

# Introducción a la Programación



## CONTENIDOS

1. Ordenadores
2. Codificación de la información
3. Programas y lenguajes de programación
  - 3.1. Código máquina
  - 3.2. Lenguaje ensamblador
  - 3.3. ¿Un programa diferente para cada ordenador?
  - 3.4. Lenguajes de programación de alto nivel
  - 3.5. Compiladores e intérpretes
  - 3.6. Lenguaje C
4. Más allá de los programas: algoritmos
5. Ejecutar aplicaciones en C
  - 5.1. Prueba de uso de la aplicación Calculadora de Pago de Automóviles
  - 5.2. Prueba de funcionamiento de Adivina el número
6. Referencias

El objetivo de la asignatura de Programación es enseñarte a *programar*, esto es, a diseñar *algoritmos* y expresarlos como *programas* escritos en un *lenguaje de programación* para poder *ejecutarlos* en un *ordenador*.

Seis términos técnicos en el primer párrafo. No está mal. Vayamos paso a paso: empezaremos por presentar en qué consiste, básicamente, un ordenador.

## 1. Ordenadores

El diccionario de la Real Academia define ordenador electrónico como

*"Máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información, capaz de resolver problemas matemáticos y lógicos mediante la utilización automática de programas informáticos".*

La propia definición nos da indicaciones acerca de algunos elementos básicos del ordenador:

1. La memoria, y
2. algún dispositivo capaz de efectuar cálculos matemáticos y lógicos.

La **memoria** es un gran almacén de información. En la memoria almacenamos todo tipo de datos: valores numéricos, textos, imágenes, etc. El dispositivo encargado de efectuar operaciones matemáticas y lógicas, que recibe el nombre de *Unidad Aritmético-Lógica (UAL)*, es como una calculadora capaz de trabajar con esos datos y producir, a partir de ellos, nuevos datos (el resultado de las operaciones). Otro dispositivo se encarga de transportar la información de la memoria a la UAL, de controlar a la UAL para que efectúe las operaciones pertinentes y de depositar los resultados en la memoria: la *Unidad de Control*. El conjunto que forman la Unidad de Control y la UAL se conoce como *Unidad Central de Proceso* (o CPU, del inglés "*Central Processing Unit*").

Podemos imaginar la memoria como un armario enorme con cajones numerados y la CPU como una persona que, equipada con una calculadora (la UAL), es capaz de buscar operandos en la memoria, efectuar cálculos con ellos y dejar los resultados en la memoria.





$$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 =$$

$$= 8 + 2 + 1 = 11$$

El bit que se encuentra más a la izquierda recibe el nombre de "*bit más significativo*" y el bit que se encuentra más a la derecha se denomina "*bit menos significativo*".

## Ejercicios

1. ¿Cuál es el máximo valor que puede representarse con 16 bits y un sistema de representación posicional como el descrito? ¿Qué secuencia de bits le corresponde?
2. ¿Cuántos bits se necesitan para representar los números del 0 al 18, ambos inclusive?

El sistema posicional es especialmente adecuado para efectuar ciertas operaciones aritméticas. Tomemos por caso la suma. Hay una "tabla de sumar" en binario que te mostramos a continuación:

sumandos	suma	acarreo
0	0	0
0	1	0
1	0	0
1	1	1

El **acarreo no nulo** indica que un dígito no es suficiente para expresar la suma de dos bits y que debe añadirse el valor uno al bit que ocupa una posición más a la izquierda. Para ilustrar la sencillez de la adición en el sistema posicional, hagamos una suma de dos números de 8 bits usando esta tabla. En este ejemplo sumamos los valores 11 y 3 en su representación binaria:

$$00001011 + 00000011$$

Empezamos por los bits menos significativos. Según la tabla, la suma 1 y 1 da 0 con acarreo 1:

$$\begin{array}{r} \text{Acarreo} \quad 1 \\ \quad 00001011 \\ + \quad 00000011 \\ \hline \quad 0 \end{array}$$

El segundo dígito empezando por derecha toma el valor que resulta de sumar a 1 y 1 el acarreo que arrastramos. O sea, 1 y 1 es 0 con acarreo 1, pero al sumar el acarreo que arrastramos de la anterior suma de bits, el resultado final es 1 con acarreo 1:

$$\begin{array}{r} \text{Acarreo} \quad 11 \\ \quad 00001011 \\ + \quad 00000011 \\ \hline \quad 10 \end{array}$$

Ya te habrás hecho una idea de la sencillez del método. De hecho, ya lo conoces bien, pues el sistema de numeración que aprendiste en tu infancia es también posicional, sólo que usando diez dígitos diferentes en lugar de dos, así que el procedimiento de suma es esencialmente idéntico. He aquí el resultado final, que es la secuencia de bits 00001110, o sea, el valor 14:

$$\begin{array}{r} \text{Acarreo} \quad 11 \\ \quad 00001011 \\ + \quad 00000011 \\ \hline \quad 00001110 \end{array}$$

La circuitería electrónica necesaria para implementar un sumador que actúe como el descrito es extremadamente sencilla.

## Ejercicios

3. Calcula las siguientes sumas de números codificados con 8 bits en el sistema posicional:

- a. 01111111 + 00000001
- b. 01010101 + 10101010
- c. 00000011 + 00000001

Debes tener en cuenta que la suma de dos números de 8 bits puede proporcionar una cantidad que requiera 9 bits. Suma, por ejemplo, las cantidades 255 (en binario de 8 bits es 11111111) y 1 (que en binario es 00000001):

```
Acarreo      1111111
              1111111
            + 0000001
            -----
            (1)0000000
```

El resultado es la cantidad 256, que en binario se expresa con 9 bits, no con 8. Decimos en este caso que la suma ha producido un *desbordamiento* (*overflow*, en inglés). Esta anomalía debe ser tenida en cuenta cuando se usa o se programa un ordenador.

Hasta el momento hemos visto cómo codificar valores positivos. ¿Podemos representar también cantidades negativas? La respuesta es sí. Consideremos brevemente tres formas de hacerlo. La primera es muy intuitiva: consiste en utilizar el bit más significativo para codificar el signo; si vale 0, por ejemplo, el número expresado con los restantes bits es positivo (con la representación posicional que ya conoces), y si vale 1, es negativo. Por ejemplo, el valor 00000010 es 2 y el valor 10000010 es -2. Efectuar sumas con valores positivos y negativos resulta relativamente complicado si codificamos así el signo de un número. Esta mayor complicación se traslada también a la circuitería necesaria. Mal asunto.

Una forma alternativa de codificar cantidades positivas y negativas es el denominado **complemento a uno**. Consiste en lo siguiente: se toma la representación posicional de un número (que debe poder expresarse con 7 bits) y se invierten todos sus bits si es negativo. La suma de números codificados así es relativamente sencilla: se efectúa la suma convencional y, si no se ha producido un desbordamiento, el resultado es el valor que se deseaba calcular; pero si se produce un desbordamiento, la solución se obtiene sumando el valor 1 al resultado de la suma (sin tener en cuenta ya el bit desbordado). Veámoslo con un ejemplo. Sumemos el valor 3 al valor -2 en complemento a uno:

```
Acarreo      1111111
              0000011
            + 1111101
            -----
            (1)0000000
              ↓
              0000000
            + 0000001
            -----
              0000001
```

La primera suma ha producido un desbordamiento. El resultado correcto resulta de sumar una unidad a los 8 primeros bits. La codificación en complemento a uno tiene algunas desventajas. Una de ellas es que hay dos formas de codificar el valor 0 (con 8 bits, por ejemplo, tanto 00000000 como 11111111 representan el valor 0) y, por tanto, sólo podemos representar 255 valores ([-127,127]), en lugar de 256. Pero el principal inconveniente es la **lentitud** con que se realizan operaciones como la suma: cuando se produce un desbordamiento se han de efectuar dos adiciones, es decir, se ha de invertir el doble de tiempo.

Una codificación alternativa (y que es la utilizada en los ordenadores) es la denominada **complemento a dos**. Para cambiar el signo a un número hemos de invertir todos sus bits y *sumar 1 al resultado*. Esta codificación, que parece poco natural, tiene las ventajas de que sólo hay una forma de representar el valor nulo (el rango de valores representados es [-128,127]) y, principalmente, de que una sola operación de suma basta para obtener el resultado correcto de una adición. Repitamos el ejemplo anterior. Sumemos 3 y -2, pero en complemento a dos:

```
Acarreo      111111
              0000011
            + 1111110
            -----
            (1)0000001
```

Si ignoramos el bit desbordado, el resultado es correcto.

## Ejercicios

4. Codifica en complemento a dos de 8 bits los siguientes valores:

- a) 4                      b) -4                      c) 0                      d) 127                      e) 1                      f) -1

5. Efectúa las siguientes sumas y restas en complemento a dos de 8 bits:

- a) 4 + 4                      b) -4 + 3                      c) 127 - 128                      d) 128 - 127                      e) 1 - 1                      f) 1 - 2

---

Ya hemos hablado bastante acerca de cómo codificar números (aunque más adelante ofreceremos alguna reflexión acerca de cómo representar valores con parte decimal). Preocupémonos por un instante acerca de cómo **representar texto**. Hay una tabla que pone en correspondencia 127 símbolos con secuencias de bits y que se ha asumido como estándar. Es la denominada tabla ASCII, cuyo nombre son las siglas de "*American Standard Code for Information Interchange*". La correspondencia entre secuencias de bits y caracteres determinada por la tabla es arbitraria, pero aceptada como estándar. La letra "a", por ejemplo, se codifica con la secuencia de bits 01100001 y la letra "A" se codifica con la secuencia de bits 01000001. En consecuencia, un texto se puede codificar, como una secuencia de bits. Aquí tienes el texto "Hola" codificado con la tabla ASCII:

01001000 01101111 01101100 01100001

Sin embargo, cuando vemos ese texto en pantalla, no vemos una secuencia de bits, sino la letra "H", seguida de la letra "o",... Lo que realmente vemos es un gráfico, un patrón de píxeles almacenado en la memoria del ordenador y que se muestra en la pantalla. Un bit de valor 0 puede mostrarse como color blanco y un bit de valor 1 como color negro. La letra "H" que ves en pantalla, por ejemplo, es la visualización de este patrón de bits:

01000010  
01000010  
01000010  
01111110  
01000010  
01000010  
01000010

En la memoria del ordenador se dispone de un patrón de bits para cada carácter (la realidad es cada vez más compleja; los sistemas más modernos almacenan los caracteres en memoria de otra forma, pero hablar de ello supone desviarnos mucho de lo que queremos contar). Cuando se detecta el código ASCII 01001000, se muestra en pantalla el patrón de bits correspondiente a la representación gráfica de la "H". Truculento, pero eficaz.

No sólo podemos representar caracteres con patrones de píxeles: todos los gráficos de ordenador son simples patrones de píxeles dispuestos como una matriz.

Como puedes ver, basta con ceros y unos para codificar la información que manejamos en un ordenador: números, texto, imágenes, etc.

## 3. Programas y lenguajes de programación

Antes de detenernos a hablar de la codificación de la información estábamos comentando que la memoria es un gran almacén con cajones numerados, es decir, identificables con valores numéricos: sus respectivas direcciones. En cada cajón se almacena una secuencia de bits de tamaño fijo. La CPU, el "cerebro" del ordenador, es capaz de ejecutar acciones especificadas mediante secuencias de *instrucciones*. Una instrucción describe una acción muy simple, del estilo de "suma esto con aquello", "multiplica las cantidades que hay en tal y cual posición de memoria", "deja el resultado en tal dirección de memoria", "haz una copia del dato de esta dirección en esta otra dirección", "averigua si la cantidad almacenada en determinada dirección es negativa", etc. Las instrucciones se representan mediante combinaciones particulares de unos y ceros (valores binarios) y, por tanto, se pueden almacenar en la memoria.

Combinando hábilmente inteligentemente las **instrucciones** en una secuencia podemos hacer que la CPU ejecute cálculos más complejos. Una **secuencia de instrucciones** es un *programa*. Si hay una instrucción para multiplicar pero ninguna para elevar un número al cubo, podemos construir un programa que efectúe este último cálculo a partir de las instrucciones disponibles. He aquí, grosso modo, una secuencia de instrucciones que calcula el cubo a partir de productos:

1. Toma el número y multiplícalo por sí mismo.
2. Multiplica el resultado de la última operación por el número original.

Las secuencias de instrucciones que el ordenador puede ejecutar reciben el nombre de **programas en código máquina**, porque el *lenguaje de programación* en el que están expresadas recibe el nombre de **código máquina**. Un **lenguaje de programación** es cualquier sistema de notación que permite expresar programas.

### 3.1. Código máquina

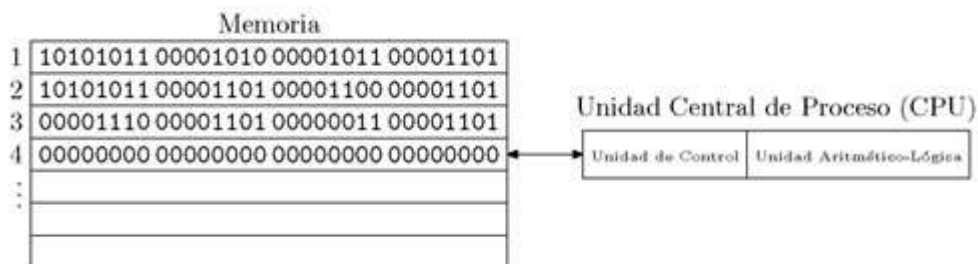
El código máquina codifica las secuencias de instrucciones como sucesiones de unos y ceros que siguen ciertas reglas. Cada familia de microprocesadores dispone de su propio repertorio de instrucciones, es decir, de su propio código de máquina.

Por ejemplo, un programa que calcula la media de tres números almacenados en las posiciones de memoria 10, 11 y 12, respectivamente, y deja el resultado en la posición de memoria 13, podría tener el siguiente aspecto expresado de forma comprensible para nosotros:

#### Memoria

- 1 Suma contenido de direcciones 10 y 11 y deja resultado en dirección 13
- 2 Suma contenido de direcciones 13 y 12 y deja resultado en dirección 13
- 3 Divide contenido de dirección 13 por 3 y dejr resultado en dirección 13
- 4 Detener

En realidad, el contenido de cada dirección estaría codificado como una serie de unos y ceros, así que el aspecto real de un programa como el descrito arriba podría ser éste:



La CPU es un ingenioso sistema de circuitos electrónicos capaz de interpretar el significado de cada una de esas secuencias de bits y llevar a cabo las acciones que codifican. Cuando la CPU ejecuta el programa empieza por la instrucción contenida en la primera de sus posiciones de memoria. Una vez ha ejecutado una instrucción, pasa a la siguiente, y sigue así hasta encontrar una instrucción que detenga la ejecución del programa.

Supongamos que en las direcciones de memoria 10, 11 y 12 se han almacenado los valores 5, 10 y 6, respectivamente. Representamos así la memoria:

Memoria	
1	Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13
2	Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13
3	Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13
4	Detener
...	
10	5
11	10
12	6
...	

Naturalmente, los valores de las posiciones 10, 11 y 12 estarán codificados en binario, aunque hemos optado por representarlos en base 10 en aras de una mayor claridad.

La ejecución del programa procede del siguiente modo. En primer lugar, se ejecuta la instrucción de la dirección 1, que dice que tomemos el contenido de la dirección 10 (el valor 5), que lo sumemos al de la dirección 11 (el valor 10) y que dejemos el resultado (el valor 15) en la dirección de memoria 13. Tras ejecutar esta primera instrucción, la memoria queda así:

Memoria



1	Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13
2	Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13
3	Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13
4	Detener
:	:
:	:
10	5
11	10
12	6
13	15
:	:
:	:

A continuación, se ejecuta la instrucción de la dirección 2, que ordena que se tome el contenido de la dirección 13 (el valor 15), se sume al contenido de la dirección 12 (el valor 6) y se deposite el resultado (el valor 21) en la dirección 13. La memoria pasa a quedar en este estado.

Memoria	
1	Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13
2	Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13
3	Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13
4	Detener
:	:
:	:
10	5
11	10
12	6
13	21
:	:
:	:

Ahora, la tercera instrucción dice que hemos de tomar el valor de la dirección 13 (el valor 21), dividirlo por 3 y depositar el resultado (el valor 7) en la dirección 13. Este es el estado en que queda la memoria tras ejecutar la tercera instrucción:

Memoria	
1	Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13
2	Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13
3	Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13
4	Detener
:	:
:	:
10	5
11	10
12	6
13	7
:	:
:	:

Y finalmente, la CPU detiene la ejecución del programa, pues se encuentra con la instrucción **Detener** en la dirección 4.

## Ejercicios

- Ejecuta paso a paso el mismo programa con los valores 2, -2 y 0 en las posiciones de memoria 10, 11 y 12, respectivamente.
- Diseña un programa que calcule la media de cinco números depositados en las posiciones de memoria que van de la 10 a la 14 y que deje el resultado en la dirección de memoria 15. Recuerda que la media  $\bar{x}$  de cinco números  $x_1, x_2, x_3, x_4$  y  $x_5$  es

$$\bar{x} = \frac{\sum_{i=1}^5 x_i}{5} = \frac{x_1 + x_2 + x_3 + x_4 + x_5}{5}.$$

- Diseña un programa que calcule la varianza de cinco números depositados en las posiciones de memoria que van de la 10 a la 14 y que deje el resultado en la dirección de memoria 15. La varianza, que se denota con  $\sigma_2$ , es

$$\sigma^2 = \frac{\sum_{i=1}^5 (x_i - \bar{x})^2}{5},$$

donde  $\bar{x}$  es la media de los cinco valores. Supon que existe una instrucción "Multiplicar el contenido de dirección  $a$  por el contenido de dirección  $b$  y dejar el resultado en dirección  $c$ ".

¿Qué instrucciones podemos usar para confeccionar programas? Ya hemos dicho que el ordenador sólo sabe ejecutar instrucciones muy sencillas. En nuestro ejemplo, sólo hemos utilizado tres instrucciones distintas:

- Una instrucción de suma de la forma "Suma el contenido de las direcciones  $p$  y  $q$  y deja el resultado en la dirección  $r$ ".
- Una instrucción de división de la forma "Divide el contenido de la dirección  $p$  por el contenido de la dirección  $q$  y deja el resultado en la dirección  $r$ ".
- Una instrucción que indica que se ha llegado al final del programa: **Detener**.

¡Pocos programas interesantes podemos hacer con tan solo estas tres instrucciones! Naturalmente, en un código máquina hay instrucciones que permiten efectuar sumas, restas, divisiones y otras muchas operaciones. Y hay, además, instrucciones que permiten escoger qué instrucción se ejecutará a continuación, bien directamente, bien en función de si se cumple o no una determinada condición (por ejemplo, "Si el último resultado es negativo, pasar a ejecutar la instrucción de la posición  $p$ ").

### 3.2. Lenguaje ensamblador

En los primeros tiempos de la informática los programas se introducían en el ordenador directamente en código de máquina, indicando uno por uno el valor de los bits de cada una de las posiciones de memoria. Para ello se insertaban manualmente cables en un panel de conectores: cada cable insertado en un conector representaba un uno y cada conector sin cable representaba un cero. Como puedes imaginar, programar así un computador resultaba una tarea ardua, extremadamente tediosa y propensa a la comisión de errores. El más mínimo fallo conducía a un programa incorrecto. Pronto se diseñaron notaciones que simplificaban la programación: cada instrucción de código de máquina se representaba mediante un *código mnemotécnico*, es decir, una abreviatura fácilmente identificable con el propósito de la instrucción.

Por ejemplo, el programa desarrollado anteriormente se podría representar con el siguiente texto:

```
SUM #10, #11, #13
SUM #13, #12, #13
DIV #13, 3, #13
FIN
```

En este lenguaje la palabra **SUM** representa la instrucción de sumar, **DIV** la de dividir y **FIN** representa la instrucción que indica que debe finalizar la ejecución del programa. La almohadilla (#) delante de un número indica que deseamos acceder al contenido de la posición de memoria cuya dirección es dicho número. Los caracteres que representan el programa se introducen en la memoria del ordenador con la ayuda de un teclado y cada letra se almacena en una posición de memoria como una combinación particular de unos y ceros (su código ASCII, por ejemplo).

Pero, ¿cómo se puede ejecutar ese tipo de programa si la secuencia de unos y ceros que la describe como texto no constituye un programa válido en código de máquina? La respuesta es: con la ayuda de otro programa: el *ensamblador*. El ensamblador es un programa traductor que lee el contenido de las direcciones de memoria en las que hemos almacenado códigos mnemotécnicos y escribe en otras posiciones de memoria sus instrucciones asociadas en código máquina.

El repertorio de códigos mnemotécnicos traducible a código de máquina y las reglas que permiten combinarlos, expresar direcciones, codificar valores numéricos, etc., recibe el nombre de *lenguaje ensamblador*, y es otro lenguaje de programación.

### 3.3. ¿Un programa diferente para cada ordenador?



Cada microprocesador tiene su propio juego de instrucciones y, en consecuencia, un código máquina y uno o más lenguajes ensambladores propios. Un programa escrito para un micro de la marca Intel no funcionará con un micro diseñado por otro fabricante, como Motorola (a menos que el micro se haya diseñado expresamente para reproducir el funcionamiento de la primera, como ocurre con los procesadores de AMD, diseñados con el objetivo de ejecutar el código máquina de los procesadores de Intel). Incluso diferentes versiones de un mismo microprocesador tienen juegos de instrucciones que no son totalmente compatibles entre sí: los modelos más evolucionados de una familia de microprocesadores pueden incorporar instrucciones que no se encuentran en los más antiguos.

Si queremos que un programa se ejecute en más de un tipo de ordenador, ¿habrá que escribirlo de nuevo para cada microprocesador particular? Durante mucho tiempo se intentó definir algún tipo de "lenguaje ensamblador universal", es decir, un lenguaje cuyos códigos mnemotécnicos, sin corresponderse con los del código de máquina de ningún ordenador concreto, fuesen fácilmente traducibles al código de máquina de cualquier ordenador. Disponer de dicho lenguaje permitiría escribir los programas una sola vez y ejecutarlos en diferentes ordenadores tras efectuar las correspondientes traducciones a cada código de máquina con diferentes programas ensambladores.

Si bien la idea es en principio interesante, presenta serios inconvenientes:

1. Un lenguaje ensamblador universal no puede tener en cuenta cómo se diseñarán ordenadores en un futuro y qué tipo de instrucciones soportarán, así que posiblemente quede obsoleto en poco tiempo.
2. Programar en lenguaje ensamblador (incluso en ese supuesto lenguaje ensamblador universal) es complicadísimo por los numerosos detalles que deben tenerse en cuenta.

## Hola Mundo!

Nos gustaría mostrarte el aspecto de los programas escritos en lenguajes ensambladores reales con un par de ejemplos. Es una tradición ilustrar los diferentes lenguajes de programación con un programa sencillo que se limita a mostrar por pantalla el mensaje "Hello, World!" ("¡Hola, mundo!"). He aquí ese programa escrito en los lenguajes ensambladores de dos CPU distintas: en la columna de la izquierda, el de los procesadores 80x86 de Intel, y en la columna de la derecha, el de los procesadores de la familia Motorola 68000 (que es el procesador de los primeros ordenadores Apple Macintosh).

<pre>.data msg: .string "Hello, World!\n" len: .long . - msg .text .globl _start _start:     push \$len     push \$msg     push \$1     movl \$0x4, %eax     call _syscall     addl \$12, %esp     push \$0     movl \$0x1, %eax     call _syscall _syscall:     int \$0x80 ret</pre>	<pre>start:     move.l #msg, -(a7)     move.w #9, -(a7)     trap #1     addq.l #6, a7     move.w #1, -(a7)     trap #1     addq.l #2, a7     clr -(a7)     trap #1     msg: dc.b "Hello, World!",10,13,0</pre>
---	--

Como puedes ver, ambos programas presentan un aspecto muy diferente. Por otra parte, los dos son bastante largos (entre 10 y 20 líneas) y de difícil comprensión.

Además, puestos a diseñar un lenguaje de programación general, ¿por qué no utilizar un lenguaje natural, es decir un lenguaje como el castellano o el inglés? Programar un computador consistiría, simplemente, en escribir (¡o pronunciar frente a un micrófono!) un texto en el que indicásemos al ordenador qué es lo que deseamos que haga, utilizando el mismo lenguaje con el que nos comunicamos con otras personas. Un programa informático podría encargarse de traducir nuestras frases al código de máquina, del mismo modo que un programa ensamblador traduce lenguaje ensamblador a código de máquina. Es una idea atractiva, pero que queda lejos de lo que sabemos hacer por varias razones:

1. La complejidad intrínseca de las construcciones de los lenguajes naturales dificulta enormemente el *análisis sintáctico* de las frases, es decir, comprender su estructura y cómo se relacionan entre sí los diferentes elementos que las constituyen.
2. El *análisis semántico*, es decir, la comprensión del significado de las frases, es aún más complicado. Las ambigüedades e imprecisiones del lenguaje natural hacen que sus frases presenten, fácilmente, diversos significados, aun cuando las podamos analizar sintácticamente. (¿Cuántos significados tiene la frase "Trabaja en un banco"?). Sin una buena comprensión del significado no es posible efectuar una traducción aceptable.

### 3.4. Lenguajes de programación de alto nivel

Hay una solución intermedia: podemos diseñar lenguajes de programación que, sin ser tan potentes y expresivos como los lenguajes naturales, eliminen buena parte de la complejidad propia de los lenguajes ensambladores y estén bien adaptados al tipo de problemas que podemos resolver con los computadores: los denominados *lenguajes de programación de alto nivel*. El calificativo "de alto nivel" señala su independencia de un ordenador concreto. En contraposición, los códigos máquina y los lenguajes ensambladores se denominan *lenguajes de programación de bajo nivel*.

He aquí el programa que calcula la media de tres números en un lenguaje de alto nivel típico (lenguaje C):

```
...
a = 5;
b = 10;
c = 6;
media = (a+b+c) / 3;
```

Las tres primeras líneas definen los tres valores y la cuarta calcula la media. Como puedes ver, resulta mucho más legible que un programa en código de máquina o en un lenguaje ensamblador. Para cada lenguaje de alto nivel y para cada CPU se puede escribir un programa que se encargue de traducir las instrucciones del lenguaje de alto nivel a instrucciones de código de máquina, con lo que se consigue la deseada independencia de los programas con respecto a los diferentes sistemas computadores. Sólo habrá que escribir una versión del programa en un lenguaje de programación de alto nivel y la traducción de ese programa al código máquina de cada microprocesador se realizará automáticamente.

### 3.5. Compiladores e intérpretes

Hemos dicho que los lenguajes de alto nivel se traducen automáticamente a código máquina, sí, pero has de saber que hay dos tipos diferentes de traductores, dependiendo de su modo de funcionamiento: *compiladores* e *intérpretes*.

Un **compilador** lee completamente un programa en un lenguaje de alto nivel y lo traduce en su integridad a un programa de código de máquina equivalente. El programa código de máquina resultante se puede ejecutar cuantas veces se desee, sin necesidad de volver a traducir el programa original.

Un **intérprete** actúa de un modo distinto: lee un programa escrito en un lenguaje de alto nivel instrucción a instrucción y, para cada una de dicha instrucciones, efectúa una traducción a las instrucciones de código de máquina equivalentes, y las ejecuta inmediatamente. No hay un proceso de traducción separado por completo del proceso de ejecución. Cada vez que ejecutamos el programa con un intérprete, se repite el proceso de traducción y ejecución, ya que ambos son simultáneos.

## Compiladores e intérpretes... de idiomas

Puede resultarte de ayuda establecer una analogía entre compiladores e intérpretes de lenguajes de programación y traductores e intérpretes de idiomas.

Un compilador actúa como un traductor que recibe un libro escrito en un idioma determinado (lenguaje de alto nivel) y escribe un nuevo libro que, con la mayor fidelidad posible, contiene una traducción del texto original a otro idioma (código máquina). El proceso de traducción (compilación) tiene lugar una sola vez y podemos leer el libro (ejecutar el programa) en el idioma destino (código de máquina) cuantas veces queramos.

Un intérprete de programas actúa como su homónimo humano en el caso de los idiomas. Supongamos que se imparte un ciclo de conferencias en inglés en diferentes ciudades y un intérprete ofrece su traducción simultánea al castellano. Cada vez que se pronuncia la conferencia, el intérprete debe realizar nuevamente la traducción. Es más, la traducción se produce sobre la marcha, frase a frase, y no de un tirón al final de la conferencia. Del mismo modo actúa un intérprete de un lenguaje de programación: traduce cada vez que ejecutamos el programa y además lo hace instrucción a instrucción.

Por regla general, los intérpretes ejecutarán los programas *más lentamente*, puesto que al tiempo de ejecución del código de máquina se suma el que consume la traducción simultánea. Además, un compilador puede examinar el programa de alto nivel abarcando más de una instrucción cada vez, por lo que es capaz de producir mejores traducciones. Un programa interpretado suele ser mucho más lento que otro equivalente que haya sido compilado (típicamente entre 2 y 100 veces más lento!).

Si tan lento resulta interpretar un programa, ¿por qué no se usan únicamente compiladores? Es pronto para que entiendas las razones, pero, por regla general, los intérpretes permiten una mayor flexibilidad que los compiladores y ciertos lenguajes de programación de alto nivel han sido diseñados para explotar esa mayor flexibilidad. Otros lenguajes de programación, por contra, sacrifican la flexibilidad en aras de una mayor velocidad de ejecución. Aunque nada impide que compilemos o interpretemos cualquier lenguaje de programación, ciertos lenguajes se consideran apropiados para que la traducción se lleve a cabo con un compilador y otros no. Es más apropiado hablar, pues, de lenguajes de programación *típicamente* interpretados y lenguajes de programación *típicamente* compilados. Entre los primeros podemos citar Python, BASIC, Perl, Tcl, Ruby, Bash, Java o Lisp. Entre los segundos, C, Pascal, C++ o Fortran.

En este curso aprenderemos a programar usando un lenguaje de programación compilado, como el lenguaje C.

### 3.6. Lenguaje C

El lenguaje de programación C es uno de los más utilizados en el mundo profesional. La mayoría de las aplicaciones comerciales y libres se han desarrollado con el lenguaje de programación C. El sistema operativo Linux, por ejemplo, se ha desarrollado en C en su práctica totalidad.

¿Por qué es tan utilizado el lenguaje C? C es un lenguaje de propósito general que permite controlar con gran precisión los factores que influyen en la eficiencia de los programas. Pero esta capacidad de control "fino" que ofrece C tiene un precio: la escritura de programas puede ser mucho más costosa, pues hemos de estar pendientes de numerosos detalles. Tan es así que muchos programadores afirman que C no es un lenguaje de alto nivel, sino de *nivel intermedio*.

C ha sufrido una evolución desde su diseño en los años 70. El C, tal cual fue concebido por sus autores, Brian Kernighan y Dennis Ritchie, de la compañía norteamericana de telecomunicaciones AT&T, se conoce popularmente por K&R C y está prácticamente en desuso. En los años 80, C fue modificado y estandarizado por el *American National Standards Institute* (ANSI), que dio lugar al denominado ANSI C y que ahora se conoce como C89 por el año en que se publicó. El estándar se revisó en los años 90 y se incorporaron nuevas características que mejoran sensiblemente el lenguaje. El resultado es la segunda edición del ANSI C, más conocida como **C99**. **Esta es la versión que estudiaremos en este curso.**

En la asignatura utilizaremos un compilador de C gratuito: el **gcc** en su versión 3.2 o superior. Inicialmente se denominó *gcc* tomando las siglas de *GNU C Compiler*. GNU es el nombre de un proyecto que tiene por objeto ofrecer un sistema operativo "libre" y todas las herramientas que es habitual encontrar en una plataforma Unix. Hoy día se ha popularizado enormemente gracias a la plataforma GNU/Linux, que se compone de un núcleo de sistema operativo de la familia del Unix (Linux) y numerosas herramientas desarrolladas como parte del proyecto GNU, entre ellas *gcc*.

### La torre de Babel

Hemos dicho que los lenguajes de programación de alto nivel pretendían, entre otros objetivos, paliar el problema de que cada ordenador utilice su propio código de máquina. Puede que, en consecuencia, estés sorprendido por el número de lenguajes de programación citados. Pues los que hemos citado son unos pocos de los más utilizados: ¡hay centenares! ¿Por qué tantos?

El primer lenguaje de programación de alto nivel fue Fortran, que se diseñó en los primeros años 50 (y aún se utiliza hoy día, aunque en versiones evolucionadas). Fortran se diseñó con el propósito de traducir fórmulas matemáticas a código de máquina (de hecho, su nombre proviene de "FORmula TRANslator", es decir, "traductor de fórmulas"). Pronto se diseñaron otros lenguajes de programación con propósitos específicos: Cobol (Common Business Oriented Language), Lisp (List Processing language), etc. Cada uno de estos lenguajes hacía fácil la escritura de programas para solucionar problemas de ámbitos particulares: Cobol para problemas de gestión empresarial, Lisp para ciertos problemas de Inteligencia Artificial, etc. Hubo también esfuerzos para diseñar lenguajes de "propósito general", es decir, aplicables a cualquier dominio, como Algol 60 (Algorithmic Language). En la década de los 60 hicieron su aparición nuevos lenguajes de programación (Algol 68, Pascal, Simula 67, Snobol 4, etc.), pero quizá lo más notable de esta década fue que se sentaron las bases teóricas del diseño de compiladores e intérpretes. Cuando la tecnología para el diseño de estas herramientas se hizo accesible a más y más programadores hubo una auténtica explosión en el número de lenguajes de programación. Ya en 1969 se habían diseñado unos 120 lenguajes de programación y se habían implementado compiladores o intérpretes para cada uno de ellos.

La existencia de tantísimos lenguajes de programación creó una situación similar a la de la torre de Babel: cada laboratorio o departamento informático usaba un lenguaje de programación y no había forma de intercambiar programas.

Con los años se ha ido produciendo una selección de aquellos lenguajes de programación más adecuados para cada tipo de tarea y se han diseñado muchos otros que sintetizan lo aprendido de lenguajes anteriores. Los más utilizados hoy día son C, C++ , Java, Python, Perl y PHP.

## 4. Más allá de los programas: algoritmos

Dos programas que resuelven el mismo problema expresados en el mismo o en diferentes lenguajes de programación pero que siguen, en lo fundamental, el mismo procedimiento, son dos *implementaciones* del mismo *algoritmo*. Un **algoritmo** es, sencillamente, una secuencia de pasos orientada a la consecución de un objetivo.

Cuando diseñamos un algoritmo podemos expresarlo en cualquiera de los numerosos lenguajes de programación de propósito general existentes. Sin embargo, esto resulta poco adecuado:

1. No todos los programadores conocen todos los lenguajes y no hay consenso acerca de cuál es el más adecuado para expresar las soluciones a los diferentes problemas,

2. Cualquiera de los lenguajes de programación presenta particularidades que pueden interferir en una expresión clara y concisa de la solución a un problema.

Podemos expresar los algoritmos en lenguaje natural, pues el objetivo es comunicar un procedimiento resolutorio a otras personas y, eventualmente, traducirlos a algún lenguaje de programación. Si, por ejemplo, deseamos calcular la media de tres números leídos de teclado podemos seguir este algoritmo:

1. Solicitar el valor del primer número.
2. Solicitar el valor del segundo número.
3. Solicitar el valor del tercer número.
4. Sumar los tres números y dividir el resultado por 3.
5. Mostrar el resultado.

Como puedes ver, esta secuencia de operaciones define exactamente el proceso que nos permite efectuar el cálculo propuesto.

Los algoritmos son independientes del lenguaje de programación. Describen un procedimiento que puedes implementar en cualquier lenguaje de programación de propósito general o, incluso, que puedes ejecutar a mano con lápiz, papel y, quizá, la ayuda de una calculadora.

¡Ojo! No es cierto que cualquier procedimiento descrito paso a paso pueda considerarse un algoritmo. Un algoritmo debe satisfacer ciertas condiciones. Una analogía con recetas de cocina (procedimientos para preparar platos) te ayudará a entender dichas restricciones.

Estudia esta primera receta:

1. Poner aceite en una sartén.
2. Encender el fuego.
3. Calentar el aceite.
4. Coger un huevo.
5. Romper la cascara.
6. Verter el contenido del huevo en la sartén.
7. Aderezar con sal.
8. Esperar a que tenga buen aspecto.

En principio ya está: con la receta, sus ingredientes y los útiles necesarios somos capaces de cocinar un plato. Bueno, no del todo cierto, pues hay unas cuantas cuestiones que no quedan del todo claras en nuestra receta:

1. ¿Qué tipo de huevo utilizamos?: ¿un huevo de gallina?, ¿un huevo de avestruz?, ¿de codorniz, quizá?
2. ¿Cuánta sal utilizamos?: ¿una pizca?, ¿un kilo?
3. ¿Cuánto aceite hemos de verter en la sartén?: ¿un centímetro cúbico?, ¿un litro?
4. ¿Cuál es el resultado del proceso?, ¿la sartén con el huevo cocinado y el aceite?

En una receta de cocina hemos de dejar bien claro con qué ingredientes contamos y cuál es el resultado final. En un algoritmo hemos de precisar cuáles son los datos del problema (datos de entrada) y qué resultado vamos a producir (datos de salida).

Esta nueva receta corrige esos fallos:

- Ingredientes: 10 cc. de aceite de oliva, una gallina y una pizca de sal.
- Método:
  1. *Eesperar a que la gallina ponga un huevo,*
  2. Poner aceite en una sartén,
  3. Encender el fuego,
  4. Calentar el aceite,
  5. Coger el huevo,
  6. Romper la cascara,
  7. Verter el contenido del huevo en la sartén,
  8. Aderezar con sal,
  9. Esperar a que tenga buen aspecto.
- Presentación: depositar el huevo frito, sin aceite, en un plato.

Pero la receta aún no está bien del todo. Hay ciertas indefiniciones en la receta:

1. ¿Cuan caliente ha de estar el aceite en el momento de verter el huevo?, ¿humeando?, ¿ardiendo?
2. ¿Cuánto hay que esperar?, ¿un segundo?, ¿hasta que el huevo esté ennegrecido?
3. Y aún peor, ¿estamos seguros de que la gallina pondrá un huevo? Podría ocurrir que la gallina no pusiera huevo alguno.

Para que la receta esté completa, deberíamos especificar con *absoluta precisión* cada uno de los pasos que conducen a la realización del objetivo y, además, cada uno de ellos debería ser realizable en *tiempo finito*.

No basta con decir *más o menos* cómo alcanzar el objetivo: hay que decir *exactamente* cómo se debe ejecutar cada paso y, además, cada paso debe ser realizable en tiempo finito. Esta nueva receta corrige algunos de los problemas de la anterior, pero presenta otros de distinta naturaleza:

- Ingredientes: 10 cc. de aceite de oliva, un huevo de gallina y una pizca de sal.
- Método:
  1. Poner aceite en una sartén.
  2. Encender el fuego a medio gas.
  3. Calentar el aceite hasta que humee ligeramente.
  4. Coger un huevo.
  5. Romper la cáscara *con el poder de la mente*, sin tocar el huevo.
  6. Verter el contenido del huevo en la sartén.
  7. Aderezar con sal.
  8. Esperar a que tenga buen aspecto.
- Presentación: depositar el huevo frito, sin aceite, en un plato.

El quinto paso no es *factible*. Para romper un huevo has de utilizar algo más que "el poder de la mente". En todo algoritmo debes utilizar únicamente instrucciones que pueden llevarse a cabo. He aquí una receta en la que todos los pasos son realizables:

- Ingredientes: 10 cc. de aceite de oliva, un huevo de gallina y una pizca de sal.
- Método:
  1. Poner aceite en una sartén.
  2. *Sintonizar una emisora musical en la radio.*
  3. Encender el fuego a medio gas.
  4. *Echar una partida al solitario.*
  5. Calentar el aceite hasta que humee ligeramente.
  6. Coger un huevo.
  7. Romper la cascara.
  8. Verter el contenido del huevo en la sartén.
  9. Aderezar con sal.
  10. Esperar a que tenga buen aspecto.
- Presentación: depositar el huevo frito, sin aceite, en un plato.

En esta nueva receta hay acciones que, aunque expresadas con suficiente precisión y siendo realizables, no hacen nada *útil* para alcanzar nuestro objetivo (sintonizar la radio y jugar a cartas). En un algoritmo, *cada paso dado debe conducir y acercarnos más a la consecución del objetivo*.

Hay una consideración adicional que hemos de hacer, aunque en principio parezca una obviedad: todo algoritmo bien construido debe finalizar tras la ejecución de un *número finito de pasos*. Aunque todos los pasos sean de duración finita, una secuencia de instrucciones puede requerir tiempo infinito. Piensa en este método para hacerse millonario:

1. Comprar un número de lotería válido para el próximo sorteo,
2. Esperar al día de sorteo,
3. Cotejar el número ganador con el nuestro,
4. Si son diferentes, volver al paso 1; en caso contrario, somos millonarios.

Como ves, cada uno de los pasos del método requiere una cantidad finita de tiempo, pero no hay ninguna garantía de alcanzar el objetivo propuesto.



En adelante, no nos interesarán más las recetas de cocina ni los procedimientos para enriquecerse sin esfuerzo (¡al menos no como objeto de estudio de la asignatura!). Los algoritmos en los que estaremos interesados son aquellos que describen procedimientos de cálculo ejecutables en un ordenador. Esto limitará el ámbito de nuestro estudio a la manipulación y realización de cálculos sobre datos (numéricos, de texto, etc.).

Un algoritmo debe poseer las siguientes características:

1. Cada paso del algoritmo ha de *estar definido con exactitud*, sin la menor ambigüedad.
2. Ha de ser *finito*, es decir, debe finalizar tras la ejecución de un número finito de pasos, cada uno de los cuales ha de ser ejecutable en tiempo finito.
3. Debe ser *efectivo*, es decir, cada uno de sus pasos ha de poder ejecutarse en tiempo finito con unos recursos determinados (en nuestro caso, con los que proporciona un sistema computador).

Además, nos interesa que los algoritmos sean *eficientes*, esto es, que alcancen su objetivo lo más rápidamente posible y con el menor consumo de recursos.

### Abu Ja'far Mohammed ibn Musa Al-Khowarizm y Euclides

La palabra *algoritmo* tiene origen en el nombre de un matemático persa del siglo IX: *Abu Ja'far Mohammed ibn Musa Al-Khowarizm* (que significa "Mohammed, padre de Ja'far, hijo de Moises, nacido en Khowarizm"). Al-Khowarizm escribió tratados de aritmética y álgebra. Gracias a los textos de Al-Khowarizm se introdujo el sistema de numeración hindú en el mundo árabe y, más tarde, en occidente.

En el siglo XIII se publicaron los libros *Carmen de Algorismo* (un tratado de aritmética ¡en verso!) y *Algorismus Vulgaris*, basados en parte en la *Aritmética* de Al-Khowarizm. Al-Khowarizm escribió también el libro "Kitab al jabr w'al-muqabala" ("Reglas de restauración y reducción"), que dio origen a una palabra que ya conoces: "álgebra".

Abelardo de Bath, uno de los primeros traductores al latín de Al-Khowarizm, empezó un texto con "Dixit Algorismi. . ." ("Dijo Algorismo. . ."), popularizando así el término *algorismo*, que pasó a significar "realización de cálculos con numerales hindo-árabigos". En la edad media los abaquistas calculaban con abaco y los algorismistas con "algorismos".

En cualquier caso, el concepto de algoritmo es muy anterior a Al-Khowarizm. En el siglo III a.C, Euclides propuso en su tratado "Elementos" un método sistemático para el cálculo del Máximo Común Divisor (MCD) de dos números. El método, tal cual fue propuesto por Euclides, dice así: "Dados dos números naturales,  $a$  y  $b$ , comprobar si ambos son iguales. Si es así,  $a$  es el MCD. Si no, si  $a$  es mayor que  $b$ , restar  $a$  el valor de  $b$ ; pero si  $a$  es menor que  $b$ , restar  $a$  el valor de  $a$ . Repetir el proceso con los nuevos valores de  $a$  y  $b$ ". Este método se conoce como "algoritmo de Euclides", aunque es frecuente encontrar, bajo ese mismo nombre, un procedimiento alternativo y más eficiente: "Dados dos números naturales,  $a$  y  $b$ , comprobar si  $b$  es cero. Si es así,  $a$  es el MCD. Si no, calcular  $c$ , el resto de dividir  $a$  entre  $b$ . Sustituir  $a$  por  $b$  y  $b$  por  $c$  y repetir el proceso".

## Ejercicios

9. Diseña un algoritmo para calcular el área de un círculo dado su radio. (Recuerda que el área de un círculo es  $\pi$  veces el cuadrado del radio.)
10. Diseña un algoritmo que calcule el IVA (16%) de un producto dado su precio de venta sin IVA.
11. ¿Podemos llamar algoritmo a un procedimiento que escriba en una cinta de papel *todos* los números decimales de  $\pi$ ?

## 5. Ejecutar aplicaciones en C

En esta sección aprenderás cómo ejecutar una aplicación de C. El compilador te permitirá ejecutar aplicaciones en C. Durante la mayor parte del curso nos centraremos en tres tipos de archivos: `.c`, `.h` y `.exe`. Los archivos con extensión `.c` (llamados **archivos de código fuente**) y `.h` (llamados **archivos de cabecera**) almacenan sentencias de lenguaje C escritas por ti, el programador. Estas sentencias indican las instrucciones que te gustaría que realizaran tus aplicaciones. Para ejecutar una aplicación, las sentencias almacenadas en el archivo `.c` primero deben ser convertidas en instrucciones que el ordenador pueda entender. El proceso de convertir las sentencias de un lenguaje de alto nivel como C en lenguaje de máquina se conoce como **compilación**. Los archivos con la extensión `.exe` (llamados **archivos ejecutables**) se crean después de *compilar* y *enlazar* el código fuente. En el siguiente tutorial obtendrás información sobre cómo compilar el archivo `.c` y ejecutar una aplicación. En la siguiente sección, podrás poner a prueba dos aplicaciones C.

El Símbolo del sistema es un programa de Windows que te permite dar instrucciones al equipo escribiendo texto en un sistema. La figura 2 muestra una ventana **Símbolo del sistema** en un equipo que ejecuta Windows 7. Puedes acceder al ventana **Símbolo del sistema** de Windows en el menú Iniciar, escribiendo en la casilla "Buscar Programas y archivos" el texto **CMD**. En otras versiones de Windows se puede abrir seleccionando **Inicio** ⇒ **Programas** ⇒ **Accesorios** ⇒ **Símbolo del Sistema**. Cuando la ventana Símbolo del sistema se inicia, el directorio de inicio es `C:\Users\Fernando`. En tu equipo, será reemplazado por tu nombre de usuario.

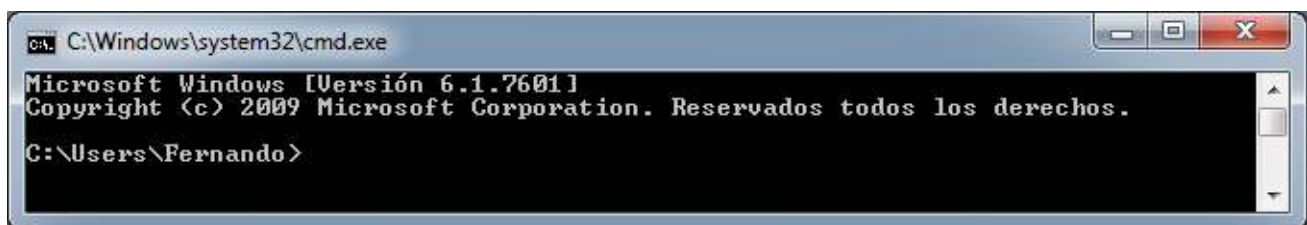


Figura 2. Ventana en Windows 7.

Comenzaremos con la ejecución de una **calculadora de pago de coches**, que calcula la cantidad que un cliente debe en un préstamo para la compra de un coche, dado el pago inicial, el precio del automóvil, la tasa de interés y la duración del préstamo. (Generarás esta aplicación más adelante). También podrás probar una aplicación de **juego** que genera un número aleatorio. El jugador gana al adivinar el número correcto. (Esta aplicación se incluye como ejercicio en una sesión posterior).

En los pasos siguientes, ejecutarás cada aplicación e interactuarás con ella. Los elementos y funcionalidades que ves en estas aplicaciones son típicos de lo que va a aprender a programar en este curso. Hemos modificado el color del fondo de la ventana **Símbolo del sistema** para que podamos destacar las entradas y las llamadas a nuevas características. [Nota: Si deseas modificar los colores del **Símbolo del sistema** en tu ordenador, abre una ventana **Símbolo del sistema**, después haz clic con el botón derecho del ratón en la barra de título y selecciona **Propiedades**. En el cuadro de diálogo **Propiedades de "Símbolo del sistema"** que aparece, haz clic en la ficha **Colores** y selecciona tus colores de texto y fondo preferidos].

### 5.1. Prueba de uso de la aplicación Calculadora de Pago de Automóviles

1. **Carpeta de Proyectos.** Crea una carpeta de Proyectos en el disco duro de tu ordenador (si estás en casa), por ejemplo, `C:\Programacion`. Si estás en el laboratorio la carpeta puede ser `Z:\Programacion`. Recuerda que el nombre de la carpeta no debe contener espacios en blanco, acentos o caracteres especiales.
2. Abre una ventana **Símbolo del sistema**. Si utilizas Windows XP o Windows 7, selecciona **Inicio** ⇒ **Todos los programas** ⇒ **Accesorios** ⇒ **Símbolo del sistema**.
3. Para cambiar a tu carpeta de proyectos, escribe `cd C:\Programacion`, y después pulsa **Intro** (figura 3). El comando `cd` se utiliza para cambiar de directorio.

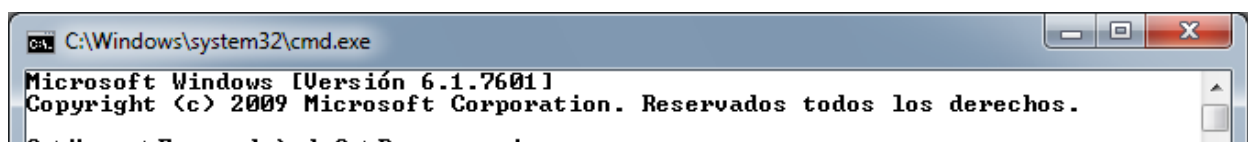




Figura 3. Abrir una ventana **Símbolo del sistema** y cambiar de carpeta.

4. Crea una carpeta para los ejemplos en tu carpeta de proyectos, llamada por ejemplo, **ejemplos**. Puedes hacerlo directamente en la ventana Símbolo del Sistema, con el comando **mkdir Ejemplos**. A continuación, utiliza de nuevo el comando **cd** para cambiar a la nueva carpeta: **cd Ejemplos**

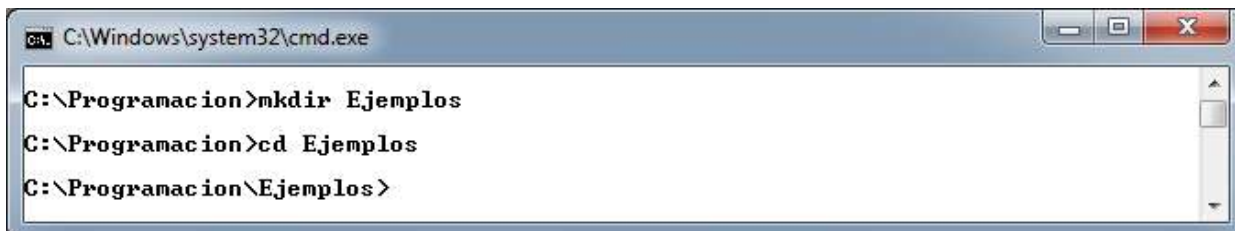


Figura 4. Crear una nueva carpeta y cambiar a dicha carpeta.

5. Descarga la aplicación pagoCoches.exe. en la carpeta **Ejemplos** que acabas de crear. Para comprobar que lo has hecho correctamente, utiliza el comando **dir**. Este comando muestra la lista de archivos y subcarpetas que contiene la carpeta actual. Si todo ha ido bien, en el listado debe aparecer el archivo PagoCoches.exe.



Figura 5. Resultado de aplicar el comando **dir** en la carpeta Ejemplos

6. Ejecuta la aplicación Calculadora de pago de coches. Escribe **PagoCoches** en la línea de comandos para ejecutar la aplicación (figura 6). Para ejecutar la aplicación lo único que se necesita es escribir su nombre. Observa que no es necesario especificar la extensión **.exe** al usar este comando. (Nota: en muchos sistemas, los comandos son sensibles a mayúsculas. Es importante que escribas el nombre de esta aplicación con una "P" en "Pago" y una "C" en "Coches". De lo contrario, puede que no se ejecute la aplicación.)



Figura 6. La ejecución de Calculadora de pago de coches.

7. **Introducción de valores en la aplicación.** Escribe 70000 en la pregunta "Escribe el precio del coche". Escribe 2000 en la pregunta "Escribe el pago inicial". Escribe 6 después de la pregunta "Ingresa el tipo de interes anual", pero no pulses **Intro**. La pantalla debe aparecer como en la figura 7.

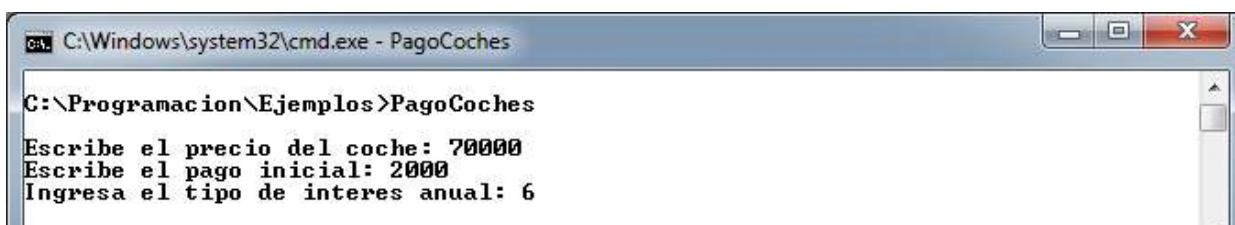
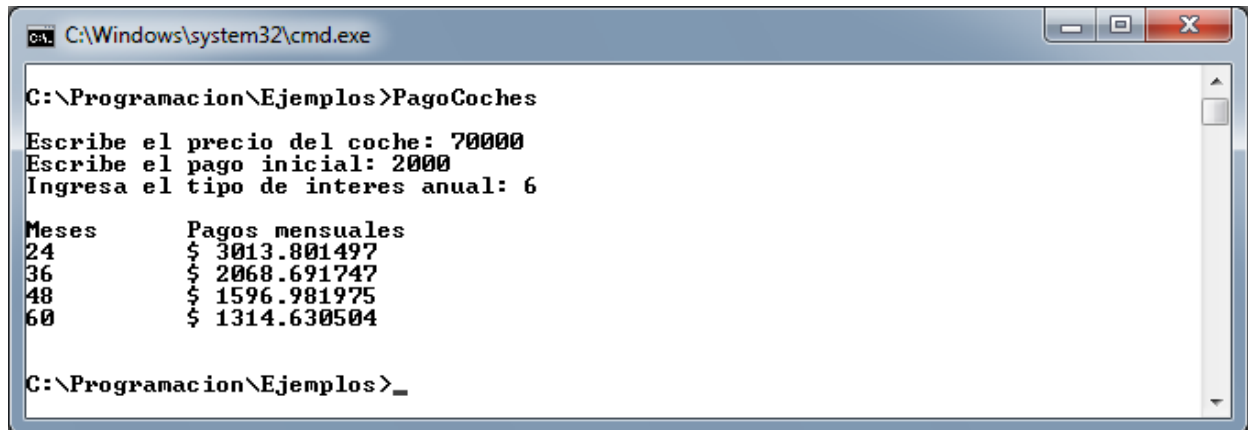


Figura 7. Calculadora de Pago de coches con los datos introducidos.

8. **Cálculo de los pagos mensuales.** Pulsa **Intro** para introducir el valor **6** en la aplicación. La aplicación muestra en formato tabular las cantidades mensuales de pago por períodos de préstamo de 24, 36, 48 y 60 meses (figura 8).



```
C:\Windows\system32\cmd.exe

C:\Programacion\Ejemplos>PagoCoches

Escribe el precio del coche: 70000
Escribe el pago inicial: 2000
Ingresa el tipo de interes anual: 6

Meses      Pagos mensuales
24          $ 3013.801497
36          $ 2068.691747
48          $ 1596.981975
60          $ 1314.630504

C:\Programacion\Ejemplos>_
```

Figura 8. Calculadora de Pago de coches mostrando los resultados del cálculo.

9. **Cierre de la ventana del sistema.** Haz clic en el cuadro de cierre de la aplicación.

## 5.2. Prueba de funcionamiento de Adivina el número

En la siguiente sección de prueba de funcionamiento de aplicaciones, se presenta la aplicación **Adivina el número**, que genera un número aleatorio entre 1 y 100 y pide al usuario que adivine el número con siete o menos intentos. El usuario introduce números en la ventana **Símbolo del Sistema** ante la solicitud Introduce numero. Si el número coincide con el calculado por la aplicación, el juego termina. Si la suposición no es correcta, la aplicación indica si el número tecleado por el usuario es mayor o menor que el número correcto. Si el usuario no ha adivinado el número correctamente después de siete intentos, la aplicación indica que el usuario no tiene turnos restantes y muestra el valor del número. [Nota: La valores de salida mostrados por esta aplicación variarán cada vez que se ejecute dicha aplicación porque el sistema elige un número al azar.]

1. Descarga la aplicación adivinaNumero.exe en la carpeta **ejemplos** de tu carpeta de proyectos.
2. Ejecuta la aplicación **AdivinaNumero**. Ahora que te encuentras en el directorio que contiene la aplicación **AdivinaNumero**, escribe el comando **AdivinaNumero** (figura 9) y pulsa **Intro**. [Nota: **AdivinaNumero.exe** es el nombre real de la aplicación; sin embargo, Windows asume la extensión **.exe** de manera predeterminada.]

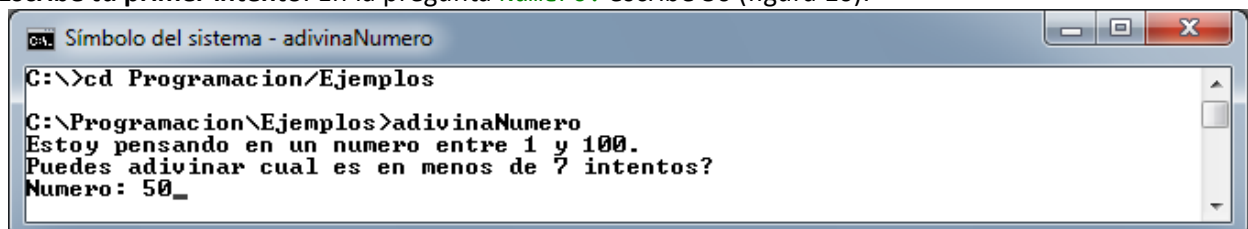


```
Símbolo del sistema

C:\>\cd Programacion/Ejemplos
C:\Programacion\Ejemplos>adivinaNumero
```

Figura 9. Ejecución de la aplicación **AdivinaNumero**.

3. **Escribe tu primer intento.** En la pregunta **Numero:** escribe 50 (figura 10).



```
Símbolo del sistema - adivinaNumero

C:\>\cd Programacion/Ejemplos
C:\Programacion\Ejemplos>adivinaNumero
Estoy pensando en un numero entre 1 y 100.
Puedes adivinar cual es en menos de 7 intentos?
Numero: 50_
```

Figura 10. Escribe tu primer intento.

4. **Escribe otro intento.** La aplicación muestra "**Demasiado alto...**", lo cual significa que el valor que has escrito es mayor que el número que ha elegido la aplicación como la respuesta correcta. Por lo tanto, debes escribir un número menor en tu siguiente intento. Escribe ahora 25 (figura 11). La aplicación muestra el mensaje "**Demasiado bajo...**", ya que el valor que has escrito es menor que el número elegido por la aplicación.

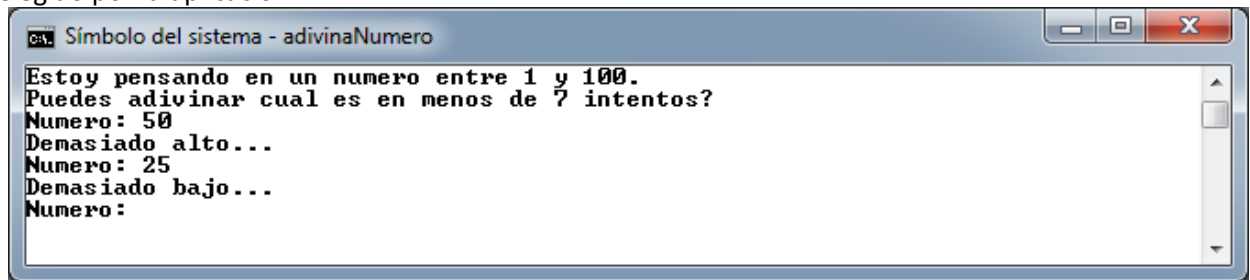


Figura 11. Escribe tu segunda opción.

5. **Inténtalo más veces.** Continúa el juego, escribiendo más valores hasta que adivines el número correcto. La aplicación mostrará el mensaje "**Acertaste!!**" y finaliza la ejecución.
6. Cierra la ventana **Símbolo del sistema**.

Estos son sólo dos ejemplos de las muchas aplicaciones que veremos a lo largo del curso. En el siguiente tutorial, podrás empezar a aprender a programar en C, escribiendo tus primeras líneas de código C. Esperamos que disfrute de aprender a construir estas aplicaciones y crear aplicaciones de tu propia cosecha mientras aprendes a programar utilizando C.

## 6. Referencias

- Joyanes Aguilar, J. "Programación en C++. Algoritmos, estructuras de datos y Objetos". Capítulo 1. Ed. McGraw-Hill.
- Pont, M.J. "Software Engineering with C++ and CASE Tools". Capítulo 1. Ed. Addison-Wesley.
- Forouzan, A. Introducción a la Ciencia de la Computación. Cap. 2 y 3. Ed. Thompson, 2003.
- Deitel & Deitel. "Simply C++ An Application Driven Tutorial Approach". Tutorial 1. Ed. Prentice Hall, 2005.



Este obra está bajo una Licencia Creative Commons Atribución-CompartirIgual 3.0 Unported