2.1 Elementos y tipos de datos digitales en Pascal

PASCAL: Es un lenguaje de Alto Nivel y propósito general desarrollado por el prof. suizo Niklaus WIRTH eb 1968.

Características:

- Excelente herramienta para aprender programacón
- Es un lenguaje de propósito general
- Lenguaje procedural (imperativo, orientado a órdenes)
- Lenguaje estructurado (soporta while, for y repeat. No necesita goto)
- Lenguaje recursivo
- Gran riqueza de tipos de datos predefinidos y definidos por el usuario
- Códigos ejecutables rápidos y eficientes

TURBO PASCAL: Lanzado en 1983 por BORLAND International.

Características adicionales:

- Entorno integrado de desarrollo
- Editor de texto
- Gráficos
- Gestión de archivos
- Compilación independiente
- Gestión de proyectos
- Enteros de gran presición
- Programación orientada a objetos
- Bilioteca de objetos

Estrucutra de un Programa PASCAL

```
Program    identificador ; {cabecera opcional en Turbo Pascal}
Uses       identificadores
Label      lista de etiquetas ; {sección de etiquetas}
Const
           definiciones de constantes
Type
           declaración de tipos de datos definidos por el usuario
Var
           declaración de variables
-------------------------------------------------------------------
Procedure
           definiciones de procedimientos
Function
           definiciones de funciones
-------------------------------------------------------------------
begin      {cuerpo del programa}
           sentencias
end.
```

Las cinco secciones de declaración -Label, Const, Type y Procedure y/o Function , así como la cláusula Uses y Program, no tiene que estar presentes en todos los programas. Turbo Pascal es muy flexible al momento de escribir las secciones de declaración, ya que se pueden hacer en cualquier orden (en Pascal estándar ISO si se require este orden). Sin embargo es conveniente seguir el orden establecido, le evitará futuros problemas.

Ejemplo:

```
Program MiPrimerPrograma; {cabecera}
Uses
  Crt; {declaraciones}
Const
  iva =0.10;
Type
  cadena =string[35];
  meses  =1..12;
Var
  sueldo :real;
  numero :integer;
nombre :cadena;
Nmes   :meses;
begin
  ClrScr; {Limpia la pantalla}
  Write ('Escribe tu nombre : ');
  {Visualiza información en pantalla}
  ReadLn(nombre);{Leer un dato del teclado}
  WriteLn ('Bienvenido ', nombre);
  {Visualiza información en pantalla}
  Readkey; {Espera la pulsación de una tecla}
  ClrScr
end.
```

Nota: Las declaraciones de constantes, tipos y variables también se pueden poner en los procedimientos y/o funciones.
Todo objeto referenciado en un programa debe haber sido previamente definido.

Ejemplo:

```
Program Incorrecto; {cabecera}
Const
  pi=3.141592;
Var
  Meses:array [1..Max] of string[15];
begin
  .....................................
end.
```

El programa anterior es incorrecto ya que hacemos referencia a la constante Max en la declaración de variables sin haberla definido en la declaración de constantes.

Identificadores

En la mayoría de los programas de computador, es necesario manejar datos de entrada o de salida, los cuales necesitan almacenarse en la memoria principal del computador en el tiempo de ejecución. Para poder manipular dichos datos, necesitamos tener acceso a las localidades de memoria donde se encuentran almacenados; esto se logra por medio de los nombres de los datos o IDENTIFICADORES.
Los identificadores también se utilizan para los nombres de los programas, los nombres de los procedimientos y los nombres de las funciones, así como para las etiquetas, constantes y variables.

Las reglas para formar los identificadores en Pascal son las siguientes :

1. Pueden estar compuestos de caracteres alfabéticos, numéricos y el carácter de subrayado (_).
2. Deben comenzar con un carácter alfabético o el carácter de subrayado.
3. Puede ser de cualquier longitud (sólo los 63 primeros caracteres son significativos).
4. No se hace distinción entre mayúsculas y minúsculas.
5. No se permite el uso de los IDENTIFICADORES RESERVADOS en los nombres de variables, constantes, programas o sub-programas.

Identificadores válidos

Nombre
Cadena
Edad_Maxima
X_Y_Z
Etiqueta2

Identificadores no válidos

Num&Dias : carácter & no válido
X nombre : Contiene un blanco
begin : es una palabra reservada
eje@s : carácter @ no válido

Elección de identificadores

La elección de identificadores permite una mejor lectura y comprensión de un programa. No es aconsejable utilizar identificadores que no sugieran ningún significado.

Declaración de etiquetas

En el remoto caso de que sea necesaria la utilización de la instrucción Goto, deberá marcarse con una etiqueta la línea a donde desea enviarse el control de flujo del programa.
La declaración deberá encabezarse con el identificador reservado Label, seguido por la lista de etiquetas separadas por comas y terminada por un punto y coma.
Pascal estándar sólo permite etiquetas formadas por números de 1 a 4 dígitos.
Turbo-Pascal permite la utilización de números y/o cualquier identificador, excepto los identificadores reservados.
Su uso no está recomendado y en este tutorial no se empleará *nunca.*

Definición de constantes

En la definición de constantes se introducen identificadores que sirven como sinónimos de valores fijos.
El identificador reservado Const debe encabezar la instrucción, seguido por una lista de asignaciones de constantes. Cada asignación de constante debe consistir de un identificador seguido por un signo de igual y un valor constante, como se muestra a continuación:

```
Const
  valor_maximo =255; precision
               =0.0001;
  palabra_clave='Tutankamen';
  encabezado   =' NOMBRE DIRECCION TELEFONO ';
```

Un valor constante puede consistir de un número (entero o real), o de una constante de caracteres.
La constante de caracteres consiste de una secuencia de caracteres encerrada entre apóstrofes ('), y, en Turbo-Pascal, también puede formarse concatenándola con caracteres de control (sin separadores), por ejemplo :

```
'Teclee su opción ==>'^G^G^G ;
```

Esta constante sirve para desplegar el mensaje :

```
Teclee su opción ==>
```

y a continuación suena el timbre tres veces.
Las constantes de caracteres pueden estar formadas por un solo carácter de control, p.ej. :

```
hoja_nueva = ^L
```

Existen dos notaciones para los caracteres de control en Turbo Pascal, a saber :

1. El símbolo # seguido de un número entero entre 0 y 255 representa el carácter al que corresponde dicho valor decimal en el codigo ASCII.
2. El símbolo ^ seguido por una letra, representa el correspondiente carácter de control.

Ejemplos :

#12 representa el valor decimal 12
(hoja_nueva o alimentación de forma).
#$1B representa el valor hexadecimal 1B (escape).
^G representa el carácter del timbre o campana.
^M representa el carácter de retorno de carro.

Pascal proporciona las siguientes CONSTANTES PREDEFINIDAS :

Nombre	Tipo	Valor
pi	real	3.1415926536 (Sólo en Turbo Pascal)
false	boolean	
true	boolean	
MaxInt	integer	32767

Además de las constantes literales para los tipos integer y real con representación decimal y hexadecimal, y las constantes literales para el conjunto de caracteres ASCII, más los caracteres especiales (no incluidos en el conjunto estándar del ASCII)

Definición de tipos

Además de identificadores, los datos deben tener asignado algún tipo que indique el espacio de memoria en que se almacenarán y que al mismo tiempo evita el error de tratar de guardar un dato en un espacio insuficiente de memoria .
Un tipo de dato en Pascal puede ser cualquiera de los tipos predefinidos (integer, real, byte, boolean, char), o algún otro definido por el programador en la parte de definición de tipos .
Los tipos definidos por el programador deben basarse en los tipos estándar predefinidos, para lo cual, debe iniciar con el identificador reservado Type , seguido de una o más asignaciones de tipo separadas por punto y coma. Cada asignación de tipo debe consistir de un identificador de tipo, seguido por un signo de igual y un identificador de tipo previamente definido.

La asignación de tipos a los datos tiene dos objetivos principales:

1. Detectar errores de operaciones en programas.
2. Determinar cómo ejecutar las operaciones.

Pascal se conoce como un lenguaje "*fuertemente tipificado*" (strongly-typed) o de tipos fuertes. Esto significa que todos los datos utilizados deben tener sus tipos declarados explícitamente y el lenguaje limita la mezcla de tipos en las expresiones. Pascal detecta muchos errores de programación *antes de que el programa se ejecute*.
Los tipos definidos por el programador pueden utilizarse para definir nuevos tipos, por ejemplo :

```
Type
  entero     = integer;
  otro_entero = entero;
```

A continuación se hace una breve descripción de los tipos predefinidos .

Tipos enteros

Tipos enteros predefinidos

Tipo	Rango	Formato
byte	0 .. 255	8 bits sin signo
integer	-32768 .. 32767	16 bits con signo
longint	-247483648 .. 2147483647	32 bits con signo
shortint	-128 .. 127	8 bits con signo
word	0 .. 65535	16 bits sin signo

- BYTE: El tipo byte es un subconjunto del tipo integer, en el rango de 0 a 255 . Donde quiera que se espere un valor byte, se puede colocar un valor integer; y viceversa (EXCEPTO cuando cuando son pasados como PARAMETROS). Asimismo, se pueden mezclar identificadores de tipo byte y de tipo integer en las expresiones.
- Los valores de tipo byte se guardan en UN OCTETO de memoria.
- INTEGER: El rango de los valores definidos por el tipo integer , en Turbo Pascal, se encuentra entre -32768 y 32767. Cada valor de este tipo se guarda en DOS OCTETOS de memoria.
- LONGINT (enteros largos): A partir de la versión 4.0 se han incorporado números que amplían el rango de variación de los enteros a -2,147,483,648. Este tipo de datos se denomina longint (enteros largos). Ocupan CUATRO OCTETOS de memoria. Existe una constante predefinida de tipo longint, denominada MaxLongInt, cuyo valor es 2,147,483,647.
- SHORTINT (enteros cortos): En ciertos casos, puede ser práctico disponer de valores enteros positivos y negativos cuyo alcance sea más restringido que el de los tipos enteros. Los tipos shortint pueden tomar valores entre -128 y 127. Ocupan UN OCTETO de memoria.
- WORD : Existen casos en los que se desea representar únicamente valores positivos. Este es el caso. Por ejemplo, cuando se desea acceder desde un programa hasta una dirección de memoria. En tal situación, no tiene sentido una dirección negativa. Turbo Pascal dispone del tipo word (o palabra, de palabra de memoria), cuyo intervalo posible de valores es de 0 a 65535. Ocupa DOS OCTETOS de memoria.

Tipos reales

- REAL: En el contexto de Pascal, un número real es aquel que está compuesto de una parte entera y una parte decimal, separadas por un punto. El rango de estos números está dado entre los valores 1E-38 y 1E+38 . Cada valor de este tipo se guarda en SEIS OCTETOS de memoria. Durante una operación aritmética con números reales, un valor mayor que 1E+38 (sobreflujo) causará la detención del programa y desplegará un mensaje de error ; mientras que un valor menor que 1E-38 (bajoflujo), producirá un resultado igual a cero.

Deben tomarse en cuenta las siguentes restricciones para los valores de tipo real :

1. No pueden utilizarse como subindices en las definiciones del tipo estructurado array.
2. No pueden formar subrangos.
3. No se pueden usar para definir el tipo base de un conjunto (tipo estructurado set)
4. No deben utilizarse para el control de las instrucciones for y case.
5. Las funciones pred y succ no pueden tomarlos como argumentos.

Los números reales están siempre disponibles en Turbo Pascal, pero si su sistema incluye un coprocesador matemático como 8087, 80287 u 80387, se dispone además de otros tipos de números *reales*:

- real (real)
- single (real corto)
- comp (entero ampliado)
- double (real de doble precisión)
- extended (real ampliado)

Computadoras sin coprocesador matemático (emulación por software)
datos disponibles : real, comp, double, extended y single.

Computadoras con coprocesador matemático
datos disponibles : real, comp, double, extended y single (reales IEEE)

Desde la versión 5.0 se permite utilizar los datos tipo coprocesador matemático aunque su computadora no lo tenga incorporado. La razón es que se emula dicho coprocesador. Los diferentes tipos reales se diferencian por el dominio de definición, el número de cifras significativas (precisión) y el espacio ocupado en memoria.

- Turbo Pascal 4.0 requiere obligatoriamente un chip coprocesador matemático para hacer uso de números reales de coma flotante IEEE.
- Turbo Pascal 5.0 a 7.0 emula el chip coprocesador matemático totalmente en software, permitiendo ejecutar tipos IEEE tanto si tiene como si no un chip 8087/287/387 instalado en su máquina.

Tipo	Rango	Cifras	Tamaño y bytes
real	2.910 E -39 .. 1.710 E 38	11 -12	6
single	1.510 E -45 .. 3.410 E 38	7 - 8	4
double	5.010 E -324 .. 1.710 E 308	15 - 16	8
extended	1.910 E -4932 .. 1.110 E 4932	19 - 20	10
comp	-2 E 63 +1 .. 2 E 63 - 1	19 - 20	8

- BOOLEAN : Un valor de tipo boolean puede asumir cualquiera de los valores de verdad denotados por los identificadores true y false, los cuales están definidos de tal manera que false < true . Un valor de tipo boolean ocupa UN OCTETO en la memoria.
- CHAR : Un valor de tipo char es cualquier carácter que se encuentre dentro del conjunto ASCII ampliado, el cual está formado por los 128 caracteres del ASCII más los 128 caracteres especiales que presenta, en este caso, IBM.

Los valores ordinales del código ASCII ampliado se encuentran en el rango de 0 a 255. Dichos valores pueden representarse escribiendo el carácter correspondiente encerrado entre apóstrofes.
Así podemos escribir :

```
'A' < 'a'
```

Que significa : " El valor ordinal de A es menor que el de a " o " A está antes que a "
Un valor de tipo char se guarda en UN OCTETO de memoria.

- CADENA (STRING): Un tipo string (cadena) es una secuencia de caracteres de cero o más caracteres correspondientes al código ASCII, escrito en una línea sobre el programa y encerrado entre apóstrofos. El tratamiento de cadenas es una característica muy potente de Turbo Pascal que contiene ISO Pascal estándar.

Ejemplos:

```
'Turbo Pascal','Tecnológico', #13#10
```

Nota:

- Una cadena sin nada entre los apóstrofos se llama cadena nula o cadena vacía.
- La longitud de una cadena es el número de carácteres encerrados entre los apóstrofos.

Instrucciones

Aunque un programa en Pascal puede contar con una sola instrucción (también llamada enunciado, sentencia o estatuto), normalmente incluye una cantidad considerable de ellas. Uno de los tipos de instrucciones más importantes lo forman las *instrucciones de asignación*; las cuales asignan a una variable (por medio del símbolo :=) , el resultado de la evaluación de una expresión.
La sintaxis para las instrucciones de asignación es :
identificador := expresión ;

Al símbolo := le llamaremos, en lo sucesivo : "simbolo de asignación"
Los siguientes son ejemplos de instrucciones de asignacion :

```
numero    := 100 ;
importe   := precio * cantidad ;
hipotenusa := sqrt(sqr(cateto_op)+sqr(cateto_ad ));
```

Es posible construir una *instrucción vacía* escribiendo sólamente el punto y coma de una instrucción.
Así podemos escribir :

```
valor := valor + 1;;
```

Lo que incluye las dos instrucciones :

```
valor := valor + 1;
y la instrucción vacía : ;
```

Bloques de instrucciones

En todo lugar donde sea válido utilizar una instrucción simple, es posible utilizar una *instrucción compuesta* o *bloque de instrucciones*, el cual se forma agrupando varias instrucciones simples por medio de los identificadores begin y end.
Por ejemplo:

```
begin
  suma  := 1000.0;
  incr  := 20.0;
  total := suma + incr
  {Obsérvese la ausencia de punto y coma
   al final de la instrucción}
end.
```

No es necesario escribir el punto y coma antes de end ya que el *punto y coma* se usa para separar instrucciones, no para terminarlas.
begin y end son *delimitadores de bloque*.

Procedimientos de entrada / salida

Desde el punto de vista del hardware, las instrucciones de *entrada y salida* le ayudan al programa a comunicarse con un periférico de la computadora tal como una terminal, una impresora o un disco.

Las instrucciones de *entrada estándar*, sirven para leer carácteres desde el *teclado*, y las instrucciones de *salida estándar* despliegan carácteres en la *pantalla*.

En Pascal todas las operaciones de *entrada/salida* se realizan ejecutando unidades de programa especiales denominados *procedimientos de entrada/salida* que forman parte del compilador y sus nombres son identificadores estándar:

Procedimientos de entrada	Read	ReadLn
Procedimientos de salida	Write	WriteLn

Procedimientos Read y Readln

Los procedimientos predefinidos Read y ReadLn (contracción de Read Line), constituyen la base para las instrucciones de entrada estándar.

Las instrucciones de entrada proporcionan datos durante la ejecución de un programa. Las instrucciones para llamar a los procedimientos Read y ReadLn son de la siguiente forma :

```
Read(lista_de_variables);
ReadLn(lista_de_variables);
```

donde :
lista_de_variables : es una lista de identificadores de variables separados por comas, los datos que se pueden leer son : enteros, caracteres, o cadenas. *No se puede leer un boolean o un elemento de tipo enumerado*.

Los datos estructurados , arrays, registros o conjuntos, no se pueden leer globalmente y se suele recurrir a diseñar procedimientos específicos.

La acción de la instrucción es obtener, del teclado, tantos valores de datos como elementos hay en lista_de_variables. Los datos deberán ser compatibles con los tipos de las variables correspondientes en la lista.

Cada valor entero o real en el flujo de entrada puede ser representado como una secuencia de caracteres en alguna de las formas permitidas para tales números, y puede estar inmediatamente precedido por un signo más o un signo menos. Cada valor entero o real puede ser precedido por cualquier cantidad de caracteres blancos o fines de línea, pero no deberá haber blancos o fines de línea entre el signo y el número.

La diferencia entre las instrucciones Read y ReadLn consiste en que Read permite que la siguiente instrucción continúe leyendo valores en la misma línea; mientras que, con ReadLn la siguiente lectura se hará después de que se haya tecleado el carácter de fin de línea.

Cuando se tienen datos de tipo char en una instrucción Read, los caracteres blancos y los de fin de línea son considerados en el conteo de los elementos de las cadenas de caracteres mientras no se complete el total especificado para cada una de ellas. Cada fin de línea es tratado como un carácter, pero el valor asignado a la variable será un carácter blanco.

Es aconsejable que cada cadena de caracteres se lea en una instrucción Read o ReadLn por separado, para evitar el tener que ir contando hasta completar la cantidad exacta de caracteres que forman la cadena (o de lo contrario se tendrán resultados sorpresivos y frustrantes al verificar los datos leídos).
Ejemplo:

```
Var
 nombre   :string[30] ;
 edad     :integer;
 estatura :real;
  .
  .
  .
 ReadLn(nombre);
 ReadLn(edad);
 ReadLn(estatura);
```

Procedimientos Write Y Writeln

La salida estándar se realiza en base a estos procedimientos predefinidos, y las instrucciones para invocarlos toman las siguientes formas :

```
Write(lista_de_salida);
WriteLn(lista_de_salida);
```

donde :
lista_de_salida es una lista de variables, expresiones y/o constantes, cuyos valores van a ser desplegados en la pantalla.

El procedimiento Write permite que la siguiente instrucción se realice en la misma línea , mientras que WriteLn alimenta una nueva línea, antes de finalizar.
Por ejemplo, las instrucciones :

```
Write  ('! HOLA ');
WriteLn('AMIGOS !');
producirán la salida :
! HOLA AMIGOS !
cursor en la siguiente línea.
Mientras que :
Write('TECLEE <CTRL> F PARA FINALIZAR ==> ');
desplegará :
TECLEE <CTRL> F PARA FINALIZAR ==>  CURSOR
Para producir un renglón en blanco, se debe escribir :
WriteLn ;
```

Un valor booleano desplegará cualquiera de las cadenas : TRUE o FALSE, así :

Write('20 + 30 = ', 20 + 30 ,' ES ', 20 + 30 = 50);
producirá :
20 + 30 = 50 ES TRUE
dejando el cursor en la misma línea, al final de TRUE.

Cuando un valor de salida se escribe sin una especificación de longitud de campo, se utilizará la especificación de campo por omisión.
La especificación de longitud de campo por omisión dependerá del tipo de valor de salida y de la implementación de Pascal.

Así tenemos que, para todas las implementaciones :

1. La especificación de longitud de campo por omisión para los valores de tipo char es 1
2. Si el valor a ser desplegado requiere menos del número de caracteres especificado, se imprimirán tantos caracteres blancos como sean necesarios para completar dicho número.
3. Si el valor a ser desplegado requiere un número de caracteres mayor que el especificado, se usa la mínima especificación de longitud de campo que se requiera para representar un valor de tipo *real* o *entero*, pero las *cadenas* serán truncadas ajustándolas al campo especificado y desechará los caracteres en exceso que se encuentren más a la derecha.
4. En el caso de valores de tipo *real*, se puede utilizar una especificación de longitud de fracción.

Por omisión en Turbo Pascal, los valores de tipo real se desplegarán en formato de *punto flotante* (también llamado exponencial), así :

a. Para el caso de valores positivos (R >= 0.0), el formato es:

bbn.nnnnnnnnnnEsnn

b. Para valores negativos (R < 0.0), el formato es :

b-n.nnnnnnnnnnEsnn

donde :
b = blanco
n = número (1-9)
s = signo (+ ó -)
E = letra "E" para exponente
La especificación de campo mínima aceptada es :

- para R >= 0.0 , SIETE caracteres.
- para R < 0.0 , OCHO caracteres.

Ejemplos:

Sentencias	Resultados
WriteLn ('Hola Mundo');	Hola Mundo
WriteLn('22' + '20');	2220
WriteLn(pi);	3.1415926536E+00
WriteLn(3.0);	3.0000000000E+00

Debido a que Pascal alinea (justifica) las impresiones hacia la derecha del campo correspondiente, cuando es necesario acomodar tabularmente los caracteres de cadenas que no ocupan la totalidad de columnas especificadas.

Por ejemplo, suponiendo que nombre es un identificador de tipo string[20] (cadena con formato de 20 caracteres), para que la impresión de todos los valores de nombre empiece en la misma columna, podemos utilizar la instrucción :

```
WriteLn(numero,'.- ',nombre,
' ':20-length(nombre),calif:3);
```

y, al ejecutar el programa se tendrá algo parecido a :

```
1.-  JUAN FLORES          90    (Nótese la justificación de los nombres
2.-  EGRID L. CASTRO      100   hacia la izquierda y los números hacia
3.-  RAMON RAMIREZ        99    la derecha)
```

Formatos de salida

Formato 1 : WriteLn (ítem:anchura...);

Anchura: Expresión entera (literal, constante, variable o llamada a función) que especifica la anchura total del campo en que se escribe ítem.

Formato 2 : WriteLn (ítem:anchura:dígitos ...);

Dígitos: dígitos decimales de un número real
Anchura: Total de dígitos del número real contando parte entera, punto decimal y dígitos decimales.

Ejemplos:
Valor:= 25.0776;

Sentencias	Resultados	Comentario
WriteLn (Valor);	2.5077600000E+01	
WriteLn(Valor:2);	2.5E+01	
WriteLn(Valor:2:1);	25.1	Notese el redondeo
WriteLn(Valor:2:4);	25.0776	
WriteLn(Valor:2:8);	25.07760000	
WriteLn('Tec':1);	Tec	
WriteLn('Tec':3);	Tec	
WriteLn('Tec':5);	_ _ Tec	Espacios vacios _

Secuencia

En este caso, las instrucciones se ejecutan una después de la otra sin omitir ninguna de ellas.
La sintaxis para las instrucciones ejecutadas en secuencia es :

```
.....
.....
<instrucción.1> ;
<instrucción.2> ;
.....
.....
<instrucción.N> ;
.....
.....
```

Tipos de datos

Pascal requiere que todos los tipos de datos sean formalmente definidos antes de ser utilizados, ya que tal definición será usada por el compilador para determinar cuanto espacio de memoria se reservará para las variables de cada tipo, así como para establecer los límites de los valores que pueden asignarse a cada variable, por lo que estableceremos las siguientes reglas generales para los diferentes tipos de datos que se utilizan en Pascal:

1. Cada variable debe tener un solo tipo en el bloque donde fue declarada.

2. El tipo de cada variable debe ser declarado antes de que la variable sea utilizada en una instrucción ejecutable.

3. A cada tipo de dato se le pueden aplicar ciertos operadores específicos.

Para facilitar su estudio, hemos clasificado a los tipos de datos que pueden ser definidos por el programador en SIMPLES y ESTRUCTURADOS.

Tipos simples

Hasta aquí, hemos considerado los tipos simples estándar integer, real, boolean, char y byte proporcionados por Turbo Pascal.

Estos tipos de datos pueden ser utilizados para declarar variables numéricas, de caracteres y/o lógicas. Sin embargo, es posible que ninguno de ellos satisfaga los requerimientos de un determinado problema, haciéndose necesaria la utilización de otros tipos como:

TIPO SUBRANGO: El tipo de dato más simple que se puede definir en un programa Pascal es el *tipo subrango o intervalo*. Estos tipos son útiles, sobre todo por la facilidad que ofrecen para verificar automáticamente errores.

Un tipo subrango se define de un tipo ordinal, especificando dos constantes de ese tipo, que actúan como *límite inferior* y *superior* del conjunto de datos de ese tipo. Un tipo subrango es un tipo ordinal y sus valores se ordenan de igual modo que en el tipo patrón de que se deducen.
Ejemplos:

1. 0..9 — este tipo subrango consta de los elementos 0,1,2,3,4,5,6,7,8,9

2. '0'..'9' — este subrango consta de los caracteres '0' a '9'

3. 'A'..'F' — este subrango consta de los caracteres 'A' a 'F'

Se pueden crear variables cuyos valores se restrinjan a un subrango dado. Las declaraciones de tipo subrango se sitúan entre las declaraciones de constantes y de variables.
Formato:

```
type
  Nombre = límite inferior .. límite superior
```

Ejemplos:

```
{$R+} {Directiva de compilador R}
Program Positivos;  {El siguiente programa realiza una validación para que sólo se acepten valores positivos entre 0 y 32767 por medio de un tipo subrango}
Uses Crt;
Type
  NumPositivo = 0..MaxInt;
Var
  numero : NumPositivo;
Begin
  ClrScr;
  {numero:=-1; (está instrucción provocaría un error}
  Write('Escribe un número entero positivo : ');
  ReadLn(numero);
  ReadKey
end.
```

Nota: Puesto que Turbo Pascal no siempre produce un error cuando el valor de un tipo subrango está fuera de su rango definido. Sin embargo se puede tener la posibilidad de visualizar dichos errores mediante la directiva de compilador:
{$R+} activada
{$R-} desactivada
Por default esta desactivada. *Sólo se debe usar durante la fase de depuración.*

TIPOS ENUMERADOS: **En estos tipos de datos simples, se listan los identificadores que serán asociados con cada valor a utilizar.**
Por ejemplo :

```
Type
 dias_semana    =(lunes,martes,miercoles,jueves,
                  viernes,sabado,domingo);
 colores_baraja =(espada,oro,basto,copa);
De igual forma las variables pueden ser de tipo enumerado:
Var
  dias    : dias_semana;
  baraja : colores_baraja;
```

Formato:

```
Type
 nombre = (constante1,constante2,...,constanteN)
```

Los datos de tipo colores_baraja sólo podrán tomar los valores denotados por : espada, oro, basto, copa . La posición de cada valor en la lista define su orden, por lo que para el tipo dias_semana tenemos que :
domingo > viernes da como resultado true
sabado < martes da como resultado false
jueves <> miercoles da como resultado true

Características:

- Un tipo de dato enumerado es un tipo ordinal cuyo orden se indica por la disposición de los valores en la definición.
- El número de orden de cada elemento comienza en 0 para el primer elemento.
- Las variables de tipo enumerado sólo pueden tomar valores de estos tipos.

- Los únicos operadores que pueden acompañar a los tipos ordinales son los operadores de relación y asignación.

A los valores de los tipos enumerados se les pueden aplicar las funciones estándar succ (de sucesor), pred (de predecesor) y ord (de ordinal).

En el caso del valor máximo de un tipo enumerado, succ no está definido, y, para su valor mínimo, no está definido pred.

La función estándar ord es aplicable a los argumentos que sean valores de tipo enumerado. La relación biunívoca se da entre los valores del tipo y los enteros comprendidos entre 0 y N-1, donde N es la cardinalidad del tipo enumerado.

El tipo estándar boolean es equivalente a un tipo enumerado de la forma :
boolean = (false, true);

Aplicación de las funciones Ord, Pred, Succ				
Argumento(x)	Tipos de datos	Ord(x)	Pred(x)	Succ(x)
lunes	dias_semana	0	Indefinido	martes
oro	colores_baraja	1	espada	basto
22	Integer	22	21	23
0	Integer	0	-1	1
-20	Integer	-20	-21	-19
MaxInt	Integer	MaxInt	MaxInt-1	Indefinido
false	Lógico	0	Indefinido	true

Ejemplo:

```
Program Dias_Semana;  {El siguiete programa muestra los dias de la semana por medio de tipos enumerados}
Uses Crt;
Type
  Dia_Semana = (Lunes,Martes,Miercoles,Jueves,
                Viernes,Sabado,Domingo);
Var
  dias :Dia_Semana;
  i    :byte;
Begin
ClrScr;
  dias:=lunes;
  for i:=1 to 7 do
  begin
    case dias of
      Lunes    :WriteLn('Lunes    ');
      Martes   :WriteLn('Martes   ');
      Miercoles :WriteLn('Miercoles');
      Jueves   :WriteLn('Jueves   ');
      Viernes  :WriteLn('Viernes  ');
      Sabado   :WriteLn('Sabado   ');
      Domingo  :WriteLn('Domingo  ')
    end;
    dias:=succ(dias)
  end;
ReadKey
end.
```

2.1. Elementos y tipos de datos digitales del Lenguaje C.

En estos apuntes se describe de forma abreviada la sintaxis del lenguaje C. No se trata de aprender a programar en C, sino más bien de presentar los recursos o las posibilidades que el C pone a disposición de los programadores.

Conocer un vocabulario y una gramática no equivale a saber un idioma. Conocer un idioma implica además el hábito de combinar sus elementos de forma semiautomática para producir frases que expresen lo que uno quiere decir. Conocer las palabras, las sentencias y la sintaxis del C no equivalen a saber programar, pero son condición necesaria para estar en condiciones de empezar a hacerlo, o de entender cómo funcionan programas ya hechos. El proporcionar la base necesaria para aprender a programar en C es el objetivo de estas páginas.

C++ puede ser considerado como una extensión de C. En principio, casi cualquier programa escrito en ANSI C puede ser compilado con un compilador de C++. El mismo programa, en un fichero con extensión ***.c*** puede ser convertido en un programa en C++ cambiando la extensión a ***.cpp***. C++ permite muchas más posibilidades que C, pero casi cualquier programa en C, con algunas restricciones, es aceptado por un compilador de C++.

Esquema general de un computador

Un ordenador es un sistema capaz de *almacenar* y *procesar* con gran rapidez una *gran cantidad* de información. Además, un ordenador tiene capacidad para *comunicarse* con el exterior, recibiendo datos, órdenes y programas como *entrada* (por medio del teclado, del ratón, de un disquete, etc.), y proporcionando resultados de distinto tipo como *salida* (en la pantalla, por la impresora, mediante un fichero en un disquete, etc.).

Los computadores modernos tienen también una gran capacidad de conectarse en *red* para comunicarse entre sí, intercambiando mensajes y ficheros, o compartiendo recursos tales como tiempo de CPU, impresoras, lectores de CD-ROM, escáners, etc. En la actualidad, estas redes de ordenadores tienen cobertura realmente mundial, y pasan por encima de fronteras, de continentes, e incluso de marcas y modelos de ordenador.

Los computadores que se utilizan actualmente tienen la característica común de ser sistemas *digitales*. Quiere esto decir que lo que hacen básicamente es trabajar a gran velocidad con una gran cantidad de *unos* y *ceros*. La memoria de un computador contiene millones de minúsculos interruptores electrónicos (*transistores*) que pueden estar en posición *on* u *off*. Al no tener partes mecánicas móviles, son capaces de cambiar de estado muchos millones de veces por segundo. La tecnología moderna ha permitido miniaturizar estos sistemas y producirlos en grandes cantidades por un precio verdaderamente ridículo.

Actualmente, los ordenadores están presentes en casi todas partes: cualquier automóvil y gran número de electrodomésticos incorporan uno o –probablemente– varios procesadores digitales. La diferencia principal entre estos sistemas y los computadores personales –PCs– que se utilizan en las prácticas de esta asignatura, está sobre todo en el carácter *especializado* o de *propósito general* que tienen, respectivamente, ambos tipos de ordenadores. El procesador que chequea el sistema eléctrico de un automóvil está diseñado para eso y probablemente no es capaz de hacer otra cosa; por eso no necesita de muchos elementos auxiliares. Por el contrario, un PC con una configuración

estándar puede dedicarse a multitud de tareas, desde contabilidad doméstica o profesional, procesamiento de textos, dibujo artístico y técnico, cálculos científicos, etc., hasta juegos (¡desde luego no en esta asignatura, al menos por el momento...!).

Existen cientos de miles de *aplicaciones* gratuitas o comerciales (la mayor parte muy baratas; algunas bastante caras) capaces de resolver los más variados problemas. Pero además, cuando lo que uno busca no está disponible en el mercado (o es excesivamente caro, o presenta cualquier otro tipo de dificultad), el usuario puede realizar por sí mismo los programas que necesite. Este es el objetivo de los *lenguajes de programación*, de los cuales el C es probablemente el más utilizado en la actualidad. Este es el lenguaje que será presentado a continuación.

PARTES O ELEMENTOS DE UN COMPUTADOR

Un computador en general, o un PC en particular, constan de distintas partes interconectadas entre sí y que trabajan conjunta y coordinadamente. No es éste el momento de entrar en la descripción detallada de estos elementos, aunque se van a enumerar de modo muy breve.

- Procesador o CPU (Central Processing Unit, o unidad central de proceso). Es el corazón del ordenador, que se encarga de realizar las operaciones aritméticas y lógicas, así como de coordinar el funcionamiento de todos los demás componentes.
- Memoria principal o memoria RAM (Random Access Memory). Es el componente del computador donde se guardan los datos y los programas que la CPU está utilizando. Se llama también a veces *memoria volátil*, porque su contenido se borra cuando se apaga el ordenador, o simplemente cuando se reinicializa.
- Disco duro. Es uno de los elementos esenciales del computador. El disco duro es capaz de mantener la información –datos y programas– de modo estable, también con el computador apagado. El computador no puede trabajar directamente con los datos del disco, sino que antes tiene que transferirlos a la memoria principal. De ordinario cada disco duro está fijo en un determinado computador.
- Disquetes. Tienen unas características y propiedades similares a las de los discos duros, con la diferencia de que los discos duros son mucho más rápidos y tienen mucha más capacidad. Los disquetes por su parte son muy baratos, son extraíbles y sirven para pasar información de un PC a otro con gran facilidad.
- Pantalla o monitor. Es el elemento "visual" del sistema. A través de él el computador nos pide datos y nos muestra los resultados. Puede ser gráfica o simplemente alfanumérica (las más antiguas, hoy día ya en desuso).
- Ratón. Es el dispositivo más utilizado para introducir información no alfanumérica, como por ejemplo, seleccionar una entre varias opciones en un menú o caja de diálogo. Su principal utilidad consiste en mover con facilidad el *cursor* por la pantalla.
- Teclado. Es el elemento más utilizado para introducir información alfanumérica en el ordenador. Puede también sustituir al ratón por medio de las teclas de desplazamiento (, , ,).
- Otros elementos. Los PCs modernos admiten un gran número de periféricos para entrada/salida y para almacenamiento de datos. Se pueden citar las impresoras, plotters, escáners, CD-ROMs, cintas DAT, etc.

LA MEMORIA: BITS, BYTES, PALABRAS

La memoria de un computador está constituida por un gran número de unidades elementales, llamadas ***bits***, que contienen unos ó ceros. Un *bit* aislado tiene muy escasa utilidad; un conjunto adecuado de *bits* puede almacenar casi cualquier tipo de información. Para facilitar el acceso y la programación, casi todos los ordenadores agrupan los *bits* en conjuntos de 8, que se llaman ***bytes*** u octetos. La memoria se suele medir en ***Kbytes*** (1024 bytes), ***Mbytes*** o simplemente "*megas*" (1024 Kbytes) y ***Gbytes*** o "*gigas*" (1024 Mbytes).

Como datos significativos, puede apuntarse que un PC estándar actual, preparado para ***Windows 95***, tendrá entre 32 y 64 Mbytes de RAM, y entre 2 y 4 Gbytes de disco. Un disquete de 3,5" almacena 1,4 Mbytes si es de *alta densidad* (HD). Un *Mbyte* de RAM puede costar alrededor de las 800 ptas., mientras que el precio de un *Gbyte* de disco está alrededor de las 15.000 ptas., dependiendo de la velocidad de acceso y del tipo de interface (SCSI, IDE, ...).

El que la CPU pudiera acceder por separado a cada uno de los *bytes* de la memoria resultaría antieconómico. Normalmente se accede a una unidad de memoria superior llamada ***palabra*** (***word***), constituida por varios *bytes*. En los PCs antiguos la *palabra* tenía 2 bytes (16 bits); a partir del procesador 386 la *palabra* tiene 4 bytes (32 bits). Algunos procesadores más avanzados tienen *palabras* de 8 bytes.

Hay que señalar que la memoria de un ordenador se utiliza siempre para almacenar diversos tipos de información. Quizás la distinción más importante que ha de hacerse es entre ***datos*** y ***programas***. A su vez, los *programas* pueden corresponder a ***aplicaciones*** (programas de usuario, destinados a una tarea concreta), o al propio ***sistema operativo*** del ordenador, que tiene como misión el arrancar, coordinar y cerrar las aplicaciones, así como mantener activos y accesibles todos los recursos del ordenador.

IDENTIFICADORES

Como se ha dicho, la memoria de un computador consta de un conjunto enorme de *palabras*, en el que se almacenan *datos* y *programas*. Las necesidades de memoria de cada tipo de dato no son homogéneas (por ejemplo, un carácter alfanumérico ocupa un *byte*, mientras que un número real con 16 cifras ocupa 8 *bytes*), y tampoco lo son las de los programas. Además, el uso de la memoria cambia a lo largo del tiempo dentro incluso de una misma sesión de trabajo, ya que el sistema *reserva* o *libera* memoria a medida que la va necesitando.

Cada posición de memoria puede identificarse mediante un número o una ***dirección***, y éste es el modo más básico de referirse a una determinada información. No es, sin embargo, un sistema cómodo o práctico, por la nula relación nemotécnica que una dirección de memoria suele tener con el dato contenido, y porque –como se ha dicho antes– la dirección física de un dato cambia de ejecución a ejecución, o incluso en el transcurso de una misma ejecución del programa. Lo mismo ocurre con partes concretas de un programa determinado.

Dadas las citadas dificultades para referirse a un dato por medio de su dirección en memoria, se ha hecho habitual el uso de ***identificadores***. *Un identificador es un nombre simbólico que se refiere a un dato o programa determinado*. Es muy fácil elegir identificadores cuyo nombre guarde estrecha relación con el sentido físico, matemático o real del dato que representan. Así por ejemplo, es lógico utilizar un identificador llamado **salario_bruto** para representar el coste anual de un empleado. El usuario no tiene nunca que preocuparse de direcciones físicas de memoria: el

sistema se preocupa por él por medio de una *tabla*, en la que se relaciona cada *identificador* con el *tipo de dato* que representa y la *posición de memoria* en la que está almacenado.

El C, como todos los demás lenguajes de programación, tiene sus propias reglas para elegir los ***identificadores***. Los usuarios pueden elegir con gran libertad los nombres de sus variables y programas, teniendo siempre cuidado de respetar las reglas del lenguaje y de no utilizar un conjunto de ***palabras reservadas*** (***keywords***), que son utilizadas por el propio lenguaje. Más adelante se explicarán las reglas para elegir nombres y cuáles son las palabras reservadas del lenguaje C. Baste decir por ahora que todos los *identificadores* que se utilicen han de ser ***declarados*** por el usuario, es decir, *hay que indicar explícitamente qué nombres se van a utilizar en el programa para datos y funciones, y qué tipo de dato va a representar cada uno de ellos*. Más adelante se volverá sobre estos conceptos.

Concepto de "programa"

Un ***programa*** –en sentido informático– está constituido por un conjunto de *instrucciones* que se ejecutan –ordinariamente– de modo *secuencial*, es decir, cada una a continuación de la anterior. Recientemente, con objeto de disminuir los tiempos de ejecución de programas críticos por su tamaño o complejidad, se está haciendo un gran esfuerzo en desarrollar programas *paralelos*, esto es, programas que se pueden ejecutar *simultáneamente* en varios procesadores. La programación paralela es mucho más complicada que la secuencial y no se hará referencia a ella en este curso.

Análogamente a los *datos* que maneja, las *instrucciones* que un procesador digital es capaz de entender están constituidas por conjuntos de *unos* y *ceros*. A esto se llama *lenguaje de máquina* o *binario*, y es muy difícil de manejar. Por ello, desde casi los primeros años de los ordenadores, se comenzaron a desarrollar los llamados *lenguajes de alto nivel* (tales como el **Fortran**, el **Cobol**, etc.), que están mucho más cerca del lenguaje natural. Estos lenguajes están basados en el uso de *identificadores*, tanto para los *datos* como para las componentes elementales del programa, que en algunos lenguajes se llaman *rutinas* o *procedimientos*, y que en C se denominan ***funciones***. Además, cada lenguaje dispone de una *sintaxis* o conjunto de reglas con las que se indica de modo inequívoco las operaciones que se quiere realizar.

Los *lenguajes de alto nivel* son más o menos comprensibles para el usuario, pero no para el procesador. Para que éste pueda ejecutarlos es necesario traducirlos *a su propio lenguaje de máquina*. Esta es una tarea que realiza un programa especial llamado ***compilador***, que traduce el programa a lenguaje de máquina. Esta tarea se suele descomponer en dos etapas, que se pueden realizar juntas o por separado. El programa de alto nivel se suele almacenar en uno o más ficheros llamados ***ficheros fuente***, que en casi todos los *sistemas operativos* se caracterizan por una terminación –también llamada ***extensión***– especial. Así, todos los ficheros fuente de C deben terminar por (***.c***); ejemplos de nombres de estos ficheros son *calculos.c*, *derivada.c*, etc. La primera tarea del compilador es realizar una traducción directa del programa a un lenguaje más próximo al del computador (llamado *ensamblador*), produciendo un ***fichero objeto*** con el mismo nombre que el fichero original, pero con la extensión (***.obj***). En una segunda etapa se realiza el proceso de *montaje* (*linkage*) del programa, consistente en producir un ***programa ejecutable*** en lenguaje de máquina, en el que están ya incorporados todos los otros módulos que aporta el sistema sin intervención explícita del programador (funciones de librería, recursos del sistema operativo, etc.). En un PC con sistema operativo ***Windows*** el programa ejecutable se guarda en un fichero con

extensión (***.exe***). Este fichero es cargado por el sistema operativo en la memoria RAM cuando el programa va a ser ejecutado.

Una de las ventajas más importantes de los lenguajes de alto nivel es la *portabilidad* de los ficheros fuente resultantes. Quiere esto decir que un programa desarrollado en un PC podrá ser ejecutado en un Macintosh o en una máquina UNIX, con mínimas modificaciones y una simple recompilación. El lenguaje C, originalmente desarrollado por D. Ritchie en los laboratorios Bell de la AT&T, fue posteriormente estandarizado por un comité del ANSI (American National Standard Institute) con objeto de garantizar su portabilidad entre distintos computadores, dando lugar al ***ANSI C***, que es la variante que actualmente se utiliza casi universalmente.

Concepto de "función"

CONCEPTOS GENERALES

Las aplicaciones informáticas que habitualmente se utilizan, incluso a nivel de informática personal, suelen contener decenas y aún cientos de miles de líneas de código fuente. A medida que los programas se van desarrollando y aumentan de tamaño, se convertirían rápidamente en sistemas poco manejables si no fuera por la *modularización*, que es el proceso consistente en dividir un programa muy grande en una serie de módulos mucho más pequeños y manejables. A estos módulos se les ha solido denominar de distintas formas (*subprogramas*, *subrutinas*, *procedimientos*, *funciones*, etc.) según los distintos lenguajes. El lenguaje C hace uso del concepto de ***función*** (***function***). Sea cual sea la nomenclatura, la idea es sin embargo siempre la misma: dividir un programa grande en un conjunto de subprogramas o funciones más pequeñas que son llamadas por el programa principal; éstas a su vez llaman a otras funciones más específicas y así sucesivamente.

La división de un programa en unidades más pequeñas o funciones presenta –entre otras– las ventajas siguientes:

1. ***Modularización***. Cada función tiene una misión muy concreta, de modo que nunca tiene un número de líneas excesivo y siempre se mantiene dentro de un tamaño manejable. Además, una misma función (por ejemplo, un producto de matrices, una resolución de un sistema de ecuaciones lineales, ...) puede ser llamada muchas veces en un mismo programa, e incluso puede ser reutilizada por otros programas. Cada función puede ser desarrollada y comprobada por separado.
2. ***Ahorro de memoria y tiempo de desarrollo***. En la medida en que una misma función es utilizada muchas veces, el número total de líneas de código del programa disminuye, y también lo hace la probabilidad de introducir errores en el programa.
3. ***Independencia de datos y ocultamiento de información***. Una de las fuentes más comunes de errores en los programas de computador son los *efectos colaterales* o perturbaciones que se pueden producir entre distintas partes del programa. Es muy frecuente que al hacer una modificación para añadir una funcionalidad o corregir un error, se introduzcan nuevos errores en partes del programa que antes funcionaban correctamente. Una función es capaz de mantener una gran independencia con el resto del programa, manteniendo sus propios datos y definiendo muy claramente la *interfaz* o comunicación con la función que la ha llamado y con

las funciones a las que llama, y no teniendo ninguna posibilidad de acceso a la información que no le compete.

Las funciones de C están implementadas con un particular cuidado y riqueza, constituyendo uno de los aspectos más potentes del lenguaje. Es muy importante entender bien su funcionamiento y sus posibilidades.

NOMBRE, VALOR DE RETORNO Y ARGUMENTOS DE UNA FUNCIÓN

Una función de C es una porción de código o programa que realiza una determinada tarea. Una función está asociada con un *identificador* o *nombre*, que se utiliza para referirse a ella desde el resto del programa. En toda función utilizada en C hay que distinguir entre su ***definición***, su ***declaración*** y su ***llamada***. Para explicar estos conceptos hay que introducir los conceptos de ***valor de retorno*** y de ***argumentos***.

Quizás lo mejor sea empezar por el concepto más próximo al usuario, que es el concepto de ***llamada***. Las *funciones* en C se llaman incluyendo su *nombre*, seguido de los *argumentos*, en una *sentencia* del programa principal o de otra función de rango superior. Los ***argumentos*** son datos que se envían a la función incluyéndolos entre paréntesis a continuación del nombre, separados por comas. Por ejemplo, supóngase una función llamada ***power*** que calcula ***x*** elevado a ***y***. Una forma de llamar a esta función es escribir la siguiente *sentencia* (las sentencias de C terminan con punto y coma):

```
power(x,y);
```

En este ejemplo **power** es el *nombre* de la función, y **x** e **y** son los *argumentos*, que en este caso constituyen los ***datos*** necesarios para calcular el resultado deseado. ¿Qué pasa con el ***resultado***? ¿Dónde aparece? Pues en el ejemplo anterior el resultado es el ***valor de retorno*** de la función, que está disponible pero no se utiliza. En efecto, el resultado de la llamada a **power** está disponible, pues *aparece sustituyendo al nombre de la función en el mismo lugar donde se ha hecho la llamada*; en el ejemplo anterior, el resultado aparece, pero no se hace nada con él. A este mecanismo de sustitución de la llamada por el resultado es a lo que se llama ***valor de retorno***. Otra forma de llamar a esta función utilizando el resultado podría ser la siguiente:

```
distancia = power(x+3, y)*escala;
```

En este caso el *primer argumento* (**x+3**) es elevado al *segundo argumento* `y`, el resultado de la potencia –el ***valor de retorno***– es multiplicado por **escala**, y este nuevo resultado se almacena en la posición de memoria asociada con el identificador **distancia**. Este ejemplo resulta típico de lo que es una *instrucción* o *sentencia* que incluye una ***llamada*** a una función en el lenguaje C.

Para poder ***llamar*** a una función es necesario que en algún otro lado, en el mismo o en algún otro fichero fuente, aparezca la ***definición*** de dicha función, que en el ejemplo anterior es la función **power**. *La definición de una función es ni más ni menos que el conjunto de sentencias o instrucciones necesarias para que la función pueda realizar su tarea cuando sea llamada*. En otras palabras, la definición es el código correspondiente a la función. Además del código, *la definición de la función incluye la definición del tipo del valor de retorno y de cada uno de los argumentos*. A continuación se presenta un ejemplo –incompleto– de cómo podría ser la definición de la función **power** utilizada en el ejemplo anterior.

```
double power(double base, double exponente)
```

```
{
     double resultado;

     ...
     resultado = ... ;
     return resultado;
}
```

La primera línea de la definición es particularmente importante. La primera palabra ***double*** indica el tipo del valor de retorno. Esto quiere decir que el resultado de la función será un número de punto flotante con unas 16 cifras de precisión (así es el tipo ***double***, como se verá más adelante). Después viene el nombre de la función seguido de –entre paréntesis– la definición de los argumentos y de sus tipos respectivos. En este caso hay dos argumentos, **base** y **exponente**, que son ambos de tipo ***double***. A continuación se abren las llaves que contienen el código de la función[1]. La primera sentencia ***declara*** la variable **resultado**, que es también de tipo ***double***. Después vendrían las sentencias necesarias para calcular **resultado** como **base** elevado a **exponente**. Finalmente, con la sentencia ***return*** se devuelve **resultado** al programa o función que ha llamado a **power**.

Conviene notar que las variables **base** y **exponente** han sido declaradas en la *cabecera* –primera línea– de la definición, y por tanto ya no hace falta declararlas después, como se ha hecho con **resultado**. Cuando la función es llamada, las variables **base** y **exponente** reciben sendas copias de los valores del primer y segundo argumento que siguen al nombre de la función en la llamada.

Una función debe ser también ***declarada*** antes de ser llamada. Además de la ***llamada*** y la ***definición***, está también la ***declaración*** de la función. Ya se verá más adelante dónde se puede realizar esta declaración. La declaración de una función se puede realizar por medio de la primera línea de la definición, de la que pueden suprimirse los nombres de los argumentos, pero no sus tipos; al final debe incluirse el punto y coma (;). Por ejemplo, la función **power** se puede declarar en otra función que la va a llamar incluyendo la línea siguiente:

```
double power(double, double);
```

La *declaración* de una función permite que el compilador chequee el número y tipo de los argumentos, así como el tipo del valor de retorno. La declaración de la función se conoce también con el nombre de *prototipo* de la función.

LA FUNCIÓN MAIN( )

Todo programa C, desde el más pequeño hasta el más complejo, tiene un *programa principal* que es con el que se comienza la ejecución del programa. Este programa principal es también una función, pero una función que está por encima de todas las demás. Esta función se llama **main()** y tiene la forma siguiente (la palabra *void* es opcional en este caso):

```
void main(void)
{
     sentencia_1
     sentencia_2
     ...
}
```

---

[1] A una porción de código encerrada entre llaves {...} se le llama sentencia compuesta o *bloque*.

Las *llaves* {...} constituyen el modo utilizado por el lenguaje C para agrupar varias sentencias de modo que se comporten como una sentencia única (*sentencia compuesta* o *bloque*). Todo el cuerpo de la función debe ir comprendido entre las llaves de apertura y cierre.

Tokens

Existen seis clases de *componentes sintácticos* o *tokens* en el vocabulario del lenguaje C: ***palabras clave***, ***identificadores***, ***constantes***, ***cadenas de caracteres***, ***operadores*** y ***separadores***. Los *separadores* –uno o varios espacios en blanco, tabuladores, caracteres de nueva línea (denominados "espacios en blanco" en conjunto), y también los *comentarios* escritos por el programador– se emplean para separar los demás *tokens*; por lo demás son ignorados por el compilador. El compilador descompone el texto fuente o programa en cada uno de sus ***tokens***, y a partir de esta descomposición genera el código objeto correspondiente. El compilador ignora también los sangrados al comienzo de las líneas.

PALABRAS CLAVE DEL C

En C, como en cualquier otro lenguaje, existen una serie de palabras clave (*keywords*) que el usuario no puede utilizar como identificadores (nombres de variables y/o de funciones). Estas palabras sirven para indicar al computador que realice una tarea muy determinada (desde evaluar una comparación, hasta definir el tipo de una variable) y tienen un especial significado para el compilador. El C es un lenguaje muy conciso, con muchas menos palabras clave que otros lenguajes. A continuación se presenta la lista de las 32 palabras clave del ANSI C, para las que más adelante se dará detalle de su significado (algunos compiladores añaden otras palabras clave, propias de cada uno de ellos. Es importante evitarlas como identificadores):

```
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
```

IDENTIFICADORES

Ya se ha explicado lo que es un ***identificador***: un nombre con el que se hace referencia a una función o al contenido de una zona de la memoria (variable). Cada lenguaje tiene sus propias reglas respecto a las posibilidades de elección de nombres para las funciones y variables. En ANSI C estas reglas son las siguientes:

1. Un ***identificador*** se forma con una secuencia de ***letras*** (minúsculas de la ***a*** a la ***z***; mayúsculas de la ***A*** a la ***Z***; y ***dígitos*** del ***0*** al ***9***).
2. El carácter ***subrayado*** o ***underscore*** (_) se considera como una letra más.
3. Un identificador no puede contener espacios en blanco, ni otros caracteres distintos de los citados, como por ejemplo (`*,;.:-+,` etc.).

4. El primer carácter de un identificador debe ser siempre una letra o un (_), es decir, no puede ser un dígito.
5. Se hace distinción entre letras mayúsculas y minúsculas. Así, **Masa** es considerado como un identificador distinto de **masa** y de **MASA**.
6. ANSI C permite definir identificadores de hasta 31 caracteres de longitud.

   Ejemplos de identificadores válidos son los siguientes:

   ```
   tiempo, distancia1, caso_A, PI, velocidad_de_la_luz
   ```

   Por el contrario, los siguientes nombres no son válidos (¿Por qué?)

   ```
   1_valor, tiempo-total, dolares$, %final
   ```

En general es muy aconsejable ***elegir los nombres*** de las funciones y las variables de forma que permitan conocer a simple vista qué tipo de variable o función representan, utilizando para ello tantos caracteres como sean necesarios. Esto simplifica enormemente la tarea de programación y –sobre todo– de corrección y mantenimiento de los programas. Es cierto que los nombres largos son más laboriosos de teclear, pero en general resulta rentable tomarse esa pequeña molestia.

CONSTANTES

Las variables pueden cambiar de valor a lo largo de la ejecución de un programa, o bien en ejecuciones distintas de un mismo programa. Además de variables, un programa utiliza también constantes, es decir, valores que siempre son los mismos. Un ejemplo típico es el número , que vale 3.141592654. Este valor, con más o menos cifras significativas, puede aparecer muchas veces en las sentencias de un programa. En C existen distintos tipos de constantes:

1. ***Constantes numéricas***. Son valores numéricos, enteros o de punto flotante. Se permiten también constantes *octales* (números enteros en base 8) y *hexadecimales* (base 16).
2. ***Constantes carácter***. Cualquier carácter individual encerrado entre apóstrofos (tal como 'a', 'Y', ')', '+', etc.) es considerado por C como una constante carácter, o en realidad como un *número entero pequeño* (entre 0 y 255, o entre -128 y 127, según los sistemas). Existe un código, llamado ***código ASCII***, que establece una equivalencia entre cada carácter y un valor numérico correspondiente.
3. ***Cadenas de caracteres***. Un conjunto de caracteres alfanuméricos encerrados entre comillas es también un tipo de constante del lenguaje C, como por ejemplo: "espacio", "Esto es una cadena de caracteres", etc.
4. ***Constantes simbólicas***. Las constantes simbólicas tienen un nombre (identificador) y 4en esto se parecen a las variables. Sin embargo, no pueden cambiar de valor a lo largo de la ejecución del programa. En C se pueden definir mediante el preprocesador o por medio de la palabra clave ***const***. En C++ se utiliza preferentemente esta segunda forma.

Más adelante se verán con más detalle estos distintos tipos de constantes, así como las constantes de tipo ***enumeración***.

OPERADORES

Los ***operadores*** son signos especiales –a veces, conjuntos de dos caracteres– que indican determinadas operaciones a realizar con las variables y/o constantes sobre las que actúan en el programa. El lenguaje C es particularmente rico en distintos tipos de operadores: *aritméticos* (`+, -, *, /, %`), *de asignación* (`=, +=, -=, *=, /=`), *relacionales* (`==, <, >, <=, >=, !=`), *lógicos* (`&&, ||, !`) y otros. Por ejemplo, en la sentencia:

```
espacio = espacio_inicial + 0.5 * aceleracion * tiempo * tiempo;
```

aparece un operador de asignación (`=`) y dos operadores aritméticos (`+` y `*`). También los ***operadores*** serán vistos con mucho más detalle en apartados posteriores.

SEPARADORES

Como ya se ha comentado, los *separadores* están constituidos por uno o varios espacios en blanco, tabuladores, y caracteres de nueva línea. Su papel es ayudar al compilador a descomponer el programa fuente en cada uno de sus ***tokens***. Es conveniente introducir espacios en blanco incluso cuando no son estrictamente necesarios, con objeto de mejorar la legibilidad de los programas.

COMENTARIOS

El lenguaje C permite que el programador introduzca ***comentarios*** en los ficheros fuente que contienen el código de su programa. La misión de los comentarios es servir de explicación o aclaración sobre cómo está hecho el programa, de forma que pueda ser entendido por una persona diferente (o por el propio programador algún tiempo después). Los comentarios son también particularmente útiles (y peligrosos...) cuando el programa forma parte de un examen que el profesor debe corregir. El compilador[2] ignora por completo los comentarios.

Los caracteres (`/*`) se emplean para iniciar un comentario introducido entre el código del programa; el comentario termina con los caracteres (`*/`). No se puede introducir un comentario dentro de otro. Todo texto introducido entre los símbolos de comienzo (`/*`) y final (`*/`) de comentario son siempre ignorados por el compilador. Por ejemplo:

```
variable_1 = variable_2;     /* En esta línea se asigna a
                                variable_1 el valor
                                contenido en variable_2 */
```

Los comentarios pueden actuar también como *separadores* de otros tokens propios del lenguaje C. Una fuente frecuente de errores –no especialmente difíciles de detectar– al programar en C, es el olvidarse de cerrar un comentario que se ha abierto previamente.

El lenguaje ANSI C permite también otro tipo de comentarios, tomado del C++. Todo lo que va en cualquier línea del código detrás de la doble barra (//) y hasta el final de la línea, se considera como un comentario y es ignorado por el compilador. Para comentarios cortos, esta forma es más cómoda que la anterior, pues no hay que preocuparse de cerrar el comentario (el fin de línea actúa como cierre). Como contrapartida, si un comentario ocupa varias líneas hay que repetir la doble

---

[2] El ***compilador*** es el programa que traduce a lenguaje de máquina el programa escrito por el usuario.

barra (//) en cada una de las líneas. Con este segundo procedimiento de introducir comentarios, el último ejemplo podría ponerse en la forma:

```
variable_1 = variable_2;     // En esta línea se asigna a
                             //   variable_1 el valor
                             //   contenido en variable_2
```

Lenguaje C

En las páginas anteriores ya han ido apareciendo algunas características importantes del lenguaje C. En realidad, *el lenguaje C está constituido por tres elementos*: el ***compilador***, el ***preprocesador*** y la ***librería estándar***. A continuación se explica brevemente en qué consiste cada uno de estos elementos.

COMPILADOR

El compilador es el elemento más característico del lenguaje C. Como ya se ha dicho anteriormente, su misión consiste en traducir a lenguaje de máquina el programa C contenido en uno o más ficheros fuente. El compilador es capaz de detectar ciertos errores durante el proceso de compilación, enviando al usuario el correspondiente mensaje de error.

PREPROCESADOR

El preprocesador es un componente característico de C, que no existe en otros lenguajes de programación. El preprocesador actúa sobre el programa fuente, antes de que empiece la compilación propiamente dicha, para realizar ciertas operaciones. Una de estas operaciones es, por ejemplo, la sustitución de *constantes simbólicas*. Así, es posible que un programa haga uso repetidas veces del valor 3.141592654, correspondiente al número . Es posible definir una constante simbólica llamada PI que se define como 3.141592654 al comienzo del programa y se introduce luego en el código cada vez que hace falta. En realidad PI no es una variable con un determinado valor: el preprocesador chequea todo el programa antes de comenzar la compilación y sustituye el texto PI por el texto 3.141592654 cada vez que lo encuentra. Las constantes simbólicas suelen escribirse completamente con mayúsculas, para distinguirlas de las variables.

El preprocesador realiza muchas otras funciones que se irán viendo a medida que se vaya explicando el lenguaje. Lo importante es recordar que actúa siempre por delante del compilador (de ahí su nombre), facilitando su tarea y la del programador.

LIBRERÍA ESTÁNDAR

Con objeto de mantener el lenguaje lo más sencillo posible, muchas sentencias que existen en otros lenguajes, no tienen su correspondiente contrapartida en C. Por ejemplo, en C no hay sentencias para entrada y salida de datos. Es evidente, sin embargo, que ésta es una funcionalidad que hay que cubrir de alguna manera. El lenguaje C lo hace por medio de ***funciones*** preprogramadas que se venden o se entregan junto con el compilador. Estas funciones *están agrupadas en un conjunto de librerías de código objeto,* que constituyen la llamada ***librería estándar*** del lenguaje. La llamada a dichas funciones se hace como a otras funciones cualesquiera, y *deben ser declaradas* antes de ser llamadas por el programa (más adelante se verá cómo se hace esto por medio de la directiva del preprocesador **#include**).

Ficheros

El código de cualquier programa escrito en C se almacena en uno o más ficheros, en el disco del ordenador. La magnitud del programa y su estructura interna determina o aconseja sobre el número de ficheros a utilizar. Como se verá más adelante, la división de un programa en varios ficheros es una forma de controlar su manejo y su modularidad. Cuando los programas son pequeños (hasta 50 100 líneas de código), un solo fichero suele bastar. Para programas más grandes, y cuando se quiere mantener más independencia entre los distintos subprogramas, es conveniente repartir el código entre varios ficheros.

Recuérdese además que *cada vez que se introduce un cambio en el programa hay que volver a compilarlo*. La compilación se realiza a nivel de fichero, por lo que sólo los ficheros modificados deben ser compilados de nuevo. Si el programa está repartido entre varios ficheros pequeños esta operación se realiza mucho más rápidamente.

Recuérdese también que todos los ficheros que contienen código fuente en C deben terminar con la extensión (.c), como por ejemplo: ***producto.c***, ***solucion.c***, etc.

Lectura y escritura de datos

La lectura y escritura (o entrada y salida) de datos se realiza por medio de llamadas a funciones de una librería que tiene el nombre de ***stdio*** (standard input/output). Las declaraciones de las funciones de esta librería están en un fichero llamado ***stdio.h***. Se utilizan funciones diferentes para leer datos desde teclado o desde disco, y lo mismo para escribir resultados o texto en la pantalla, en la impresora, en el disco, etc.

Es importante considerar que *las funciones de entrada y salida de datos son verdaderas funciones*, con todas sus características: *nombre*, *valor de retorno* y *argumentos*.

Interfaz con el sistema operativo

Hace algún tiempo lo más habitual era que el compilador de C se llamase desde el entorno del sistema operativo ***MS-DOS***, y no desde ***Windows***. Ahora los entornos de trabajo basados en ***Windows*** se han generalizado, y el ***MS-DOS*** está en claro restroceso como entorno de de sarrollo. En cualquier caso, la forma de llamar a un compilador varía de un compilador a otro, y es necesario disponer de los manuales o al menos de cierta información relativa al compilador concreto que se esté utilizando.

De ordinario se comienza escribiendo el programa en el fichero fuente correspondiente (extensión ***.c***) por medio de un editor de texto. ***MS-DOS*** dispone de un editor estándar llamado ***edit*** que puede ser utilizado con esta finalidad. En ***Windows 3.1*** o ***Windows 95*** puede utilizarse ***Notepad***.

Existen también entornos de programación más sofisticados que se pueden utilizar desde ***Windows***, como por ejemplo el ***Visual C++*** de Microsoft, o el ***C++*** de Borland. Estos programas ofrecen muchas más posibilidades que las de un simple compilador de ANSI C. En cualquier caso, lo que hay que hacer siempre es consultar el manual correspondiente al compilador que se vaya a utilizar. Estos sistemas disponen de editores propios con ayudas suplementarias para la programación, como por ejemplo criterios de color para distinguir las ***palabras clave*** del lenguaje C.

TIPOS DE DATOS FUNDAMENTALES. VARIABLES

El C, como cualquier otro lenguaje de programación, tiene posibilidad de trabajar con datos de distinta naturaleza: texto formado por caracteres alfanuméricos, números enteros, números reales con parte entera y parte fraccionaria, etc. Además, algunos de estos tipos de datos admiten distintos números de cifras (rango y/o precisión), posibilidad de ser sólo positivos o de ser positivos y negativos, etc. En este apartado se verán los *tipos fundamentales* de datos admitidos por el C. Más adelante se verá que hay otros tipos de datos, *derivados* de los fundamentales. Los tipos de datos fundamentales del C se indican en la Tabla 2.1.

Tabla 2.1. Tipos de datos fundamentales (notación completa)

Datos enteros	char	signed char	unsigned char
	signed short int	signed int	signed long int
	unsigned short int	unsigned int	unsigned long int
Datos reales	float	double	long double

La palabra ***char*** hace referencia a que se trata de un carácter (una letra mayúscula o minúscula, un dígito, un carácter especial, ...). La palabra ***int*** indica que se trata de un número entero, mientras que ***float*** se refiere a un número real (también llamado de punto o coma flotante). Los números enteros pueden ser positivos o negativos (***signed***), o bien esencialmente no negativos (***unsigned***); los caracteres tienen un tratamiento muy similar a los enteros y admiten estos mismos cualificadores. En los datos enteros, las palabras ***short*** y ***long*** hacen referencia al número de cifras o rango de dichos números. En los datos reales las palabras ***double*** y ***long*** apuntan en esta misma dirección, aunque con un significado ligeramente diferente, como más adelante se verá.

Esta nomenclatura puede simplificarse: las palabras ***signed*** e ***int*** son las opciones por defecto para los números enteros y pueden omitirse, resultando la Tabla 2.2, que indica la nomenclatura más habitual para los tipos fundamentales del C.

Tabla 2.2. Tipos de datos fundamentales (notación abreviada).

Datos enteros	char	signed char	unsigned char
	short	int	long
	unsigned short	unsigned	unsigned long
Datos reales	float	double	long double

A continuación se va a explicar cómo puede ser y cómo se almacena en C un dato de cada tipo fundamental.

Recuérdese que *en C es necesario declarar todas las variables que se vayan a utilizar*. Una variable no declarada produce un mensaje de error en la compilación. Cuando una variable es declarada se le reserva memoria de acuerdo con el *tipo* incluido en la declaración. Es posible *inicializar* –dar un valor inicial– las variables en el momento de la declaración; ya se verá que en ciertas ocasiones el compilador da un valor inicial por defecto, mientras que en otros casos no se realiza esta inicialización y la memoria asociada con la variable correspondiente contiene *basura*

informática (combinaciones sin sentido de unos y ceros, resultado de operaciones anteriores con esa zona de la memoria, para otros fines).

Caracteres (tipo *char*)

Las variables carácter (***tipo char***) contienen un único carácter y se almacenan en un ***byte*** de memoria (8 bits). En un bit se pueden almacenar dos valores (0 y 1); con dos bits se pueden almacenar $2^2 = 4$ valores (00, 01, 10, 11 en binario; 0, 1 2, 3 en decimal). Con 8 bits se podrán almacenar $2^8 = 256$ valores diferentes (normalmente entre 0 y 255; con ciertos compiladores entre -128 y 127).

La declaración de variables tipo carácter puede tener la forma:

```
char nombre;
char nombre1, nombre2, nombre3;
```

Se puede declarar más de una variable de un tipo determinado en una sola sentencia. Se puede también inicializar la variable en la declaración. Por ejemplo, para definir la variable carácter **letra** y asignarle el valor **a**, se puede escribir:

```
char letra = 'a';
```

A partir de ese momento queda definida la variable **letra** con el valor correspondiente a la letra **a**. Recuérdese que el valor **'a'** utilizado para inicializar la variable **letra** es una constante carácter. En realidad, **letra** se guarda en un solo byte como un número entero, el correspondiente a la letra **a** en el código ASCII, que se muestra en la Tabla 2.3 para los caracteres estándar (existe un código ASCII extendido que utiliza los 256 valores y que contiene caracteres especiales y caracteres específicos de los alfabetos de diversos países, como por ejemplo las *vocales acentuadas* y la *letra ñ* para el castellano).

Tabla 2.3. Código ASCII estándar.

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	“	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

La Tabla 2.3 se utiliza de la siguiente forma. La primera cifra (las dos primeras cifras, en el caso de los números mayores o iguales que 100) del número ASCII correspondiente a un carácter

determinado figura en la primera columna de la Tabla, y la última cifra en la primera fila de la Tabla. Sabiendo la fila y la columna en la que está un determinado carácter puede componerse el número correspondiente. Por ejemplo, la letra A está en la fila 6 y la columna 5. Su número ASCII es por tanto el 65. El carácter % está en la fila 3 y la columna 7, por lo que su representación ASCII será el 37. Obsérvese que el código ASCII asocia números consecutivos con las letras mayúsculas y minúsculas ordenadas alfabéticamente. Esto simplifica notablemente ciertas operaciones de ordenación alfabética de nombres.

En la Tabla 2.3 aparecen muchos caracteres no imprimibles (todos aquellos que se expresan con 2 ó 3 letras). Por ejemplo, el **ht** es el tabulador horizontal, el **nl** es el carácter *nueva línea*, etc.

Volviendo al ejemplo de la variable **letra**, su contenido puede ser variado cuando se desee por medio de una sentencia que le asigne otro valor, por ejemplo:

```
letra = 'z';
```

También puede utilizarse una variable ***char*** para dar valor a otra variable de tipo ***char*****:**

```
caracter = letra;       // Ahora caracter es igual a 'z'
```

Como una variable tipo ***char*** es un número entero pequeño (entre 0 y 255), se puede utilizar el contenido de una variable ***char*** de la misma forma que se utiliza un entero, por lo que están permitidas operaciones como:

```
letra = letra + 1;
letra_minuscula = letra_mayuscula + ('a' - 'A');
```

En el primer ejemplo, si el contenido de **letra** era una **a**, al incrementarse en una unidad pasa a contener una **b**. El segundo ejemplo es interesante: puesto que la diferencia numérica entre las letras minúsculas y mayúsculas es siempre la misma (según el código ASCII), la segunda sentencia pasa una letra mayúscula a la correspondiente letra minúscula sumándole dicha diferencia numérica.

Recuérdese para concluir que las variables tipo ***char*** son y se almacenan como números enteros pequeños. Ya se verá más adelante que se pueden escribir como caracteres o como números según que formato de conversión se utilice en la llamada a la función de escritura.

Números enteros (tipo *int*)

De ordinario una variable tipo ***int*** se almacena en 2 bytes (16 bits), aunque algunos compiladores utilizan 4 bytes (32 bits). El ANSI C no tiene esto completamente normalizado y existen diferencias entre unos compiladores y otros. Los compiladores de Microsoft para PCs utilizan 2 bytes.

Con 16 bits se pueden almacenar 2^{16} = 65536 números enteros diferentes: de 0 al 65535 para variables sin signo, y de -32768 al 32767 para variables con signo (que pueden ser positivas y negativas), que es la opción por defecto. Este es el ***rango*** de las variables tipo ***int***.

Una variable entera (tipo ***int***) se declara, o se declara y se inicializa en la forma:

```
unsigned int numero;
int nota = 10;
```

En este caso la variable ***numero*** podrá estar entre 0 y 65535, mientras que ***nota*** deberá estar comprendida entre -32768 al 32767. Cuando a una variable ***int*** se le asigna en tiempo de ejecución

un valor que queda fuera del rango permitido (situación de ***overflow*** o valor excesivo), se produce un error en el resultado de consecuencias tanto más imprevisibles cuanto que de ordinario *el programa no avisa al usuario de dicha circunstancia*.

Cuando el ahorro de memoria es muy importante puede asegurarse que el computador utiliza 2 bytes para cada entero declarándolo en una de las formas siguientes:

```
short numero;
short int numero;
```

Como se ha dicho antes, ANSI C no obliga a que una variable ***int*** ocupe 2 bytes, pero declarándola como ***short*** o ***short int*** sí que necesitará sólo 2 bytes (al menos en los PCs).

Números enteros (tipo *long*)

Existe la posibilidad de utilizar enteros con un rango mayor si se especifica como tipo ***long*** en su declaración:

```
long int numero_grande;
```

o, ya que la palabra clave ***int*** puede omitirse en este caso,

```
long numero_grande;
```

El rango de un entero ***long*** puede variar según el computador o el compilador que se utilice, pero de ordinario se utilizan 4 bytes (32 bits) para almacenarlos, por lo que se pueden representar 2^{32} = 4.294.967.296 números enteros diferentes. Si se utilizan números con signo, podrán representarse números entre -2.147.483.648 y 2.147.483.647. También se pueden declarar enteros ***long*** que sean siempre positivos con la palabra ***unsigned***:

```
unsigned long numero_positivo_muy_grande;
```

En algunos computadores una variable ***int*** ocupa 2 bytes (coincidiendo con ***short***) y en otros 4 bytes (coincidiendo con ***long***). Lo que garantiza el ANSI C es que el rango de ***int*** no es nunca menor que el de ***short*** ni mayor que el de ***long***.

Números reales (tipo *float*)

En muchas aplicaciones hacen falta variables reales, capaces de representar magnitudes que contengan *una parte entera y una parte fraccionaria* o *decimal*. Estas variables se llaman también de ***punto flotante***. De ordinario, en base 10 y con notación científica, estas variables se representan por medio de la ***mantisa***, que es un número mayor o igual que 0.1 y menor que 1.0, y un ***exponente*** que representa la potencia de 10 por la que hay que multiplicar la mantisa para obtener el número considerado. Por ejemplo, se representa como $0.3141592654 \cdot 10^{1}$. Tanto la ***mantisa*** como el ***exponente*** pueden ser positivos y negativos.

Los computadores trabajan en base 2. Por eso un número de tipo ***float*** se almacena en 4 bytes (32 bits), utilizando *24 bits para la mantisa* (1 para el signo y 23 para el valor) y *8 bits para el exponente* (1 para el signo y 7 para el valor). Es interesante ver qué clase de números de punto flotante pueden representarse de esta forma. En este caso hay que distinguir el ***rango*** de la

precisión. La *precisión* hace referencia al número de cifras con las que se representa la *mantisa*: con 23 bits el número más grande que se puede representar es,

$$2^{23} = 8.388.608$$

lo cual quiere decir que se pueden representar todos los números decimales de 6 cifras y la mayor parte –aunque no todos– de los de 7 cifras (por ejemplo, el número 9.213.456 no se puede representar con 23 bits). Por eso se dice que las variables tipo ***float*** tienen entre 6 y 7 cifras decimales equivalentes de precisión.

Respecto al *exponente de dos* por el que hay que multiplicar la *mantisa* en base 2, con 7 bits el número más grande que se puede representar es 127. El ***rango*** vendrá definido por la potencia,

$$2^{127} = 1.7014 \cdot 10^{38}$$

lo cual indica el número más grande representable de esta forma. El número más pequeño en valor absoluto será del orden de

$$2^{-128} = 2.9385 \cdot 10^{-39}$$

Las variables tipo ***float*** se declaran de la forma:

```
float numero_real;
```

Las variables tipo ***float*** pueden ser inicializadas en el momento de la declaración, de forma análoga a las variables tipo ***int***.

Números reales (tipo *double*)

Las variables tipo ***float*** tienen un *rango* y –sobre todo– una *precisión* muy limitada, insuficiente para la mayor parte de los cálculos técnicos y científicos. Este problema se soluciona con el tipo ***double***, que utiliza 8 bytes (64 bits) para almacenar una variable. Se utilizan *53 bits para la mantisa* (1 para el signo y 52 para el valor) y *11 para el exponente* (1 para el signo y 10 para el valor). La ***precisión*** es en este caso,

$$2^{52} = 4.503.599.627.370.496$$

lo cual representa entre 15 y 16 cifras decimales equivalentes. Con respecto al ***rango***, con un exponente de 10 bits el número más grande que se puede representar será del orden de 2 elevado a 2 elevado a 10 (que es 1024):

$$2^{1024} = 1.7977 \cdot 10^{308}$$

Las variables tipo ***double*** se declaran de forma análoga a las anteriores:

```
double real_grande;
```

Por último, existe la posibilidad de declarar una variable como ***long double***, aunque el ANSI C no garantiza un *rango* y una *precisión* mayores que las de ***double***. Eso depende del compilador y del tipo de computador. Estas variables se declaran en la forma:

```
long double real_pero_que_muy_grande;
```

cuyo *rango* y *precisión* no está normalizado. Los compiladores de Microsoft para PCs utilizan 10 bytes (64 bits para la mantisa y 16 para el exponente).

Duración y visibilidad de las variables: Modos de almacenamiento.

El ***tipo*** de una variable se refiere a la naturaleza de la información que contiene (ya se han visto los tipos *char*, *int*, *long*, *float*, *double*, etc.).

El ***modo de almacenamiento*** (storage class) es otra característica de las variables de C que determina cuándo se crea una variable, cuándo deja de existir y desde dónde se puede acceder a ella, es decir, desde dónde es visible.

En C existen 4 ***modos de almacenamiento*** fundamentales: *auto*, *extern*, *static* y *register*. Seguidamente se exponen las características de cada uno de estos modos.

1. ***auto*** (automático). Es la opción por defecto para las variables que se declaran dentro de un bloque {...}, incluido el bloque que contiene el código de las funciones. *En C la declaración debe estar siempre al comienzo del bloque. En C++ la declaración puede estar en cualquier lugar* y hay autores que aconsejan ponerla justo antes del primer uso de la variable. No es necesario poner la palabra *auto*. Cada variable *auto* es creada al comenzar a ejecutarse el bloque y deja de existir cuando el bloque se termina de ejecutar. Cada vez que se ejecuta el bloque, las variables *auto* se crean y se destruyen de nuevo. Las variables *auto* son variables *locales*, es decir, sólo son ***visibles*** en el bloque en el que están definidas y en otros bloques *anidados*[3] en él, aunque pueden ser ocultadas por una nueva declaración de una nueva variable con el mismo nombre en un bloque anidado. *No son inicializadas por defecto*, y –antes de que el programa les asigne un valor– pueden contener *basura informática* (conjuntos aleatorios de unos y ceros, consecuencia de un uso anterior de esa zona de la memoria).

   A continuación se muestra un ejemplo de uso de variables de modo *auto*.

```
{
     int i=1, j=2;           // se declaran e inicializan i y j
     ...
     {
          float a=7., j=3.;  // se declara una nueva variable j
          ...
          j=j+a;             // aqui j es float
          ...                // la variable int j es invisible
          ...                // la variable i=1 es visible
     }
     ...                     // fuera del bloque, a ya no existe
     ...                     // la variable j=2 existe y es entera
}
```

2. ***extern***. Son variables globales, que se definen fuera de cualquier bloque o función, por ejemplo antes de definir la función **main()**. Estas variables *existen durante toda la ejecución del programa*. Las variables *extern son visibles por todas las funciones que están entre la definición y el fin del fichero*. Para verlas desde otras funciones definidas anteriormente o desde otros ficheros, deben ser declaradas en ellos como variables *extern*. Por defecto, *son inicializadas a cero*.

---

[3] Se llama *bloque anidado* a un bloque contenido dentro de otro bloque.

Una variable *extern* es *definida o creada* (una variable se crea en el momento en el que se le reserva memoria y se le asigna un valor) una sola vez, pero puede ser *declarada* (es decir, reconocida para poder ser utilizada) varias veces, con objeto de hacerla accesible desde diversas funciones o ficheros. También estas variables pueden ocultarse mediante la declaración de otra variable con el mismo nombre en el interior de un bloque. La variables *extern* permiten transmitir valores entre distintas funciones, pero ésta es una práctica considerada como *peligrosa*. A continuación se presenta un ejemplo de uso de variables *extern*.

```
int i=1, j, k;                        // se declaran antes de main()

main()
{
     int i=3;                       // i=1 se hace invisible int
     func1(int, int);
     ...                            // j, k son visibles
}

int func1(int i, int m)
{
     int k=3;                       // k=0 se hace invisible
     ...                            // i=1 es invisible
}
```

3. ***static***. Cuando ciertas variables son declaradas como *static* dentro de un bloque, estas variables *conservan su valor entre distintas ejecuciones de ese bloque*. Dicho de otra forma, las variables *static* se declaran dentro de un bloque como las *auto*, pero permanecen en memoria durante toda la ejecución del programa como las *extern*. Cuando se llama varias veces sucesivas a una función (o se ejecuta varias veces un bloque) que tiene declaradas variables *static*, los valores de dichas variables se conservan entre dichas llamadas. La inicialización sólo se realiza la primera vez. Por defecto, son inicializadas a cero.

   Las variables definidas como ***static extern*** son visibles sólo para las funciones y bloques comprendidos desde su definición hasta el fin del fichero. No son visibles desde otras funciones ni aunque se declaren como *extern*. Ésta es una forma de restringir la visibilidad de las variables.

   Por defecto, y por lo que respecta a su visibilidad, las ***funciones*** tienen modo *extern*. Una función puede también ser definida como *static*, y entonces sólo es visible para las funciones que están definidas después de dicha función y en el mismo fichero. Con estos modos se puede controlar la visibilidad de una función, es decir, desde qué otras funciones puede ser llamada.

4. ***register***. Este modo es una recomendación para el compilador, con objeto de que –si es posible– ciertas variables sean almacenadas en los registros de la CPU y los cálculos con ellas sean más rápidos. No existen los modos *auto* y *register* para las funciones.

Conversiones de tipo implícitas y explícitas(casting)

Más adelante se comentarán (ver Sección 4.2) las *conversiones implícitas de tipo* que tienen lugar cuando en una expresión se mezclan variables de distintos tipos. Por ejemplo, para poder sumar dos

variables hace falta que ambas sean del mismo tipo. Si una es *int* y otra *float*, la primera se convierte a *float* (es decir, la variable del tipo de menor rango se convierte al tipo de mayor rango), antes de realizar la operación.

A esta conversión automática e implícita de tipo (el programador no necesita intervenir, aunque sí conocer sus reglas), se le denomina ***promoción***, pues la variable de menor rango es *promocionada* al rango de la otra.

Así pues, cuando dos tipos diferentes de constantes y/o variables aparecen en una misma expresión relacionadas por un operador, el compilador convierte los dos operandos al mismo tipo de acuerdo con los rangos, que de mayor a menor se ordenan del siguiente modo:

long double > *double* > *float* > *unsigned long* > *long* > *unsigned int* > *int* > *char*

Otra clase de conversión implícita tiene lugar cuando el resultado de una expresión es asignado a una variable, pues dicho resultado se convierte al tipo de la variable (en este caso, ésta puede ser de menor rango que la expresión, por lo que esta conversión puede perder información y ser peligrosa). Por ejemplo, si **i** y **j** son variables enteras y **x** es double,

```
x = i*j – j + 1;
```

En C existe también la posibilidad de realizar *conversiones explícitas de tipo* (llamadas ***casting***, en la literatura inglesa). El casting es pues una conversión de tipo, forzada por el programador. Para ello basta preceder la constante, variable o expresión que se desea convertir por el tipo al que se desea convertir, encerrado entre paréntesis. En el siguiente ejemplo,

```
k = (int) 1.7 + (int) masa;
```

la variable **masa** es convertida a tipo *int*, y la constante **1.7** (que es de tipo *double*) también. El ***casting*** se aplica con frecuencia a los valores de retorno de las funciones.

CONSTANTES

Se entiende por ***constantes*** aquel tipo de información numérica o alfanumérica que no puede cambiar más que con una nueva compilación del programa. Como ya se ha dicho anteriormente, en el código de un programa en C pueden aparecer diversos tipos de constantes que se van a explicar a continuación.

Constantes numéricas

CONSTANTES ENTERAS.

Una ***constante entera decimal*** está formada por una secuencia de dígitos del 0 al 9, constituyendo un número entero. Las constantes enteras decimales están sujetas a las mismas restricciones de rango que las variables tipo ***int*** y ***long***, pudiendo también ser ***unsigned***. El tipo de una constante se puede determinar automáticamente según su magnitud, o de modo explícito postponiendo ciertos caracteres, como en los ejemplos que siguen:

```
23484              constante tipo int
45815              constante tipo long (es mayor que 32767)
253u ó 253U        constante tipo unsigned int
739l ó 739L        constante tipo long
583ul ó 583UL      constante tipo unsigned long
```

En C se pueden definir también ***constantes enteras octales***, esto es, expresadas en base 8 con dígitos del 0 al 7. Se considera que una constante está expresada en base 8 si el primer dígito por la izquierda es un cero (0). Análogamente, una secuencia de dígitos (del 0 al 9) y de letras (A, B, C, D, E, F) precedida por `0x` o por `0X`, se interpreta como una ***constante entera hexadecimal***, esto es, una constante numérica expresada en base 16. Por ejemplo:

```
011           constante octal (igual a 9 en base 10)
11            constante entera decimal (no es igual a 011)
0xA           constante haxadecimal (igual a 10 en base 10)
0xFF          constante hexadecimal (igual a 16²-1=255 en base 10)
```

Es probable que no haya necesidad de utilizar constantes octales y hexadecimales, pero conviene conocer su existencia y saber interpretarlas por si hiciera falta. La ventaja de los números expresados en base 8 y base 16 proviene de su estrecha relación con la base 2 (8 y 16 son potencias de 2), que es la forma en la que el ordenador almacena la información.

CONSTANTES DE PUNTO FLOTANTE

Como es natural, existen también ***constantes de punto flotante***, que pueden ser de tipo ***float***, ***double*** y ***long double***. Una constante de punto flotante se almacena de la misma forma que la variable correspondiente del mismo tipo. Por defecto –si no se indica otra cosa– *las constantes de punto flotante son de tipo* ***double***. Para indicar que una constante es de tipo ***float*** se le añade una **f** o una **F**; para indicar que es de tipo ***long double***, se le añade una **l** o una **L**. En cualquier caso, el punto decimal siempre debe estar presente si se trata de representar un número real.

Estas constantes se pueden expresar de varias formas. La más sencilla es un conjunto de dígitos del 0 al 9, incluyendo un punto decimal. Para constantes muy grandes o muy pequeñas puede utilizarse la *notación científica*; en este caso la constante tiene una parte entera, un punto decimal,

una parte fraccionaria, una e o E, y un exponente entero (afectando a la base 10), con un signo opcional. Se puede omitir la parte entera o la fraccionaria, pero no ambas a la vez. Las constantes de punto flotante son siempre positivas. Puede anteponerse un signo (-), pero no forma parte de la constante, sino que con ésta constituye una ***expresión***, como se verá más adelante. A continuación se presentan algunos ejemplos válidos:

```
1.23          constante tipo double (opción por defecto)
23.963f       constante tipo float
.00874        constante tipo double
23e2          constante tipo double (igual a 2300.0)
.874e-2       constante tipo double en notación científica (=.00874)
.874e-2f      constante tipo float en notación científica
```

seguidos de otros que no son correctos:

```
1,23          error: la coma no esta permitida
23963f        error: no hay punto decimal ni carácter e ó E
.e4           error: no hay ni parte entera ni fraccionaria
-3.14         error: sólo el exponente puede llevar signo
```

Constantes carácter

Una constante carácter es un carácter cualquiera encerrado entre apóstrofos (tal como 'x' o 't'). El valor de una constante carácter es el valor numérico asignado a ese carácter según el código ASCII (ver Tabla 2.3). Conviene indicar que en C no existen constantes tipo ***char***; lo que se llama aquí ***constantes carácter*** son en realidad constantes enteras.

Hay que señalar que el valor ASCII de los números del 0 al 9 no coincide con el propio valor numérico. Por ejemplo, el valor ASCII de la constante carácter '7' es 55.

Ciertos caracteres no representables gráficamente, el apóstrofo (') y la barra invertida (\) y otros caracteres, se representan mediante la siguiente tabla de ***secuencias de escape***, con ayuda de la barra invertida (\)[4]

Nombre completo	Constante	en C	ASCII
sonido de alerta	BEL	\a	7
nueva línea	NL	\n	10
tabulador horizontal	HT	\t	9
retroceso	BS	\b	8
retorno de carro	CR	\r	13
salto de página	FF	\f	12
barra invertida	\	\\	92
apóstrofo	'	\'	39
comillas	"	\"	34
carácter nulo	NULL	\0	0

Los caracteres ASCII pueden ser también representados mediante el número octal correspondiente, encerrado entre apóstrofos y precedido por la barra invertida. Por ejemplo, '\07' y '\7' representan el número 7 del código ASCII (sin embargo, '\007' es la representación octal del carácter '7'), que es el sonido de alerta. El ANSI C también admite secuencias de escape hexadecimales, por ejemplo '\x1a'.

---

[4] Una *secuencia de escape* está constituida por la barra invertida (\) seguida de otro carácter. La finalidad de la secuencia de escape es cambiar el significado habitual del carácter que sigue a la barra invertida.

Cadenas de caracteres

Una cadena de caracteres es una secuencia de caracteres delimitada por comillas ("), como por ejemplo: "Esto es una cadena de caracteres". Dentro de la cadena, pueden aparecer caracteres en blanco y se pueden emplear las mismas secuencias de escape válidas para las constantes carácter. Por ejemplo, las comillas (") deben estar precedidas por (\), para no ser interpretadas como fin de la cadena; también la propia barra invertida (\). Es muy importante señalar que el compilador sitúa siempre un byte nulo (\0) adicional al final de cada cadena de caracteres para señalar el final de la misma. Así, la cadena "mesa" no ocupa 4 bytes, sino 5 bytes. A continuación se muestran algunos ejemplos de cadenas de caracteres:

```
"Informática I"
"'A'"
"        cadena con espacios en blanco         "
"Esto es una \"cadena de caracteres\".\n"
```

Constantes de tipo Enumeración

En C existen una clase especial de constantes, llamadas constantes ***enumeración***. Estas constantes se utilizan para definir los posibles valores de ciertos identificadores o variables que sólo deben poder tomar unos pocos valores. Por ejemplo, se puede pensar en una variable llamada **dia_de_la_semana** que sólo pueda tomar los 7 valores siguientes: **lunes**, **martes**, **miercoles**, **jueves**, **viernes**, **sabado** y **domingo**. Es muy fácil imaginar otros tipos de variables análogas, una de las cuales podría ser una variable booleana con sólo dos posibles valores: SI y NO, o TRUE y FALSE, u ON y OFF. El uso de este tipo de variables hace más claros y legibles los programas, a la par que disminuye la probabilidad de introducir errores.

En realidad, las constantes ***enumeración*** son los posibles valores de ciertas variables definidas como de ese tipo concreto. Considérese como ejemplo la siguiente declaración:

```
enum dia {lunes, martes, miercoles, jueves, viernes, sabado, domingo};
```

Esta declaración crea un nuevo ***tipo de variable*** –el tipo de variable ***dia***– que sólo puede tomar uno de los 7 valores encerrados entre las llaves. Estos valores son en realidad constantes tipo ***int***: **lunes** es un 0, **martes** es un 1, **miercoles** es un 2, etc. Ahora, es posible definir variables, llamadas ***dia1*** y ***dia2***, que sean de tipo ***dia***, en la forma (obsérvese que en C deben aparecer las palabras ***enum dia***; en C++ basta que aparezca la palabra ***dia***)

```
enum dia  dia1, dia2;       // esto es C
dia dia 1, dia 2;           // esto es C++
```

y a estas variables se les pueden asignar valores en la forma

```
dia1 = martes;
```

o aparecer en diversos tipos de expresiones y de sentencias que se explicarán más adelante. Los valores enteros que se asocian con cada constante tipo enumeración pueden ser controlados por el programador. Por ejemplo, la declaración,

```
enum dia {lunes=1, martes, miercoles, jueves, viernes, sabado, domingo};
```

asocia un valor 1 a **lunes**, 2 a **martes**, 3 a **miercoles**, etc., mientras que la declaración,

```
enum dia {lunes=1, martes, miercoles, jueves=7, viernes, sabado, domingo};
```

asocia un valor 1 a **lunes**, 2 a **martes**, 3 a **miercoles**, un 7 a **jueves**, un 8 a **viernes**, un 9 a **sabado** y un 10 a **domingo**.

Se puede también hacer la definición del tipo ***enum*** y la declaración de las variables en una única sentencia, en la forma

```
enum palo {oros, copas, espadas, bastos}  carta1, carta2, carta3;
```

donde **carta1**, **carta2** y **carta3** son variables que sólo pueden tomar los valores ***oros***, ***copas***, ***espadas*** y ***bastos*** (equivalentes respectivamente a 0, 1, 2 y 3).

CUALIFICADOR CONST

Se puede utilizar el cualificador ***const*** en la declaración de una variable para indicar que esa variable no puede cambiar de valor. Si se utiliza con un array, los elementos del array no pueden cambiar de valor. Por ejemplo:

```
const int i=10;
const double x[] = {1, 2, 3, 4};
```

El lenguaje C no define lo que ocurre si en otra parte del programa o en tiempo de ejecución se intenta modificar una variable declarada como ***const***. De ordinario se obtendrá un mensaje de error en la compilación si una variable ***const*** figura a la izquierda de un operador de asignación. Sin embargo, al menos con el compilador de Microsoft, se puede modificar una variable declarada como ***const*** por medio de un puntero de la forma siguiente:

```
const int i=10;
int *p;
p = &i;
*p = 1;
```

C++ es mucho más restrictivo en este sentido, y no permite de ningumamanera modificar las variables declaradas como ***const***.

El cualificador ***const*** se suele utilizar cuando, por motivos de eficiencia, se pasan argumentos por referencia a funciones y no se desea que dichos argumentos sean modificados por éstas.