

Curso de Introducción al lenguaje C

- 1. Identificadores, palabras reservadas y caracteres especiales
- 2. Tipos de datos
- 3. Estructuras de datos
- 4. Declaración de variables
- 5. Arrays
- 6. Constantes
 - ◆ 6.1. Constantes enteras
 - ◆ 6.2. Constantes en coma flotante
 - ◆ 6.3. Constantes de caracteres
 - ◆ 6.4. Constantes de cadenas de caracteres
 - ◆ 6.5. Constantes enumeradas
- 7. Punteros
 - ◆ 7.1. Declaración de punteros
 - ◆ 7.2. Punteros a una estructura
 - ◆ 7.3. Paso de punteros a una función
 - ◆ 7.4. Punteros y arrays unidimensionales
 - ◆ 7.5. Operaciones con punteros
 - ◆ 7.6. Punteros y arrays multidimensionales
 - ◆ 7.7. Arrays de punteros
- 8. Expresiones
 - ◆ 8.1. Operaciones aritméticas
 - ◆ 8.2. Operaciones lógicas
 - ◆ 8.3. Operaciones de bits
 - ◆ 8.4. Comparaciones
 - ◆ 8.5. Asignaciones
 - ◆ 8.6. Operadores especiales
 - ◆ 8.7. Precedencia de los operadores
 - ◆ 8.8. Incremento ++ y decremento --
 - ◆ 8.9. Detalle sobre los operadores de asignación
 - ◆ 8.10. Conversión de tipos
- 9. Sentencias

9.1 Sentencia while

9.2 Sentencia for

9.3 Sentencia if

9.4 Sentencia do-while

9.5 Sentencia switch

9.6 Sentencia break

9.7 Sentencia continue

9.8 Sentencia goto

- 10. Funciones
- 11. Librerías
 - ◆ 11.1. Operaciones sobre ficheros
 - ◆ 11.2. Manejo de cadenas
 - ◆ 11.3. Algunas librerías y funciones importantes en ANSI C
 - ◆ 11.4. Funciones de entrada/salida más importantes
- 12. Características avanzadas de C

- ◆ 12.1. Operador ?:
- ◆ 12.2. Punteros a funciones
- 13. Estructura de un programa en C
- 14. Preprocesador de C
 - ◆ 14.1. #include
 - ◆ 14.2. #define y #undef
 - ◆ 14.3. #if, #ifdef, #ifndef, #else y #endif
- 15. Convenios a la hora de escribir en C
 - ◆ 15.1. Reglas de tipo general
 - ◆ 15.2. Legibilidad
 - ◆ 15.3. Reglas que atañen al diseño
 - ◆ 15.4. Declaraciones
 - ◆ 15.5. Sentencia switch
 - ◆ 15.6. Reglas de estilo
 - ◆ 15.7. Disposición de las llaves
 - ◆ 15.8. Sangrado
 - ◆ 15.9. La función main()

• Identificadores, palabras reservadas y caracteres especiales

Un identificador puede estar compuesto de cualquier combinación de letras (minúsculas y mayúsculas), dígitos y el símbolo subrayado '_'. La única restricción es que el primer carácter debe ser una letra o un subrayado.

Identificadores válidos Identificadores no válidos

x	4num (primer carácter no es letra)
y2	x (carácter ilegal)
suma_1	orden-no (carácter ilegal -)
_t	ind lis (espacio ilegal)
TABLA	

Figura 2 Ejemplos de identificadores legales e ilegales

No se limita la longitud de los identificadores. Pero algunas implementaciones sólo reconocen los 8 primeros y otras (ANSI) los 31 primeros caracteres.

Se diferencia entre mayúsculas y minúsculas.

Existe un conjunto de caracteres que tienen un significado especial en el lenguaje C. Se muestran en la figura 3.

```
! * + \ " <
# ( = | { >
% ) ~ ; } /
^ _ [ : , ?
& - ] ' . (blanco)
```

Figura 3 Caracteres especiales.

Las palabras reservadas de C, que no pueden ser definidas por el usuario son las que se listan en la Figura 4. También existen algunas otras que se han añadido en implementaciones posteriores.

```
auto    extern    sizeof
```

```

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

```

Figura 4 Palabras reservadas de C

• Tipos de Datos

C utiliza 5 palabras reservadas para definir los tipos de datos fundamentales. A diferencia de Pascal, un determinado tipo de datos puede ir cualificado por un conjunto de modificadores que estudiaremos más adelante.

Los tipos de datos fundamentales son:

```

char          short int      int
long int      unsigned char  unsigned short int
unsigned int  unsigned long int double
float         long float      void

```

Figura 5 Tipos de datos fundamentales

char Representa un carácter en código ASCII, también se puede interpretar como un entero.

short int Indica un entero de tamaño corto.

int Entero igual que integer en Pascal.

long int Entero largo.

unsigned short int Como short int pero sin signo.

unsigned int Como int pero sin signo.

unsigned long int Como long int pero sin signo.

float Flotante corto. Análogo al single de Pascal.

double Flotante largo. Análogo al double de Pascal.

void No indica ningún tipo. Es el tipo de las funciones que no devuelven nada.

Los tipos short int, long int, unsigned int y long float se pueden escribir como: short, long, unsigned y double.

Con respecto al tamaño que ocupan en memoria variables de estos tipos, todo lo que garantiza C es:

```
sizeof(char) = 1
```

```
sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(unsigned) = sizeof(int)
sizeof(float) =< sizeof(double)
```

Donde sizeof es un operador que está incorporado en C y devuelve el número de bytes que tiene un objeto.

Hay un grupo de modificadores que indican la forma que se almacena una determinada variable de un determinado tipo. Se indica antes del tipo de la variable.

static

Cuando se invoca a una función por segunda vez se pierden los valores que las variables locales de la función tenían al acabar la anterior llamada. Declarando una variable de este tipo cuando se llama por segunda vez a la subrutina la variable static (estática) contiene el mismo valor que al acabar la llamada anterior.

auto

Es lo mismo que si no se usara ningún modificador

volatile

El compilador debe asumir que la variable está relacionada con un dispositivo y que puede cambiar de valor en cualquier momento.

register

El compilador procurará almacenar la variable cualificada de este modo en un registro de la CPU.

extern

La variable se considera declarada en otro fichero. No se le asignará dirección ni espacio de memoria.

• Estructuras de datos

struct Es análoga al constructor record de Pascal. Su sintaxis es:

```
struct nombre {
    tipo miembro1;
    tipo miembro2;
    ...
} identificador1, identificador2, ...;
```

Nombre es el nombre de la estructura y los identificadores corresponden a variables del tipo de la estructura. Tanto el nombre como las variables pueden no existir. Los miembros son las componentes de la estructura. La forma de declarar variables del tipo definido en la estructura es:

[cualificador] struct nombre identificador1, identificador2, ...;

Para acceder a los campos de la estructura se usa la misma técnica que en Pascal: variable.miembro
Se admite el uso de estructuras dentro de la declaración de otra estructura ya que los miembros, en general, pueden tener cualquier tipo.

```
struct fecha {
    int mes;
    int día;
    int año;
};    /* Estructura fecha */
```

```

struct cuenta {
    int cuen_no;
    char cuen_tipo;
    float saldo;
    struct fecha ultimo_pago;
} cliente[100]; /* Estructura cuenta y declaración */

struct cuenta clientes[20];
clientes[0].cuen_no = cliente[99].ultimo_pago.anio;

```

Figura 6 Ejemplo de struct

union Es una construcción análoga al registro con variante de Pascal. Al igual que las estructuras constan de varios miembros aunque éstos comparten la misma ubicación de memoria. Se usa para hacer referencia a una misma área de memoria de varias formas. El tamaño de la unión es el tamaño del miembro mayor. La forma de declarar una unión es:

```

modificador union nombre {
    tipo miembro1;
    tipo miembro2;
    ...
} identificador1, identificador2, ...;

```

La forma de declarar variables del tipo unión es:

cualificador union nombre variable1, variable2, ...;

```

union palabra {          Se declaran 4 variables
    unsigned char l[2];   a,b,c y d
    unsigned int x;       A cada variable se puede
} a,b,c,d;               acceder como entero (x) o
                        como 2 bytes (l[2]).

```

```

a.x = 65535U;  a.x es un entero
a.l[0] = 0;    a.l[0] es un byte
a.l[1] = 255;  a.l[1] es un byte
¿Cuánto vale a.x en este punto?

```

Figura 7 Ejemplo de unión

typedef Permite definir un nuevo tipo de datos en función de los ya existentes. Es análogo a las declaraciones type de Pascal. La forma general de definir un nuevo tipo es:

```
typedef tipo nuevo-tipo;
```

nuevo-tipo es el nombre del tipo que se va a definir y tipo puede ser todo lo complicado que se quiera siempre que esté definido con los tipos ya existentes.

```

typedef struct {
    int mes;
    int día;
    int anio;
} fecha; /* Declaración de un nuevo tipo llamado fecha */

```

```

fecha hoy; /* Declaración de una variable de tipo fecha */
typedef unsigned char byte; /* Tipo byte de Pascal */
typedef unsigned int word; /* Tipo word de Pascal */

```

Figura 8 Ejemplos de typedef

enum Es análogo a un tipo enumerado de Pascal. Sirve para especificar los miembros de un determinado tipo.

```

enum nombre { miembro0 = valor0 , miembro1 = valor1 , ... }
variable1, variable2, ...

```

Las expresiones = valorX son opcionales, encargándose el compilador de asignar un valor constante a cada miembro que no la posea.

```

enum colores {negro = -1, azul, cian, magenta, rojo = 2, blanco} fondo, borde;
enum colores primer_plano;
fondo = cian;

```

Figura 9 Ejemplo de enum

• Declaración de variables

La forma general de declarar variables en C es la siguiente:

cualificador tipo identificador = valor, identificador = valor, ... ;

Las expresiones = valor sirven para inicializar la variable y pueden ser opcionales.

Las variables pueden ser declaradas en dos puntos: dentro de un bloque antes de la primera línea ejecutable; en este caso el ámbito de la variable es el cuerpo del bloque y fuera de todos los procedimientos, en este caso, el ámbito abarca a todas las funciones, es decir son declaraciones globales. El cuerpo de una función es considerado como un bloque.

```

int a,b,c;           Tres variables enteras.
float raiz1, raiz2;   Dos variables de tipo real.
char caracter, texto[80];  Un caracter y una cadena de 80.
short int a;         Entero corto.
long int b;          Entero largo.
unsigned short int d; Entero corto sin signo.
unsigned char a;      Caracter sin signo.
signed char b;        Caracter con signo.
char texto[3] = "abc"; Declaración e inicialización
char a = '\n';        Inicialización con Return.
char texto[] = "abc"; Sin especificar tamaño.
extern unsigned short int  Variable externa.

```

Figura 10 Ejemplos de declaración de variables.

• Arrays

En C los arrays son de tamaño fijo, es decir, su tamaño se especifica en tiempo de compilación.

No se comprueba la longitud del array a la hora de acceder. Es responsabilidad del programador controlar los

accesos fuera de rango.

Si se pretende escribir en la posición 20 de un array y este ha sido declarado de 10 componentes, no se generará ningún error (ni siquiera en tiempo de ejecución) y se escribirá en memoria en la dirección donde debería ir el elemento 20 si lo hubiera, por lo que se puede variar el contenido de otra variable, o incluso de parte del programa, o peor aún, se puede destruir parte del sistema operativo.

La forma de declarar arrays en C es la siguiente:

cualificador tipo variable[expresión][expresión]... = {valor, ...};

La expresión y los valores deben ser constantes. Si se declara la parte de valores se puede eliminar la expresión. En este caso el tamaño del vector es el mismo que el número de valores que se especifican. No se admite el cualificador register.

La forma de hacer referencia a un elemento del vector es:

nombre[expresión][expresión]...

Las cadenas son consideradas como vectores de caracteres. Acaban con el carácter nulo '\0'. Pueden tener una longitud arbitraria y el primer byte corresponde al primer carácter. No se almacena la longitud de la cadena como parte de la misma (como ocurre en Turbo Pascal).

• Constantes

6.1 Constantes enteras

Se pueden escribir en decimal (número que no empieza por 0), octal (el primer dígito es 0) y hexa (comienza con 0x ó 0X). Además pueden ser con signo o sin signo. Si es sin signo se les añade el sufijo U. El tamaño de las constantes puede ser normal o largo, en este caso se añade el sufijo L.

	tamaño normal	tamaño largo
con signo	L	
sin signo	U	UL (en este orden)

6.2 Constantes en coma flotante

Debe aparecer el punto decimal o la letra E (o e) de exponente. Si se desea especificar que se represente la constante en simple precisión se añade el sufijo F y en larga precisión el sufijo L o sin ninguno.

6.3 Constantes de caracteres

Es un sólo carácter o una secuencia de escape encerrado con comillas simples.

Las secuencias de escape son las que se presentan en la Figura 11:

sonido (campana)	'\a'
backspace	'\b'
tab horizontal	'\t'
tab vertical	'\v'
nueva línea	'\n'
form feed	'\f'
retorno de carro	'\r'
comillas("")	'\"'
comilla simple(')	'\''
signo interrogación	'\?'

backslash (\)	'\\'
nulo	'\0'

Figura 11 Secuencias de escape

Mediante secuencias de escape se puede expresar cualquier carácter ASCII indicando su código en octal (\ooo) o en hexa (\xhh). Donde los símbolos 'o' representan dígitos octales y las 'h' dígitos hexadecimales.

6.4. Constantes de cadenas de caracteres

Constan de cualquier número de caracteres o secuencias de escape consecutivos entre comillas dobles. También se admite ausencia de caracteres (""), El carácter nulo no puede aparecer en medio de una constante de cadena de caracteres puesto que se usa para indicar fin de cadena. El carácter de fin de cadena '\0' se coloca de forma automática en las cadenas literales; así en la cadena "abc" hay 4 caracteres: a, b, c y '\0'.

6.5 Constantes enumeradas

Son las definidas de tipo enum

0	Entero
9999	Entero
0x	Entero (hexa)
0x1	Entero (hexa)
0X7FFF	Entero (hexa)
0xabcd	Entero (hexa)
05270	Entero (octal)
50000U	Entero (sin signo)
123456789L	Entero (largo)
123456789UL	Entero (largo y sin signo)
01234L	Entero (octal y largo)
077777UL	Entero (octal, largo y sin signo)
0x456L	Entero (hexa y largo)
0XFFFFUL	Entero (hexa, sin signo y largo)
0.	Flotante (double)
0.2	Flotante (double)
.8	Flotante (double)
2E-2	Flotante (double)
2e3	Flotante (double)
2.2e+5	Flotante (double)
0.12L	Flotante representado con larga precisión
.12e6F	Flotante representado con simple precisión
'A'	Caracter A
'\$'	Caracter dolar
'\n'	Caracter retorno de carro
'\x20'	Caracter espacio
"verde"	Cadena
"verde\n"	Cadena que termina con un carry return
"\"RETURN\""	Cadena que contiene RETURN entre comillas terminado en retorno de carro.

Figura 12 Ejemplos de constantes

• Punteros

Los punteros son una de las características más útiles y a la vez más peligrosas de que dispone el lenguaje C. En C se permite declarar una variable que contiene la dirección de otra variable, o sea, un puntero. Cuando se declara un puntero éste contiene una dirección arbitraria, si leemos a dónde apunta nos dará un valor indefinido y si se escribe en tal dirección estamos variando el contenido de una posición de memoria que no conocemos por lo que podemos hacer que el sistema tenga comportamientos no deseados.

Antes de hacer uso de un puntero debemos asignarle una dirección de memoria en nuestro espacio de trabajo.

7.1 Declaración de punteros

La forma de declarar un puntero es la siguiente:

cualificador tipo *nombre, *nombre;

El tipo indica al tipo de datos a los que apuntará el puntero, pero como efecto de la declaración se reservará espacio en memoria para guardar un puntero, no para el tipo de datos al que apunta.

Existe un carácter especial que se usa como prefijo y aplicado a las variables indica la dirección de memoria que ocupa la variable, no el contenido (valor). Este símbolo es &. Además existe otro prefijo, *, que aplicado a una variable de tipo puntero indica el contenido de la dirección a la que apunta dicho puntero. A estos dos símbolos se les llama dirección e indirección respectivamente.

Hay 3 formas de inicializar un puntero:

a) Inicializarlo con el valor NULL (definido en un fichero header). De este modo estamos indicando que el puntero no apunta a ninguna memoria concreta.

b) Inicializarlo haciendo que tome como valor la dirección de una variable.

```
int *p, a;
```

```
p = &a;
```

A partir de estas sentencias, *p y a son alias.

c) Asignarle memoria dinámica a través de una función de asignación de memoria. Las funciones más habituales son calloc y malloc, definidas en el fichero alloc.h o bien en stdlib.h

```
void *malloc(size_t size)
```

```
void *calloc(size_t n_items, size)
```

Una inicialización de un puntero a un tipo T tendría la forma:

```
p = (T*)malloc(sizeof(T));
```

7.2 Punteros a una estructura

Existe una pequeña variación a la hora de acceder a una estructura mediante un puntero, por ejemplo, (*p).miembro se puede escribir como p->miembro

7.3. Paso de punteros a una función

El lenguaje C sólo admite paso de parámetros por valor, pero tiene una forma de simular un paso por referencia (variable), pasando un puntero que es la dirección donde están los datos (p. ej. &v). En realidad se pasa un valor que es una dirección de una variable.

```
void func(int *pa,int b) {
```

```

    *pa = 1;
    b = 2;
    return ;
}

main(void) {
    int a, b;
    a = b = 0;
    func(&a, b);
    /* En este punto ¿cuánto valen a y b? */
    return;
}

```

Figura 13 Ejemplo de paso de punteros a una función

7.4 Punteros y arrays unidimensionales

El identificador de un array se considera un puntero al primer elemento del array. Cualquier forma de acceder como un array puede ser sustituida por su forma equivalente como puntero.

```

int x[100];    Declaración de un array de 100 enteros
x[0]  *x      Primer elemento del array
x[2]  *(x + 2) Tercer elemento del array
x  x        Dirección del array
&x[3] (x + 3) Dirección del tercer elemento del array

```

```
char a[] = "Pulsa Return";
```

```

a[0] *a      Carácter "P"
a[i] *(a + i) Carácter i-ésimo
&a[0] a      Dirección de la cadena

```

Figura 14 Ejemplos de punteros y arrays unidimensionales

Las strings (cadenas de caracteres) se consideran arrays de caracteres a todos los efectos.

7.5 Operaciones con punteros

A los punteros se les puede añadir o restar una cierta cantidad entera. Admiten comparaciones e incrementos y decrementos. Cuando un puntero es incrementado en uno pasa a apuntar al siguiente elemento del array a que apuntaba, no al siguiente byte, es decir, se incrementa en el número de bytes que ocupa el tipo al que apunta. También se permite restar dos punteros para calcular la distancia entre ellos.

```

int *px, *py;
px < py
px <= py
px > py
px >= py
px == py
px != py
px == NULL
a = *(px++)    Postincremento del puntero

```

a = *(++px) Preincremento del puntero
px - py Distancia entre los punteros px y py

Figura 15 Operaciones con punteros

7.6 Punteros y arrays multidimensionales

Una matriz bidimensional es implementada en C como un vector cuyos elementos son vectores. Es decir, la matriz se implementa en forma de vector, sin embargo, cuando accedemos a ella sigue teniendo la forma de matriz.

La forma de declarar un array multidimensional es:

tipo nombre[expresion1][expresion2]...;

La forma de acceder a un elemento de la matriz es:

nombre[expresion1][expresion2]...

También se puede hacer referencia a los elementos de una matriz usando la técnica de punteros pero el texto se hace demasiado críptico.

7.7 Arrays de punteros

Al igual que de cualquier otro tipo de dato se permite declarar un array de punteros. La forma de hacerlo es:

tipo *nombre[expresion1][expresion2];

• Expresiones

El C permite operar dos expresiones de cualquier tipo con cualquier operador. Es responsabilidad del programador conocer la interpretación que hará el compilador.

En operaciones lógicas y comparaciones el valor falso se representa con un cero y el verdadero es cualquier otro número, siendo 1 el que devuelven dichas operaciones.

8.1. Operaciones aritméticas

+	Suma
-	Resta
*	Multiplicación
/	División
++	Incremento
--	Decremento
%	Módulo

8.2 Operaciones lógicas

	OR lógico
&&	AND lógico
!	NOT lógico
^	XOR lógico

8.3 Operaciones de bits

| OR
 & AND
 ~ NOT
 >> Desplazamiento a la derecha
 << Desplazamiento a la izquierda

8.4 Comparaciones

< Mayor que
 > Menor que
 <= Menor o igual que
 >= Mayor o igual que
 == Igual que
 != Distinto que

8.5 Asignaciones

= Asignación
 += ejemplo: $a = a + 3$ es equivalente a: $a += 3$
 -= ejemplo: $a = a - 3$ es equivalente a: $a -= 3$
 *= ejemplo: $a = a * 3$ es equivalente a: $a *= 3$
 /= ejemplo: $a = a / 3$ es equivalente a: $a /= 3$
 %= ejemplo: $a = a \% 3$ es equivalente a: $a \% = 3$
 ^= ejemplo: $a = a \wedge 3$ es equivalente a: $a \wedge = 3$
 &= ejemplo: $a = a \& 3$ es equivalente a: $a \& = 3$
 |= ejemplo: $a = a | 3$ es equivalente a: $a |= 3$
 >>= ejemplo: $a = a >> 3$ es equivalente a: $a >> = 3$
 <<= ejemplo: $a = a << 3$ es equivalente a: $a << = 3$

8.6 Operadores especiales

? : Operador If de expresiones
 , Separador de expresiones
 * Valor apuntado por un puntero (indirección)
 & Dirección de una variable
 (type) Typecasting, donde type indica el tipo de datos al que se convierte el resultado de la expresión.
 sizeof type Devuelve el número de bytes necesarios para representar un determinado tipo de datos.
 . Cualificador. Separador de campos
 -> (p*).a es equivalente a: p->a

8.7 Precedencia de los operadores

() [] -> .
 ! ~ ++ -- - (type) * & sizeof (todos son unarios)
 * / %
 + -
 << >>
 < <= > >=
 == !=
 &

```

^
|
&&
||
?:
= += -= *= /= %= ^= &= |=
,

```

8.8 Incremento ++ y decremento --

Existen dos formas de incrementar y decrementar variables: postincremento y postdecremento, y preincremento y predecremento. En los primeros se calcula la expresión con el valor de la variable y luego se incrementa o decrementa dicha variable, con los segundos se incrementa o decrementa y luego se calcula la expresión.

8.9 Detalle sobre los operadores de asignación

Las asignaciones son operadores, es decir devuelven un valor. El valor que devuelven es el correspondiente al operando de la parte derecha.

```

int a, b, c;
a = b = c = 0;
a += b = c;

```

Figura 16 Ejemplos de asignaciones

Esta propiedad permite que se realicen expresiones como el condicional:

```
if ((x = f(y) == N) ...
```

Preferiremos habitualmente el código equivalente:

```

x = f(y);
if (x == N) ...

```

8.10 Conversión de tipos

Existe un tipo de conversión automática cuando tratamos con expresiones aritméticas ($x \text{ op } y$) que cumple las siguientes reglas:

1. Todos los char o short se convierten a int.
Todos los unsigned char o unsigned short se convierten a unsigned.
2. Si después del paso anterior la expresión es de tipo mixto,
el operando de tipo menor se convierte al de tipo mayor según la jerarquía:
 $\text{int} < \text{unsigned} < \text{long} < \text{unsigned long} < \text{float} < \text{double}$

Considérense las declaraciones:

```

char c; float f; int i;
short s; unsigned u;

```

<u>Expresión</u>	<u>Tipo</u>
------------------	-------------

```

c - s / i    int
u * 3 - i    unsigned
u * 3.0 - i  double
f * 3 - i    float
c + 1        int
c + 1.0      double

```

Figura 17 Conversiones implícitas

También podemos realizar conversiones de tipo explícitas llamadas *type casts* (invitación de tipos), indicando el tipo anfitrión entre paréntesis.

• Sentencias

En C hay tres tipos de sentencias: las de expresión, las compuestas y las de control de flujo.

Las sentencias de expresión son aquellas en las que se especifica una expresión para evaluar y que devuelven un valor. Acaban siempre con el símbolo ";".

Las sentencias compuestas son las que comienzan por llaves abiertas y acaban con llaves cerradas, conteniendo en su interior en primer lugar las posibles declaraciones (locales al ámbito de la sentencia) y a continuación una lista de sentencias.

Las sentencias de control de flujo son las de bucle (**for**, **while**, **switch** y **do**), la condicional (**if**) y las de salto (**goto**, **continue** y **break**).

9.1 Sentencia while

`while (expresión) sentencia`

Se repite la sentencia mientras el valor de expresión sea cierto (no 0). La condición se evalúa antes de ejecutar la sentencia.

```

/* Cálculo de la media de un vector */
int v[100], i = 0, media, suma = 0;
while (i < 100)
    suma += v[i++];
media = suma / 100;

```

Figura 18 Ejemplo de sentencia while

9.2 Sentencia for

`for (expresión1; expresión2; expresión3) sentencia`

Es la sentencia de control más potente y la más usada. Consta de tres expresiones: la primera es la inicialización del bucle, la segunda indica la condición en la que se debe continuar el bucle y la tercera es la que se encarga de incrementar los índices del bucle. Expresión1 se ejecuta una sola vez al principio del bucle. La sentencia se ejecuta mientras la expresión2 sea verdadera (no 0). Esta expresión es evaluada antes que la sentencia por lo que es posible que el bucle no se ejecute ni siquiera una vez. La expresión3 se ejecuta después de la sentencia. Las expresiones 1 y 3 pueden ser compuestas, expresiones simples separadas por comas. La instrucción `for` equivale directamente a lo siguiente.

```

expresion1;

```

```
while (expresion2) {
    sentencia;
    expresion3;
}
```

```
/* Cálculo de la media de un vector */
int v[100], i, media, suma = 0;
for (i = 0; i < 100; i++)
    suma += v[i];
media = suma / 100;
```

Figura 19 Ejemplo de bucle for

9.3 Sentencia if

```
if (expresión) sentencia1 else sentencia2
```

Igual que en Pascal la parte else es opcional. La sentencia1 se ejecuta si expresión tiene un valor verdadero (distinto de 0) y si tiene un valor falso (0) se ejecuta la sentencia2.

```
if (estado == 'S')
    tasa = 0.20 * pago;
else
    tasa = 0.14 * pago;

/* Nótese que en C se escribe ";" antes del else */
```

Figura 20 Ejemplo de sentencia if-else

9.4 Sentencia do-while

```
do sentencia while (expresión);
```

Se repite la sentencia mientras expresión sea cierta (no 0). Es análoga al repeat de Pascal. La expresión se evalúa después de ejecutar la sentencia, por lo que esta se ejecuta al menos una vez.

```
/* Cálculo de la media de un vector */
int v[100], i, media, suma = 0;
i = 0;
do
    suma += v[i++];
while (i < 100);
media = suma / 100;
```

Figura 21 do-while

9.5 Sentencia switch

```
switch (expresión) {
    case expresión1: sentencia; sentencia; ...
    case expresión2: sentencia; sentencia; ...
    case expresión3: sentencia; sentencia; ...
    default: sentencia; sentencia; ...
```

```
}
```

Es análoga a la sentencia case de Pascal. La expresión se evalúa y si su valor coincide con el valor de alguna expresión indicada en los case se ejecutan todas las acciones asociadas que le siguen. Las expresiones deben ser de tipo entero o carácter. Si el valor de expresión no se encuentra en la lista case se ejecuta/n la/s sentencia/s correspondiente/s a la opción default, si ésta no existe se continúa con la sentencia situada a continuación de switch. Una vez se elija una opción se ejecutan las sentencias asociadas y se continúan ejecutando todas las sentencia a partir de ésta (incluso las correspondientes a otras opciones) hasta que aparezca una sentencia break.

9.6 Sentencia break

break

Se usa para romper una sentencia **while**, **do-while**, **for** o **switch**. Si se ejecuta se sale del bucle más interno o de la sentencia **switch** que se esté ejecutando.

```
char color;
switch (color) {
    case 'a':
    case 'A': printf("AMARILLO\n");
        break;
    case 'r':
    case 'R': printf("ROJO\n");
    case 'b':
    case 'B': printf("BLANCO\n");
    default: printf("OTRO\n");
}
```

Figura 22 break

9.7 Sentencia continue

continue

Esta sentencia anula la pasada actual de un bucle. Se ejecuta la siguiente pasada del bucle más interno en el que se encuentre ignorándose el código que se tendría que ejecutar para acabar la pasada actual. Se puede usar en los bucles **for**, **while** y **do-while**

9.8 Sentencia goto

goto etiqueta

Es análoga a su equivalente Pascal. Le debe seguir una etiqueta a la que transferir control. Sólo se permiten saltos dentro del cuerpo de una función. La forma de definir las etiquetas es la misma que en Pascal, es decir, un identificador seguido de dos puntos. No es necesario declarar las etiquetas previamente.

• Funciones

El lenguaje C sólo permite funciones, no hay procedimientos. La forma de declarar las funciones es la siguiente:

tipo nombre(tipo parámetro_1, ... tipo parámetro_N) sentencia

Donde tipo es el tipo del dato que devuelve la función, que si no aparece se supone siempre int, y nombre es el identificador de la función.

La forma de invocar a una función es la misma que en Pascal, pero aunque la función no tenga parámetros se deben colocar los paréntesis. Por otra parte, es lícito no realizar ninguna acción con el resultado que devuelve una función, pero en ese caso debiera ponerse un type cast (void) delante de la invocación de la función:

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

Las listas de parámetros formales y de parámetros actuales no tienen por qué coincidir en número, e incluso en tipo. Si hay más parámetros formales que actuales se les asigna un valor indefinido a los formales; si ocurre al revés se descartan los parámetros actuales que sobren. Para evitar este problema se puede declarar el prototipo de la función antes de invocarla. La forma de escribir el prototipo de una función es la siguiente:

tipo nombre(tipo parámetro_1, ... tipo parámetro_N);

O sea, la cabecera de la función acabada en punto y coma. Es análoga una declaración.

Como se comentó anteriormente sólo existe paso por valor.

Para terminar la ejecución de una función se usa la sentencia return. Su sintaxis es:

```
return [expresión];
```

Donde la expresión es el valor que retorna la función. Debe ser del mismo tipo que el de la función.

```
void funcion1(int a); /* Prototipo de la función */
void funcion2(void); /* Prototipo de la función */
```

```
/* trozo de programa */
```

```
void funcion1(int a) { /* Función 1 */
    /* Cuerpo de la función */
    funcion2();
}
```

```
void funcion2(void) { /* Función 2 */
    /* Cuerpo de la función */
    funcion1(2);
}
```

Figura 23 Funciones

• Librerías

En el lenguaje C toda la entrada y salida de un programa se realiza a través de funciones definidas en librerías. También se encuentran definidas en librerías otros tipos de funciones. Dichas librerías, o la mayoría, son estándar (las funciones en ellas definidas tienen nombres estándar) lo que facilita la portabilidad de los programas. Al inicio de cada fichero se debe indicar las declaraciones de las librerías que se utilizarán. Esto se realiza utilizando la directiva #include, cuya sintaxis es:

```
#include nombre_fichero
```

donde el nombre del fichero donde se realizan las definiciones se coloca entre ángulos (<nombre_fichero>) o entre comillas dobles ("nombre_fichero") según el lugar en que haya que buscar el fichero sea en los directorios asignados por defecto a los ficheros 'include' o en el actual. Estos ficheros de definiciones suelen tener la extensión .h (de header, cabeceras) y contienen definiciones necesarias para la utilización de las funciones contenidas en las librerías. Por otra parte, utilizando la directiva #include se puede hacer inclusión de ficheros de código del mismo modo que en cualquier otro lenguaje.

```
#include <stdio.h>  Incluye la cabecera para usar las librerías  
                    de entrada/salida desde el  
                    directorio estándar
```

```
#include "stdio.h"  Igual al anterior pero las busca  
                    en el directorio actual
```

```
#include "a:\librería\mia.h"  Incluye el fichero mia.h  
                             desde la unidad a:
```

Figura 24 Ejemplo de inclusión de librerías

11.1 Operaciones sobre ficheros

Existen dos niveles para trabajar con ficheros. Un nivel bajo compatible con UNIX en donde las lecturas y escrituras en ficheros se realizan sobre cadenas de bytes y el programador debe implementar los buffers. Y un nivel alto que permite lectura-escritura de ficheros usando datos, estructuras y buffering. Las funciones del nivel bajo se encuentran fundamentalmente en el fichero io.h y las de alto nivel en el fichero stdio.h. Funciones para abrir, leer, escribir y cerrar ficheros en alto nivel (modo texto) son **fopen**, **fscanf**, **fprintf**, **fclose** mientras que sus análogos en bajo nivel (modo binario) son **open**, **read**, **write** y **close**.

11.2 Manejo de cadenas

Existe una librería que contiene todas las funciones para manejar cadenas cuyas declaraciones estan en string.h. Todas estas funciones consideran a una string como una secuencia de caracteres acabada en nulo ('\0').

11.3 Algunas librerías y funciones importantes en ANSI C

- **alloc.h** Contiene las funciones para obtener y liberar memoria.
 - ◆ void *calloc(size_t n_elem, size_t ele_size);

Reserva espacio contiguo para n_elem elementos de tamaño cada uno ele_size bytes. Devuelve un puntero al bloque o NULL si no hay suficiente espacio.

- ◆ void *malloc(size_t size);

Reserva size bytes de memoria. Devuelve un apuntador al comienzo de dicha memoria NULL.

- ◆ void free(void *block);

Libera bloques reservados previamente con calloc o malloc.

- **conio.h** Es la librería que contiene las funciones de entrada salida desde la consola.
- **ctype.h** Contiene funciones que indican características de los caracteres, por ejemplo, si está en mayúscula o en minúscula, si es un dígito hexa valido, etc.
 - ◆ `int isalnum(char c);` Verdad si c es una letra o un dígito.
 - ◆ `int isalpha(char c);` Verdad si c es una letra.
 - ◆ `int isdigit(char c);` Verdad si c es un digito.
 - ◆ `int islower(char c);` Verdad si c está en minúscula.
 - ◆ `int isupper(char c);` Verdad si c está en mayúscula.
 - ◆ `int isspace(char c);` Verdad si c es un espacio, tabulador, retorno de carro,

nueva linea, tabulador vertical o salto de página.

- **errno.h** Declara una lista de constantes de error que devuelven las funciones de entrada salida.
- **fcntl.h** Define los flags para los modos de apertura de un fichero.
- **stdlib.h** Conjunto de funciones estándar.
 - ◆ `double atof(char *s);` Convierte una cadena en un flotante double.
 - ◆ `int atoi(char *s);` Convierte una cadena a un int.
 - ◆ `long atol(char *s);` Convierte una cadena a un long.
 - ◆ `void exit(int status);` Termina el programa devolviendo status.
 - ◆ `char *ecvt(double value, int ndig, int *dec, int *sign);`

Convierte u número de flotante a cadena.

◆ `char *itoa(int value, char *string, int radix);`

Convierte de int a cadena.

◆ `char *ltoa(long value, char *string, int radix);`

Convierte de long a cadena.

◆ `char *ultoa(unsigned long value, char *string, int radix);`

Convierte de unsigned long a cadena.

- **math.h** Conjunto de funciones matemáticas.
 - ◆ `int abs(int x);` Valor absoluto de un entero.
 - ◆ `double sin(double x);` Seno.
 - ◆ `double cos(double x);` Coseno.
 - ◆ `double tan(double x);` Tangente.
 - ◆ `double atan(double x);` Arcotangente.
 - ◆ `double exp(double x);` Calcula e elevado a x.
 - ◆ `double log(double x);` Logaritmo neperiano.
 - ◆ `double sqrt(double x);` Raiz cuadrada.
 - ◆ `double pow(double x, double y);` Obtiene la potencia x elevado a y.
- **string.h** Funciones de manejo de cadenas.
 - ◆ `int strcmp(char *s1, char *s2);`

Compara s1 y s2, devolviendo <0, 0 ó >0 según sea s1<s2, s1==s2 ó s1>s2.

- ◆ `int strncmp(char *s1, char *s2, size_t maxlen);`

Igual que strcmp, pero con los maxlen primeros caracteres.

- ◆ `int stricmp(char *s1, char *s2);`

Igual que strcmp pero sin diferencial mayúsculas de minúsculas.

- ◆ `int strnicmp(char *s1, char *s2, size_t maxlen);`

Mexcla de stricmp y strncmp.

- ◆ `size_t strlen(char *s);`

Devuelve el número de caracteres en s, sin contar /0.

- ◆ `char *strchr(char *str, int c);`

Busca el primer caracter c en str y retorna un puntero a dicha c o NULL si no hay.

- ◆ `char *strrchr(char *str, int c);`

Busca el último caracter c en str y retorna un puntero a dicha c o NULL si no hay.

- ◆ `char *strpbrk(char *s1, char *s2);`

Busca dentro de s1 el primer caracter de los de s2, devolviendo un puntero a dicha posición o NULL si no hay.

- ◆ `char *strcat(char *dest, char *src);`

Añade src al final de dest.

- ◆ `char *strncat(char *dest, char *src, size_t maxlen);`

Añade a lo sumo maxlen caracteres de src a dest.

- ◆ `char *strcpy(char *dest, char *src);`

Copia la cadena src en dest, devolviendo dest.

- ◆ `char *strlwr(char *s);`

Convierte s a minúsculas.

- ◆ `char *strupr(char *s);`

Convierte la cadena a mayúsculas.

11.4 Funciones de entrada/salida más importantes

Los ficheros estandar definidos en todo sistema C son:

stdin (entrada estandar, asociado al teclado)

stdout (salida estandar, asociado a la pantalla)

stderr (error estandar, asociado a la pantalla)

stdaux (auxiliar estandar)

stdprn (impresora estandar)

- `putchar(char c)`
Función de salida de carácter. Imprime el carácter `c` por la salida estándar.
- `char getchar(void)`
Entrada estándar de carácter. Obtiene un carácter de la entrada estándar y este es el valor que devuelve la función.
- `int printf(const char *format, ...)`
Imprime por la salida estándar una secuencia de caracteres cuya estructura está definida en la string `format`. Se permite dar salida a cualquier tipo de dato predefinido. La string `format` tiene la estructura de una string normal pero admite además caracteres de conversión (%) para indicar qué tipo de dato se ha de imprimir.
La estructura de una cadena de formato es:

`%[flags][.width][.prec][F|N|h|l]type`

donde `type` es un caracter de conversión. El campo `flag` afecta al tipo de justificación; `width` a la anchura utilizada para imprimir; `.prec` a la precisión a la hora de escribir números en coma flotante. El último campo opcional afecta a la interpretación del argumento.

`%c` Caracter
`%d` Entero decimal
`%e` Flotante se representa con exponente
`%f` Flotante se representa sin exponente
`%g` Menor entre `%e` y `%f`
`%o` Entero octal, sin el cero inicial
`%u` Entero decimal sin signo
`%x` Entero representado en hexa sin `0x`
`%s` Strings (cadenas)

Figura 25 Caracteres de conversión

- `int scanf(const char *format, ...)`
Es el equivalente a `printf` para la entrada estándar de datos de forma estructurada. En la cadena de formato se pueden poner todos los caracteres de conversión (%) de la tabla anterior. Además admite `%[cadena]`, donde se acepta cualquier cadena formada por elementos pertenecientes a `cadena`, o si la cadena comienza con '^', los no pertenecientes.
- `FILE *fopen(const char *filename, const char *mode)`
Abre un fichero en alto nivel. `filename` es el nombre del fichero y `mode` indica el modo de apertura del fichero, en la Figura 26 se indican los valores posibles. Retorna un puntero al descriptor del fichero (`FILE`).

"r" Abrir un archivo existente solo para lectura
 "w" Abrir un archivo solo para escritura
 "a" Abrir un archivo para añadir. Si no existe se crea uno
 "r+" Abrir un archivo existente para lectura y escritura
 "w+" Abrir un archivo nuevo para lectura y escritura
 "a+" Abrir un archivo nuevo para leer y añadir

Figura 26 Modos de apertura de un fichero

- `int fclose(FILE *fp)`
 Cierra un fichero cuyo puntero al descriptor es `fp`.
- `int fprintf(FILE *fp, const char *format, ...)`
 Escribe en el fichero descrito por `fp` datos con el formato indicado por `format`.
- `int fscanf(FILE *fp, const char *format, ...)`
 Lee del fichero descrito por `fp`. Tanto esta llamada como la anterior son las equivalentes de `printf` y `scanf` que operan sobre ficheros.

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

```
int main(void) {
    FILE *fp;
    char a;
    int b, c;

    fp = fopen("fichero.txt", "r+");
    if (fp == NULL) {
        printf("ERROR");
        exit(-1);
    }
    fscanf(fp, "%c %d, %d", &a, &b, &c);
    fclose(fp);
    return 0;
}
```

Figura 27 Lectura de datos de un fichero

• Características avanzadas de C

12.1 Operador ?:

Es un operador ternario equivalente a `if-else`. Su sintaxis es:

(condición) ? expresión1 : expresión2

Antes del signo de interrogación debe aparecer una condición. Si la condición es verdadera se evalúa `expresión1` y el valor devuelto por el operador `?:` será su resultado. Si la condición es falsa se devuelve el resultado de `expresión2`.

```
if (a > b) /* Cálculo del máximo de dos variables */
    c = a;
```

```

else
c = b;

c = (a > b) ? a : b; /* Equivalente con el operador ?: */

```

Figura 28 Operador ?:

12.2 Punteros a funciones

Existe un tipo especial de punteros que es el puntero a una función. La forma de declararlo es:

```

tipo (*nombre)();

```

Donde tipo es el tipo que devuelve la función y nombre es el identificador del puntero a la función. Para darle una dirección a este puntero se realiza una asignación donde en la parte derecha debe haber un nombre de función. Cuando se haga uso del puntero se estará haciendo una llamada a la función.

```

int funcion1(int a) {
/* Cuerpo de la función */
}

void funcion2(void) {
int a, (*pun_fun)();

pun_fun = funcion1;
a = (*pun_fun)(2);
return ;
}

```

Figura 29 Punteros a funciones

• Estructura de un programa en C

En C existe una función principal (main) que es la primera que se ejecuta. Puede tener dos o tres parámetros. El primer elemento es el número de elementos de la línea de comandos (contando el propio nombre del programa que se ha ejecutado), el segundo es un puntero a un vector de punteros que a su vez apuntan a cadenas que contienen cada uno de los elementos de la línea de comandos y el tercero es un puntero al entorno.

```

main(int argc, char *argv[])
/* Típica declaración de la función main() */

main(int argc, char *argv[]) {
/* Cuerpo de la función main */
}

```

Figura 30 Función main()

El compilador C lee el fichero fuente una sola vez. Esto quiere decir que si queremos utilizar un identificador tenemos que declararlo antes. Esto no ocurre con las funciones que pueden ser declaradas en el orden que se desee, pero en este caso el compilador no comprueba que los tipos ni que la cantidad de los parámetros formales y actuales coincidan.

• Preprocesador de C

El preprocesador actúa antes que el compilador. El nombre del preprocesador es CPP (CPP.EXE en el caso de DOS). Es una fase previa a éste, y se encarga de realizar ciertas tareas básicas; entre estas tareas están la inclusión de ficheros, expansión de macros y proceso de directivas.

14.1. #include

Se usa para incluir un fichero

14.2. #define y #undef

#define declara una macro o una constante y #undef borra una macro de la tabla de definiciones.

14.3 #if, #ifdef, #ifndef, #else y #endif

Se puede preguntar por el valor de una constante o la existencia o no de una macro. En caso de ser cierta la condición se compila el código entre #if y #else, en caso de ser falsa se compila el código entre #else y #endif.

```
#define NULL 0 /* Definición de constante */

#ifdef DEBUG
    /* Código para depuración */
#else
    /* Código definitivo (sin depuración) */
#endif

#undef VGA /* Quita de la tabla la definición de VGA */
```

Figura 31 Algunas directivas del Preprocesador

• Convenios a la hora de escribir en C

Existen una serie de convenios que no pertenecen de por sí al lenguaje C, pero que se usan para hacer más legibles los programas, sobre todo para programadores que no han realizado la implementación.

Un buen programador debería codificar aprovechando toda oportunidad que asegure que su código es claro y fácil de comprender. No debe utilizar "trucos inteligentes" a los cuales C se presta con facilidad. Considérese a modo de ejemplo los siguientes fragmentos de código:

1º:

```
while ('\n' != *p++=*q++);
```

2º:

```
while (1) {
    *destination_ptr = *source_ptr;
    destination_ptr++;
    source_ptr++;
    if (*destination_ptr == '\n')
        break; /* abandonar el bucle si se ha acabado */
}
```

Aunque la segunda versión es más larga, es más clara y fácil de comprender. Incluso un programador sin

demasiada experiencia en C puede entender que este código tiene algo que ver con mover datos desde una fuente hacia un destino. Al compilador no le importa cuál de las dos versiones se utilice: un buen compilador generará el mismo código máquina para ambas. Es el programador quien se beneficia de la claridad del código.

15.1 Reglas de tipo general

- Comentar, comentar, comentar. Documentar convenientemente los programas es imprescindible en cualquier lenguaje.
- Utilizar siempre la regla KISS: Keep It Simple, Stupid. Claro y sencillo es siempre preferible a complejo y maravilloso.
- Evitar efectos laterales. Utilizar los operadores ++ y -- siempre en una única línea por sí mismos (no imbuídos en otras sentencias complejas).
- Nunca colocar asignaciones dentro de los condicionales. Nunca colocar una asignación dentro de otra sentencia.
- Considere la diferencia entre = y ==. Utilizar = en lugar de == es un error muy común y difícil de encontrar, especialmente entre programadores procedentes de otros lenguajes.
- Nunca haga "nada" de forma "silenciosa".

```
/* Nunca programe de este modo */  
for (index = 0; data[index] < key; index++);
```

```
/*¿Ha notado el punto y coma al final de la línea anterior?*/
```

```
/* Forma correcta para hacer lo mismo: */  
for (index = 0; data[index] < key; index++)  
; /* No hacer nada */
```

15.2 Legibilidad

- Haga todo lo posible para incrementar la legibilidad de sus programas. En concreto:
 - ◆ Coloque siempre un espacio después de las comas y puntos y comas: Es preferible

```
for (i = 1; i <= MAX; i++)
```

que

```
for(i=1;i<MAX;i++)
```

- ◆ Coloque siempre espacios a ambos lados de un operador binario:
Es preferible a + b
que a+b

15.3 Reglas que atañen al diseño

- Cuando diseñe sus programas, tenga en mente la "ley del mínimo asombro", que establece que su programa se debe comportar de modo que asombre lo menos posible al usuario.
- Haga la interface de usuario tan simple y consistente como sea posible.
- Provea al usuario final de tanta ayuda como le sea posible.
- Identifique claramente todos los mensajes de error con la palabra "error" y trate de dar al usuario

alguna idea de cómo corregir el problema.

15.4 Declaraciones

- Coloque una declaración de variable por cada línea, y comente el significado de cada variable significativa para comprender el código.
- Utilice identificadores lo suficientemente significativos como para comprender su significado y lo suficientemente cortos como para que sean fáciles de escribir.
- Nunca utilice declaraciones por defecto: si una función retorna un entero, declárela de tipo `int`. Todos los parámetros de una función deben ser declarados y comentados.
- Los identificadores de las constantes y macros se suelen escribir en mayúscula y el resto de identificadores en minúscula.

15.5 Sentencia `switch`

- Coloque siempre una opción default en las sentencias **`switch`** (incluso en caso que no haga nada).
- Todas las opciones de una sentencia **`switch`** deben finalizar con un **`break`**.

15.6. Reglas de estilo

- Los programas de un cierto tamaño se suelen dividir en varios ficheros fuente. Cada uno de los ficheros realiza una determinada parte del trabajo, pudiéndose compilar por separado para posteriormente enlazarlos juntos y crear el ejecutable.
- Un único bloque de código delimitado por `{ }` no debe extenderse más de tres pantallas. Si se da ese caso, el código debería estar dividido en funciones más pequeñas y simples.
- Cuando el código comienza a salirse de la pantalla por la parte derecha de la misma es el momento de dividirlo en módulos más pequeños y simples.

15.7 Disposición de las llaves

Las llaves izquierdas (abiertas) se suelen añadir a otras líneas de manera que no ocupen ellas solas una línea. Por ejemplo, se suelen colocar en la misma línea que un `if`, `for`, `while`, etc., cerrarla justo antes de `else` y abrirla justo después, etc.

15.8 Sangrado

El sangrado que se utiliza es el normal de cualquier lenguaje de programación, sangrando más a la derecha el contenido de los bloques (**`if`**, **`for`**, **`while`**, **`{`**, ...) con anidamientos más profundos.

15.9 La función `main()`

Normalmente la función `main` no hace nada, es decir, en ella no se codifica la solución del problema. Sólo se encarga de llamar a la funciones que implementan la solución del problema.