2 Bachillerato

Algoritmos y programación en lenguaje C

David Arboledas Brihuega
Pilar Sobrino de Toro

educàlia editorial

2 Bachillerato

Algoritmos y programación en lenguaje C

David Arboledas Brihuega
Pilar Sobrino de Toro

Primera edición, 2018

Autores: David Arboledas Brihuega y Pilar Sobrino de Toro

Maquetación: Raquel Garzón Montagut

Edita: Educàlia Editorial

Imprime: Grupo Digital 82, S.L.

ISBN: 978-84-17493-03-5

Depósito legal: V-1130-2018

Printed in Spain/Impreso en España.

Todos los derechos reservados. No está permitida la reimpresión de ninguna parte de este libro, ni de imágenes ni de texto, ni tampoco su reproducción, ni utilización, en cualquier forma o por cualquier medio, bien sea electrónico, mecánico o de otro modo, tanto conocida como los que puedan inventarse, incluyendo el fotocopiado o grabación, ni está permitido almacenarlo en un sistema de información y recuperación, sin el permiso anticipado y por escrito del editor.

Alguna de las imágenes que incluye este libro son reproducciones que se han realizado acogiéndose al derecho de cita que aparece en el artículo 32 de la Ley 22/1987, del 11 de noviembre, de la Propiedad intelectual. Educàlia Editorial agradece a todas las instituciones, tanto públicas como privadas, citadas en estas páginas, su colaboración y pide disculpas por la posible omisión involuntaria de algunas de ellas.

Educàlia Editorial

Avda de les Jacarandes 2 loft 327 46100 Burjassot-València

Tel. 960 624 309 - 963 768 542 - 610 900 111

Email: educaliaeditorial@e-ducalia.com

www.e-ducalia.com

ÍNDICE

Introducción 7

UNIDAD 1 INTRODUCCIÓN AL LENGUAJE C 9

- 1.1. Conceptos 9
- 1.2. Lenguajes de programación 10
- 1.3. Algoritmos 11
  - 1.3.1. Cómo escribir un algoritmo 13
  - 1.3.2. Estructuras básicas en un algoritmo 16
- 1.4. El lenguaje C 17
- 1.5. Del IDE a la aplicación 19
- 1.6. Elementos de un programa en C 20
  - 1.6.1. Identificadores 20
  - 1.6.2. Tipos de datos 21
  - 1.6.3. Variables 24
  - 1.6.4. Constantes y macros 25
  - 1.6.5. Palabras reservadas 26
  - 1.6.6. Comentarios 26
- 1.7. Entrada y salida estándar 27
  - 1.7.1. Salida de datos 27
  - 1.7.2. Entrada de datos 30
- 1.8. El primer programa 31
- Resumen 32

UNIDAD 2 LA ARITMÉTICA DE C 33

- 2.1. Iniciación y asignación de variables 33
- 2.2. Operadores aritméticos 34
  - 2.2.1. Prioridad de los operadores aritméticos 35
  - 2.2.2. Otros operadores de asignación 36
  - 2.2.3. Operadores de incremento y decremento 37
- 2.3. Operadores relacionales 37
- 2.4. Operadores lógicos 38
- 2.5. Operador condicional 39
- 2.6. El operador `sizeof` 39
- 2.7. Conversión de tipos 40
  - 2.7.1. Conversión implícita 40
  - 2.7.2. Conversión explícita 40
- 2.8. Prioridad de operadores 41
- Resumen 42

UNIDAD 3 PROGRAMACIÓN ESTRUCTURADA......43
3.1. Estructura de un programa en C......43
3.1.1. Directivas del compilador......45
3.1.2. Declaraciones globales......46
3.1.3. La función principal......46
3.2. Estructuras de control......47
3.2.1. Estructuras de control secuencial......47
Resumen......53
UNIDAD 4 ESTRUCTURAS DE......55
CONTROL SELECTIVAS O CONDICIONALES......55
4.1. Selección simple y doble......55
4.2. Selecciones anidadas......62
4.3. Selección múltiple......66
Resumen......72
UNIDAD 5 ESTRUCTURAS DE CONTROL REPETITIVAS......73
5.1. El bucle `for`......73
5.2. El bucle `while`......77
5.2.1. Bucles controlados por centinelas......79
5.2.2. Bucles controlados por indicadores......80
5.3. El bucle `do…while`......82
5.4. Sentencias de control en los bucles......85
5.4.1. Break......85
5.4.2. Continue......86
Resumen......88
UNIDAD 6 DATOS DE TIPO ESTRUCTURADO......89
6.1. Arreglos......89
6.1.1. Vectores o arreglos unidimensionales......89
6.1.2. Matrices o arreglos bidimensionales......94
6.2. Cadenas de caracteres......96
6.2.1. Funciones para el manejo de cadenas......99
6.2.1.1. Funciones de transcripción de cadenas......99
6.3. Estructuras......103
6.3.1. Arreglos de estructuras......106
6.4. Enumeraciones......108
6.5. Punteros......110
6.5.1. Iniciación de punteros......112
6.5.2. Aritmética de punteros......114
Resumen......115

UNIDAD 7 PROGRAMACIÓN MODULAR 117

7.1. Funciones 118

7.1.1. Declaración y estructura de una función 118

7.1.2. Prototipo de las funciones 119

7.1.3. Paso de parámetros 120

7.2. Ámbito y visibilidad 125

7.2.1. Variables globales y locales 125

7.2.2. Visibilidad 126

7.3. Funciones de la biblioteca estándar 127

7.3.1. Funciones de carácter 128

7.3.2. Funciones numéricas 128

7.3.3. Otras funciones para el tratamiento de cadenas .. 129

Resumen 130

INTRODUCCIÓN

Los ordenadores son capaces de hacer casi cualquier actividad que imaginemos con un reducido conjunto de instrucciones. Pero, ¿qué idioma habla un computador? La respuesta, a priori, es fácil: el lenguaje de máquina; el cómo interactúan con nosotros, más complicado, pues trasladar nuestra lengua natural materna a código máquina no es tan sencillo como podría parecer.

Un lenguaje de programación es un idioma formal e inventado diseñado para expresar procesos que pueden ser llevados a cabo por las computadoras. Está formado por un conjunto de símbolos, reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones.

C es un lenguaje de programación de propósito general que destaca por su economía de expresión, control de flujo y un rico conjunto de operadores. Por otro lado, este lenguaje no está destinado a ninguna área de aplicación, por lo que es efectivo para un gran número de tareas sin restricciones previas.

En sus comienzos, C fue diseñado para el sistema operativo UNIX y Dennis Ritchie (9 de septiembre de 1941 - 12 de octubre de 2011), su creador, lo implantó sobre dicho sistema en 1970 en la serie de minicomputadoras DEC PDP-11. El sistema operativo, el compilador de C y básicamente todas las aplicaciones de UNIX están escritos en C. No obstante, este lenguaje no está supeditado a ningún hardware en particular, por lo que cualquier programa en C puede correr en cualquier máquina. Con un poco de cuidado es sencillo escribir programas portables, es decir, programas que puedan ejecutarse sin cambios en toda una variedad de computadores.

Muchas de las ideas de C provienen del lenguaje BCPL (Basic Combined Programming Language), desarrollado por Martin Richards en 1966. La influencia de BCPL sobre C se continuó indirectamente a través del lenguaje B, escrito por Ken Thompson en 1969 para el primer sistema UNIX de la DEC PDP-7.

Mientras que BCPL y B son lenguajes no tipados, C proporciona una gran variedad de tipos de datos, como caracteres, enteros y números en coma flotante de varios tamaños. Además de toda una jerarquía de tipos de datos derivados creados con punteros, arreglos y estructuras. No obstante, C no proporciona operaciones para tratar directamente con objetos compuestos, como cadenas, arreglos o listas. Este hecho, que podría parecer como una grave carencia, permite que el lenguaje pueda tener un tamaño modesto, lo que le hace realmente atractivo para empezar a programar y aprender con rapidez.

La finalidad de este libro es enseñar al estudiante a aprender cómo programar un ordenador en C. Se supone que este ya posee conocimientos básicos del funcionamiento de un computador. Aun así, todos los conceptos se explican desde el punto de vista de una persona que comienza su andadura en el mundo de la programación. El método de aprendizaje se basa en el análisis de un problema y la descripción de los pasos necesarios para llegar a la solución. El objetivo prioritario es mostrar con detalle cómo crear un programa y cómo entenderlo.

Durante muchos años la definición de C fue el manual de referencia de la primera edición de *The C Programming Language*, escrito por Brian Kernighan y Dennis Ritchie en 1978. En 1983,

el Instituto Nacional Estadounidense de Estándares (ANSI) organizó un comité para establecer una especificación estándar de C. Tras un largo y arduo proceso se completó el estándar en 1989 y se ratificó como "Lenguaje de Programación C" ANSI X3.159-1989. Esta versión del lenguaje se conoce a menudo como ANSI C, o a veces, simplemente, como C89.

Tras el proceso de estandarización de ANSI C, la especificación del lenguaje permaneció relativamente estable durante algún tiempo, mientras que C++ (un C orientado a objetos) seguía evolucionando. Sin embargo, el estándar continuó bajo revisión, lo que llevó a la publicación en 1999 del conocido como estándar C99. Se adoptó como tal por la ANSI en marzo de 2000. El último estándar publicado para C es el C11, oficialmente ratificado y publicado el 8 de diciembre de 2011.

Todas los ejemplos del libro se han realizado con la versión 5.11 del entorno de desarrollo integrado Dev-C++. Se trata de un software libre para la plataforma Windows que usa MinGW, una implementación de los compiladores del proyecto GNU, como su propio compilador de C/C++ para la programación rápida de aplicaciones. Puedes descargar el proyecto sin ninguna restricción desde su página oficial *https://sourceforge.net/projects/orwelldevcpp/*.

Existen otras alternativas gratuitas a Dev-C++, como el entorno de desarrollo Code::-Blocks, *http://www.codeblocks.org/*, muy popular entre los nuevos programadores y con versiones para Windows y Linux.

Aunque se ha puesto todo el esmero posible en asegurar la perfecta ejecución del contenido en cualquier plataforma, los errores son propios de las personas, y en un libro intervienen muchas, por lo que las erratas existirán. Así pues, si encuentras algún error en el texto o en el código fuente, no dudes en ponerlo en conocimiento de los autores a través del correo *davarbri@hotmail.com*. De este modo, ahorrarás a futuros estudiantes importantes quebraderos de cabeza y permitirá su corrección en siguientes ediciones.

UNIDAD 1
INTRODUCCIÓN AL LENGUAJE C

Programar un ordenador consiste en diseñar, codificar, depurar y mantener el código fuente de un programa. El código fuente es el conjunto de instrucciones que se escribe en un determinado lenguaje de programación. El propósito de la programación es crear aplicaciones que muestren el comportamiento deseado en la fase de diseño. El proceso de escribir código requiere frecuentemente conocimientos en varias áreas distintas, además del dominio del lenguaje empleado, algoritmos especializados y lógica formal.

En este primer capítulo repasaremos los conceptos básicos de las ciencias de la computación para después describir los elementos de un programa escrito en lenguaje C, los datos e instrucciones de entrada y salida de los mismos.

1.1. Conceptos

Todo ordenador está formado por dos subsistemas complementarios y esenciales: el **hardware**, que constituye la parte física y tangible de la máquina y el **software**, que es su parte lógica, los programas, que dan las instrucciones necesarias al microprocesador para que ejecute las tareas deseadas.

Toda la electrónica de los ordenadores se fundamenta en componentes de estado sólido entre los que destacan los transistores. Esto hace que en su más bajo nivel los microprocesadores solo puedan interpretar estados altos y bajos de tensión. En otras palabras, es como si solo existieran interruptores, cada uno de los cuales únicamente pudiera encontrarse en uno de los dos estados siguientes: abierto o cerrado. Si a uno de esos niveles le identificamos con el 1, por ejemplo, al otro le corresponde el 0; es decir, que asociamos a los niveles eléctricos los dígitos binarios 0 o 1. Cada uno de estos dígitos se denomina en informática **bit**, acrónimo inglés de ***b**inary dig**it***.

Así pues, los ordenadores digitales solo pueden manejar estos estados discretos binarios a los que hemos denominado bit y representado por los dígitos 0 y 1. Pero, ¿cómo se consigue trasladar nuestro lenguaje natural al **código máquina** que lee e interpreta un ordenador? Esa es la pregunta clave. Hemos de ser capaces de traducir el algoritmo que resuelve un problema a **lenguaje de máquina** o **código máquina**, el único interpretable y ejecutable por un microprocesador.

En primer lugar, hemos de representar con cadenas de bits nuestro lenguaje natural, al igual que traducir después secuencias de bits a algo entendible por nosotros. En definitiva, tenemos que ser capaces de establecer comunicaciones no solo entre los ordenadores, sino también entre ellos y nosotros.

La información numérica es fácil de convertir del sistema decimal que utilizamos al sistema binario que manejan los ordenadores digitales. Observa la siguiente tabla:

1000100110000001101110011011100111011001000000010011001001100110000001101110011001100111001100100110001000110011
0111001001110011000001110000011001100111011001001100

Decimal	0	1	2	3	4	5	6	7	8	9
Binario	0	1	10	11	100	101	110	111	1000	1001

En ella hemos conseguido relacionar todos los dígitos que empleamos en nuestro mundo real con los del sistema binario. ¿Qué ocurre, entonces, con las letras del alfabeto? La pregunta también tuvo una fácil respuesta. Los ingenieros consiguieron desarrollar códigos alfanuméricos que relacionaban una combinación binaria específica con cada carácter. A nivel conceptual estos códigos no presentan ninguna dificultad, ya que por muy grande que sea el número de caracteres, siempre será finito, por lo que el único problema reside en consensuar con la comunidad informática qué combinación binaria se asigna a cada carácter. Por ejemplo, al usar el **código ASCII** (Código Estadounidense Estándar para el Intercambio de Información), el patrón `1010011` se corresponde con la letra S (mayúscula).

Para programar los ordenadores, por tanto, necesitamos asignar significados reales a los patrones de bits. No solo para los caracteres alfanuméricos, sino también para las instrucciones. Los ordenadores necesitan **datos**, pero también **instrucciones** y, de algún modo, tendremos que decirle: "toma los números enteros 4 y 5, súmalos y devuelve el resultado en el sistema decimal", por ejemplo. Esto, como puede imaginarse, ya no es tan sencillo, pues esos datos e instrucciones que acabamos de poner por escrito habrá de suministrárselos el programador en código máquina como una secuencia de bits. Para esto, precisamente, han ido surgiendo los diferentes lenguajes de programación a lo largo del tiempo.

1.2. Lenguajes de programación

Un lenguaje de programación es un lenguaje formal y artificial diseñado para expresar procesos que pueden ser llevados a cabo por las computadoras. Está formado por un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones

Como no podemos trabajar directamente en código máquina, los primeros intentos consistieron en implementar traductores para reemplazar los bits por palabras o abstracción de palabras del idioma inglés para formar un conjunto de mnemotécnicos que constituye el **lenguaje ensamblador**. Este lenguaje es de **bajo nivel** y específico de la arquitectura de la máquina o circuito que desea programarse, lo que le hace aún más complicado. Por ejemplo, para un procesador de arquitectura *x*86 la sentencia en lenguaje ensamblador `MOV AL, 037h`, asigna el valor hexadecimal 37 al registro `AL`.

El lenguaje ensamblador lee la sentencia de arriba y la traduce a su equivalente binario en lenguaje de máquina, que resulta ser `1011 0000 00110111`. El código máquina generado por el ensamblador consiste en dos *bytes*. El primero contiene empaquetado la instrucción `MOV` y el código del registro hacia donde se mueve el dato, mientras que el segundo byte especifica el número que se asignará al registro (Figura 1.1).

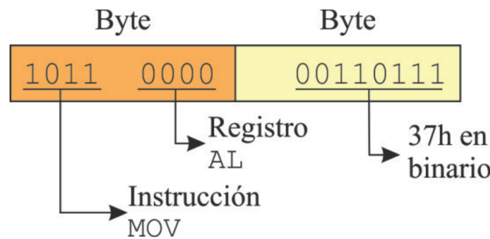


Figura 1.1. Código máquina de la instrucción ensamblador MOV AL, 037h

Al ser un lenguaje específico de los circuitos que se programan, rápidamente puede quedar obsoleto, pues los cambios en el hardware avanzan a un ritmo desorbitado y, con ellos, el conjunto de instrucciones necesarias. Por tanto, desde el punto de vista del diseño, lo que necesita un programador es disponer de un lenguaje que se pueda aplicar en distintas plataformas, independientemente de la arquitectura del microprocesador o del sistema operativo que aquel maneje. Este, en principio, es el objetivo que buscan desde finales de la década de 1950 los lenguajes de programación de alto nivel.

Un **lenguaje** de **alto nivel** se caracteriza porque se encuentra más cercano al lenguaje natural que hablamos los humanos que al lenguaje de máquina. Se trata de lenguajes independientes de la arquitectura del ordenador, por lo que en principio un programa escrito en un lenguaje de alto nivel se puede migrar a otra máquina sin grandes complicaciones. Permiten al programador olvidarse por completo del funcionamiento interno de la máquina para la que está implementando el programa.

Una vez escrito el programa, para poder ejecutarse, deberá traducirse a código máquina, bien mediante el proceso de **compilación** o de **interpretación**. Un **compilador** es un programa que permite traducir el código fuente de un programa escrito en un lenguaje de alto nivel, como lo es C, a lenguaje máquina. El resultado del proceso es un módulo denominado **objeto**. Un **intérprete**, por el contrario, es un programa capaz de realizar la traducción a medida que sea necesaria, típicamente instrucción por instrucción, hasta finalizar el programa. Perl, Python, Ruby y PHP son lenguajes interpretados de uso muy común.

1.3. Algoritmos

La resolución de cualquier problema mediante el uso de un ordenador requiere codificar un programa que lo resuelva. Los procesos necesarios para la creación de un programa se pueden resumir en cuatro puntos:

1. Especificación y análisis del problema en cuestión.
2. Diseño de un algoritmo que resuelva el problema.
3. Codificación del algoritmo en un lenguaje de programación.
4. Validación del programa.

Como recoge la Real Academia española de la lengua, un **algoritmo** es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema. La palabra algoritmo proviene de al-Juarismi, sobrenombre del matemático persa del siglo IX que introdujo nuestro sistema de numeración arábigo.

Las características fundamentales que debe tener todo algoritmo son cuatro:

- **Precisión**. Un algoritmo debe ser preciso, es decir, indicar el orden de realización de cada paso sin equívocos.
- **Definición**. El algoritmo debe estar definido, esto es, si se ejecuta varias veces partiendo de las mismas condiciones iniciales debe dar el mismo resultado.
- **Finitud**. Debe ser también finito, lo que significa que debe tener un número contable de pasos.
- **Independencia**. El algoritmo debe ser independiente del lenguaje de programación que se emplee para implementarlo.

En cualquier algoritmo se pueden distinguir tres partes: la **entrada** de datos o información sobre la que se van a efectuar las distintas operaciones, **procesamiento** o conjunto de operaciones que se realizan con los datos de entrada y **salida**, es decir, el resultado que debe proporcionar.

Toda actividad que realizamos la podemos expresar en forma de algoritmo. Existen dos tipos de algoritmos, los que se desarrollan para ser ejecutados por una computadora, llamados **algoritmos computacionales**, y los que realiza el ser humano, es decir, **algoritmos no computacionales**.

Algunos ejemplos clásicos en matemática son el algoritmo de la multiplicación, para calcular el producto, el algoritmo de la división para hallar el cociente de dos números, el algoritmo de Euclides para obtener el máximo común divisor de dos números naturales, o el método de Gauss para resolver un sistema de ecuaciones lineales.

Veamos cómo podríamos describir el algoritmo de Euclides para hallar el máximo común divisor (MCD) de dos números naturales:

- **Paso 1**. Tomar el número mayor como dividendo y el menor como divisor.
- **Paso 2**. Calcular el resto de la división entera de los dos números.
- **Paso 3**. Si el resto es igual a cero, entonces ir al paso 4. En caso contrario, tomar el divisor del paso 1 como nuevo dividendo y el resto del paso 2 como divisor y volver al paso anterior.
- **Paso 4**. El MCD es el divisor de la última división.

El algoritmo de Euclides es **definido**, pues prevé todas las situaciones posibles para el resto de la división (ser cero o diferente de cero); **no es ambiguo**, pues las acciones de dividir y comparar el resto con cero son claras y precisas, al igual que la obtención del número mayor; es **finito**, pues tiene cuatro pasos diferentes; y acaba en un tiempo finito cuando se cumple la condición del paso 3 que hace avanzar al paso 4, que es el punto de fin.

Como aplicación práctica, considera el caso en el que $A = 6$ y $B = 20$:

- **Paso 1**. Dividendo, $D = 20$; divisor, $d = 6$.
- **Paso 2**. Resto, $R = 2$ (20 mod 6).
- **Paso 3**. $D = 6$, $d = 2$.
- **Paso 2**. $R = 0$ (6 mod 2).
- **Paso 4**. MCD = 2

1.3.1. Cómo escribir un algoritmo

Un algoritmo debe escribirse sin ceñirse a las reglas de un lenguaje determinado. Existen dos formas básicas en algoritmia para describir las operaciones de las que consta un algoritmo:

- **Diagramas de flujo**. Son una representación gráfica de la lógica de un algoritmo mediante un conjunto de símbolos y líneas para indicar los pasos del algoritmo. Se emplea texto abreviado para describir las distintas tareas.
- **Pseudocódigo**. Se utilizan palabras clave para identificar las estructuras del algoritmo, como alternativas, repeticiones, etc. Es un lenguaje creado a medida por el programador.

1.3.1.1. Diagramas de flujo y pseudocódigo

Como acabas de leer, los diagramas de flujo son una representación gráfica de los pasos del algoritmo en los que se emplean símbolos normalizados. Los símbolos han sido reglamentados por el Instituto Nacional de Normalización Estadounidense (ANSI, American National Standards Institute) y los apreciamos en la Figura 1.2. Habitualmente, se utilizan solo cinco de ellos: **elipses** u **óvalos**, para indicar el inicio y el fin; **rectángulos**, para la asignación de un valor en memoria y/o la ejecución de una operación aritmética; **romboides**, para las operaciones de entrada y salida; **rombos** para las decisiones y **rectángulos con dos barras verticales** cerca de sus extremos para señalizar rutinas y subprogramas.

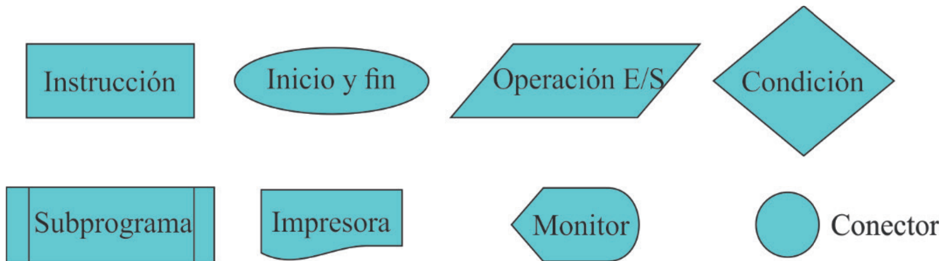


Figura 1.2. Símbolos normalizados habituales utilizados en los diagramas de flujo

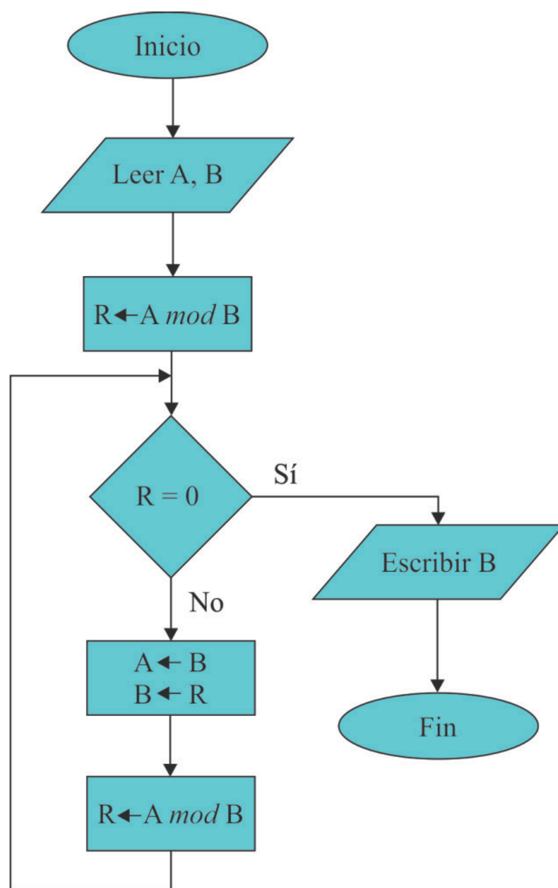
En un diagrama el **flujo del programa**, es decir, el orden en el que se ejecuta la lista de operaciones, se obtiene siguiendo las flechas que conectan los símbolos.

10001001100000110111001101110011101100100000001001100
0100110000001101110011001100111011001001100010001100011
0111001001100110000011100000110011001110110010011000

Las características más importantes de todo diagrama de flujo son las siguientes:

- Debe haber un inicio y un fin.
- Se deben trazar los símbolos de manera que se puedan leer de arriba hacia abajo y de izquierda a derecha.
- Solo deben emplearse líneas de flujo horizontales y/o verticales.
- Debe evitarse el cruce de líneas.
- No deben quedar líneas de flujo sin conectar.

Como ejemplo, un diagrama de flujo correspondiente al algoritmo de Euclides podría ser el siguiente:



Donde por A ← B se denota la asignación del valor B a la variable A.

El **pseudocódigo**, a diferencia de los diagramas de flujo, es una combinación del lenguaje natural, símbolos y términos utilizados dentro de la programación. Se creó para superar las

dos principales desventajas del diagrama de flujo: es lento de crear y difícil de modificar sin un nuevo dibujo. Por otra parte, el pseudocódigo es más fácil de utilizar, ya que es similar al lenguaje natural.

Cabe destacar que el pseudocódigo utiliza las convenciones estructurales de un lenguaje de programación real, pero está diseñado para la lectura humana en lugar de la lectura mediante máquina y es independiente de cualquier lenguaje de programación. Normalmente, el pseudocódigo omite detalles que no son esenciales para la comprensión humana del algoritmo, tales como declaraciones de variables, código específico del sistema y algunas subrutinas. El pseudocódigo se complementará, donde sea conveniente, con descripciones detalladas en lenguaje natural o con notación matemática compacta. Al contrario de los lenguajes de programación de alto nivel no tiene normas que definan con precisión lo que es y lo que no es pseudocódigo, por lo tanto varía de un programador a otro.

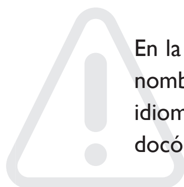
En ciencias de la computación se emplea mucho el pseudocódigo, pues es más fácil de entender que el código en un lenguaje de programación convencional. De este modo todos los programadores pueden entenderlo aunque no todos conozcan el mismo lenguaje de programación.

Un programador que tiene que aplicar un algoritmo específico generalmente comienza con una descripción en pseudocódigo y luego codifica esa descripción en un lenguaje de programación específico.

Tomando como base el algoritmo de Euclides para hallar el máximo común divisor de dos números naturales, el pseudocódigo correspondiente podría ser el siguiente:

```
PROGRAMA PRINCIPAL
INICIO
   A, B, R números enteros
   Leer A, B
   R ← A % B
   Mientras R != 0
      A ← B
      B ← R
      R ← A % B
   FinMientras
   Escribir B
FIN
```

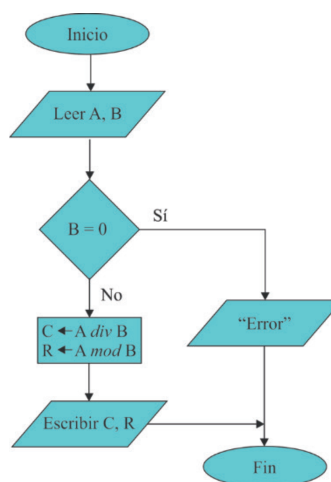
Como observas, el pseudocódigo es más parecido a un lenguaje de programación, más compacto incluso que un diagrama de flujo, aunque menos visual que este último.



En la actualidad, y por lo general, el pseudocódigo, como su nombre indica, no obedece a las reglas de sintaxis de ningún idioma en particular. Dependiendo del programador, el pseudocódigo puede variar mucho en su estilo.

EJEMPLO 1.1. Escribe el pseudocódigo y dibuja el diagrama de flujo correspondientes al algoritmo para hallar el cociente de dos números enteros y el resto de su división entera.

```
PRINCIPAL
INICIO
   A, B, C, R números
   enteros
   Leer A, B
   Si B != 0
      C ← A div B
      R ← A % B
      Escribir C, R
   SiNo
      Escribir "Error"
   FinSi
FIN
```



El algoritmo funciona del siguiente modo: tras leer los números correspondientes al dividendo (*A*) y divisor (*B*), se comprueba si este último es o no cero. Si es nulo, entonces se imprime la cadena "Error"; en caso contrario, se halla el cociente y el resto de la división entera y se muestra el resultado. El algoritmo presentado es definido, prevé todas las situaciones posibles para el divisor (ser cero o diferente de cero); no es ambiguo, pues las instrucciones son claras y precisas y es finito.

1.3.2. Estructuras básicas en un algoritmo

Un programa se puede definir como una secuencia ordenada de instrucciones cuyo propósito es realizar una determinada tarea. Por tanto, definimos el concepto de **flujo** de ejecución de un programa como el orden que siguen dichas instrucciones durante la ejecución del programa. En principio, el flujo de ejecución de un programa será el orden en el cual se escriban las instrucciones, lo que se conoce como ejecución **secuencial**. Sin embargo, este esquema de ejecución impone serias limitaciones, pues a menudo hay ciertos fragmentos de código que solo se deben ejecutar si se cumplen ciertas **condiciones** y otras tareas **repetitivas** que

hay que hacer un gran número de veces. Por ello, se han definido una serie de estructuras de programación que permiten crear programas cuyo flujo de ejecución sea más versátil que una ejecución secuencial.

Los tipos de estructuras básicas que se pueden emplear en un algoritmo son:

- **Secuenciales**. Constituidas por 1, 2 o *N* instrucciones que se ejecutan según el orden en el que han sido escritas. Es la estructura más simple y la pieza más básica a la hora de componer estructuras.
- **Selectivas**. Constan de una instrucción especial de decisión y de una o más secuencias de instrucciones. La sentencia de decisión genera un resultado delimitado dentro de un rango preseleccionado (generalmente verdadero o falso) y, dependiendo del resultado obtenido, se ejecuta o no la secuencia de instrucciones.
- **Iterativas**. Están formadas por una instrucción de decisión y de una secuencia. La instrucción de decisión genera dos tipos de resultado, como en el caso anterior, y la secuencia de instrucciones se ejecutará de modo reiterativo mientras que la instrucción de decisión genere el resultado verdadero; en caso contrario, finalizará la ejecución de la secuencia. Los bucles pueden tener la instrucción de decisión al principio o al final. Si la condición está al final, el bucle siempre se ejecuta al menos una vez.

Los diagramas de flujo para estas tres estructuras de control se muestran en la figura siguiente:

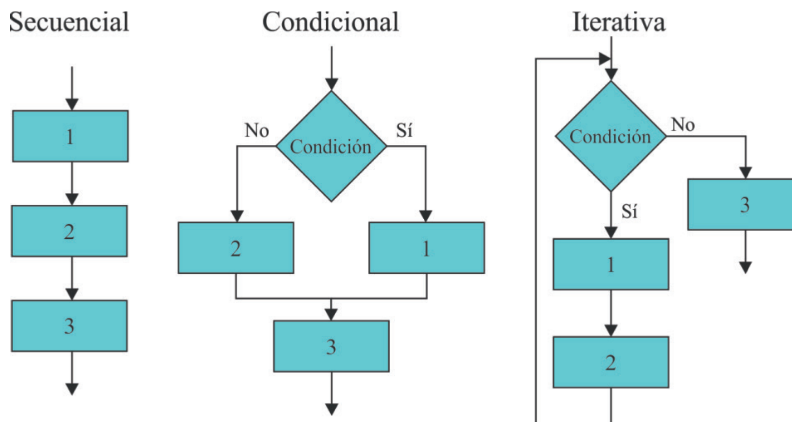


Figura 1.3. Diagramas de flujo de las distintas estructuras de control

1.4. El lenguaje C

El **lenguaje** de programación **C** fue creado por Brian Kernighan y Dennis Ritchie entre 1969 y 1972 en los Laboratorios Bell. La primera implementación del mismo la realizó Dennis Ritchie sobre un computador DEC PDP-11 con sistema operativo UNIX. C es el resultado de un proceso de desarrollo que comenzó con un lenguaje anterior, el BCPL, el cual influyó en el desarrollo por parte de Ken Thompson de un lenguaje llamado B, antecedente directo del lenguaje C.

Se trata de un lenguaje orientado a la **programación estructurada** y de **propósito general**, aunque muy dirigido a la implementación de sistemas operativos. Incluye lo más característico de los lenguajes de **alto nivel**, pero también dispone de construcciones del lenguaje que permiten un control a muy **bajo nivel**. Al romper con esta inflexibilidad de otros lenguajes es capaz de ofrecer un rendimiento que permite más posibilidades, lo que ha llevado a que haya mucha discusión sobre si este lenguaje no debería ocupar una categoría de **medio nivel**.

El lenguaje C, aunque con una curva de aprendizaje más alta que otros como Pascal, es una elección perfecta para aprender a programar por tres razones: C es un lenguaje con una estructura sencilla y tiene un reducido número de palabras reservadas. Además, y esto es lo más importante, enseña buenos hábitos de programación para enfrentarse después con cualquier otro lenguaje como C++, C# o Java.

Ilustremos su relativa sencillez con el tradicional "Hola mundo" con el que se presentan todos los lenguajes de programación. En un lenguaje como C++, sería algo así:

```
#include <iostream>
int main(void) {
   std::cout << "Hola mundo." << std::}l;
}
```

En Java, el resultado aún se complica un poco más:

```
public class HolaMundo {
   public static void main(String [] args) {
      System.out.println("Hola mundo.");
   }
}
```

Y en C# el grado de dificultad para un novato crece hasta casi lo inverosímil:

```
using System;
namespace HolaMundo
{
   class Hola
   {
      static void Main()
      {
         Console.WriteLine("Hola mundo");
      }
   }
}
```

¿Cómo se puede explicar a alguien que no ha visto un programa antes tal galimatías de órdenes? C va mucho más directo al grano:

```
#include <stdio.h>
main() {
   printf("Hola mundo.\n");
}
```

Naturalmente, este programa no es determinante a la hora de escoger un lenguaje como primera opción, pero sí debería suscitar una seria reflexión por parte de quien va a aprender su primer lenguaje y de quien va a enseñar a profanos.

1.5. Del IDE a la aplicación

Ya sabemos que cualquier lenguaje con características de alto nivel, como lo es C, permite a los programadores implementar aplicaciones dando instrucciones al microprocesador con una sintaxis más parecida a nuestro modo de hablar. Lógicamente, el microprocesador no entiende C, ni Pascal, ni Python, por tanto, el código fuente C deberá ser traducido a señales electrónicas digitales que sean interpretables por la máquina. Este proceso de **traducción** es lo que hemos denominamos **compilación** y produce, si no se encuentran errores, un módulo llamado **objeto**. Este módulo es la traducción a lenguaje de máquina del código fuente escrito por el programador, pero aún no es ejecutable. Es necesario someterlo, a su vez, a otro proceso conocido como **enlazado**, tras el cual se obtendría el programa ejecutable (Figura 1.4).

Cualquier entorno de desarrollo integrado para C de los ya comentados —Dev-C++ o Code::Blocks— emplea un compilador de C para producir su propio código objeto. Presentan, en una única aplicación, todas las herramientas y funciones que necesitarás para programar en C y obtener un ejecutable partiendo solo de la idea. Son herramientas para el desarrollo rápido de aplicaciones (RAD).

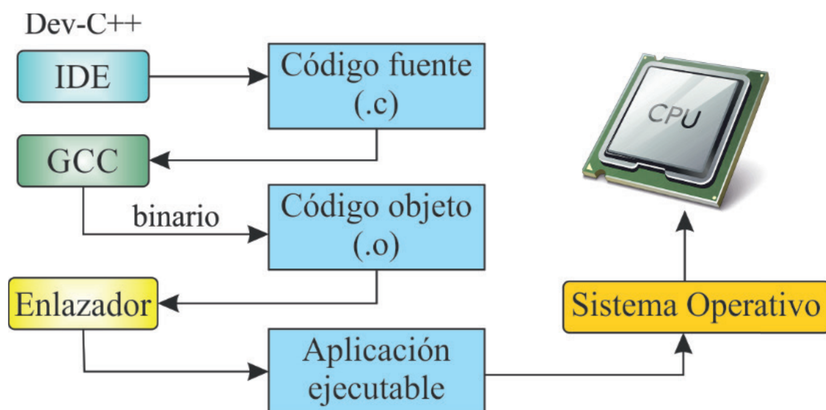


Figura 1.4. Del IDE a la CPU. Desarrollo completo de una aplicación

1.6. Elementos de un programa en C

La sintaxis de un lenguaje de programación es el conjunto de reglas que debemos seguir para que el compilador sea capaz de reconocer nuestro programa como un programa válido. C es un lenguaje de programación estructurado bastante explícito y más legible que otros, como viste en los ejemplos anteriores. Se caracteriza por no ser fuertemente tipado y por tener una rígida sintaxis, pero esto, lejos de ser un inconveniente para empezar, es una ventaja.

Un programa en C puede estar formado por diferentes **módulos** o fuentes. Los diferentes módulos se compilan de forma separada, para después ser **enlazados** o combinados entre ellos y con las bibliotecas necesarias para formar el programa en su versión ejecutable.

La estructura general de cualquier programa es similar:

- Una zona de **directivas del preprocesador**, que contiene la cláusula **`#include`**, seguida del nombre del archivo de cabecera que posee las declaraciones y definiciones necesarias para un propósito específico.
- Una sección **declarativa**, en la que se indica las declaraciones globales de los datos, sus tipos y las funciones que se van a emplear a lo largo del programa.
- El **cuerpo** del programa, constituido por las diferentes funciones que realizan las operaciones necesarias. La función principal es `main()`, de quien depende todo el control de la aplicación.

Así pues, todos los programas presentarán, como mínimo, la siguiente estructura general:

```
#include <stdio.h>
main()
{
   // Instrucciones
}
```

1.6.1. Identificadores

En cualquier programa siempre operan distintos elementos: unos creados por el programador, como variables, constantes y funciones; otros presentes en las bibliotecas del propio compilador. Cada uno de estos elementos necesita un nombre único y exclusivo para diferenciarse de otros empleados en el mismo programa. Estos nombres se conocen como **identificadores**.

Como programador, uno tiene plena libertad para nombrar como desee al programa, sus variables, constantes y funciones, siempre y cuando se atenga a las siguientes **restricciones** y **sugerencias**:

- Debe comenzar con una letra del alfabeto, mayúscula o minúscula o el guion bajo, _, y no puede contener espacios en blanco.
- El lenguaje C distingue entre mayúsculas y minúsculas.

- Tras el primer carácter del identificador pueden emplearse letras, números y el guion bajo.
- El número de caracteres de un identificador puede ser ilimitado, aunque algunos compiladores de C no reconocen más de 31.
- Los nombres deben de ser lo más compactos posibles, pero claramente descriptivos.
- No se pueden emplear palabras reservadas de C como identificadores.
- En lenguaje C es usual escribir variables en minúscula, dejando las mayúsculas para las constantes. En los casos de nombres compuestos se suele poner la inicial de la segunda palabra en mayúscula.

A continuación, una muestra de identificadores válidos:

x	nombreAlumno	Superficie_esfera	R2
Z1	_codigo	VOLUMEN	resultado_1

Ahora, unos cuantos ejemplos más de identificadores incorrectos y la razón por la que no son válidos en C:

Identificador	Explicación
2Radio	El primer carácter no puede ser un número, solo una letra o el guion bajo _
volumen@	@ no es un carácter permitido.
matricula coche	El espacio en blanco no se permite
codigo-alumno	El signo ortográfico guion no puede emplearse
"ave"	Las comillas no son caracteres válidos

1.6.2. Tipos de datos

Los ordenadores trabajan con dos tipos de entidades: **instrucciones** y **datos**. Los **datos** constituyen la representación simbólica de un atributo o variable cuantitativa. Describen hechos empíricos, sucesos y entidades, tales como las fechas, los salarios de los empleados o las notas de un examen. Representan la información que el programador manipula en la construcción de una solución o en el desarrollo de un algoritmo.

Por el contrario, las **instrucciones**, escritas como código en el programa, representan órdenes para la CPU del ordenador. A través de ellas, el programador le indica cómo operar con los datos y cómo interactuar con el resto de dispositivos del computador.

Todos los programas necesitan, entonces, procesar datos de distinta naturaleza. Dependiendo de su tipo, se representarán y almacenarán en memoria de forma específica, es decir, cada tipo de dato determina la cantidad de memoria RAM requerida para almacenarlo.

Las **variables** y las **constantes** son los objetos de datos básicos que se manipulan en un programa. Las declaraciones muestran las variables que se van a utilizar y establecen el

tipo que tienen y, algunas veces, cuáles son sus valores iniciales. Todos los datos tienen un tipo asociado con ellos que establece el conjunto de valores que puede tomar, así como las operaciones aplicables sobre esa variable.

Los datos que C puede utilizar se clasifican en **simples** y **compuestos**. Un dato **simple** es indivisible, no se puede descomponer. Un dato **compuesto** está formado por varios datos.

Los tipos de datos simples son **numéricos** (enteros y reales), **lógicos** y **caracteres**. Su uso es corriente en programación. Casi todos los lenguajes disponen de ellos, por lo que se les llama también tipos de datos **predefinidos** o **estándar**.

Hay **cinco** tipos de **datos básicos** en lenguaje C: entero (int), coma flotante (float), coma flotante con doble precisión (double), carácter (char) y sin valor (void). Ciertamente, es posible ampliar nuestro lenguaje con otros objetos construidos a la medida de los problemas que se planteen, aunque hemos de mencionar que no todos los compiladores de C son capaces de trabajar con datos de tipo lógico.

1.6.2.1. Enteros

El tipo entero se corresponde con el concepto de número entero que empleamos en matemáticas, pero atendiendo a su tamaño, se diferencian los siguientes seis tipos predefinidos: **`short`**, **`unsigned short`**, **`int`**, **`unsigned int`**, **`unsigned long`** y **`long`**.

Cada uno de estos tipos tiene un dominio determinado por el tamaño asignado en memoria. En C, el tamaño de los tipos de datos básicos puede variar de una plataforma a otra. Esta característica está detrás de buena parte de las críticas que recibe este lenguaje, pues de ella se derivan problemas de compatibilidad En principio, deben usarse los tipos que consuman menos espacio en memoria para acelerar el funcionamiento del programa y ocupar la menor cantidad de RAM durante su ejecución.

A continuación, te presentamos una tabla con los tipos de enteros predefinidos en C y su tamaño en memoria en una plataforma x64.

Tamaño en bytes	Tipo predefinido
2	`short, unsigned short`
4	`int, unsigned int, long, unsigned long`

Tabla 1.1. Tipos numéricos enteros predefinidos en C

1.6.2.2. Reales

Los números reales admiten la expresión ordinaria y la notación científica. Así, escribimos en C el número π en notación decimal como 3.1415926 y la carga del electrón, en notación científica, como −1.6E−19. Dependiendo de la precisión que se requiera, podrán emplearse distintos tipos. En general, estos son números en coma flotante de 4, 8 y 16 bytes en su expresión.

Los números reales permiten trabajar con magnitudes muy grandes o muy pequeñas respecto de la unidad, positivas o negativas. En cualquier caso, si se trabaja con reales de 32 y 64 bits, se consiguen entre 8 y 16 cifras significativas de precisión.

Las variables de tipo real realmente compatibles en C aparecen en la siguiente tabla:

Tamaño en bytes	Tipo predefinido
4	`float`
8	`double`
16	`long double`

Tabla 1.2. Tipos numéricos reales predefinidos en C

1.6.2.3. Lógicos

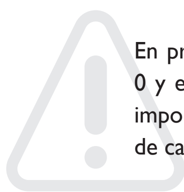
El **tipo booleano** es quizá la característica más importante de cualquier lenguaje de programación. Indica el valor verdadero o falso de una proposición cuando esta solo admite dos valores mutuamente excluyentes. En C, sin embargo, no existe el tipo lógico, pero se suele implementar con un número entero: 0 es **falso** y cualquier número diferente de cero es **verdadero**.

1.6.2.4. Carácter

El **tipo carácter** en C, **`char`**, tiene exactamente un tamaño de 1 byte y representa valores enteros en el rango −128 a +127. El lenguaje C proporciona el tipo `unsigned char` para plasmar valores de 0 a 255 y así representar todos los caracteres ASCII.

Los caracteres se almacenan internamente como números y por lo tanto se pueden realizar operaciones aritméticas con datos tipo char.

Para indicar al compilador un carácter, este se escribe entrecomillado, empleando para tal fin la comilla simple (') que aparece en el teclado junto con el signo de interrogación `?`, como en el ejemplo siguiente: `'a'` o `'A'`. También es posible indicar un carácter a través de su código ASCII. Por ejemplo, `97` sería para el compilador lo mismo que `'a'`.



En principio, aquellos caracteres con código ASCII entre el 0 y el 31 no son imprimibles, pero sí tienen un significado importante en programación, como salto de línea, retorno de carro o fin de fichero.

1.6.2.5. Tipo void

Son datos vacíos o sin valor. Cuando se utiliza como un tipo de valor devuelto de una función, la palabra clave **`void`** especifica que la función no devuelve ningún valor, como en `void`

`main()`. Cuando se utiliza para la lista de parámetros de una función, `void` especifica que la función no toma ningún parámetro, por ejemplo en `void main(void)`.

1.6.3. Variables

Para que un programa pueda ejecutarse, es necesario que los datos estén almacenados en memoria junto con las instrucciones. En muchas situaciones, los datos no son proporcionados por el programador, sino por el usuario del programa en tiempo de ejecución o mediante el procesamiento de otros datos.

Una variable es un identificador simbólico asociado a un espacio en memoria que contiene una cierta información que puede modificarse en el transcurso del programa.

Existen cuatro tipos diferentes de variables: **enteras**, **reales**, **caracteres** y **cadenas**. Una variable que es de un tipo dado solo puede tomar valores que correspondan a ese tipo. Si se intenta asignar un valor de tipo diferente se producirá un error.

C obliga a **declarar** las **variables** antes de que se puedan usar. Cada variable tiene por tanto asociado un tipo, un identificador y un valor inicial. Esta declaración tiene la forma:

```
tipo_dato identificador;
```

Puedes declarar varias variables del mismo tipo en la misma línea separándolas por comas; por ejemplo:

```
float largo, ancho, alto;
```

Una vez declarada cualquier variable, se puede hacer uso de ella en el programa a través del **operador de asignación** =. Por ejemplo, las siguientes son asignaciones válidas en C:

```
// declaraciones
char activo;
short longitud;
main() {
   // asignaciones con =
   activo = 1;
   longitud = 35;
}
```

Las variables pueden iniciarse en el momento de declararse; lo hacemos así:

```
short altura = 23;
```

El signo **=** solo se usa para asignar valores a las variables.

Recuerda que la memoria RAM es un conjunto de celdas direccionables, es decir, que se puede tener acceso a ellas mediante su dirección y en ellas se pueden almacenar datos.

Cuando se declara una variable, el programador no necesita conocer la dirección de la celda o celdas en las que se van a almacenar los datos de su programa, simplemente declara las variables y el compilador se encarga del resto.

En la siguiente figura se muestra, de manera descriptiva, cómo podría quedar la memoria tras haber declarado las variables `activo` y `longitud` anteriores:

0	1	2	3	4	5	6	7
		longitud	longitud		activo		

Figura 1.5. Reserva de memoria para dos variables, una de tipo entero corto y otra de carácter

Suponiendo que cada celda de la figura ocupase un byte, los espacios sombreados representarían la reserva de memoria que se ha hecho de acuerdo a la declaración de las variables. El identificador longitud ocupa las celdas 2 y 3 (dos bytes) y activo la celda 5 (1 byte). Date cuenta de que los espacios se han asignado de forma arbitraria, pues las variables no tienen que almacenarse de manera consecutiva en la memoria.

No lo habíamos mencionado, pero te habrás dado cuenta de que la variable siempre se encuentra en el lado izquierdo de la asignación y, en su lado derecho, el valor que almacenará. Por ejemplo, no podríamos asignar

```
0 = activo;
```

porque el compilador se detendría para mostrar un error: "`[Error] value required as left operand of assignment`".

1.6.4. Constantes y macros

Las **constantes** son valores que no pueden modificarse durante la ejecución de una aplicación. Se declaran al principio del programa mediante la palabra **`const`** o a través de la directiva del procesador **`#define`**:

```
#define identificador valor
const tipo identificador = valor;
```

Como norma, se estima que si un mismo objeto aparece más de tres veces en el programa debiera definirse como constante. Esta restricción mejora la lectura del código de un programa y permite modificar en un único punto del código fuente un valor que se utiliza en varias zonas del mismo.

Cualquier número entero, real, carácter o cadena puede ser una constante. Por ejemplo, son constantes válidas para C, las siguientes:

```
const float PI = 3.141592;
const char VOCAL = 'a';
const short INTENTOS = 3;
```

La directiva del procesador **`#define`** presenta una ventaja notoria sobre la cláusula **`const`**, pues nos permite definir macros en el programa. Una **macro** es la asociación de un

identificador o de un identificador parametrizado con una cadena de caracteres que tenga un significado coherente en el lenguaje. Esto significa que las constantes así definidas no necesitan ser explícitamente declaradas por tipo. El compilador decidirá a partir de su valor cuál es el más apropiado. Así, podríamos escribir:

```
#define        AVOGADRO       6.022E+23
#define        LIMITE         100
#define        IDE            "Dev-C++"
```

Antes de compilar, el preprocesador del compilador de C sustiturá en el archivo el identificador por su valor cada vez que aquel aparezca. Si la macro contiene argumentos, estos se cambian en el texto de reemplazo. Por ejemplo, la instrucción

```
#define suma(n1,n2) n1 + n2
```

hace que `suma(3,7)` se sustituya en el punto del programa donde se encuentre por el numero entero `10`.

1.6.5. Palabras reservadas

Las palabras reservadas de C son aquellas que ya tienen asignado un uso específico en el lenguaje. Con ellas se escriben las instrucciones de los programas, por lo que el programador no puede emplear identificadores con sus mismos nombres.

Las palabras reservadas en C, según recoge el estándar ANSI C89, son las siguientes:

```
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
```

1.6.6. Comentarios

En general, los **comentarios** son cadenas de texto que describen zonas del programa que el programador desea aclarar. Estos no forman parte del código fuente, sino que describen regiones del mismo, por lo que el compilador los obviará en el proceso de compilación.

Los comentarios generales pueden abarcar desde una única línea, para los que se suele usar la doble diagonal **//** precediendo al comentario, a varias líneas, encerrando el comentario entre los símbolos **/* */**. Por ejemplo:

```
// Este es un comentario de una línea
/* Y este un comentario
    que ocupa dos líneas */
```

1.7. Entrada y salida estándar

Un programa es un conjunto de instrucciones que el ordenador ejecuta con el fin de obtener un resultado o la solución de un problema determinado. Generalmente, lo datos dependen del propio usuario y es este el que tendrá que conocer los resultados del programa que esté empleando. Para ello, se necesita contar con instrucciones que permitan a los usuarios introducir datos y otras que muestren los resultados generados por los programas.

El lenguaje C cuenta con las funciones **printf()** y **scanf()** para la salida y entrada estándar de datos, respectivamente. Ambas están definidas en la biblioteca estándar de C —*stdio.h*—, de ahí que sea necesaria la sentencia #include <stdio.h>.

1.7.1. Salida de datos

El ordenador tiene diferentes formas para proporcionar la salida de datos, como archivos, impresoras o el propio monitor. Precisamente, C emplea la función **printf()** para imprimir un mensaje por pantalla. La función utiliza una *cadena de formato* que incluye las instrucciones para mezclar múltiples cadenas en la cadena final. Su sintaxis es la siguiente:

```
printf("cadena de formato",[argumentos]);
```

La **cadena de formato** contiene dos tipos de elementos: **caracteres ordinarios**, que se copian literalmente al flujo de salida, y **especificaciones de conversión**, cada uno de los cuales causa la conversión e impresión de los siguientes argumentos sucesivos de **printf**.

Cada especificación de conversión o **código de formato** comienza con el símbolo % e indica la posición dentro de la cadena en donde se imprimirá el siguiente argumento de la función. Los argumentos pueden ser cualquier tipo de expresión que represente un elemento imprimible en la salida estándar, por ejemplo, un número, una cadena, una constante o una variable.

A continuación, te mostramos algunos ejemplos sencillos con su salida:

printf("Hola");	Se visualiza la palabra Hola
printf("Hola, %s","David");	Se imprime Hola, David
printf("Numero: %d", 35);	La salida es Numero: 35

El código de formato debe tener la siguiente estructura:

```
%[parameter][flags][width][.precision][length]type
```

Cada uno de los nombres (*parameter*, *flags*, *width*, *precision*, *length* y *type*) representa un conjunto de valores posibles que se explican a continuación.

Parameter	**Descripción**
`n$`	Se reemplaza "n" por un número para cambiar el orden en el que se procesan los argumentos. Por ejemplo, `%3$d` se refiere al tercer argumento independientemente del lugar que ocupe en la cadena de formato.
Flags	**Descripción**
`número`	Rellena con espacios (o con ceros, ver siguiente *flag*) a la izquierda hasta el valor del número.
`0`	Se rellena con ceros a la izquierda hasta el valor dado por el *flag* anterior. Por ejemplo `%03d` imprime un número justificado con ceros hasta tres dígitos.
`+`	Imprimir el signo de un número.
`-`	Justifica el campo a la izquierda, por defecto se hace siempre a la derecha.
`#`	Formato alternativo. Para números en coma flotante se dejan ceros al final y se imprime siempre la coma. Para números que no están en base 10, se añade un prefijo denotando la base.
Width	**Descripción**
`número`	Tamaño del ancho del campo donde se imprimirá el valor.
`*`	Igual que el caso anterior, pero el número que se va a utilizar se pasa como parámetro justo antes del valor. Por ejemplo `printf("%*d", 5, 10)` imprime el número 10, pero con un ancho de cinco dígitos (es decir, rellenará con 3 espacios en blanco a la izquierda).
Precision	**Descripción**
`número`	Tamaño de la parte decimal para números reales. Número de caracteres en las cadenas de texto.
`*`	Igual que el caso anterior, pero el número se pasa como parámetro justo antes del valor. Por ejemplo, `printf("%.*s",3,"abc def")` imprime `abc`.
Length	**Descripción**
`hh`	Convertir variable de tipo char a entero e imprimir.
`h`	Convertir variable de tipo short a entero e imprimir.
`l`	Para enteros, se espera una variable de tipo long.
`ll`	Para enteros, se espera una variable de tipo long long.
`L`	Para reales, se espera una variable de tipo long double.
`z`	Para enteros, se espera un argumento de tipo `size_t`.
Type	**Descripción**
`%c`	Imprime el carácter ASCII correspondiente.
`%d, %i`	Conversión decimal con signo de un entero.
`%x, %X`	Conversión hexadecimal sin signo.
`%p`	Dirección de memoria (puntero).
`%e, %E`	Conversión a coma flotante con signo en notación científica.

`%f, %F`	Conversión a coma flotante con signo, usando punto decimal.
`%g, %G`	Conversión a coma flotante, usando la notación que requiera menor espacio.
`%o`	Conversión octal sin signo de un entero.
`%u`	Conversión decimal sin signo de un entero.
`%s`	Cadena de caracteres (terminada en ‘\0’).
`%%`	Imprime el símbolo %.

Ahora, una serie de ejemplos de uso de distintas especificaciones de conversión para la función `printf()` y el valor que se obtendría en la salida estándar:

`printf("%.3e", 1235.0);`	`1.235e+003`
`printf("%#x", 1310);`	`0x51e`
`printf("%.2f", M_PI);`	`3.14`
`printf("%.3s", "Diario");`	`Dia`

Si recordamos, los primeros 33 símbolos ASCII no son imprimibles, pero sí podemos utilizarlos para realizar una función concreta en la máquina. Entre estos caracteres se encuentran las **secuencias de escape**, que especifican acciones como retornos de carro y movimientos de tabulación en terminales e impresoras. También se utilizan para proporcionar representaciones literales de caracteres no imprimibles y de caracteres que normalmente tienen significados especiales, como las comillas dobles. Una secuencia de escape se considera un carácter individual y por tanto es válida como constante de caracteres.

Las secuencias de escape se escriben con una barra diagonal invertida (\) seguida de una letra o una combinación de dígitos con distinto significado, como muestra la siguiente tabla.

Secuencia de escape	**Representa**
`\a`	Campana (alerta)
`\b`	Retroceso
`\f`	Avance de página
`\n`	Nueva línea
`\r`	Retorno de carro
`\t`	Tabulación horizontal
`\v`	Tabulación vertical
`\’`	Comilla simple
`\”`	Comillas dobles

Secuencia de escape	Representa
`\\`	Barra diagonal invertida
`\?`	Signo de interrogación literal
`\ooo`	Carácter ASCII en notación octal
`\xhh`	Carácter ASCII en notación hexadecimal
`\xhhhh`	Carácter Unicode en notación hexadecimal si esta secuencia de escape se utiliza en una constante de caracteres anchos o un literal de cadena Unicode.

Tabla 1.3. Secuencias de escape ANSI

1.7.2. Entrada de datos

De la misma manera que damos salida a los datos desde un programa, también tenemos la opción de poder introducirlos desde la entrada estándar, el teclado. Para este caso C nos brinda la función **`scanf()`**, cuya sintaxis es la siguiente:

```
scanf("cadena de formato", &identificador_variable);
```

La función lee caracteres de la entrada estándar, los interpreta de acuerdo con las especificaciones que están en su formato y almacena los resultados a través de los argumentos restantes. El segundo argumento es un apuntador que indica dónde deberá almacenarse la entrada correspondientemente convertida. `scanf()` se detiene cuando termina con su cadena de formato, o cuando alguna entrada no coincide con la especificación de control.

La cadena de formato consisten en el carácter %, un número optativo que especifica el ancho máximo de campo y un carácter de conversión. Los caracteres de conversión se muestran en la siguiente tabla.

Carácter	Tipo de entrada esperada
`c`	Carácter de un byte.
`d`	Entero decimal.
`i`	Entero. Puede estar en hexadecimal o en octal.
`o`	Entero octal.
`p`	Dirección de puntero en dígitos hexadecimales.
`u`	Entero decimal sin signo.

Carácter	Tipo de entrada esperada
`x`	Entero hexadecimal.
`e,f, g`	Valor de coma flotante.
`s`	Cadena, no entrecomillada, hasta el primer carácter de espacio en blanco (espacio, tabulación o nueva línea).

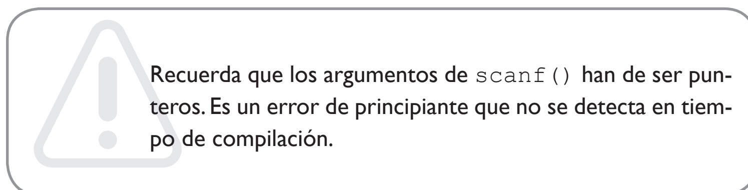
Tabla 1.4. Conversiones básicas de scanf()

Los caracteres de conversión `d`, `i`, `o`, `u`, `x` pueden ser precedidos por `h` para indicar que en la lista de argumentos aparece un apuntador a short en lugar de a int, o por l (letra ele) para indicar que aparece un apuntador a long en la lista de argumentos. En forma semejante, los caracteres de conversión `e`, `f`, `g` pueden ser precedidos por la letra l para indicar que hay un apuntador a double en lugar de a float en la lista de argumentos.

Supongamos que deseamos recoger por teclado fechas de nacimiento de la forma `12 Enero 1971`, entonces, la cláusula `scanf` podría ser:

```
scanf("%d %s %d", &dia, mes, &anio);
```

No se emplea `&` con `mes` puesto que un nombre de arreglo es ya un apuntador por sí mismo.



Recuerda que los argumentos de `scanf()` han de ser punteros. Es un error de principiante que no se detecta en tiempo de compilación.

1.8. El primer programa

Asumimos que ya tienes instalado en tu ordenador un IDE como Dev-C++ o Code::Blocks. Abre un nuevo archivo en el editor, Ctrl + N en Dev-C++, y copia el código fuente del siguiente ejemplo.

EJEMPLO 1.2. Imprime en pantalla tu nombre y edad

```
#include <stdio.h>

void main() {
      char nombre[20];
      unsigned short edad;

      printf("Escribe tu nombre: ");
      scanf("%s", nombre);
      printf("\nAhora tu edad: ");
      scanf("%hd", &edad);
      printf("\nHola, %s. Tu edad es %d", nombre, edad);
}
```

En este programa se han declarado dos variables: `nombre` y `edad`. La función `printf()` se emplea para solicitar por pantalla el nombre y la edad del usuario, que se almacenarán en las variables homónimas. Por último, `printf()` escribe la información suministrada por el usuario en la salida estándar.

Graba el código fuente como *ejemplo1.2* con la extensión *.c* —para indicar que está escrito en lenguaje C— y compila y ejecuta el prormama con **[F11]**. Verás una serie de mensajes en la **Ventana de mensajes** y, si todo ha ido bien, aparecerá una ventana en modo consola con su ejecución. Prueba su funcionamiento.

RESUMEN

Un ordenador es una máquina que procesa datos e instrucciones en código máquina. El programador, sin embargo, puede emplear distintos lenguajes de programación para comunicarse con el microprocesador: lenguajes de alto nivel, como C, o de bajo nivel, como ensamblador.

El lenguaje que vamos a estudiar a lo largo del curso es C, un lenguaje de propósito general que utilizaremos desde algún entorno de desarrollo integrado (IDE). Como otros lenguajes, emplea identificadores, variables, constantes y comentarios.

Los programas en C tienen una o más directivas `#include` al principio del código fuente. Estas directivas proporcionan información adicional para crear el programa, como son las funciones y macros definidas en los diferentes archivos de su biblioteca. Cada programa debe incluir al menos la función `main()`, aunque lo normal será que tenga varias más.

Los nombres de los identificadores empleados por el programador deben comenzar por un carácter alfabético, seguido de otros más, un número o el guion bajo. El compilador normalmente ignora los caracteres posteriores al 32.

Los tipos de datos básicos en C son: enteros (**`int`**), enteros largos (**`long`**), carácter (**`char`**) y reales en coma flotante (**`float`** y **`double`**). Cada uno de los tipos enteros tiene un calificador **`unsigned`** para almacenar solo valores positivos.

Los tipos de datos **`char`** emplean 1 byte de memoria, el tipo **`int`** 2 bytes, el entero largo, 4 bytes y los reales 4, 8 o 10 bytes de almacenamiento.

La instrucción más usada para leer datos por la entrada estándar —teclado— es `scanf`. Para imprimir los datos en el monitor se utiliza la función `printf()`, que permite hacerlo con distintos tipos de formato según las especificaciones de conversión.