1;

// Asigna mediante codigo ASCII
char cadena[] = {72, 111, 108, 97, 0};
```

- Elementos de una cadena:

- Se referencian con el nombre seguido de un subíndice entre corchetes.
- El subíndice representa la posición del elemento dentro de la cadena.
- La **primera posición** tiene el **subíndice 0**.
- La **última posición** es el carácter nulo \0.

```
#include <stdio.h>

int main()
{
 char cadena[5] = "Hola";
 printf("%c \t %c \t %c \t %c \t %c\n",
       cadena[0], cadena[1],
       cadena[2], cadena[3],
       cadena[4]);
 return 0;
}
```

Error

```
char cadena[5];

//Error de compilacion
cadena = "Hola";
```

Solución provisional

Mejor con `strcpy` de `string.h`

```
char cadena[5];
cadena[0] = 'H';
cadena[1] = 'o';
cadena[2] = 'l';
cadena[3] = 'a';
cadena[4] = '\0';
```

- Lectura y escritura de una cadena:

- Usamos el especificador `%s` con `printf` y `scanf`.
- En `scanf` **debemos** especificar el **límite de caracteres** en el especificador de formato.
- En `scanf` **no** ponemos `&` delante del identificador.

```
#include <stdio.h>

int main()
{
 char texto[31];

 printf("Dime algo: \n");
 // Deja de leer cuando detecta un espacio
 // Imponemos el límite de caracteres
 scanf("%30s", texto);
 printf("Has dicho %s", texto);
 return 0;
}
```

- Lectura de una cadena con espacios:

  - scanf con %s termina de leer cuando recibe un espacio o salto de línea.
  - Para leer cadenas de caracteres que incluyan espacios se emplea el identificador %[^\n]

```c
#include <stdio.h>

int main()
{
 char texto[31];

 printf("Dime algo: \n");
 // Deja de leer cuando detecta un salto de línea
 // o al alcanzar el límite de caracteres
 scanf("%30[^\n]", texto);
 // En printf seguimos usando %s
 printf("Has dicho %s\n", texto);
 return 0;
}
```

- Recorrido de los elementos: se emplea generalmente el bucle while usando el carácter nulo para terminar, aunque el bucle for también se puede emplear.

- El bucle while es el más indicado, usando el carácter nulo para terminar:

```c
#include <stdio.h>

int main()
{
 char cadena[5] = "Hola";
 int i = 0;
 printf("Los caracteres son:\n");
 while (cadena[i] != '\0')
  {
   printf("%c \t", cadena[i]);
   i++;
  }
 return 0;
}
```

```c
#include <stdio.h>

int main()
{
 char cadena[5] = "Hola";
 int i;
 printf("Los caracteres son:\n");
 for(i = 0; cadena[i] != '\0'; i++)
  {
   printf("%c \t", cadena[i]);
  }
 return 0;
}
```

- Pasar a mayúsculas:

```c
#include <stdio.h>

int main() {
 char cadena[5] = "Hola";
 // Distancia entre A y a
 int inc = 'A' - 'a';
 int i = 0;
 // Recorremos la cadena
 while(cadena[i] != '\0')
  { // Si el caracter es letra minuscula
   if (cadena[i] >= 'a' && cadena[i] <= 'z')
    //sumamos la distancia para pasar a mayuscula
    cadena[i] += inc;
   i++;
  }
 printf("%s\n", cadena);

 return 0;
}
```

- Librería string.h:

La librería `string.h` incluye numerosas funciones dedicadas a cadenas de caracteres:

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

Longitud de una cadena `strlen`
Paso a mayúsculas `_strup`
Copiar cadenas `strcpy`
Concatenar cadenas `strcat`
Comparación de cadenas `strcmp`

- Longitud de una cadena: strlen

- `strlen` devuelve un entero con el número de caracteres.

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

int main()
{
 char palabra[21];
 int longitud;
 printf("Introduce una palabra: ");
 scanf("%20s", palabra);
 longitud = strlen(palabra);
 printf("Esta palabra tiene %d caracteres\n",
       longitud);
 return 0;
}
```

- Paso a mayúsculas: _strup

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

int main()
{
  char nombre[100];
  printf("Introduce tu nombre: ");
  scanf("%s",nombre);
  //¡Sin asignacion!
  _strupr(nombre); //atencion al guion inicial
  printf("En mayusculas %s\n", nombre);
  return 0;
}
```

- Copiar cadenas: strcpy

Con `strcpy` tenemos una solución óptima para la **asignación de contenido**.

```c
#include <stdio.h>
#include <string.h>
int main()
{
 char s1[50], s2[50];
 strcpy(s1, "Hello World!");
 strcpy(s2, s1);
 printf("%s\n", s2);
 return 0;
}
```

La cadena receptora debe tener espacio suficiente: *los caracteres sobrantes serán eliminados*.

- Concatenar cadenas: strcat

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

int main() {
  char nombre_completo[50];
  char nombre[ ] = "Juana";
  char apellido[ ] = "de Arco";
  // Copiamos por tramos:
  // Primero el nombre
  strcpy(nombre_completo, nombre);
  // A continuacion un espacio
  strcat(nombre_completo, " ");
  // Finalmente el apellido
  strcat(nombre_completo, apellido);
  printf("El nombre completo es: %s.\n",
         nombre_completo);
  return 0;
}
```

- Comparación de cadenas: strcmp

- **Si las dos cadenas son iguales entrega un 0.**

```
#include <stdio.h>
#include <string.h>
int main()
{
  char color[] = "negro";
  char respuesta[11];
  do // El bucle se repite mientras
    {// las cadenas *no* coincidan
      printf("Adivina un color: ");
      scanf ("%10s", respuesta);
    } while (strcmp(color, respuesta) != 0);
  printf("¡Correcto!\n");
  return 0;
}
```

- Comparación de cadenas: strcmp

- Si hay diferencias, es positivo si el valor ASCII del primer carácter diferente es mayor en la cadena 1.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char s1[] = "abcdef";
  char s2[] = "abCdef";
  char s3[] = "abcdff";
  int res;
  res = strcmp(s1, s2);
  printf("strcmp(s1, s2) = %d\n",
        res);
  res = strcmp(s1, s3);
  printf("strcmp(s1, s3) = %d\n",
        res);
return 0;
}
```

- Funciones y cadenas:

Una función acepta una cadena como argumento: **paso por referencia** (igual que un vector).

```
#include <stdio.h>
void imprime(char cadena[]);

int main() {
  char saludo[]="Hola";
  imprime(saludo);
  return 0;
}
void imprime(char cadena[]) {
  int i=0;
  while(cadena[i]!='\0') {
     printf("%c", cadena[i]);
     i++;
    }
  printf("\n");
}
```

8) Tema 8: Matrices:

Una matriz es un conjunto de valores del mismo tipo (`int`, `char`, `float`, etc.), de dos o más dimensiones

```
tipo identificador[dimension_1][dimension_2] ... [dimension_n];
```

tipo Tipo de los elementos de la matriz.
identificador Nombre de la matriz.
$dimensión_n$ Dimensión n-ésima de la matriz.

Ejemplo

```
// Crea una matriz de datos enteros, llamada
// tabla, de dos dimensiones y 9 elementos.
int tabla[3][3];
```

- Elementos de una matriz:

```c
#include <stdio.h>
int main (){
  int matriz[2][2]; // Matriz 2 x 2
  int fila, columna;
  // Inicializacion de elementos
  matriz[0][0] = 1;
  matriz[0][1] = 2;
  matriz[1][0] = 3;
  matriz[1][1] = 4;
  // Recorre matriz con un bucle for anidado
  for(fila = 0; fila < 2; fila++) {
     for(columna = 0; columna < 2; columna++)
       printf("%d\t", matriz[fila][columna]);
     printf("\n\n");
   }
  return 0;
}
```

- Inicialización de una matriz: Las dos formas son semejantes pero la primera por claridad es más recomendable.

```c
#include <stdio.h>
int main() {
  int matriz[2][3] = // Matriz 2 x 3
    {
      {10, 20, 30}, // 1a fila
      {40, 50, 60} // 2a fila
    };
```

```c
int matriz[2][3] = {10, 20, 30, 40, 50, 60};
```

- Operaciones con matrices: en este caso es una suma.

```c
#include <stdio.h>
int main() {
  int i,j;
  int m1[2][3] = {1, 2, 3, 4, 5, 6};
  int m2[2][3] = {4, 5, 12, 23, -5, 6};
  int m3[2][3]; // Matriz resultado
  // Realiza la suma con bucle anidado
  for(i = 0; i < 2; i++) // Filas
    for(j = 0; j < 3; j++) // Columnas
      m3[i][j] = m1[i][j] + m2[i][j];
```

- Funciones con matrices:

- Una matriz siempre se pasa por referencia: no se pasa la matriz completa sino la dirección del primer elemento.

```c
void funcion (int matriz[3][3], int nFil, int nCol);
```

- Se puede omitir el número de filas pero **no** el número de columnas.

```c
void funcion (int matriz[][3], int nFil, int nCol);
```

```c
//Error de sintaxis
void funcion (int matriz[][], int nFil, int nCol);
```

```c
#include <stdio.h>
void imprime_matriz(int M[][2], int f, int c);

int main()
{
  int tabla[2][2] = {{1,2}, {3,4}};
  imprime_matriz(tabla, 2, 2);
  return 0;
}

void imprime_matriz(int M[][2], int f, int c)
{
  int i, j;
  for(i = 0; i < f; i++) {
     for(j = 0; j < c; j++)
       printf("%d ", M[i][j]);
     printf("\n");
   }
}
```

```c
#include <stdio.h>
void absMatriz(int M[][2], int f, int c);

int main() {
  int tabla[2][2] = {{-1,2}, {-3,4}};
  printf("Antes: %d\n", tabla[0][0]);
  absMatriz(tabla, 2, 2); // ¡Sin asignacion!
  printf("Después: %d\n", tabla[0][0]);
  return 0;
}

void absMatriz(int M[][2], int f, int c) {
  int i, j;
  for(i = 0; i < f; i++)
     for(j = 0; j < c; j++)
       if (M[i][j] < 0)
         M[i][j] = -M[i][j];
}
```

9) Tema 9: Estructuras:

- Crea un nuevo tipo de datos que lleva como nombre el identificador de la estructura

Permiten almacenar valores de diferentes tipos bajo un mismo identificador.

```c
struct identificador
{
  tipo_1 comp_1;
  tipo_2 comp_2;
  ...
  tipo_n comp_n;
};
```

identificador Nombre de la estructura

tipo_n Tipo de datos del componente comp_n.

comp_n Componente n-ésimo de la estructura.

- Inicialización de estructuras: es mas recomendable usar el tipydef para así luego no tener que usar el struct para cada vez que se crea una nueva variable con el identificador.

```c
typedef struct
{
  tipo_1 comp_1;
  tipo_2 comp_2;
  ...
  tipo_n comp_n;
} identificador;
```

```c
struct identificador
{
  tipo_1 comp_1;
  tipo_2 comp_2;
  ...
  tipo_n comp_n;
};
```

- Ejemplo de cada tipo de inicialización:

```c
struct contacto
{
  char nombre[30];
  int telefono;
  int edad;
};

int main()
{
  struct contacto person1;

  return 0;
}
```

```c
typedef struct
{
  char nombre[30];
  int telefono;
  int edad;
} contacto;

int main()
{
  contacto person1;
  return 0;
}
```

- Inicialización de valores en estructuras:

```c
typedef struct {
  char nombre[50];
  char apellidos[50];
  int matricula;
} ficha;

int main ()
{
  ficha alumno1 = {"Yo", "Soy Aquel", 1234};
  return 0;
}
```

```c
typedef struct {
  char nombre[50];
  char apellidos[50];
  int matricula;
} ficha;

int main ()
{
  ficha alumno1 = {.apellidos = "Soy Aquel",
                   .matricula = 1234,
                   .nombre = "Yo"};
  return 0;
}
```

- Asignación de valores en estructuras:

```c
typedef struct {
  int day;
  int month;
  int year;
} date;

int main () {
  date d1, d2, d3;
  // Asignacion por componentes
  d1.day = 31;
  d1.month = 12;
  d1.year = 1999;
  // Asignacion con el operador cast
  d2 = (date) {1, 1, 2000};
  // Asignacion por copia
  d3 = d1;
  return 0;
}
```

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

typedef struct {
  char nombre[50];
  char apellidos[50];
  int matricula;
} ficha;

int main ()
{
  ficha alumno1, alumno2, alumno3;
  // Para asignar cadenas usamos strcpy
  strcpy(alumno1.nombre, "Yo");
  strcpy(alumno1.apellidos, "Soy Aquel");
  alumno1.matricula = 1234;
  return 0;
}
```

```c
#include <stdio.h>

typedef struct {
  char nombre[50];
  char apellidos[50];
  int matricula;
} ficha;

int main () {
  ficha alumno;
  printf("Nombre:");
  scanf("%s", alumno.nombre);
  printf("Apellidos:");
  scanf("%s", alumno.apellidos);
  printf("Numero de matricula:");
  scanf("%d", &alumno.matricula);
  return 0;
}
```

- Estructuras dentro de estructuras:

```c
typedef struct
{
  int d, m, a;
} fecha;

typedef struct
{
  char nombre[50];
  char apellidos[50];
  int matricula;
  fecha fNacimiento;
} ficha;
```

```c
ficha alumno1, alumno2;

alumno1.fNacimiento.d = 31;
alumno1.fNacimiento.m = 12;
alumno1.fNacimiento.a = 1999;

alumno2.fNacimiento = (fecha){1, 1, 2000};
```

- Vector de estructuras:

A partir de una estructura previamente definida se pueden generar vectores basados en esa estructura.

```c
#include <stdio.h>

typedef struct
{
  int day;
  int month;
  int year;
} date;

int main() {
  date fechas[3] = {// Vector de 3 fechas
    {1, 1, 1999},
    {31, 12, 2000},
    {15, 5, 1980}
  };
  return 0;
}
```

La asignación de valores sigue las mismas reglas que para vectores de tipos simples (mediante []).

```c
#include <stdio.h>

typedef struct
{
  int day;
  int month;
  int year;
} date;

int main() {
  date fechas[3]; // Vector de 3 fechas
  fechas[1].day = 1;
  fechas[2] = (date) {31, 12, 1999};
  fechas[3] = fechas[2];
  return 0;
}
```

- Funciones y estructuras:

Una función acepta estructuras (**paso por valor**).

```c
#include <stdio.h>
#include <math.h>
//Definicion de estructura
typedef struct {
  float real, imaginaria;
} complejo;
//Funcion que acepta una estructura
float modulo(complejo c);

int main(){
  complejo comp={1, 3};
  printf("El modulo es: %f\n", modulo(comp));
  return 0;
}
//Implementacion de la funcion
float modulo(complejo c){
  return sqrt(c.real * c.real + c.imaginaria * c.imaginaria);
}
```

Una función puede devolver una estructura

```c
#include <stdio.h>

typedef struct {
  float real, imaginaria;
} complejo;
//Funcion que devuelve estructura
complejo conjugado(complejo c);

int main(){
  complejo comp1 = {1, 3}, comp2;
  comp2 = conjugado(comp1);
  printf("%f", comp2.imaginaria);
  return 0;
}

complejo conjugado(complejo c){
  return (complejo) {c.real, -c.imaginaria};
}
```

10) Tema 10: Enumeraciones en C

Definición

Con enum se pueden definir tipos de datos enteros que tengan un rango limitado de valores, y darle un nombre significativo a cada uno de los posibles valores.

Código

```c
enum nombre_enum {lista_de_valores};
```

Ejemplo

```c
enum dia{ //valores enteros: 0 al 6
  lunes, martes, miercoles, jueves, viernes, sabado, domingo
};
```

```c
#include <stdio.h>
enum dia{ //valores enteros: 0 al 6
  lunes, martes, miercoles, jueves, viernes, sabado, domingo
};

int main()
{
  enum dia hoy, manana;
  hoy = lunes;
  manana = hoy + 1;
  printf("%d\n", hoy);
  printf("%d\n", manana);
  return 0;
}
```

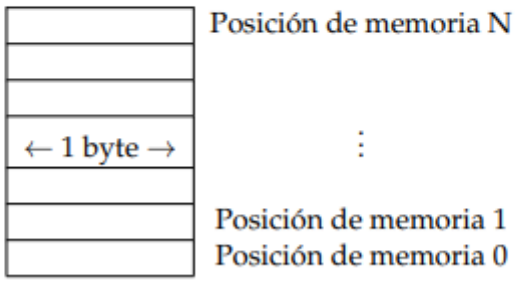
```c
#include <stdio.h>
enum dia{ //valores enteros: 1 en adelante
  lunes = 1, martes, miercoles, jueves, viernes, sabado, domingo
};

int main()
{
  enum dia hoy, manana;
  hoy = lunes;
  manana = hoy + 1;
  printf("%d\n", hoy);
  printf("%d\n", manana);
  return 0;
}
```

11) <u>Tema 11: Punteros</u>

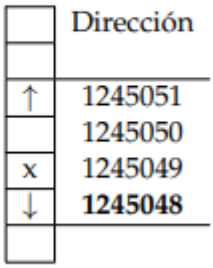
Datos y Memoria

- Los datos de un programa se almacenan en la memoria del ordenador.
- La memoria del ordenador está estructurada en **bytes** (8 bits).
- Cada byte tiene una posición en la memoria (dirección).



- Ejemplo: un dato `int` ocupa 4 bytes.

```c
int x;
```



- El operador & recibe el primer valor que ocupa el puntero, es decir, el primer valor que toma la variable en la memoria RAM.

Operador &

El operador & (*ampersand*) aplicado a una variable cualquiera proporciona su dirección de memoria.

```
#include <stdio.h>

int main()
{
  // No hace falta asignar valor inicial
  // para que la variable
  // tenga dirección de memoria
  int x;

  printf("La variable x está almacenada en %lli.\n",
        &x);

  return 0;
}
```

¿Qué es un puntero?

Un puntero apunta a una variable

Un **puntero** (*pointer*) es una **variable** (tipo número entero) que contiene la dirección de memoria de una variable:

- El puntero es una referencia de la variable a la que apunta.
- El valor del puntero es la dirección de memoria de la variable.
- La variable está apuntada por el puntero.

Declaración de un puntero

Un puntero se declara:

- Indicando el tipo de datos de la variable a la que apunta.
- Incluyendo un asterisco * antes del identificador.

```
int main()
{
  // p1: puntero apuntando a una variable de tipo entero
  int *p1;
  // p2: puntero apuntando a una variable de tipo caracter
  char *p2;
  // p3: puntero apuntando a una variable de tipo real
  float *p3;
  // p4: puntero apuntando a una variable genérica
  void *p4;

  return 0;
}
```

Un puntero es una variable `int`

- El contenido de una variable puntero es la dirección de memoria, un valor de tipo entero.
- Su tamaño depende del sistema:
  - Sistemas de 32 bits ocupan 4 bytes.
  - Sistemas de 64 bits ocupan 8 bytes.

```c
#include <stdio.h>

int main()
{
  int x, *p;
  // sizeof devuelve el numero de bytes de una variable o tipo de datos
  printf("La variable x ocupa %i bytes.\n",
        sizeof(x));
  printf("El puntero p ocupa %i bytes.\n",
        sizeof(p));

  return 0;
}
```

Asignación

```c
int main()
{
  int x, y;
  // p1 apunta a x
  int *p1 = &x, *p2, *p3;
  // p2 apunta a y
  p2 = &y;
  // p3 apunta a la misma variable que p1, es decir, x
  p3 = p1;

  return 0;
}
```

Operador $*$

El operador $*$ aplicado a un puntero proporciona el valor de la variable apuntada por el puntero.

```c
#include <stdio.h>

int main()
{
  int x = 10, *p;
  // Operador & para obtener la direccion de x
  p = &x;
  // *p y x son lo mismo
  printf("La variable apuntada vale %i",
        *p);
}
```

Operaciones con punteros

- La **suma o resta de un entero a un puntero** produce una **nueva localización de memoria**.
- Se pueden **comparar punteros** utilizando expresiones lógicas para **comprobar si apuntan a la misma dirección de memoria**.
- La **resta de dos punteros** da como resultado el **número de variables entre las dos direcciones**.

Punteros y vectores

El identificador de un vector es un puntero *constante* que apunta al primer elemento del vector.

```
#include <stdio.h>

int main()
{
  int vector[3] = {1, 2, 3};
  int *p;
  // p apunta al primer elemento del vector
  p = &vector[0];
  printf("El primer elemento es %i\n", *p);
  // De forma mas concisa
  p = vector;
  printf("El primer elemento es %i\n", *p);

  return 0;
}
```

- Cuando se avanza por un puntero p=p+1 el puntero p avanza los bytes necesarios en función del tipo de puntero para llegar a la siguiente dirección de memoria libre para introducir un valor

Recorrido de un vector

Podemos recorrer un vector a través de su puntero con sumas y restas

```
#include <stdio.h>
int main()
{
  int i, vector[3] = {1, 2, 3};
  int *p, *p1;
  p = vector;
  // p apunta a vector[0].
  printf("%i\t", *p);
  // p + 1 apunta a vector[1]
  p1 = p + 1;
  printf("%i\t", *p1);
  // *(p + 1) es equivalente a v[i + 1]
  printf("%i\t", *(p + 1));
  printf("%i\t", vector[1]);
  return 0;
}
```

Modificación de un vector

Podemos modificar un vector a través de su puntero

```c
#include <stdio.h>
int main(){
  int i, vector[3];
  int *p;
  p = vector;
  //vector[0] - 1
  *p = 1;
  //vector[1] - 2
  *(p + 1) = 2;
  //vector[2] - 3
  *(p + 2) = 3;

  printf("El vector es %i, %i, %i.\n",
        vector[0], vector[1], vector[2]);

  return 0;
}
```

Aritmética de punteros con vectores

```c
#include <stdio.h>
#define N 10

int main(){
  int vector[N] = {1};
  int *pVec, *pFin;
  // Puntero apuntando al segundo elemento
  pVec = vector + 1;
  // Puntero apuntando al ultimo elemento
  pFin = vector + N - 1;
  // Comparamos los punteros para avanzar
  while (pVec <= pFin)
    { // vector[i] - vector[i - 1] + 1
      *pVec = *(pVec - 1) + 1;
      printf("%i\t", *pVec);
      ++pVec; // Movemos el puntero por el vector
    }
  return 0;
}
```

Punteros y cadenas

El identificador de una cadena es un puntero *constante* que apunta al primer elemento.

```c
#include <stdio.h>

int main()
{
  char letras[3] = {'a', 'b', 'c'};
  char *p;
  // p apunta al primer elemento
  p = &letras[0];
  printf("El primer elemento es %c\n", *p);
  // De forma mas concisa
  p = letras;
  printf("El primer elemento es %c\n", *p);

  return 0;
}
```

Recorrido de una cadena

```c
#include <stdio.h>
int main()
{
  char mensaje[] = "Hola Mundo";
  char *p = mensaje;
  int i = 0;
  // Movemos el puntero por la cadena
  while(*p != '\0')
    {
      printf("%c", *p);
      p++; // Incrementa el puntero para
    } // pasar al siguiente caracter
  printf("\n");
}
```

Aritmética de punteros con cadenas

```c
#include <stdio.h>

int main(){
  char texto[] = "Hola Mundo";
  char *pChar = texto;

  while (*pChar != '\0')
    ++pChar; // Movemos el puntero por la cadena

  // El identificador "texto" es un puntero que
  // apunta al primer caracter de la cadena.
  // Si lo restamos del puntero movil
  // tenemos el total.
  printf("La cadena tiene %i caracteres.\n",
        pChar - texto);

  return 0;
}
```

Punteros a estructuras

- Un puntero a una estructura se declara igual que un puntero a un tipo simple.
- Para acceder a un miembro de la estructura se emplea el operador ->.

```c
#include <stdio.h>
typedef struct
{
  int y, m, d;
} fecha;

int main(){
  fecha f = {2000, 10, 15}, *p;
  // p apunta a la estructura
  p = &f;

  printf("%i-%i-%i",
        p->d, p->m, p->y);

  return 0;
}
```

Ejemplo

```c
#include <stdio.h>
typedef struct
{
  int y, m, d;
} fecha;

int main()
{
  fecha f, *p = &f;
  // Rellenamos la estructura a través de su puntero
  p->d = 15;
  p->m = 10;
  p->y = 2000;

  printf("%i-%i-%i",
        f.d, f.m, f.y);

  return 0;
}
```

Paso por referencia

- El uso de punteros en funciones permite el **paso por referencia**. De esta forma la función puede acceder (y **modificar**) a la variable original (*sin copia*).
- Las funciones que emplean **vectores y cadenas** como argumentos funcionan con **paso por referencia** (*el identificador de un vector es un puntero*).

Ejemplo

```
#include <stdio.h>
void operaciones (float x, float y,
                  float *s, float *p, float *d);
int main(){
  float a = 1.0, b = 2.0; // Datos
  float suma, producto, division; // Resultados
  operaciones(a, b, &suma, &producto, &division);
  printf("S: %f \t P: %f \t D: %f \t",
         suma, producto, division);
  return 0;
}
//Funcion con varios resultados
void operaciones (float x, float y,
                  float *s, float *p, float *d)
{// Cada puntero sirve para un resultado
  *s = x + y;
  *p = x * y;
  *d = x / y;
}
```

`malloc` y `free`

- La **asignación dinámica** de memoria permite definir **objetos (p.ej. vectores) de dimensión variable**.
- La función `malloc` permite asignar, durante la ejecución del programa, un bloque de memoria de n bytes consecutivos para almacenar los datos (devuelve `NULL` si no es posible la asignación)
- La función `free` permite liberar un bloque de memoria previamente asignado.

- La función exit(-1) señala un error por consola.

Uso de malloc y free

```
int *pInt;
...
// Reservamos la memoria suficiente para almacenar
// un int y asignamos su dirección a pInt
pInt = malloc(sizeof(int));

// Comprobamos si la asignación
// se ha realizado correctamente
if (pInt == NULL) {
     printf("Error: memoria no disponible.\n");
     exit(-1);
}

... // Codigo usando el puntero

free(pInt); // Liberamos memoria al terminar
```

Ejemplo

```
#include <stdio.h>
#include <stdlib.h> //Necesaria para malloc y free
int main () {
  int *vec, i, N = 100;
  vec = malloc(sizeof(int) * N);
  //Comprueba si malloc ha funcionado
  if (vec == NULL) {
     printf("Error: memoria no disponible.\n");
     exit(-1);
    }
  // El resultado es un puntero-vector de N elementos
  for (i = 0; i < N; ++i)
    vec[i] = i * i; // Rellenamos el puntero-vector
  for (i = 0; i < N; ++i) // Mostramos contenido
    printf("%i \t", *(vec + i));
  free(vec); // Liberamos el puntero-vector
  return 0;
}
```

12) Tema 12: Ficheros

- Hasta ahora:
  - Introducción de datos desde el **teclado**.
  - Presentación de datos en **pantalla**.
  - **Los datos se pierden** cuando finaliza el programa.
- Ahora vamos a ver:
  - **Almacenamiento de datos** en ficheros que pueden ser leídos por el programa.
  - **Operaciones con ficheros**: apertura, lectura y/o escritura, y cierre.

Tipo FILE

En C se emplea la estructura de datos de tipo `FILE` (declarada en `stdio.h`):

```
#include <stdio.h>

int main ()
{
  FILE *pf;

  return 0;
}
```

- Cuando un fichero se pone en modo w, reescribe el fichero es decir borra el contenido anterior y si se pone en modo a se añade sobre lo anterior. Es importante devolver un error si no se pudo abrir el fichero.

`fopen` abre un fichero para para leer y/o escribir en él.

```
FILE *fopen (const char *nombre, const char *modo);
```

- `nombre`: nombre del fichero (*debe respetar las normas del sistema operativo en el que se ejecute el programa*).
- `modo`: indica cómo se va a abrir el fichero:
  - lectura: `r`
  - escritura: `w`
  - añadir: `a`
- Devuelve un puntero a una estructura de tipo `FILE` o un puntero nulo `NULL` si se ha producido un error.

Ejemplo de fopen

```
#include <stdio.h>
int main ()
{
  FILE *pf;
  // Atención a los separadores en la ruta del fichero,
  //y a las comillas dobles
  pf = fopen("c:/ejemplos/fichero.txt", "r");

  if (pf == NULL)
    {
      printf("Error al abrir el fichero.\n");
      return -1;
    }
  else
    {
      printf("Fichero abierto correctamente.\n");
      return 0;
    }
}
```

Cerrar un fichero: `fclose`

`fclose` cierra un fichero previamente abierto con `fopen`

```
int fclose (FILE *pf);
```

- El puntero `pf`, de tipo `FILE`, apunta al fichero.
- La función devuelve 0 si el fichero se cierra correctamente o `EOF` si se ha producido un error.

Escritura de ficheros: `fprintf`

```
int fprintf(FILE *stream, const char *format, ...)
```

- Escribe en un fichero con el formato especificado (**igual que** `printf`)
- Devuelve el número de caracteres escritos, o un valor negativo si ocurre un error.

Ejemplo de `fprintf`

```
#include <stdio.h>

int main(){
  FILE *pf;
  int vals[3] = {1, 2, 3};
  // Abrimos fichero para escritura
  pf = fopen("datos.txt", "w");
  if (pf == NULL) {// Si el resultado es NULL mensaje de error
    printf("Error al abrir el fichero.\n");
    return -1;
  }
  else {// Si ha funcionado, comienza escritura
    fprintf(pf, "%i, %i, %i",
          vals[0], vals[1], vals[2]);
    fclose(pf); // Cerramos fichero
    return 0;
  }
}
```

- EOF es la marca de final de fichero

Lectura de ficheros: `fscanf`

```
int fscanf(FILE *stream, const char *format, ...)
```

- Lee desde un fichero con el formato especificado (**igual que** `scanf`)
- Devuelve el número de argumentos que han sido leídos y asignados o `EOF`[1] si se detecta el final del fichero.

Ejemplo de `fscanf`

```
#include <stdio.h>

int main()
{
  int i, n, vals[3];
  FILE *pf;
  // Abrimos fichero para lectura
  pf = fopen("datos.txt", "r");
  // Leemos datos separados por comas
  n = fscanf(pf, "%i, %i, %i",
       &vals[0], &vals[1], &vals[2]);
  printf("Se han leído %i argumentos.\n", n);
  fclose(pf);
  // Mostramos en pantalla lo leído
  for (i = 0; i < 3; i++)
    printf("%i\t", vals[i]);

  return(0);
}
```

Lectura de datos con separadores

Si en un fichero hay datos numéricos junto con cadenas de caracteres, hay que usar `[^X]`, donde `X` es el carácter empleado como separador.
Por ejemplo, para leer datos separados por punto y coma empleamos: `[^;]`

Ejemplo

Sea un fichero con el siguiente contenido:

Jorge Rodríguez; Profesor; 35; 84.4

Para leerlo:

```
fscanf(pf, "%[^;];%[^;];%d;%f\n",
     nombre, tipo, &edad, &peso);
```

Marca de final de fichero EOF

- Cuando se crea un fichero nuevo con `fopen` se añade automáticamente al final la marca de fin de fichero `EOF` (*end of file*).
- Es una marca escrita al final de un fichero que indica que no hay más datos.
- Cuando se realizan operaciones de lectura o escritura es necesario comprobar si se ha alcanzado esta marca.

Comprobación de EOF

- `feof` detecta el final del fichero: devuelve un valor distinto de cero después de la primera operación que intente leer después de la marca final del fichero.

```
while (feof(pf) == 0)
{
// Operaciones de L/E
}
```

- `fscanf` y `fprintf` devuelven `EOF` cuando alcanzan la marca. Se puede emplear directamente este resultado (sin necesidad de `feof`)

```
while(fscanf(...) !=EOF )
{
// Sentencias
}
```

Ejemplo: número de líneas de un fichero

```
#include <stdio.h>

int main()
{
  int i, nLineas = 0;
  char x; // Variable auxiliar
  FILE *pf;
  pf = fopen("lorem_ipsum.txt", "r");
  // Leemos caracter a caracter
  while (fscanf(pf, "%c", &x) != EOF)
    //Si lo leído es un salto de línea
      if (x == '\n')
        //incrementamos el contador
        ++nLineas;
  printf(" %i", nLineas);
  return 0;
}
```

stdin, stdout, and stderr

- Al ejecutarse un programa de C se abren tres ficheros de forma automática (identificados por tres punteros de tipo `FILE`):
  - `stdin`: entrada estándar del programa (habitualmente el teclado).
  - `stdout`: salida estándar del programa (habitualmente la pantalla).
  - `stderr`: fichero estándar de error.

Ejemplo

```
#include <stdio.h>

int main(){
  printf("hello there.\n");
  fprintf(stdout, "hello there.\n");
  return 0;
}
```

Movimiento en un fichero

`fseek`

```
int fseek(FILE *stream, long int offset, int whence)
```

- Desplaza a una posición en un fichero
- `offset` (long): valor (en bytes) a ir desde `whence`
- `whence`:
  - `SEEK_SET` Comienzo del fichero
  - `SEEK_CUR` Posición actual
  - `SEEK_END` Final del fichero

`ftell`

```
long int ftell(FILE *stream)
```

- Devuelve la posición actual respecto del inicio del fichero.
- Las unidades suelen ser **bytes**.
- Es una función de tipo `long`

- Es importante saber que a la hora de usar estos comandos para saber el número de bytes en un fichero depende del número de caracteres y no siempre es fiable.

Ejemplo: nº de bytes de un fichero

```
#include <stdio.h>

int main()
{
  long int fsize; // tamaño del fichero
  FILE *pf;
  pf = fopen("datos.txt", "r");
  // Desplaza al final
  fseek(pf, 0, SEEK_END);
  //Almacena la posición
  fsize = ftell(pf);
  printf("El fichero tiene %li bytes.\n",
       fsize);
  return 0;
}
```