Tema 2

Elementos básicos de programación

En este tema se presenta un conjunto mínimo de elementos de un lenguaje de programación imperativo. Este conjunto se particulariza para el lenguaje **C±**. Con los elementos presentados se podrán construir programas completos aunque con una estructura muy simple, ya que sólo pueden estar formados por una secuencia de sentencias. Para que estos primeros programas produzcan resultados, se introducen también varios mecanismos de escritura simple.

El objetivo que trata de alcanzar este tema es permitir el desarrollo de programas completos desde el principio. Estos programas se podrán realizar como prácticas con el computador de manera inmediata y directa utilizando un compilador de C/C++.

2.1 Lenguaje C±

El lenguaje de programación **C±** (léase C-más-menos), que se utilizará a lo largo de todo este libro para introducir los diferentes conceptos de programación, está constituido por un subconjunto del vocabulario de los lenguajes C y C++. Por lo tanto, cualquier programa escrito en el *lenguaje* **C±** se podrá editar, compilar y ejecutar en un entorno de desarrollo para C++ que incluya como subconjunto al lenguaje C. Conviene señalar que los ficheros fuente que contengan los programas **C±** deberán tener extensión `.cpp` como si fueran programas en C++.

El objetivo fundamental de utilizar el lenguaje **C±** es la introducción de los conceptos fundamentales de programación de una manera progresiva, sistemática y sin ambigüedades con el fin de que se adquiera una buena metodología de programación. Lamentablemente C o C++ no fueron diseñados para la formación de programadores y disponen de ciertas estructuras que conceptualmente son poco rigurosas y que por ello no forman parte de **C±**.

El lenguaje **C±** se irá presentando de manera simultánea a la introducción de los conceptos según se avance en el curso. La presentación de cada nuevo elemento de **C±** se realizará formalmente mediante la notación BNF (ver siguiente epígrafe de este tema). En el apéndice A de este libro se recopila la descripción formal en BNF de la sintaxis completa del lenguaje **C±**.

En cada tema sólo se utilizarán aquellos elementos del lenguaje **C±** que ya hayan sido presentados. Dado el enfoque metodológico de la asignatura, cualquier programa o práctica que no se realice en el lenguaje **C±** y siguiendo las pautas marcadas a lo largo del libro se considerará incorrecto. Aunque el exigir restricciones como éstas pueda parecer de carácter solo formativo o estrictamente académico, el poner ciertas restricciones para la codificación en determinados lenguajes de programación resulta una práctica bastante habitual en el desarrollo de software a nivel industrial.

Siguiendo las pautas de buenas prácticas de ingeniería de software, cualquier empresa o equipo de desarrollo de software debe disponer antes del inicio de cada desarrollo de un *Manual de Estilo*. Para lograr la adecuada claridad, homogeneidad y mantenibilidad de los programas, en el *Manual de Estilo*, se establecen prohibiciones expresas de uso de algunas estructuras del lenguaje de programación empleado, el formato de escritura de cada sentencia, recomendaciones de uso de los distintos elementos del lenguaje (constantes, variables, tipos y subprogramas) y otros muchos aspectos. Para garantizar que todo el desarrollo sigue estas pautas establecidas, una o varias personas del departamento de calidad son las encargadas del garantizar la calidad requerida de todos los programas y para ello tienen la potestad de exigir las correcciones o modificaciones que consideren necesarias a los programadores.

2.2 Notación BNF

Un lenguaje de programación sigue unas reglas gramaticales similares a las de cualquier idioma humano, aunque más estrictas. Para la definición formal de dichas reglas sintácticas utilizaremos la *notación BNF* (Backus-Naur Form) basada en la descripción de cada elemento gramatical en función de otros más

sencillos, según determinados esquemas o construcciones. Cada uno de estos esquemas se define mediante una *regla de producción*.

Estas reglas sobre cómo han de escribirse los elementos del lenguaje en forma de símbolos utilizan a su vez otros símbolos, que se denominan *metasímbolos*. Son los siguientes:

::= Metasímbolo de definición. Indica que el elemento a su izquierda puede desarrollarse según el esquema de la derecha.

| Metasímbolo de alternativa. Indica que puede elegirse uno y sólo uno de los elementos separados por este metasímbolo.

{ } Metasímbolos de repetición. Indican que los elementos incluidos dentro de ellos se pueden repetir cero o más veces.

[ ] Metasímbolos de opción. Indican que los elementos incluidos dentro de ellos pueden ser utilizados o no.

( ) Metasímbolos de agrupación. Agrupan los elementos incluidos en su interior.

Estos metasímbolos se escriben con el tipo de letra especial indicado para distinguirlos de los paréntesis, corchetes, etc. que forman parte del lenguaje **C±**. También se emplearán distintos estilos de letra para distinguir los elementos simbólicos siguientes:

Elemento_no_terminal Este estilo se emplea para escribir el nombre de un elemento gramatical que habrá de ser definido por alguna regla. Cualquier elemento a la izquierda del metasímbolo **::=** será no terminal y aparecerá con este estilo.

`Elemento_terminal` Este estilo se emplea para representar los elementos que forman parte del lenguaje **C±**, es decir, que constituyen el texto de un programa. Si aparecen en una regla deberán escribirse exactamente como se indica.

2.3 Valores y tipos

El computador, como máquina de tratamiento de información, manipula diferentes datos. Un *dato* es un elemento de información que puede tomar un *valor* entre varios posibles. Si un dato tiene siempre necesariamente un valor fijo, diremos que es una *constante*.

Los valores de los datos pueden ser de diferentes clases. En general un dato sólo puede tomar valores de una clase. Por ejemplo, la estatura de una persona no puede tomar el valor "Felipe", ni el nombre de una persona puede ser "175".

En programación a las distintas clases de valores se les denomina *tipos*. Un dato tiene asociado un tipo, que representa la clase de valores que puede tomar. Por ejemplo, son tipos diferentes:

- Los números enteros.
- Los días de la semana.
- Los meses del año.
- Los títulos de libros.
- ... etc. ...

Es importante destacar que el concepto de tipo es algo abstracto, e independiente de los símbolos concretos que se empleen para *representar* los valores. Por ejemplo, aunque podemos representar los meses del año mediante números enteros de 1 a 12, los meses no son números enteros, pues no tiene sentido, por ejemplo, sumar Enero (1) y Marzo (3) para obtener Abril (4).

Con más precisión se habla de *tipos abstractos de datos*, que identifican tanto el conjunto de valores que pueden tomar los datos de ese tipo como las operaciones significativas que pueden hacerse con dichos valores.

En la comunicación humana usamos habitualmente dos grandes clases de valores: los *números* y los *textos*. Los lenguajes de programación llevan incluidas formas de representación concretas de estas clases de valores, que se traducen en la existencia de tipos de datos predefinidos, ya incorporados al lenguaje, y que pueden usarse, en su caso, para representar también valores de otros nuevos tipos de datos definidos por el programador. Aunque en la práctica los números han de escribirse externamente en forma de texto para poder ser leídos por las personas, desde el punto de vista abstracto son valores de tipos diferentes a los de los caracteres que los representan.

2.4 Representación de valores constantes

Uno de los objetivos de los lenguajes de programación es evitar las ambigüedades o imprecisiones que existen en los lenguajes humanos. Por ejemplo, la representación de valores numéricos en los países anglosajones se realiza separando por comas (,) los millares. Así, trescientos cuarenta y ocho mil quinientos treinta y seis se representa de la siguiente manera:

348,536

Sin embargo, nosotros utilizamos la coma para separar la parte entera de la parte decimal de un número no entero. Por lo tanto, la interpretación con esta regla del número anterior sería: trescientos cuarenta y ocho con quinientas treinta y seis milésimas.

A continuación se indican las reglas particulares de **C±** para la representación de valores básicos, tanto numéricos como de texto.

2.4.1 Valores numéricos enteros

Los valores enteros representan un número exacto de unidades, y no pueden tener parte fraccionaria. Un *valor entero* se escribe mediante una secuencia de uno o más dígitos del 0 al 9 sin separadores de ninguna clase entre ellos y precedidos opcionalmente de los símbolos más (+) o menos (−). Son enteros válidos los siguientes:

```
2
+56
0
-234567745
1000000000
```

Sin embargo, no son valores enteros válidos los siguientes:

`123,234`	No se pueden usar comas
`22.56`	No se pueden usar puntos
`13E5`	No se pueden usar letras

Usando la notación BNF podemos representar de manera precisa las reglas para escribir estos valores:

Valor_entero **::=** **[** + **|** - **]** *Secuencia_dígitos*

Secuencia_dígitos **::=** *Dígito* **{** *Dígito* **}**

Dígito **::=** 0 **|** 1 **|** 2 **|** 3 **|** 4 **|** 5 **|** 6 **|** 7 **|** 8 **|** 9

▮ NOTA: El lenguaje **C±** como derivado de C/C++ considera que cuando el valor entero comienza por un primer dígito 0 se está escribiendo en base 8 (octal) en lugar de en base 10 (decimal). Así, el valor númerico 020 es un número octal que equivale a 16 en decimal. En este curso no se hace uso de los valores octales y carece de sentido poner ceros a la izquierda de un valor numérico. En cualquier caso, el compilador de C/C++ da un error si al escribir un valor octal se utilizan los dígitos 8 ó 9.

2.4.2 Valores numéricos reales

Los valores numéricos reales permiten expresar cualquier cantidad, incluyendo fracciones de unidad. Se pueden representar de dos maneras distintas: en la notación decimal habitual, o en la notación científica. En la notación decimal habitual un *valor real* se escribe con una parte entera terminada siempre por un punto (`.`), y seguida opcionalmente por una secuencia de dígitos que constituyen la parte fraccionaria decimal. De acuerdo con ello son valores reales válidos los siguientes:

```
5.
-0.78
+234.53
0.0000000034
1234.000
```

En la notación científica un número real se escribe como una *mantisa*, que es un número real en la notación decimal habitual, seguida de un factor de escala que se escribe como la letra `E` seguida del exponente entero de una potencia de 10 por la que se multiplica la mantisa. Son valores reales válidos en notación científica:

`-23.2E+12`	equivalente a -23.2×10^{12}
`14567.823E4`	equivalente a 14567.823×10^{4}
`126.E-34`	equivalente a 126×10^{-34}

Sin embargo, no son valores reales válidos los siguientes:

`4,78`	No se pueden usar comas
`56.7F-56`	No se puede usar la letra `F`

A diferencia de los valores enteros, un mismo valor real puede tener muy diversas representaciones válidas. Por ejemplo, todas las representaciones siguientes corresponden al mismo valor:

```
45.6      456.E-1      4.56E+1      45.60E+0      456000.00E-4
```

Las reglas anteriores, expresadas en notación BNF son:

Valor_real **::=** *Valor_entero* **.** **[** *Secuencia_dígitos* **]** **[** *Escala* **]**

Escala **::=** `E` *Valor_entero*

2.4.3 Caracteres

Además de los valores numéricos enteros o reales, empleados para la realización de cálculos numéricos, los lenguajes de programación nos deben permitir representar valores correspondientes a los caracteres de un texto, y que están disponibles en cualquier teclado, pantalla o impresora.

Dentro del texto de un programa en **C±** el *valor de un carácter* concreto se escribe poniendo dicho carácter entre apóstrofos (`'`). Ejemplos de valores de caracteres son los siguientes:

```
'a'   'Ñ'   '7'   '?'   '.'   ' '   '}'
'A'   'ñ'   '5'   '!'   ';'   '{'   '"'
```

Es interesante hacer las siguientes observaciones:

- el espacio en blanco (`' '`) es un carácter válido como los demás
- hay que distinguir entre un valor entero de un dígito (p.ej. `7`) y el carácter correspondiente a dicho dígito (p.ej. `'7'`)

La colección o *juego de caracteres* (*charset*) que pueden manipularse en un programa depende de la máquina que se esté usando. Sólo se pueden representar de la forma indicada (escribiéndolos entre apóstrofos) aquellos caracteres que tengan asociado un símbolo gráfico (letra, dígito, signo de puntuación, etc.) que pueda introducirse en el texto del programa. Otros caracteres definidos, tales como los *caracteres de control*, que no tienen símbolo gráfico, se representan mediante una *secuencia de escape* con la siguiente notación:

`'\n'`	Salto al comienzo de una nueva línea de escritura
`'\r'`	Retorno al comienzo de la misma línea de escritura
`'\t'`	Tabulación
`'\''`	Apóstrofo
`'\\'`	Barra inclinada
`'\f'`	Salto a una nueva página o borrado de pantalla

2.4.4 Cadenas de caracteres (*strings*)

Es frecuente que los caracteres no se utilicen de forma aislada, sino formando palabras o frases. Una *cadena de caracteres* (en inglés *string*) se escribe como una secuencia de caracteres incluidos entre comillas (`"`). Por ejemplo:

```
"Palabra"
"Este texto es una cadena de caracteres"
"&"
"El resultado de A+B es: "
"Incluir entre 'apóstrofos' el texto"
"Conteste \"Si\" o \"No\""
"¿Año de fabricación?"
""
```

Conviene observar que:

- si una cadena incluye comillas en su interior se escribirá mediante `\"`
- no hay que confundir un valor de tipo carácter (`'x'`) con una cadena del mismo único carácter (`"x"`). La distinción se produce por el delimitador utilizado comillas (`"`) para una cadena y apóstrofo (`'`) para un carácter
- es posible definir una cadena vacía que no contenga ningún carácter, como en el último ejemplo

Una cadena puede contener cualquier número de caracteres y puede incluir cualquier carácter alfabético o de puntuación que sea representable dentro del texto del programa. Aquí se aplican las mismas observaciones que se han hecho antes respecto al juego de caracteres particular de cada máquina.

2.5 Tipos predefinidos

Las diferentes formas de representación de valores constantes presentadas en los apartados anteriores distinguen ya varias clases de datos, pero que no llegan a ser tipos en sí mismos. En efecto, tal como vamos a ver a continuación, dentro de una misma clase de valores pueden distinguirse varios tipos diferentes, tanto a nivel de *tipos predefinidos* en el lenguaje, como de *tipos definidos* por el programador.

Recordaremos que un tipo de datos define:

1. Una colección de valores posibles
2. Las operaciones significativas sobre ellos

En el lenguaje **C±** hay cuatro tipos de datos predefinidos, que se designan con los nombres `int`, `float`, `char`, `bool`, así como mecanismos para definir nuevos tipos a partir de ellos. En los lenguajes C y C++ hay tipos predefinidos adicionales. En las secciones siguientes se describen los tipos predefinidos fundamentales, excepto el tipo `bool`, que se describe más adelante.

2.5.1 El tipo entero (`int`)

Los valores de este tipo son los valores numéricos enteros positivos y negativos. Como tipo abstracto su definición coincide con el concepto matemático de los números enteros. Sin embargo, dado el carácter físico de los computadores, el rango de valores nunca podrá ser infinito como se establece en el concepto matemático. En cada caso el rango de valores del *tipo* `int` depende de la *plataforma* (combinación de: procesador, sistema operativo y compilador) que se esté utilizando. En general se corresponde con el rango de valores que pueden manipularse con instrucciones básicas del lenguaje de máquina y viene a ser simétrico en torno al valor cero. Dentro de dicho rango la representación de cualquier valor es exacta. Son rangos comunes los siguientes:

Tamaño de palabra	**Rango de valores enteros**
16 bits	-32.768 ... 0 ... 32.767
32 bits	-2.147.483.648 ... 0 ... 2.147.483.647
64 bits	-9.223.372.036.854.775.808 ... 0 ... 9.223.372.036.854.775.807

Estos rangos obedecen a que los computadores suelen emplear la codificación en base 2 de los valores enteros. Para el signo del número se utiliza un bit, quedando, por tanto, 15, 31 ó 63 para el valor absoluto:

$$2^{15} = 32.768$$
$$2^{31} = 2.147.483.648$$
$$2^{63} = 9.223.372.036.854.775.808$$

▮ NOTA: Para facilitar la escritura de programas que tengan en cuenta la limitación particular de rango existente en cada caso, C y C++ permiten hacer referencia al valor mínimo mediante el nombre simbólico `INT_MIN`, y al valor máximo mediante `INT_MAX`. El rango admisible será, por tanto: `INT_MIN` ... 0 ... `INT_MAX`. Estos nombres están definidos en el módulo `limits` de la librería estándar de C (cabecera `<limits.h>`).

Asociadas al tipo `int` están las operaciones que se pueden realizar con los valores de este tipo. Las operaciones predefinidas entre valores enteros son las operaciones aritméticas básicas, que se realizan entre enteros y devuelven como resultado valores enteros. Para invocar estas operaciones se dispone de los siguientes símbolos de operación u *operadores*:

`+`	Suma de enteros	`a + b`
`-`	Resta de enteros	`a - b`
`*`	Multiplicación de enteros	`a * b`
`/`	División de enteros	`a / b`
`%`	Resto de la división	`a % b`
`+`	Identidad de un entero	`+ a`
`-`	Cambio de signo de un entero	`- a`

Siguiendo la representación aritmética habitual, los símbolos `+` y `-` tienen un doble significado, según se usen como operadores infijos entre 2 operandos o como operadores prefijos con un único operando.

El operador `/` realiza la división entre dos números enteros y obtiene como resultado el cociente entero truncado al valor más próximo a cero. Cuando el divisor es cero se obtiene como resultado un error. Por ejemplo:

Operación	**Resultado**
`10 / 3`	3
`(-20) / (-7)`	2
`(-15) / 4`	-3
`17 / (-3)`	-5
`34 / 0`	*Error*

El operador `%` obtiene el resto de la división de enteros realizada con `/`. Por ejemplo:

Operación	**Resultado**
`10 % 3`	1
`(-20) % (-7)`	-6
`(-15) % 4`	-3
`17 % (-3)`	2
`34 % 0`	*Error*

Entre los operadores `/` y `%` se cumple la regla aritmética habitual:

$$Dividendo = Divisor \times Cociente + Resto$$

que en **C±** se expresaría así:

```
a = b * (a/b) + (a%b)
```

Cuando se realiza una operación con enteros se debe tener en cuenta el rango de valores disponible en la plataforma que se esta utilizando. Si se produce un resultado fuera del rango disponible se producirá un error. En algunos casos este tipo de errores no se indica y puede ser difícil su detección. Por ejemplo, para enteros de 16 bits con un rango entre -32.768 y 32.767, se obtienen los siguientes resultados:

Operación	Resultado
`234 + 89`	323
`345 * 97`	*Error*
`214 * (-203)`	*Error*
`15456 + 18678`	*Error*
`(-20) - 32750`	*Error*

2.5.2 El tipo real (`float`)

Con el *tipo* `float` se trata de representar en el computador los valores numéricos reales positivos y negativos. Sin embargo, al contrario que en caso del tipo `int`, esta representación puede no ser exacta. Además, dado que la capacidad de los computadores es limitada, la representación sólo se puede considerar válida dentro de un rango, de forma semejante a como sucede con los enteros.

Tanto el rango como la precisión dependen de la plataforma concreta utilizada. Dentro de dicho rango para algunos valores concretos es posible una representación exacta. Sin embargo, dado el carácter discreto de los datos que siempre se manejan en un computador, nunca será posible una representación exacta de valores tales como los valores irracionales π o e o, en general, de valores cuya precisión sea superior a la disponible en la plataforma que estemos utilizando. En estos casos se manejan valores aproximados.

Los valores reales se suelen representar internamente de forma equivalente a la notación científica, con una mantisa y un factor de escala. El rango de valores representables está limitado tanto para valores grandes como pequeños. Los valores más pequeños que un límite dado se confunden con el cero. Al igual que en el caso de los valores enteros, el rango y precisión de los valores reales puede cambiar de una plataforma a otra. Algunos de los rangos habituales son los siguientes:

Tamaño de palabra y precisión	Rango de valores reales
32 bits; 6 cifras decimales	-3.4E+38 ... -1.2E-38 0 +1.2E-38 ... +3.4E+38
64 bits; 15 cifras decimales	-1.7E+308 ... -2.3E-308 0 +2.3E-308 ... +1.7E+308

Estos rangos dependen del número concreto de bits y de la codificación que se emplean para la mantisa y el exponente del valor `float`. En el caso de valores

representados con 32 bits no existe ningún valor intermedio entre $-1,2\times10^{-38}$ y el valor 0, ni tampoco entre 0 y $1,2\times10^{-38}$ (y análogamente para valores en 64 bits).

Asociadas al tipo `float` están las operaciones que se pueden realizar con él. Las operaciones entre valores reales son las operaciones aritméticas básicas, que se realizan entre reales y devuelven como resultado valores reales. Los correspondientes operadores son los siguientes:

`+`	Suma de reales	`a + b`
`-`	Resta de reales	`a - b`
`*`	Multiplicación de reales	`a * b`
`/`	División de reales	`a / b`
`+`	Identidad de un real	`+ a`
`-`	Cambio de signo de un real	`- a`

Los símbolos empleados para estos operadores son los mismos que para los operadores enteros. Sin embargo, en todos los casos son operadores distintos de los operadores enteros. Las operaciones entre reales dan como resultado un real con la precisión de la plataforma. Así, para valores reales no se cumple siempre exactamente la relación básica:

```
(a / b) * b  =  a
```

Es importante tener en cuenta imprecisiones como ésta cuando los cálculos sean más complejos y se puedan acumular errores. Ejemplos de operaciones entre valores reales son las siguientes:

Operación	**Resultado**
`10.5 / 0.2`	52.5
`-20.6 * 2.4`	-49.44
`-15.4E2 + 450.0`	-1090.0
`23.4 - 2E-1`	23.2

La representación sólo aproximada de los valores reales se pone de manifiesto si tratamos de expresar con más precisión de la realmente existente el resultado de una operación. Por ejemplo:

Operación	**Resultado**
`10.0 / 3.0`	`3.3333332538604736E+0`

En este caso concreto el valor es inexacto a partir de la 7ª cifra decimal. En cada plataforma se podrá obtener un resultado ligeramente diferente.

2.5.3 El tipo carácter (`char`)

Para comprender bien el manejo de valores de *tipo carácter* en un computador es necesario conocer cómo se definen y representan esos valores de caracteres. Cada carácter no se representa internamente como un dibujo (el *glifo* del carácter), sino como un valor numérico entero que es su código. La colección concreta de caracteres y sus códigos numéricos se establecen en una tabla (*charset*) que asocia a cada carácter el código numérico (*codepoint*) que le corresponde.

Dependiendo del número de bits reservado para representar el código de cada carácter podremos tener tablas más o menos amplias. Algunas tablas de caracteres de amplio uso son:

Tabla (*charset*)	**Tamaño del carácter**	**Repertorio de caracteres**
ASCII	7 bits	Letras inglesas mayúsculas y minúsculas. Algunos signos de puntuación y códigos de control.
ISO-8859-1 (llamado también Latin-1)	8 bits	Lo anterior, más letras con acentos y nuevos signos de puntuación
UNICODE-BMP (Basic Multilingual Plane)	16 bits	Incluye además los alfabetos griego, cirílico, árabe, chino/japonés/coreano, signos matemáticos, etc.
UNICODE completo	32 bits	Incluye la práctica totalidad de caracteres utilizados en cualquier idioma o notación textual existente en nuestro mundo actual.

Las tablas mencionadas son compatibles entre sí en el sentido de que cada una de ellas incluye la anterior, manteniendo los códigos numéricos de los caracteres. Lamentablemente la compatibilidad no se extiende a otras muchas tablas de caracteres de amplio uso. Por ejemplo, otras tablas de caracteres de 8 bits muy conocidas son:

Tabla (*charset*)	**Tamaño del carácter**	**Repertorio de caracteres**
ISO-8859-7	8 bits	Repertorio ASCII más el alfabeto griego
ISO-8859-15	8 bits	Repertorio Latin-1 revisado. Incluye el símbolo de Euro
IBM-PC-437 (original del sistema operativo MS-DOS)	8 bits	Repertorio ASCII más simbolos semigráficos y letras con acentos, pero con códigos distintos de Latin-1
IBM-PC-850 (usado también en MS-DOS)	8 bits	Casi el mismo repertorio de Latin-1 pero con códigos diferentes, y diferentes también de IBM-PC-437.
Windows-1252 (usado en los sistemas operativos MS-Windows)	8 bits	Coincide con Latin-1 excepto en un rango de 32 códigos de Latin-1 que repiten códigos de control

En **C±** (como en C/C++) los valores del tipo `char` ocupan 8 bits e incluyen el repertorio ASCII. Además incluyen otros caracteres no-ASCII que dependen de la tabla de caracteres establecida. En los ejemplos de este libro asumiremos que se dispone de los caracteres comunes a Latin-1 y Windows-1252. Por lo tanto la colección de valores del tipo `char` incluye caracteres alfabéticos, numéricos, de puntuación y caracteres de control.

Como ya se ha dicho, en el texto de un programa se pueden escribir los valores de los caracteres, bien directamente, o mediante una secuencia de escape, p.ej. para los caracteres de control. También se puede representar cualquier carácter mediante la notación `char(`x`)` siendo x el código del carácter. Por ejemplo, en ASCII:

`char(10)` Salto al comienzo de una nueva línea. Posición 10ª
`char(13)` Retorno al comienzo de la misma línea. Posición 13ª
`char(65)` Letra A mayúscula. Posición 65ª

En sentido inverso, el código numérico de un determinado carácter c se expresa como `int(`c`)`. Por ejemplo:

`int('A')` 65 (65ª posición de la tabla ASCII)
`int('Z')` 90 (90ª posición de la tabla ASCII)

De forma inmediata se puede decir que, para cualquier carácter c, cuyo código sea x, se cumplirá que:

$$\texttt{char(int(}c\texttt{))} = c$$
$$\texttt{int(char(}x\texttt{))} = x$$

Además conviene saber que la tabla ASCII posee las siguientes características:

- Los caracteres correspondientes a las letras mayúsculas de la 'A' a la 'Z' están ordenados en posiciones consecutivas y crecientes según el orden alfabético.
- Los caracteres correspondientes a las letras minúsculas de la 'a' a la 'z' están ordenados en posiciones consecutivas y crecientes según el orden alfabético.
- Los caracteres correspondientes a los dígitos del '0' al '9' están ordenados en posiciones consecutivas y crecientes.

Esto facilita el obtener por cálculo el valor numérico equivalente al carácter de un dígito decimal, o la letra mayúscula correspondiente a una minúscula o viceversa.

En C (y en **C±**) se puede usar también el módulo de librería `ctype` (cabecera `<ctype.h>`), que facilita el manejo de diferentes clases de caracteres. Este módulo incluye funciones tales como:

`isalpha( c )`	Indica si `c` es una letra
`isascii( c )`	Indica si `c` es un carácter ASCII
`isblank( c )`	Indica si `c` es un carácter de espacio o tabulación
`iscntrl( c )`	Indica si `c` es un carácter de control
`isdigit( c )`	Indica si `c` es un dígito decimal (0-9)
`islower( c )`	Indica si `c` es una letra minúscula
`isspace( c )`	Indica si `c` es espacio en blanco o salto de línea o página
`isupper( c )`	Indica si `c` es una letra mayúscula
`tolower( c )`	Devuelve la minúscula correspondiente a `c`
`toupper( c )`	Devuelve la mayúscula correspondiente a `c`

▮ NOTA: El concepto de función se introduce en el tema 7, y el de módulo en el tema 14.

2.6 Expresiones aritméticas

Una *expresión aritmética* representa un cálculo a realizar con valores numéricos (más adelante se verán expresiones que utilizan también valores de otros tipos). Una expresión aritmética es una combinación de *operandos* y *operadores*.

Para indicar el orden en que se quieren realizar las operaciones parciales se pueden utilizar paréntesis. Si no se utilizan paréntesis el orden de las operaciones depende de una jerarquía entre los operadores empleados, que para los operadores numéricos es la siguiente:

1º Operadores multiplicativos: `* / %`
2º Operadores aditivos: `+ -`

Dentro del mismo nivel las operaciones se ejecutan en el orden en que están escritas en la expresión aritmética de izquierda a derecha.

Si una expresión va precedida del signo más o menos, se entiende que solamente le afecta al primer operando. Si se quiere que afecte a toda la expresión, ésta deberá incluirse entre paréntesis.

Ejemplos de expresiones entre datos enteros son las siguientes:

Expresión	**Resultado**
`5 * 30 + 5`	155
`334 / 6 % 4 * 5`	15
`-5 * 10 % 3 / 2`	-1

Cuando la complejidad de la expresión puede dar lugar a posibles errores de interpretación, es preferible utilizar paréntesis para clarificar cuál es el cálculo exacto que se quiere realizar. Así las expresiones anteriores son equivalentes a las siguientes:

```
(5 * 30) + 5
((334 / 6) % 4) * 5
(((-5) * 10) % 3) / 2
```

Igualmente, ejemplos de expresiones entre valores reales son las siguientes:

Expresión	**Resultado**	**Expresión equivalente**
`35.3 * 5.1 / 7.6 - 4.5`	19.18816	`((35.3*5.1)/7.6)-4.5`
`-23.1 / 6.2 * 5.4 / 2.4`	-8.38306	`(((-23.1)/6.2)*5.4)/2.4`

Aunque se represente por los mismos símbolos, los operadores aritméticos para valores reales y enteros son en realidad diferentes. Así, si se mezclaran en una misma expresión valores de tipos diferentes, las expresiones aritméticas son completamente ambiguas. Por ejemplo en las siguientes operaciones:

`33.7 / 5` ¿La división a realizar es entera o real?
`33 / 5.3` ¿La división a realizar es entera o real?
`25 * 3.5` ¿La multiplicación a realizar es entera o real?

Evidentemente, los resultados serán diferentes según el tipo de operación que se realice. Además no queda claro si el resultado que se pretende obtener es un valor entero o real. Para poder realizar estas operaciones combinadas es necesario que previamente se realice una conversión de la representación de los datos al tipo adecuado. La representación real de un dato entero se indica de la siguiente manera:

`float(45)` Representa el valor numérico 45.0 con tipo `float`

De forma similar la representación entera de un dato real (correspondiente a la parte entera, truncando el valor) se consigue de la siguiente forma:

`int(34.7)` Representa el valor numérico 34 con tipo `int`

Por tanto, si queremos obtener un resultado entero, las operaciones entre enteros y reales anteriormente indicadas se tienen que realizar de la siguiente manera:

Expresión	**Resultado**	**Tipo**
`int(33.7) / 5`	6	`int`
`33 / int(5.3)`	6	`int`
`25 * int(3.5)`	75	`int`

Si el resultado deseado es un valor real, es necesario realizar previamente las conversiones a real de los operandos enteros y lógicamente obtendremos resultados completamente distintos:

Expresión	**Resultado**	**Tipo**
`33.7 / float (5)`	6.74	`float`
`float(33) /5.3`	6.22642	`float`
`float(25) * 3.5`	87.5	`float`

El lenguaje **C±** permite la ambigüedad que supone la mezcla de tipos de datos diferentes en la misma expresión sin exigir una conversión explícita. Para salvar la ambigüedad, **C±** utiliza el convenio de C/C++ de convertir previamente de manera automática todos los valores de una misma expresión al tipo del valor con mayor rango y precisión. Por tanto, el resultado siempre se obtendrá también en el mayor rango y precisión utilizado en la expresión. El desconocimiento de esta regla implícita, puede dar lugar a que el resultado de una expresión aritmética sea completamente inesperado. Para evitar esta situación, en el *Manual de Estilo* para la realización de programas en esta asignatura es obligatorio que se realice siempre una conversión explícita de tipos.

2.7 Operaciones de escritura simples

El objetivo de un programa es obtener unos resultados. Estos resultados deben ser emitidos al exterior del computador a través de un dispositivo de salida de datos: impresora, pantalla, trazador (*plotter*), línea de comunicaciones, etc. Las acciones que envían resultados al exterior se llaman, en general, *operaciones de escritura*, con independencia de que se trate de una impresión en papel, o la simple visualización en pantalla, o la grabación de los datos en un soporte donde queden registrados, o su envío a otro equipo remoto.

Existe una gran variedad de dispositivos periféricos, que se diferencian mucho en los detalles de su manejo. Para simplificar la escritura de resultados los lenguajes de programación prevén sentencias de escritura apropiadas para ser usadas con cualquier tipo de dispositivo, facilitando la tarea de programación al especificar la escritura de resultados de una manera uniforme, con independencia de las particularidades del dispositivo físico que se utilice en cada caso.

Al diseñar un lenguaje de programación se puede optar por usar sentencias o instrucciones especiales para ordenar la escritura de resultados, o bien ordenar la escritura del resultado con las mismas sentencias generales que se empleen para invocar operaciones definidas por el usuario. Los primeros lenguajes de programación solían emplear la primera alternativa. Los lenguajes más modernos utilizan con preferencia la segunda, que simplifica la complejidad del lenguaje en sí, a costa de permitir a veces una cierta variación en las operaciones de escritura entre diferentes versiones del lenguaje.

El lenguaje **C±** adopta también la segunda alternativa que es la utilizada en C/C++. Las operaciones de escritura se definen como procedimientos (ver tema 7), que se invocan escribiendo el nombre de la operación, seguido de una serie de valores o argumentos entre paréntesis. Estos *procedimientos* están definidos en *módulos de librería* (ver tema 14) disponibles de antemano.

En todas las versiones de C/C++ deben estar disponibles ciertos módulos estándar con la definición de operaciones de escritura normalizadas. En este apartado describiremos una operación disponible en el módulo llamado `stdio`.

2.7.1 El procedimiento `printf`

Este procedimiento pertenece al módulo `stdio` (cabecera `<stdio.h>`). La forma más sencilla de invocarlo es escribir:

`printf(` *cadena-de-caracteres* `);`

▮ NOTA: Esta forma sencilla sólo es válida si la cadena de caracteres a escribir no contiene el carácter %.

El procedimiento `printf` escribe en la pantalla del computador la cadena de caracteres. Por ejemplo, para cada una de las siguientes operaciones de escritura se obtiene el resultado que se muestra a su derecha. Para visualizar con detalle el resultado se ha utilizado el símbolo '·' para representar el carácter de espacio en blanco.

Operación de escritura	Resultado en pantalla
`printf( "En un lugar de " );`	`En·un·lugar·de·`
`printf( "¿Año de nacimiento?" );`	`¿Año·de·nacimiento?`

Si lo que se quiere escribir es la representación como texto de una serie de valores de cualquier tipo de los vistos hasta el momento (enteros, reales, caracteres, etc.), habrá que usar la forma general de la orden `printf`:

`printf(` *cadena-con-formatos, valor1, valor2, ... valorN* `);`

Una cadena de caracteres con formatos deberá incluir en su interior una especificación de formato por cada valor que se quiera insertar. La forma más simple de especificar un formato es mediante `%`*x*, es decir, usando el carácter fijo `%` seguido de una letra de código que indica el tipo de formato a aplicar. Algunos códigos de formato habituales son:

Código	Nemotécnico (*inglés*)	Tipo de valor
`d`	**d***ecimal*	entero
`f`	**f***ixed point*	real
`e`	**e***xponential*	real con notación exponencial
`g`	**g***eneral*	real con/sin notación exponencial
`c`	**c***haracter*	un carácter
`s`	**s***tring*	una cadena de caracteres

Por ejemplo, para cada una de las siguientes operaciones de escritura se obtiene el resultado que se muestra a su derecha:

Operación de escritura	Resultado
`printf( "%d", 120 / 12 );`	`10`
`printf( "Datos:%d#%d", 23*67, -50 );`	`Datos:1541#-50`
`printf( "Datos: %d # %d", 23*67, -50 );`	`Datos:·1541·#·-50`

Como se puede apreciar en los ejemplos estos formatos simples usan sólo el número de caracteres estrictamente necesarios para escribir el valor de cada dato, sin añadir espacios en blanco. Si se quiere separar con espacios unos valores de otros, entonces hay que incluirlos en el formato.

Otra forma de conseguir espacios en los resultados es indicar explícitamente cuántos caracteres debe ocupar el valor de cada dato escrito. Esto se hace poniendo el número de caracteres entre el símbolo de `%` y el código del formato. Ejemplos:

Operación de escritura	Resultado
`printf( "%5d", 120/12 );`	`···10`
`printf( "Datos:%7d#%5d", 23*6, -50 );`	`Datos:····138#··-50`
`printf( "%3d", 1000*34 );`	`34000`

Cuando el número de caracteres indicado es insuficiente para representar completamente el valor, como ocurre en el último ejemplo, se utilizan tantos caracteres como sean necesarios para que el resultado aparezca completo.

Además, cuando se utiliza un formato `f`, `e` ó `g` se puede especificar también el número de cifras decimales que se deben escribir después del punto decimal. Por ejemplo:

Operación de escritura	Resultado
`printf( "%10.3f", 1.2 );`	`·····1.200`
`printf( "%10.4e", 23.1*67.4 );`	`0.1557E+04`
`printf( "%15.3g", -50.6E-6 );`	`·····-0.506E-04`

Salvo que se indique otra cosa, los resultados obtenidos mediante sucesivas sentencias de escritura van apareciendo en el dispositivo de salida uno tras otro en la misma línea de texto. Por ejemplo, las siguientes sentencias de escritura:

```
printf( "Area = " );
printf( "%10.4f", 24.45 );
printf( "Mi ciudad es Avila" );
printf( "Descuento: " );
printf( "%5.2d", 12.5 );
printf( "%c", '%' );
```

producen el resultado siguiente (se prescinde ya del símbolo '`·`' para representar un espacio en blanco):

```
Area =    24.4500Mi ciudad es AvilaDescuento: 12.50%
```

Para escribir resultados en varias líneas de texto habrá que recordar que dentro de una cadena se pueden incluir caracteres especiales mediante secuencias de escape. Más concretamente, si queremos dar por terminada una línea de resultados y pasar a escribir en la siguiente bastará con incluir en el punto adecuado la secuencia de escape `\n`. Por ejemplo, si se modifican ligeramente las anteriores operaciones de escritura:

```
printf( "Area = " );
printf( "%10.4f\n", 24.45 );
printf( "Mi ciudad es Avila\n" );
printf( "Descuento: " );
printf( "%5.2d", 12.5 );
printf( "%c\n", '%' );
```

se obtendrá como resultado:

```
Area =    24.4500
Mi ciudad es Avila
Descuento: 12.50%
```

2.8 Estructura de un programa completo

Con los elementos introducidos hasta este momento pueden formarse ya programas completos. Sólo falta indicar cuál es la estructura global del programa.

Un programa en **C±** se engloba dentro de una estructura principal o `main()`. Un ejemplo de programa muy sencillo es el siguiente:

```
/** Programa: Hola */
/* Este programa escribe "Hola" */

#include <stdio.h>

int main() {
  printf( "Hola\n" );
}
```

Podemos observar que en el texto del programa aparece una línea precedida del símbolo #. Con este símbolo comienzan lo que se llaman *directivas para el compilador*. En concreto con la directiva `#include` se indica al compilador que utilice el módulo de librería `stdio` (cabecera `<stdio.h>`) para las operaciones de escritura que se realizarán en el programa. De hecho la directiva `#include` será la única que se usará en **C±**.

El cuerpo del programa contiene las sentencias ejecutables correspondientes a las acciones a realizar, escritas entre los símbolos `{` de comienzo y `}` final. Cada sentencia del programa termina con un punto y coma (`;`).

Es conveniente recordar que todos los programas en **C±** se deben guardar en un fichero con el nombre del programa y la extensión `.cpp`. Así, el nombre del fichero fuente de este programa deberá ser: `hola.cpp`.

Una vez compilado y ejecutado este programa de ejemplo produce, como es de esperar, el siguiente resultado:

```
Hola
```

En el anterior ejemplo de programa aparece también un nuevo elemento no mencionado hasta el momento, y que se explica a continuación.

2.8.1 Uso de comentarios

El código de un programa en un lenguaje de programación tal como C, C++ o **C±** puede no ser suficiente, en muchos casos, para comprender el sentido del programa. Casi siempre es conveniente alguna aclaración adicional que explique el significado exacto de los elementos usados para desarrollar el programa. Estas aclaraciones facilitan la labor de una posible modificación posterior del programa por nosotros mismos u otros programadores.

Todos los lenguajes permiten incluir dentro del texto del programa *comentarios* que faciliten su comprensión. Estos comentarios sirven sólo como documentación del programa fuente, y son ignorados por el compilador, en el sentido de que no pasan a formar parte del código objeto al que se traducirá el programa.

En **C±** los comentarios se incluyen dentro de los símbolos `/*` y `*/` . Por ejemplo:

```
/* ¡Ojo!. Esto es un comentario */
```

2.8.2 Descripción formal de la estructura de un programa

La descripción formal de la estructura (simplificada) de un programa es la siguiente:

Programa **::=** **{** *Include* **}** `int main()` *Bloque*

Include **::=** `#include` <*Nombre_módulo*`.h`>

Cada directiva debe ocupar una línea del programa ella sola. En los lenguajes C y C++ hay una gran variedad de directivas, pero en **C±** se utilizará casi exclusivamente la directiva `#include`, que sirve para indicar que el programa utilizará un determinado módulo de librería. El parámetro *Nombre_módulo* corresponde en realidad al nombre del fichero de cabecera (*header*) del módulo.

El resto del código del programa se podría repartir en líneas de código como se desee, aunque en el *Manual de Estilo* de **C±** se exigirá un estilo de presentación uniforme, tal como se irá indicando a medida que se introduzcan los elementos del lenguaje.

La estructura de un bloque solamente puede indicarse de momento de forma simplificada, ya que sólo se han visto los elementos mínimos necesarios para escribir un programa. Por ahora diremos que un bloque puede contener una secuencia de sentencias:

Bloque **::=** { *Parte_ejecutiva* }

Parte_ejecutiva **::=** **{** *Sentencia* **}**

La única sentencia que se ha descrito hasta el momento es la orden de escritura `printf`. Más adelante se irán introduciendo nuevas sentencias de **C±**.

2.9 Ejemplos de programas

En este apartado se muestran programas completos que pueden ser compilados y ejecutados de manera directa e inmediata. Aunque estos ejemplos son de una gran sencillez, permiten ilustrar los conceptos introducidos en este tema.

2.9.1 Escribir una fecha

En este ejemplo se escribe una fecha. El día y el año se escriben como valores numéricos, y el mes como un texto.

```
/** Programa: EscribirFecha */
/* Escribe la fecha del descubrimiento de América */

#include <stdio.h>

int main() {
  printf( "%2d", 12 );
  printf( " de Octubre de" );
  printf( "%5d\n", 1492 );
}
```

La ejecución del programa produce el siguiente resultado (usando el símbolo '·' para representar el espacio en blanco):

```
12·de·Octubre·de·1492
```

Obsérvese cómo se han incluido los espacios en blanco en las sentencias de escritura.

2.9.2 Suma de números consecutivos

Con este programa se trata de obtener la suma de una serie de números consecutivos, desde uno inicial a otro final. Para su obtención utilizaremos la fórmula de la suma de una progresión aritmética. Si n es el número de términos, a_1 el primer término, y a_n el último, la fórmula general de la suma es:

$$\sum_{i=1}^{n} a_i = n \times \frac{a_1 + a_n}{2}$$

Y para una serie de números enteros consecutivos se cumple que:

$$n = a_n - a_1 + 1$$

El listado del programa es el siguiente:

```
/** Programa: SumarNumeros */
/* Este programa calcula e imprime la suma
   de los números correlativos desde 4 hasta 45

   El algoritmo empleado es el utilizado para
   calcular la suma de una progresión aritmética:
   Suma = (Final - Inicial + 1) * (Inicial + Final) / 2
*/
#include <stdio.h>

int main() {
  printf( "La suma de los números desde 4 hasta 45\n" );
  printf( "es igual a: ");
  printf( "%5d\n",(45 - 4 + 1) * (45 + 4) / 2 );
}
```

La ejecución del programa produce el siguiente resultado:

```
La·suma·de·los·números·desde·4·hasta·45
es·igual·a:··1029
```

Es interesante analizar la manera en que se evalúa la expresión aritmética, dado que / trunca el resultado. Si se hubiera escrito la expresión de la forma que sugiere la fórmula de la suma de la progresión aritmética, tal como se ha presentado anteriormente:

```
(45 - 4 + 1) * ( (45 + 4) / 2 )
```

se obtendría un resultado erróneo igual a 1008, al producirse el truncamiento de la división por 2 antes de multiplicar.

2.9.3 Área y volumen de un cilindro

En este tercer programa sencillo se obtienen el área y el volumen de un cilindro a partir de su radio R y su altura A:

$$\acute{a}rea \;=\; 2\pi R^2 + 2\pi RA \;=\; 2\pi R(R+A)$$

$$volumen \;=\; \pi R^2 A$$

El listado del programa es el siguiente:

```
/** Programa: Cilindro */
/* Cálculo del área y el volumen de un cilindro */

#include <stdio.h>

int main() {
  printf( "%s\n", "Dado un cilindro de dimensiones:" );
  printf( "%s\n", "radio = 1,5 y altura = 5,6" );
  printf( "%s", "su area es igual a: " );
  printf( "%g\n", 2.0*3.141592*1.5*(1.5+5.6) );
  printf( "%s", "y su volumen es igual a:" );
  printf( "%20.8f\n", 3.141592*1.5*1.5*5.6 );
}
```

La ejecución del programa produce el siguiente resultado:

```
Dado un cilindro de dimensiones:
radio = 1,5 y altura = 5,6
su área es igual a: 66.9159
y su volumen es igual a:     39.58405920
```

En este ejemplo se ha renunciado a marcar expresamente los espacios en blanco impresos por el programa, para facilitar la lectura de los resultados.