TEMA 4. ELEMENTOS BÁSICOS DE PROGRAMACIÓN

4.1 Definiciones y palabras reservadas básicas del C

El lenguaje C++ está **compuesto** por:

- 32 **palabras clave** (estándar ANSI), comunes a todos los compiladores de C.
- palabras clave añadidas por cada compilador de C (no estándar).
- **Sintaxis** formal del lenguaje.

  Indica **cómo** se organiza (estructura) un programa; **cómo** se terminan las sentencias; **cómo** se escriben cada una de las instrucciones, …es decir, **son las reglas para escribir correctamente un programa en C.**

Algunas **características** del lenguaje C:

- **Distingue** entre MAYÚSCULAS y minúsculas.
- **Todas** las **palabras clave** se escriben en **minúscula**.
- Los nombres de las palabras clave **no pueden usarse** para identificar a variables o funciones.
- Las 32 palabras claves definidas por el estándar ANSI son:

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**

Estructura general de un programa en C

- Todos los programas en C constan de **una** o **más funciones**.
- La función principal **main()** SIEMPRE está presente, es la única que obligatoriamente debe existir.
  - es la **primera función** llamada cuando se ejecuta un programa
  - **controla** toda la actividad desarrollada por el programa (es la encargada de hacer las llamadas al resto de funciones).

Sección de **includes** (para la inclusión de librerías/bibliotecas)

```
#include <librería>  o bien
#include "librería"
```

Sección de **defines**

```
#define constante valor
```

Sección para la declaración de **constantes**

```
const <tipo> <identificador> = <valor>;
```

Prototipos de **funciones** y **procedimientos**

```
Tipo_devuelto Nombre(lista de parámetros);
```

Función **main** (Programa principal)

```
int main()
{ Declaración de variables locales;
  Sentencia;
   . . .
  return 0;
}
```

Implementación de **funciones** y **procedimientos**

```
Tipo_devuelto Nombre(lista de parámetros)
{ Declaración de variables locales;
  sentencia;
  . . .
  return resultado;
}
```

Cuando haga falta **devolver un resultado**. **Obligatorio**

En un programa en C hay dos clases de instrucciones:

- Las instrucciones de control de flujo: *if* (si), *while* (mientras), etc.
- Funciones que realizan acciones o cálculos (leer un nº por teclado, escribir en pantalla, hacer cálculos, etc.).

Respecto a las funciones, en C se distinguen dos clases:

- Funciones de la biblioteca estándar de C: Son funciones de uso general que trae el compilador implementadas: *prinft( ), gets( )*,etc.
- Funciones que define el programador (debido a que no existen en ninguna de las bibliotecas).

```
#include <stdio.h> /* ficheros cabecera */
#include <conio.h> // contiene prototipos de funciones

float factorial(int n) { /* calcula el factorial */
 int j; float fact = 1;
 for(j = 1; j <= n; j++)
   fact = fact * j;
 return fact;
}

int main(void) { // primera funcion que se ejecuta
 int n;
 printf("Dame número: ");
 scanf("%i", &n);
 printf("El factorial de %i es %.0f: ", n, factorial(n) );
 getch(); /* espera a que se pulse una tecla */
 return 0;
}
```

- Se pueden poner **Comentarios** (líneas que no serán compiladas y por tanto no afectan a la ejecución del programa) para describir el programa.

  /* comentario, que puede ocupar varias líneas ****/***

  // comentario de una sola línea **(PERMITIDO EN C++)**

- Toda instrucción en C debe terminar con un punto y coma (;).

4.2 Tipos, Variables y Constantes

4.2.1 Tipos de Datos Fundamentales

El tipo de una variable (contenedor de información) indica que **tipo de información** puede contener y **qué operaciones** puede realizar.

Existen 5 tipos simples en C:

- *char* (carácter), ocupa 8 bits (1 bytes)
- *int (entero),* ocupa 16 bits (2 bytes)
- *float (coma flotante),* ocupa 32 bits (4 bytes)
- *double (coma flotante doble precisión)* ocupa 64 bits (8 bytes)
- *void (sin valor).* ocupa 0 bits (0 bytes)

Los tipos básicos (excepto *void*) pueden tener modificadores:

MODIFICADORES DE TIPO ➔ Preceden al **tipo de dato** (excepto a void), para modificar el valor del tipo base ➔ Nos permiten ajustar los valores de las variables a nuestras necesidades. Son los siguientes:

- **signed** (por defecto): Puede tomar valores positivos y negativos
- **unsigned**: Indica que sólo puede tomar valores positivos,
- **short** (por defecto): Indica que la variable tiene rango corto.
- **long**: Indica que la variable tiene rango o tamaño doble

Los modificadores *signed* y *short* pueden omitirse:

Tipo	Tamaño en bits	Rango
char	8	−128 a 127
unsigned char	8	0 a 255
signed char	8	−128 a 127
int	16	−32768 a 32767
unsigned int	16	0 a 65535
signed int	16	−32768 a 32767
short int	16	−32768 a 32767
unsigned short int	16	0 a 65535
signed short int	16	−32768 a 32767
long int	32	−2147483648 a 2147483647
signed long int	32	−2147483648 a 2147483647
float	32	3.4E−38 a 3.4E+38
double	64	1.7E−308 a 1.7E+308
long double	64	1/7E−308 a 1.7E+308

- **SIN VALOR** (void), se utilizan para:
  - Indicar que una funcion **no devuelven ningún valor**
  - Indicar que una función **no tiene parámetros**.

- **CARÁCTER** (char) ➔ Se usa para almacenar caracteres o nº enteros pequeños. Por defecto pueden ser positivos o negativos, aunque podemos forzar que sólo sean positivos (**unsigned**). Posibilidades:
  - unsigned char ➔ 2^{8bits} = [0, 255] = [0, 2^8-1]
  - char ➔ 2^{8bits} = [-128, 127] = [-2^7, 2^7-1]

- **ENTERO** (int) ➔ Se usa para almacenar nº enteros. Por defecto pueden ser positivos o negativos, aunque podemos obligar a que sólo puedan ser positivos (**unsigned**). Posibilidades:
  - Pueden ocupar **2bytes:**
    - unsigned int ➔ 2^{16bits} = [0, 65535] = [0, 2^{16}-1]
    - int ➔ 2^{16bits} = [-32768, 32767] = [-2^{15}, 2^{15}-1]
  - Pueden ocupar **4bytes:**
    - unsigned long int ➔ 2^{32bits} = [0, 4294967295] = [0, 2^{32}-1]
    - long int ➔ 2^{32bits} = [-2147483648, 2147483647] = [-2^{31}, 2^{31}-1]

- **REAL** (float, double) ➔ Sus valores son número reales, con un componente entero y uno fraccionario. Posibilidades:
  - Pueden ocupar **4bytes**: float
  - Pueden ocupar **8bytes**: double ➔ Mayor precisión que los float.

Si en tiempo de ejecución se asigna un valor mayor del permitido a una variable ➔ **Overflow** (valor excesivo). Provoca un error en el resultado

Ejemplo:

unsigned char a;	Reserva en memoria 8 bits para la variable a. Como es *unsigned* no hay bits de signo y los 8 bits se usan para guardar el nº. El rango de valores es: 0 al 255 (0 a 2^8-1)
a = 300;	300 en binario es: 100101100 Solo almacena 8 bits: 00101100 El equivalente decimal es: 44 ¡El nº que almacena es 44 en vez de 300!
a = a + 5;	a = 44 + 5 → a = 49

¡El resultado que devolvería seria 49 en vez de 305!

4.2.2 Variables y Constantes

Una **variable** es un contenedor de información (datos), y su valor se encuentra almacenado en memoria.

Para referirnos a los datos contenidos en la memoria utilizamos **su dirección** ➔ no es práctico, porque una dirección **no es significativa de su contenido** ➔ Se hacen necesarios los **identificadores** ⇨ *Nombre simbólico para referirse a variables, funciones y cualquier otro objeto definido por el usuario en un programa.*

Todas las variables o funciones que usemos deben ser **declarados** y deben tener un nombre (identificador): al comienzo del programa se deben indicar los nombres de las variables y el tipo de los datos que almacenan

Porque se debe hacer reserva de memoria para cada variable, y establecer la correspondencia entre nombre de variable y posición de memoria asignada.

Un identificador debe cumplir las siguientes normas:

1. Longitud máxima: 32 caracteres. Debe comenzar por una letra o un símbolo de subrayado (_) seguido por letras, números o subrayado.

   Bien: cont, prueba25, a_3 Mal: 1cont, hola!, b...total
2. C distingue entre mayúsculas y minúsculas: a y A son <>.
3. No puede llamarse igual que una palabra clave, o que una función.
4. Es aconsejable elegir nombres que sean significativos.

   No aconsejable: *float funcion1(int a, int b)*

   Si aconsejable: *float media(int teoria, int practica)*

Una **constante** es un valor que **no puede cambiar** durante la ejecución de un programa ➔ No pueden aparecer en la parte izquierda de una asignación. Pueden ser:

- De **cualquiera** de los **tipos** vistos para las **variables** (excepto void):
  - Constantes enteras: 25, -126, 0x7FC (hexadecimal), 0767 (octal).
  - Constantes de Punto Flotante: 23.5, -0.05, -3.8E+17, 2.5E-4.
  - Constantes carácter: ‘x’, ‘t’, ‘\n’, ‘\t’. (Su valor es su código ASCII).
- Una **cadena de caracteres** (conjunto de caracteres encerrado entre ” ”).
  - Constantes de cadenas: “constante de cadena”, ”Bienvenidos”.

4.2.3 Declaración de variables

Toda variable debe ser declarada para poder ser usada. Hay que indicar:

- el tipo de los valores que podrá contener. Lo que determinará qué cantidad de memoria habrá que reservar.
- el nombre de la variable.

Ejemplo: int x, z=2; ← Podemos inicializar la variable en su declaración.

La forma general de declaración es la siguiente:

```
[Modif.tipo] <tipo> {<identificador> [= valor],...} ;
```

donde:

- modificador tipo: *signed* (por defecto), *unsigned, long*, *short*.
- tipo de dato: *char, int, float, double, void.*
- identificador: nombre con el que nos vamos a referir a la variable.
- valor: Podemos inicializar la variable en la misma declaración.
- [ ], indica que es opcional. < >, indica que es obligatori
- { } , indica que se puede repetir varias veces, separando con comas

Toda declaración debe finalizar con un punto y coma (;).

4.2.4 Definición de constantes

Hay dos formas de crear constantes: *const* y #define

- const se usa para indicar que una variable no puede cambiar de valor.

```
const <tipo> <identificador> = valor;
```

const int i = 10; /* la variable i la declaramos como constante */

i = 15; /* el compilador dará error ya que i es constante */

- #define se usa para definir y eliminar constantes.

```
#define nombre_constante expresion_constante
```

donde: **no termina en ";"**

- *nombre_constante* es un identificador (por convenio en mayúsculas).
- *expresion_constante*. No puede contener variables, aunque si constantes.

Una constante puede usarse para definir otras constantes:

```
#define MENSAJE "Pulse una tecla para continuar \n"
#define PI 3.1416
#define PIDOBLE (2 * PI)
```

El compilador sustituye en el fichero fuente la constante por su definición:

```
printf(MENSAJE);
printf("\nEl area de radio 2 es %.2f \n", PIDOBLE * 2);
```

Son sustituidas por el compilador por estas otras:

```
printf("Pulse una tecla para continuar \n");
printf("\nEl area de radio 2 es %.2f \n", (2 * 3.1416) * 2);
```

El uso de constantes facilita la edición de los programas: si queremos cambiar el mensaje, sólo hay que cambiar la definición de la constante MENSAJE: el compilador lo sustituye en todos los lugares donde MENSAJE aparece.

4.3 Operaciones básicas aritmético-lógicas. Operadores y expresiones.

4.3.1 Operación de asignación

Provoca que la variable de la izquierda tome el valor de la expresión derecha.

```
<nombre variable> = <expresión>;
```

o bien, las dos siguientes que son equivalentes:

```
<nombre variable> operador= expresión;
<nombre variable> = <nombre variable> operador expresión;
```

Ej:

n = n + 1; → n += 1;	n = n - t; → n -= t;	n = 5;	n = 5 * (7 + a) / 2;
n = n * 5; → n *= 5;	n = n / 2; → n /= 2;	n = n + 5 * b → n += (5 * b)	

4.3.2 Operadores aritméticos

Operador	Acción
-	Resta, Cambio de signo
+	Suma
*	Multiplicación
/	División
%	Módulo ⇨ Resto de una división entera
--	Decremento en 1 unidad
++	Incremento en 1 unidad

Operadores aritméticos.

Debe tenerse en cuenta que:

- Cuando se aplica / a operandos *int* o un *char* se realiza una división entera.
- El operador % no puede aplicarse a los tipos float ni double.
- Los operadores ++ y -- incrementan o disminuyen en una unidad el valor de la variable a la que afectan. Si preceden a la variable, ésta es incrementada/decrementada antes de que el valor de dicha variable sea utilizado en la expresión en la que aparece. Si es la variable la que precede al operador, la variable es incrementada/decrementada después de ser utilizada en la expresión. Ejemplo:

```
i = 2;
j = 2;
m = i++;          // primero se asigna y después se incrementa m = 2 e i = 3
n = ++j;          // primero se incrementa y después se asigna n = 3 y j = 3
```

La precedencia de los operadores aritméticos es la siguiente:

```
mayor     ++   --
          -    (cambio de signo)
          *  /  %
menor     +  -
```

Los operadores del mismo nivel de precedencia son evaluados por el compilador de izquierda a derecha.

4.3.3 Operadores relacionales

Permiten comparar unas expresiones con otras, devolviendo ***true*** cuando es cierta la comparación y ***false*** en caso contrario.

Operadores relacionales: >, >=, <, <=, = = (iguales), ! = (distinto)

Su forma general es:

```
expresion1 op expresion2
```

se evalúa *expresion1* y *expresion2* y si la condición representado por el operador relacional se cumple devuelve *true* y si no *false*

4.3.4 Operadores lógicos

Permiten combinar los resultados de los operadores relacionales, y así ver si se cumplen varias condiciones simultáneamente o alguna de ellas.

Operadores lógicos:

&& (y)	expr1 && expr2	deben cumplirse simultáneamente las 2.
|| (o)	expr1 | | expr2	basta con que se cumpla una de ellas.
! (negación)	! expr	niega expr. Si verdad -> falso y viceversa

4.3.5 Expresiones

Una expresión en C es cualquier combinación de operadores, constantes y variables. En C existen distintos tipos de expresiones.

- **Expresiones aritméticas**: Formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales. Por ejemplo, la solución de la ecuación de segundo grado:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$ x = **(**-b + sqrt ((b * b) – (4 *a *c)) **)** / (2 * a);

- **Expresiones Lógicas**: Formadas por variables y/o constantes, **operadores lógicos (| |, &&** y **!), valores lógicos (*true***, distinto de 0 y ***false***, igual a 0), así como **operadores relacionales**. Estas expresiones devuelven siempre un valor 1 (*true*) o 0 (*false*).

  Por ejemplo:

  a = **(** (b > c) && (c > d) **)** || (c == e) || (e == b);

4.3.5.1 Reglas de precedencia y asociatividad

El resultado de una expresión depende del orden en que se ejecutan las operaciones. Este orden viene determinado por las reglas de ***precedencia***.

El orden de evaluación puede modificarse con paréntesis.

Mayor ↑	( )
	! ++ - - - (monario)
	* / %
	+ -
	< <= > >=
	= = !=
	&&
	||
Menor	= += -= *= /=

4.3.5.2 Conversiones de tipos en las expresiones

Existen dos formas de realizar una conversión de tipo:

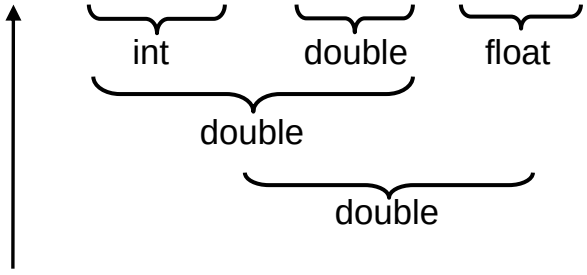
- **Implícita:** Si mezclamos variables y constantes de distinto tipo en una misma expresión, se convierten operación a operación al tipo más fuerte (al de mayor precisión), según el siguiente orden de promoción:

char → int → unsigned int → long → unsigned long → float → double → long double

Ejemplo:

```
char ch;
int i, rdo;
float f;
double d;

rdo = (ch / i) + (f * d) – (f + i)
```



¡ojo ¡ se almacena como un int ➔ se pierde la parte decimal

- **Explícita (cast o molde):** El cambio de tipo se produce a petición del programador, y tiene la siguiente sintaxis:

```
(tipo) expresión
```

donde *tipo* es uno de los tipos estándar de C o uno definido por el usuario. Por ejemplo, si se quiere asegurar que la expresión **x / 2** se evalúe como de tipo ***float***, se puede escribir: **(*float*) x / 2**.

Ejemplo:

```
#include <stdio.h>
int main(void) {
 int a = 5, b = 2;
 float c;
 c = a / b;          // division entera
 printf("%f", c);   // imprime 2.0
 return 0;
}
```

```
#include <stdio.h>
int main(void) {
 int a = 5, b = 2;
 float c;
 c = (float) a / b;  // division real
 printf("%f", c);    // imprime 2.5
 return 0;
}
```

4.3.5.3. Conversiones de tipos con y sin pérdida de información

Podemos encontrar 3 tipos distintos de conversión de tipos:

- **Sin pérdida de información:** *float ← int; double ← float;*
- **Con pérdida de información:**

  unsigned char ← char; (pierde el signo)

  char ← short int; char ← int; (los 8 bits más significativos)

  int ← float; int ← double; (parte fraccionaria o mas significativa)

- **Con redondeo:** *float ← double; double ← long double;*

```
int x, y; char ch; float f;
ch = x;  // ch solo se queda con los 8 bits menos significativos de x
x = f;   // x se queda con la parte no fraccionaria de f
f = ch;  // f convierte el valor de 8 bits de ch a formato coma flotante
f = x;   // f convierte el valor de 16 bits de x a formato coma flotante
f = y / (float) x; // casting manual: el tipo de x es cambiado de int a float
```

4.4 Operaciones básicas de entrada-salida

El lenguaje C **no dispone de sentencias de E/S** para recibir/enviar datos del/al exterior → Utiliza las funciones de las librerías estándar para realizar dichas actividades → Para su utilización hay que incluir al principio del programa el archivo ***stdio.h*** (Standard Input-Output), que nos permitirá utilizar las funciones de E/S, y se hará del siguiente modo:

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

4.4.1 E/S por consola sin formato

La E/S por consola se refiere a las operaciones que se producen en el teclado (entrada estándar) y la pantalla (salida estándar) del PC.

Funciones para leer o escribir un carácter:

- *getche( ):* Espera a que se pulse una tecla. El carácter pulsado aparece en pantalla y lo devuelve para ser recogido por una variable.
  Ej: char v; v = getche();
- *getch( ):* Igual que *getche( )* pero sin mostrar el caracter en pantalla.
- *putchar(c):* Imprime el carácter c en pantalla en la posición del cursor.
  Ej: char letra = 's'; putchar(letra);
- *putch(c):* Igual que *putchar( )*, pero lo imprime en la ventana activa.

Funciones para leer o escribir una cadena de caracteres:

- *gets(cad):* Lee una cadena de caracteres desde teclado y lo guarda en cad. La función espera hasta que se pulse INTRO.
- *puts(cad):* Imprime la cadena de caracteres cad en pantalla.

Ejemplo:

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

int main(void) {
  char string[] = "Buenos Dias\n", nombre[80];
  puts(string);
  printf("Introduce tu nombre:");
  gets(nombre); /*scanf("%s",nombre); */
  printf("Tu nombre es: %s\n", nombre);
  getch(); /* espera a que se pulse una tecla */
  return 0;
}
```

4.4.2 E/S por consola con formato

El término *con formato* se refiere a que estas funciones leen y escriben datos en varios formatos bajo control del programador.

- ***printf( ):*** Imprime una cadena de datos en pantalla

  su prototipo es el siguiente:

```
int printf("cadena_de_formato", arg1, arg2, ...)
```

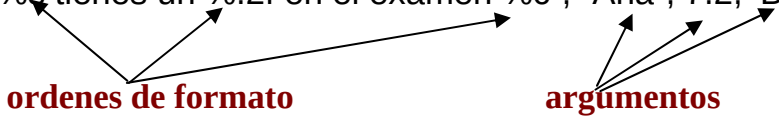
  donde la *cadena de formato* consiste en 2 tipos de elementos:

  - caracteres que se mostrarán en la pantalla.
  - órdenes de formato: definen la forma en que se muestran los argumentos posteriores. Una *orden de formato* empieza con % y va seguido por uno de los siguientes códigos de formato.

Código	Formato
%c	Un único carácter
%d	Decimal
%i	Decimal
%e	Notación científica
%f	Decimal en punto flotante
%g	Usar %e o %f, el más corto
%o	Octal
%s	Cadena de caracteres
%u	Decimales sin signo
%x	Hexadecimales
%%	Imprime un signo %
%p	Muestra un puntero
%n	El argumento asociado es un puntero a entero al que se asigna el número de caracteres escritos hasta entonces

Debe haber el mismo nº de argumentos que de órdenes de formato y deben aparecer en el mismo orden y coincidir en el tipo. Por ejemplo:

printf(“Hola %s tienes un %.2f en el examen %c”, “Ana”, 7.2, ‘B’);



Mostraría en pantalla:

Hola Ana tienes un 7.20 en el examen B

Las órdenes de formato pueden tener modificadores que indiquen la longitud de campo, el nº de decimales y el ajuste a la izquierda.

- ***scanf( ):*** Es la rutina de entrada por consola de propósito general. Puede leer todos los tipos de datos que suministra el compilador y convierte los números automáticamente al formato interno apropiado. Es el complemento de *printf( )*. Su prototipo es:

```
int scanf("%x1%x2...", &arg1, &arg2, ...);
```

Los %xi indican qué tipo de dato se va a leer a continuación:

Código	Significado
%c	Leer un único carácter
%d	Leer un entero decimal
%i	Leer un entero decimal
%e	Leer un número en coma flotante
%f	Leer un número en coma flotante
%h	Leer un entero corto
%o	Leer un número octal
%s	Leer una cadena
%x	Leer un número hexadecimal
%p	Leer un puntero
%n	Recibe un valor entero igual al número de caracteres leídos hasta entonces

Ejemplo: int edad; float nota, char nombre[20];
scanf("%i %f %s", &edad, &nota, nombre);

ordenes de formato **argumentos**

Todas los argumentos (excepto las cadenas de caracteres) deben estar precedidos del operador &(dirección).

4.4.2.1 Constantes de carácter con barra invertida

- Son necesarias para poder imprimir caracteres especiales, como un retorno de carro, comillas dobles, comillas simples, tabuladores, etc.

Código	Significado	Código	Significado
\b	Espacio atrás	\t	Tabulación horizontal
\n	Salto de línea	\r	Retorno de carro
\a	Alerta	\\	Barra invertida
\"	Comillas dobles	\'	Comilla simple
\x	Constante hexadecimal	\o	Constante octal

printf("Mis datos son: \nNombre:\nApellido1\tApellido2");

Mostraría en pantalla:

```
Mis datos son:
Nombre:
Apellido1       Apellido2
```

APENDICE: La Biblioteca estándar de C

Debido a que el C tiene muy pocas palabras clave, suple esta carencia con una serie de **funciones**, que realizan acciones muy frecuentes, agrupadas en unas **librerías** (cada **librería** contiene un determinado número de funciones), a cuyo conjunto se le llama **Biblioteca**.

Para poder utilizar una determinada **librería** hay que incluir su ***archivo de cabecera***, al principio de nuestro programa, mediante #include:

#include <nombre archivo cabecera>

Los principales archivos de cabecera estándar son:

Archivo de cabecera	Propósito
CONIO.H	Funciones de manejo de pantalla
CTYPE.H	Funciones de manejo de caracteres (ANSI C)
DIR.H	Funciones de manejo de directorio
IO.H	Rutinas de E/S de tipo UNIX
MATH.H	Funciones matemática (ANSI C)
STDIO.H	Funciones de E/S estándar
STDLIB.H	Declaraciones variadas y funciones de propósito general (ANSI C)
STRING.H	Funciones para trabajar con cadenas de caracteres (ANSI C)
TIME.H	Funciones para manipular la hora y fecha del sistema (ANSI C)

Ejemplo: Programa que calcula la edad del usuario

```
#include <stdio.h>  // printf( ), scanf( )
#include <stdlib.h> // system( )
#include <conio.h>  // getch( )
int main(void) {
 int agno, edad, actual;  // declaramos las variables
 system("cls");           // borra la pantalla
 printf("Indica en qué año naciste: ");  scanf("%i", &agno);
 printf("Indica en qué año estamos: ");  scanf("%i", &actual);
 edad = actual - agno;  // calculamos la edad
 printf("Estamos en %i y naciste en %i, tienes %i años", actual, agno, edad);
 getch( );
 return 0;
}
```