

Unidad 1: Algoritmos y programas

Fundamentos de Programación. 1º de ASI



Esta obra está bajo una licencia de Creative Commons.

Autor: Jorge Sánchez Asenjo (año 2008) <http://www.jorgesanchez.net>
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons

Para ver una copia de esta licencia, visite:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

o envíe una carta a:

Creative Commons, 559 Nathan Abbot

(1) algoritmos y programas

(1.1) esquema de la unidad

| | |
|--|--------------------------------------|
| (1.1) computadora y sistema operativo | 6 |
| (1.1.1) computadora..... | 6 |
| (1.1.2) hardware y software..... | 8 |
| (1.1.3) Sistema Operativo | 9 |
| (1.2) codificación de la información | 11 |
| (1.2.1) introducción..... | 11 |
| (1.2.2) sistemas numéricos..... | 11 |
| (1.2.3) sistema binario de numeración..... | 12 |
| (1.2.4) representación de texto en el sistema binario | 13 |
| (1.2.5) representación binaria de datos no numéricos ni de texto | 15 |
| (1.2.6) múltiplos para medir dígitos binarios | 15 |
| (1.3) algoritmos | 16 |
| (1.3.1) noción de algoritmo | 16 |
| (1.3.2) características de los algoritmos..... | 17 |
| (1.3.3) elementos que conforman un algoritmo..... | 17 |
| (1.3.4) fases en la creación de algoritmos..... | iError! Marcador no definido. |
| (1.4) aplicaciones | 18 |
| (1.4.1) programas y aplicaciones | 18 |
| (1.4.2) historia del software. La crisis del software..... | 18 |
| (1.4.3) el ciclo de vida de una aplicación | 19 |
| (1.5) errores | 20 |
| (1.6) lenguajes de programación | 21 |
| (1.6.1) breve historia de los lenguajes de programación..... | 21 |
| (1.6.2) tipos de lenguajes | 26 |
| (1.6.3) intérpretes | 27 |
| (1.6.4) compiladores | 28 |
| (1.7) programación | 29 |
| (1.7.1) introducción | 29 |
| (1.7.2) programación desordenada | 29 |
| (1.7.3) programación estructurada..... | 29 |
| (1.7.4) programación modular | 30 |
| (1.7.5) programación orientada a objetos | 30 |
| (1.8) índice de ilustraciones | 31 |

(1.2) computadora y sistema operativo

(1.2.1) computadora

Según la **RAE** (Real Academia de la lengua española), una computadora es una **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**.

En cualquier caso cualquier persona tiene una imagen clara de lo que es una computadora, o como se la conoce popularmente, un **ordenador**. La importancia del ordenador en la sociedad actual es importantísima; de hecho casi no hay tarea que no esté apoyada en la actualidad por el ordenador.

Debido a la importancia y al difícil manejo de estas máquinas, aparece la **informática** como el **conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores**.

Inicialmente, las primeras computadoras eran máquinas basadas en el funcionamiento de relés o de ruedas. Por ello sólo eran capaces de realizar una única tarea.

A finales de los años cuarenta **Von Newman** escribió en un artículo lo que serían las bases del funcionamiento de los ordenadores (seguidos en su mayor parte hasta el día de hoy).

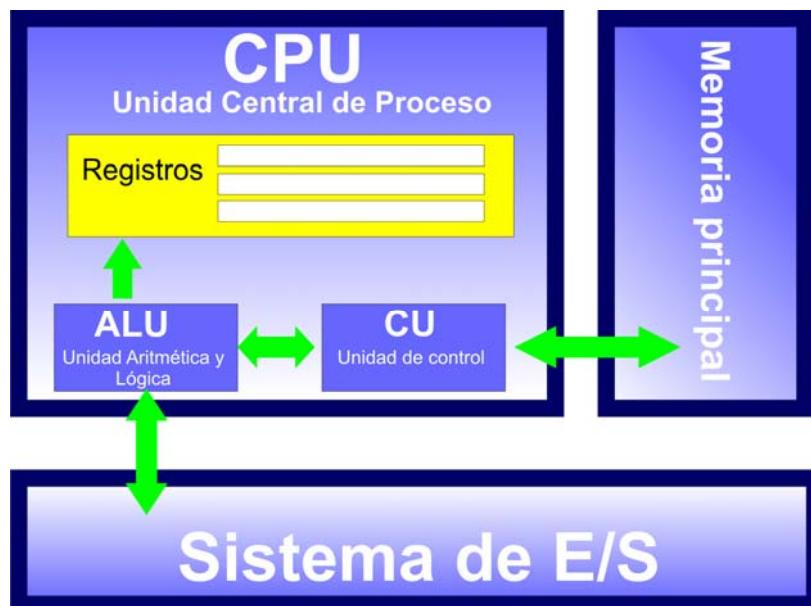


Ilustración 1, Modelo de Von Newman

Las mejoras que consiguió este modelo (entre otras) fueron:

- ◆ Incluir el modelo de **Programa Almacenado** (fundamental para que el ordenador pueda realizar más de una tarea)
- ◆ Aparece el concepto de **Lenguaje de Programación**.
- ◆ Aparece el concepto de programa como amo secuencia de instrucciones secuenciales (aunque pueden incluir bifurcaciones y saltos).

El modelo no ha cambiado excesivamente hasta la actualidad de modo que el modelo actual de los ordenadores es el que se indica en la Ilustración 2.

De los componentes internos del ordenador, cabe destacar el **procesador** (o microprocesador, muchas veces se le llama **microprocesador** término que hace referencia al tamaño del mismo e incluso simplemente **micro**). Se trata de un chip que contiene todos los elementos de la **Unidad Central de Proceso**; por lo que es capaz de realizar e interpretar instrucciones. En realidad un procesador sólo es capaz de realizar tareas sencillas como:

- ◆ Operaciones aritméticas simples: suma, resta, multiplicación y división
- ◆ Operaciones de comparación entre valores
- ◆ Almacenamiento de datos

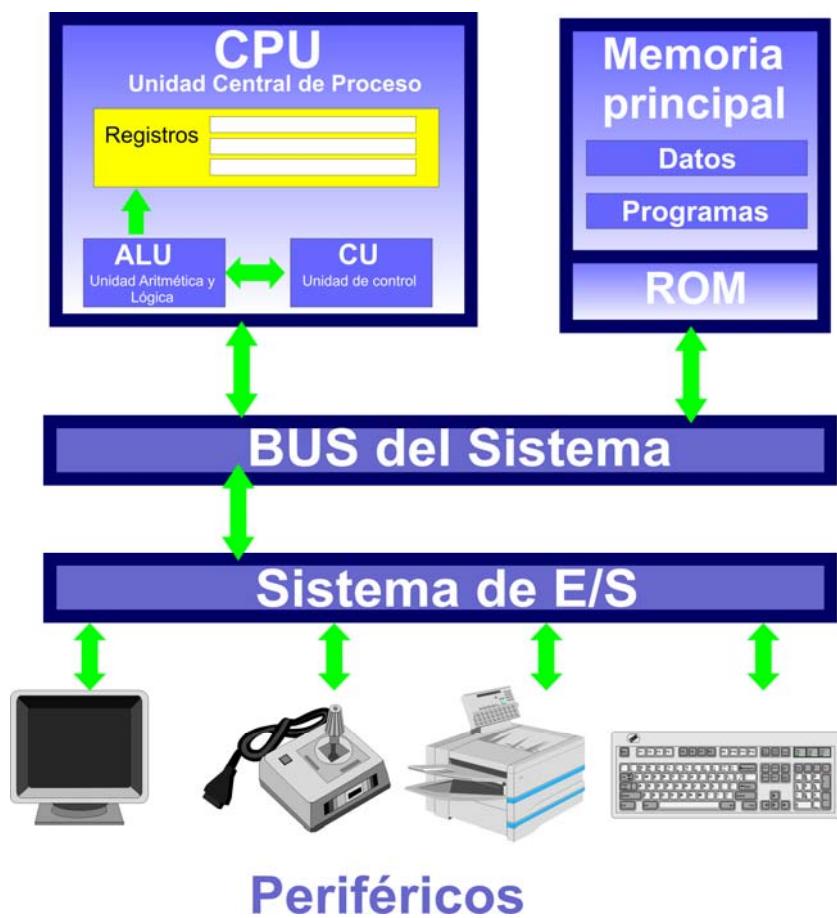


Ilustración 2, arquitectura de los ordenadores actuales

En definitiva los componentes sobre los que actualmente se hace referencia son:

- ◆ **Procesador.** Núcleo digital en el que reside la CPU del ordenador. Es la parte fundamental del ordenador, la encargada de realizar todas las tareas.
- ◆ **Placa base.** Circuito interno al que se conectan todos los componentes del ordenador, incluido el procesador.
- ◆ **Memoria RAM.** Memoria principal del ordenador, formada por un circuito digital que está conectado mediante tarjetas a la placa base. Su contenido se pierde cuando se desconecta al ordenador. Lo que se almacena no es permanente. Mientras el ordenador está funcionando contiene todos los programas y datos con los que el ordenador trabaja.
- ◆ **Memoria caché.** Memoria ultrarrápida de características similares a la RAM, pero de velocidad mucho más elevada por lo que se utiliza para almacenar los últimos datos utilizados de la memoria RAM.
- ◆ **Periféricos.** Aparatos conectados al ordenador mediante tarjetas o ranuras de expansión (también llamados puertos). Los hay de **entrada** (introducen datos en el ordenador: teclado, ratón, escáner,...), de **salida** (muestran datos desde el ordenador: pantalla, impresora, altavoces,...) e incluso de **entrada/salida** (módem, tarjeta de red).
- ◆ **Unidades de almacenamiento.** En realidad son periféricos, pero que sirven para almacenar de forma permanente los datos que se deseen del ordenador. Los principales son el **disco duro** (unidad de gran tamaño interna al ordenador), la **disquetera** (unidad de baja capacidad y muy lenta, ya en desuso), el **CD-ROM** y el **DVD**.

(1.2.2) hardware y software

hardware

Se trata de todos los componentes físicos que forman parte de un ordenador (o de otro dispositivo electrónico): procesador, RAM, impresora, teclado, ratón,...

software

Se trata de la parte conceptual del ordenador. Es decir los datos y aplicaciones que maneja. De forma más práctica se puede definir como cualquier cosa que se pueda almacenar en una unidad de almacenamiento es software (la propia unidad sería hardware).

(1.2.3) Sistema Operativo

Se trata del software (programa) encargado de gestionar el ordenador. Es la aplicación que oculta la física real del ordenador para mostrarnos un interfaz que permita al usuario un mejor y más fácil manejo de la computadora.

funciones del Sistema Operativo

Las principales funciones que desempeña un Sistema Operativo son:

- ◆ Permitir al usuario comunicarse con el ordenador. A través de comandos o a través de una interfaz gráfica.
- ◆ Coordinar y manipular el hardware de la computadora: memoria, impresoras, unidades de disco, el teclado,...
- ◆ Proporcionar herramientas para organizar los datos de manera lógica (carpetas, archivos,...)
- ◆ Proporcionar herramientas para organizar las aplicaciones instaladas.
- ◆ Gestionar el acceso a redes
- ◆ Gestionar los errores de hardware y la pérdida de datos.
- ◆ Servir de base para la creación de aplicaciones, proporcionando funciones que faciliten la tarea a los programadores.
- ◆ Administrar la configuración de los usuarios.
- ◆ Proporcionar herramientas para controlar la seguridad del sistema.

algunos sistemas operativos

- ◆ **Windows.** A día de hoy el Sistema Operativo más popular (instalado en el 95% de computadoras del mundo). Es un software propiedad de Microsoft por el que hay que pagar por cada licencia de uso.
- ◆ **Unix.** Sistema operativo muy robusto para gestionar redes de todos los tamaños. Actualmente en desuso debido al uso de Linux (que está basado en Unix), aunque sigue siendo muy utilizado para gestionar grandes redes (el soporte sigue siendo una de las razones para que se siga utilizando)
- ◆ **Solaris.** Versión de Unix para sistemas de la empresa **Sun**.
- ◆ **MacOs.** Sistema operativo de los ordenadores **MacIntosh**. Muy similar al sistema Windows y orientado al uso de aplicaciones de diseño gráfico.

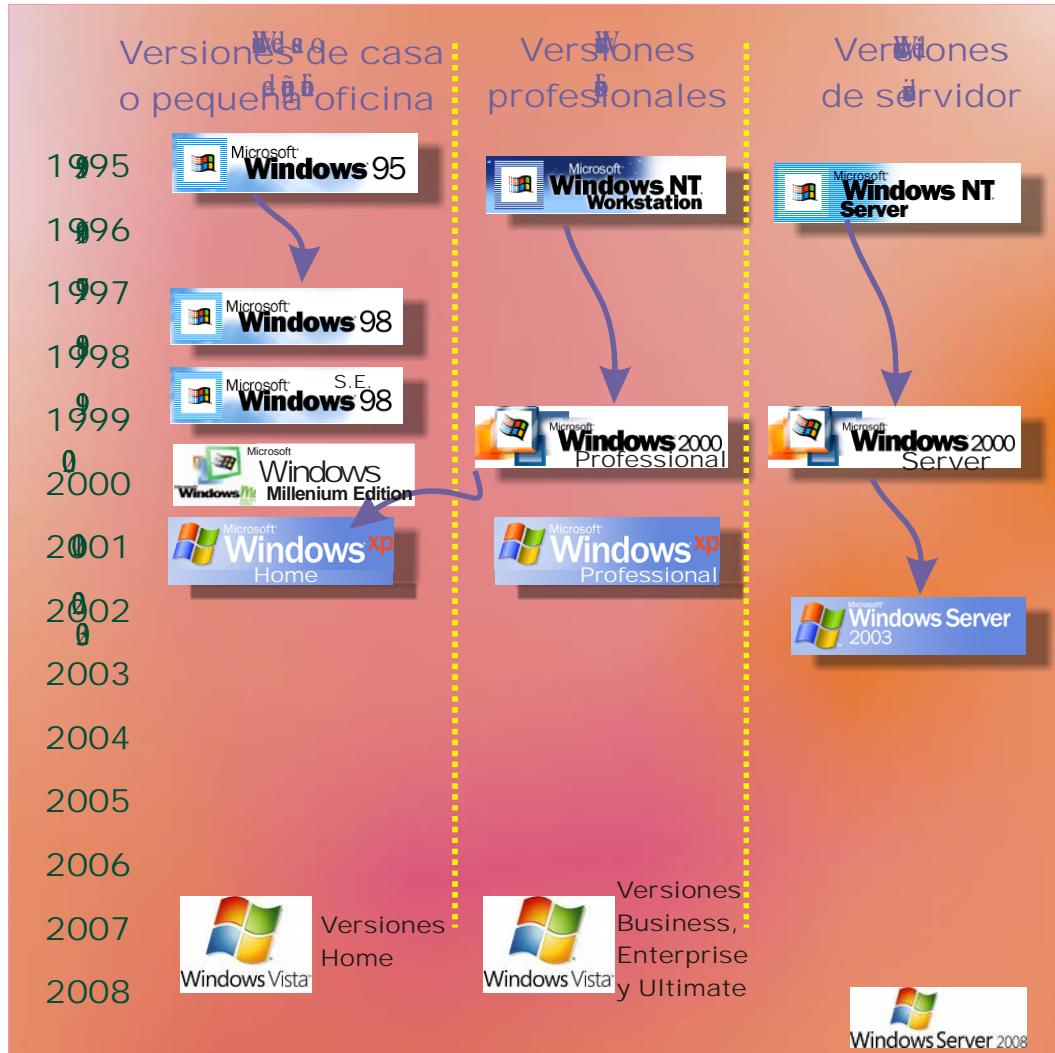


Ilustración 3, Histórico de versiones de Windows

♦ **Linux.** Sistema operativo de código abierto, lo que significa que el código fuente está a disposición de cualquier programador, lo que permite adecuar el sistema a las necesidades de cada usuario.

Esta libertad ha hecho que posea numerosas distribuciones, muchas de ellas gratuitas. La variedad de distribuciones y opciones complica su aprendizaje al usuario inicial, pero aumenta las posibilidades de selección de un sistema adecuado.

La sintaxis de Linux está basada en Linux, de hecho se trata de un Unix de código abierto pensado fundamentalmente para los ordenadores de tipo PC.

Actualmente las distribuciones Linux más conocidas son:

- **Red Hat**
- **Fedora** (versión gratuita de Red Hat)
- **Debian**

- **Ubuntu** (variante de Debian de libre distribución, quizá el Linux más exitoso de la actualidad)
- **Mandriva**
- **SUSE**

(1.3) codificación de la información

(1.3.1) introducción

Un ordenador maneja información de todo tipo. Nuestra perspectiva humana nos permite rápidamente diferenciar lo que son números, de lo que es texto, imagen,...

Sin embargo al tratarse de una máquina digital, el ordenador sólo es capaz de representar números en forma binaria. Por ello todos los ordenadores necesitan codificar la información del mundo real a el equivalente binario entendible por el ordenador.

Desde los inicios de la informática la codificación ha sido problemática y por la falta de acuerdo en la representación. Pero hoy día ya tenemos numerosos estándares.

Fundamentalmente la información que un ordenador maneja es: Números y Texto. Cualquier otro tipo de información (imagen, sonido, vídeo,...) se considera binaria (aunque como ya hemos comentado, toda la información que maneja un ordenador es binaria).

(1.3.2) sistemas numéricos

En general, a lo largo de la historia han existido numerosos sistemas de numeración. Cada cultura o civilización se ha servido en la antigüedad de los sistemas que ha considerado más pertinentes. Para simplificar, dividiremos a todos los sistemas en dos tipos:

- ◆ **Sistemas no posicionales.** En ellos se utilizan símbolos cuyo valor numérico es siempre el mismo independientemente de donde se sitúen. Es lo que ocurre con la numeración romana. En esta numeración el símbolo **I** significa siempre **uno** independientemente de su posición.
- ◆ **Sistemas posicionales.** En ellos los símbolos numéricos cambian de valor en función de la posición que ocupen. Es el caso de nuestra numeración, el símbolo **2**, en la cifra **12** vale **2**; mientras que en la cifra **21** vale veinte.

La historia ha demostrado que los sistemas posicionales son mucho mejores para los cálculos matemáticos por lo que han retirado a los no posicionales. La razón: las operaciones matemáticas son más sencillas utilizando sistemas posicionales.

Todos los sistemas posicionales tienen una **base**, que es el número total de símbolos que utiliza el sistema. En el caso de la numeración decimal la base es 10; en el sistema binario es 2.

El **Teorema Fundamental de la Numeración** permite saber el valor decimal que tiene cualquier número en cualquier base. Dicho teorema utiliza la fórmula:

$$\dots + X_3 \cdot B^3 + X_2 \cdot B^2 + X_1 \cdot B^1 + X_0 \cdot B^0 + X_{-1} \cdot B^{-1} + X_{-2} \cdot B^{-2} + \dots$$

Donde:

- ◆ **X_i** Es el símbolo que se encuentra en la posición número i del número que se está convirtiendo. Teniendo en cuenta que la posición de las unidades es la posición 0 (la posición -1 sería la del primer decimal)
 - ◆ **B** Es la base del sistemas que se utiliza para representar al número

Por ejemplo si tenemos el número **153,6** utilizando el sistema octal (base ocho), el paso a decimal se haría:

$$1 \cdot 8^2 + 5 \cdot 8^1 + 3 \cdot 8^0 + 6 \cdot 8^{-1} = 64 + 40 + 3 + 6/8 = 107,75$$

(1.3.3) sistema binario de numeración

conversión binario a decimal

El **teorema fundamental de la numeración** se puede aplicar para saber el número decimal representado por un número escrito en binario. Así para el número binario 10011011011 la conversión se haría:

$$1 \cdot 2^{10} + 0 \cdot 2^9 + 0 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1243$$

Puesto que simplemente es sumar las potencias de los dígitos con el número 1, se puede convertir asignando un peso a cada cifra. De modo que la primera cifra (la que está más a la derecha) se corresponde con la potencia 2^0 o sea uno, la siguiente 2^1 o sea dos, la siguiente 2^2 que es cuatro y así sucesivamente; de esta forma, por ejemplo para el número 10100101, se haría:

| | | | | | | | | |
|--------|----------|----------|----------|----------|----------|----------|----------|----------|
| Pesos: | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

ahora se suman los pesos que tienen un 1: $128 + 32 + 4 + 1 = 165$

conversión decimal a binario

El método más utilizado es ir haciendo divisiones sucesivas entre dos. Consiste en dividir el número decimal continuamente entre 2 tomando los restos a la inversa y el último cociente. Por ejemplo para pasar el 179:

$$\begin{array}{r}
 179 \overline{)2} \\
 19 \quad 89 \overline{)2} \\
 \textcircled{1} \quad 09 \quad 44 \quad \overline{)2} \\
 \textcircled{1} \quad 04 \quad 02 \quad \overline{)2} \\
 \textcircled{0} \quad \textcircled{0} \quad \textcircled{1} \quad 11 \quad \overline{)2} \\
 \textcircled{0} \quad \textcircled{1} \quad \textcircled{1} \quad 5 \quad \overline{)2} \\
 \textcircled{0} \quad \textcircled{1} \quad \textcircled{1} \quad 2 \quad \overline{)2} \\
 \textcircled{0} \quad \textcircled{1}
 \end{array}$$

Ahora las cifras binarias se toman al revés. Con lo cual, el número 10110011 es el equivalente en binario de 179.

Otra posibilidad más rápida es mediante restas sucesivas. En este caso se colocan todas las potencias de dos hasta sobrepasar el número decimal. La primera potencia (el primer peso) que es menor a nuestro número será un 1 en el número binario. Se resta nuestro número menos esa potencia y se coge el resultado. Se ponen a 0 todas las potencias mayores que ese resultado y un 1 en la primera potencia menor. Se vuelve a restar y así sucesivamente. Ejemplo:

Número decimal: 179

1) Obtener potencias de 2 hasta sobrepasar el número:

1 2 4 8 16 32 64 128 256

2) Proceder a las restas (observar de abajo a arriba):

| Potencias | Pesos | Restas |
|-----------|-------|------------|
| 1 | 1 | 1-1=0 |
| 2 | 1 | 3-2=1 |
| 4 | 0 | |
| 8 | 0 | |
| 16 | 1 | 19-16=3 |
| 32 | 1 | 51-32=19 |
| 64 | 0 | |
| 128 | 1 | 179-128=51 |
| 256 | 0 | |

2) El número binario es el 10110011

sistema octal

Un caso especial de numeración es el sistema octal. En este sistema se utiliza base ocho, es decir cifras de 0 a 7. Así el número 8 en decimal, pasa a escribirse 10 en octal.

Lo interesante de este sistema es que tiene una traducción directa y rápida al sistema binario, ya que una cifra octal se corresponde exactamente con tres cifras binarias (tres bits). La razón es que el número ocho es 2^3 . De ahí que tres bits signifique una cifra octal. Los números del 0 al 7 en octal pasados a binario quedarían así:

| Decimal | Binario (tres bits) |
|---------|---------------------|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

De este modo el número binario 1100011, se agrupa de tres en tres (empezando por la derecha), es decir 1 100 011 y en octal quedaría: 143. Del mismo modo, el número octal 217 en binario es el 010 001 111

sistema hexadecimal

Se utiliza muchísimo en informática por razones parecidas al sistema octal. Es el sistema de numeración en base 16. Tiene la ventaja de que una cifra hexadecimal representa exactamente 4 bits (lo que comúnmente se conoce como un **nibble**). De este modo 2 cifras hexadecimales representan un byte (8 bits), que es la unidad básica para medir.

Puesto que en nuestro sistema no disponemos de 16 cifras, sino solo de 10. se utilizan letras para representar los números del 10 al 15. Tabla:

| Hexadecimal | Decimal | Binario (cuatro bits) |
|-------------|---------|-----------------------|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

Así el número binario **10110011** se agruparía como 1011 0011 y sería el **B3** en hexadecimal. A su vez el número hexadecimal **CA** quedaría **1100 1010**

(1.3.4) representación de texto en el sistema binario

Puesto que una computadora no sólo maneja números, habrá dígitos binarios que contengan información que no es traducible a decimal. Todo depende de cómo se interprete esa traducción.

Por ejemplo en el caso del texto, lo que se hace es codificar cada carácter en una serie de números binarios. El código **ASCII** ha sido durante mucho tiempo el más utilizado. Inicialmente era un código que utilizaba 7 bits para representar texto, lo que significaba que era capaz de codificar 127 caracteres. Por ejemplo el número 65 (1000001 en binario) se utiliza para la A mayúscula.

Poco después apareció un problema: este código es suficiente para los caracteres del inglés, pero no para otras lenguas. Entonces se añadió el octavo bit para representar otros 128 caracteres que son distintos según idiomas (Europa Occidental usa unos códigos que no utiliza Europa Oriental).

Eso provoca que un código como el 190 signifique cosas diferentes si cambiamos de país. Por ello cuando un ordenador necesita mostrar texto, tiene que saber qué juego de códigos debe de utilizar (lo cual supone un tremendo problema).

Una ampliación de este método de codificación es el código **Unicode** que puede utilizar hasta 4 bytes (32 bits) con lo que es capaz de codificar cualquier carácter en cualquier lengua del planeta utilizando el mismo conjunto de códigos. Poco a poco es el código que se va extendiendo; pero la preponderancia histórica que ha tenido el código ASCII, complica su popularidad.

(1.3.5) representación binaria de datos no numéricos ni de texto

En el caso de datos más complejos (imágenes, vídeo, audio) se necesita una codificación más compleja. Además en estos datos no hay estándares, por lo que hay decenas de formas de codificar.

En el caso, por ejemplo, de las imágenes, una forma básica de codificarlas en binario es la que graba cada **píxel** (cada punto distingible en la imagen) mediante tres bytes: el primero graba el nivel de rojo, el segundo el nivel de azul y el tercero el nivel de verde. Y así por cada píxel.

Por ejemplo un punto en una imagen de color rojo puro

11111111 00000000 00000000

Naturalmente en una imagen no solo se graban los píxeles sino el tamaño de la imagen, el modelo de color,... de ahí que representar estos datos sea tan complejo para el ordenador (y tan complejo entenderlo para nosotros).

(1.3.6) múltiplos para medir dígitos binarios

Puesto que toda la información de un ordenador se representa de forma binaria, se hizo indispensable el utilizar unidades de medida para poder indicar la capacidad de los dispositivos que manejaban dichos números.

Así tenemos:

- ◆ **BIT (de *Binary diGIT*)**. Representa un dígito binario. Por ejemplo se dice que el número binario 1001 tiene cuatro BITS.
- ◆ **Byte**. Es el conjunto de 8 BITS. Es importante porque normalmente cada carácter de texto almacenado en un ordenador consta de 8 bits (luego podemos entender que en 20 bytes nos caben 20 caracteres).
- ◆ **Kilobyte**. Son 1024 bytes.
- ◆ **Megabyte**. Son 1024 Kilobytes.
- ◆ **Gigabyte**. Son 1024 Megabytes.
- ◆ **Terabyte**. Son 1024 Gigabytes.
- ◆ **Petabyte**. Son 1024 Terabytes.

(1.4) algoritmos

(1.4.1) noción de algoritmo

Según la **RAE**: *conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.*

Los algoritmos, como indica su definición oficial, son una serie de pasos que permiten obtener la solución a un problema. La palabra algoritmo procede del matemático Árabe **Mohamed Ibn Al Kow Rizmi**, el cual escribió sobre los años 800 y 825 su obra **Quitad Al Mugabala**, donde se recogía el sistema de numeración hindú y el concepto del cero. **Fibonacci**, tradujo la obra al latín y la llamó: **Algoritmi Dicit**.

El lenguaje algorítmico es aquel que implementa una solución teórica a un problema indicando las operaciones a realizar y el orden en el que se deben efectuar. Por ejemplo en el caso de que nos encontremos en casa con una bombilla fundida en una lámpara, un posible algoritmo sería:

- (1)** Comprobar si hay bombillas de repuesto
- (2)** En el caso de que las haya, sustituir la bombilla anterior por la nueva
- (3)** Si no hay bombillas de repuesto, bajar a comprar una nueva a la tienda y sustituir la vieja por la nueva

Los algoritmos son la base de la programación de ordenadores, ya que los **programas de ordenador** se puede entender que son algoritmos escritos en un código especial entendible por un ordenador.

Lo malo del diseño de algoritmos está en que no podemos escribir lo que deseemos, el lenguaje ha utilizar no debe dejar posibilidad de duda, debe recoger todas las posibilidades.

Por lo que los tres pasos anteriores pueden ser mucho más largos:

Comprobar si hay bombillas de repuesto

- [1.1]** Abrir el cajón de las bombillas
- [1.2]** Observar si hay bombillas

Si hay bombillas:

- [1.3]** Coger la bombilla
- [1.4]** Coger una silla
- [1.5]** Subirse a la silla
- [1.6]** Poner la bombilla en la lámpara

Si no hay bombillas

- [1.7]** Abrir la puerta
- [1.8]** Bajar las escaleras....

Cómo se observa en un algoritmo las instrucciones pueden ser más largas de lo que parecen, por lo que hay que determinar qué instrucciones se pueden utilizar y qué instrucciones no se pueden utilizar. En el caso de los algoritmos preparados para el ordenador, se pueden utilizar sólo instrucciones muy concretas.

(1.4.2) características de los algoritmos

características que deben de cumplir los algoritmos obligatoriamente

- ◆ **Un algoritmo debe resolver el problema para el que fue formulado.** Lógicamente no sirve un algoritmo que no resuelve ese problema. En el caso de los programadores, a veces crean algoritmos que resuelven problemas diferentes al planteado.
- ◆ **Los algoritmos son independientes del ordenador.** Los algoritmos se escriben para poder ser utilizados en cualquier máquina.
- ◆ **Los algoritmos deben de ser precisos.** Los resultados de los cálculos deben de ser exactos, de manera rigurosa. No es válido un algoritmo que sólo aproxime la solución.
- ◆ **Los algoritmos deben de ser finitos.** Deben de finalizar en algún momento. No es un algoritmo válido aquel que produce situaciones en las que el algoritmo no termina.
- ◆ **Los algoritmos deben de poder repetirse.** Deben de permitir su ejecución las veces que haga falta. No son válidos los que tras ejecutarse una vez, ya no pueden volver a hacerlo por la razón que sea.

características aconsejables para los algoritmos

- ◆ **Validez.** Un algoritmo es válido si carece de errores. Un algoritmo puede resolver el problema para el que se planteó y sin embargo no ser válido debido a que posee errores
- ◆ **Eficiencia.** Un algoritmo es eficiente si obtiene la solución al problema en poco tiempo. No lo es si es lento en obtener el resultado.
- ◆ **Óptimo.** Un algoritmo es óptimo si es el más eficiente posible y no contiene errores. La búsqueda de este algoritmo es el objetivo prioritario del programador. No siempre podemos garantizar que el algoritmo hallado es el óptimo, a veces sí.

(1.4.3) elementos que conforman un algoritmo

- ◆ **Entrada.** Los datos iniciales que posee el algoritmo antes de ejecutarse.
- ◆ **Proceso.** Acciones que lleva a cabo el algoritmo.
- ◆ **Salida.** Datos que obtiene finalmente el algoritmo.

(1.5) aplicaciones

(1.5.1) programas y aplicaciones

- ◆ **Programa.** La definición de la **RAE** es: *Conjunto unitario de instrucciones que permite a un ordenador realizar funciones diversas, como el tratamiento de textos, el diseño de gráficos, la resolución de problemas matemáticos, el manejo de bancos de datos*, etc. Pero normalmente se entiende por programa un **conjunto de instrucciones ejecutables por un ordenador**.
Un **programa estructurado** es un programa que cumple las condiciones de un algoritmo (finitud, precisión, repetición, resolución del problema,...)
- ◆ **Aplicación.** Software formado por uno o más programas, la documentación de los mismos y los archivos necesarios para su funcionamiento, de modo que el conjunto completo de archivos forman una herramienta de trabajo en un ordenador.

Normalmente en el lenguaje cotidiano no se distingue entre aplicación y programa; en nuestro caso entenderemos que la aplicación es un software completo que cumple la función completa para la que fue diseñado, mientras que un programa es el resultado de ejecutar un cierto código entendible por el ordenador.

(1.5.2) historia del software. La crisis del software

Los primeros ordenadores cumplían una única programación que estaba definida en los componentes eléctricos que formaban el ordenador.

La idea de que el ordenador hiciera varias tareas (ordenador programable o multipropósito) hizo que se idearan las **tarjetas perforadas**. En ellas se utilizaba código binario, de modo que se hacían agujeros en ellas para indicar el código 1 o el cero. Estos “primeros programas” lógicamente servían para hacer tareas muy concretas.

La llegada de ordenadores electrónicos más potentes hizo que los ordenadores se convirtieran en verdaderas máquinas digitales que seguían utilizando el 1 y el 0 del código binario pero que eran capaces de leer miles de unos y ceros. Empezaron a aparecer los primeros lenguajes de programación que escribían código más entendible por los humanos que posteriormente era convertido al código entendible por la máquina.

Inicialmente la creación de aplicaciones requería escribir pocas líneas de código en el ordenador, por lo que no había una técnica específica para especificar a la hora de crear programas. Cada programador se defendía como podía generando el código a medida que se le ocurría.

Poco a poco las funciones que se requerían a los programas fueron aumentando produciendo miles de líneas de código que al estar desorganizada hacían casi imposible su mantenimiento. Sólo el programador que había escrito el código era capaz de entenderlo y eso no era en absoluto práctico.

La llamada **crisis del software** ocurrió cuando se percibió que se gastaba más tiempo en hacer las modificaciones a los programas que en volver a crear el

software. La razón era que ya se habían codificado millones de líneas de código antes de que se definiera un buen método para crear los programas.

La solución a esta crisis ha sido la definición de la **ingeniería del software** como un oficio que requería un método de trabajo similar al del resto de ingenierías. La búsqueda de una metodología de trabajo que elimine esta crisis parece que aún no está resuelta, de hecho los métodos de trabajo siguen redefiniéndose una y otra vez.

(1.5.3) el ciclo de vida de una aplicación

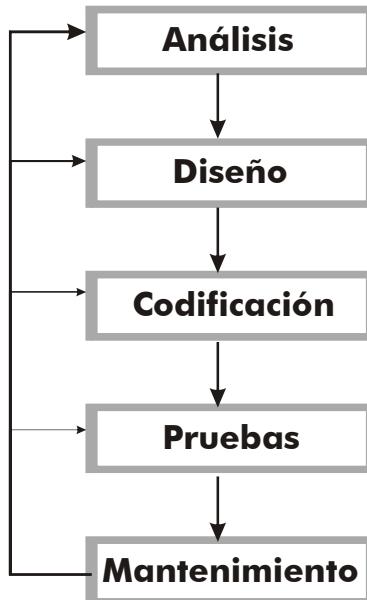


Ilustración 4, Ciclo de vida de una aplicación

Una de las cosas que se han definido tras el nacimiento de la ingeniería del software ha sido el ciclo de vida de una aplicación. El ciclo de vida define los pasos que sigue el proceso de creación de una aplicación desde que se propone hasta que finaliza su construcción. Los pasos son:

- (1) Análisis.** En esta fase se determinan los requisitos que tiene que cumplir la aplicación. Se anota todo aquello que afecta al futuro funcionamiento de la aplicación. Este paso lo realiza un analista
- (2) Diseño.** Se especifican los esquemas de diseño de la aplicación. Estos esquemas forman los **planos** del programador, los realiza el analista y representan todos los aspectos que requiere la creación de la aplicación.
- (3) Codificación.** En esta fase se pasa el diseño a código escrito en algún lenguaje de programación. Esta es la primera labor que realiza el programador
- (4) Pruebas.** Se trata de comprobar que el funcionamiento de la aplicación es la adecuada. Se realiza en varias fases:

- a) **Prueba del código.** Las realizan programadores. Normalmente programadores distintos a los que crearon el código, de ese modo la prueba es más independiente y generará resultados más óptimos.
 - b) **Versión alfa.** Es una primera versión terminada que se revisa a fin de encontrar errores. Estas pruebas conviene que sean hechas por personal no informático. El producto sólo tiene cierta apariencia de acabado.
 - c) **Versión beta.** Versión casi definitiva del software en la que no se estiman fallos, pero que se distribuye a los clientes para que encuentren posibles problemas. A veces esta versión acaba siendo la definitiva (como ocurre con muchos de los programas distribuidos libremente por Internet).
- (5) **Mantenimiento.** Tiene lugar una vez que la aplicación ha sido ya distribuida, en esta fase se asegura que el sistema siga funcionando aunque cambien los requisitos o el sistema para el que fue diseñado el software. Antes esos cambios se hacen los arreglos pertinentes, por lo que habrá que retroceder a fases anteriores del ciclo de vida.

(1.6) errores

Cuando un programa obtiene una salida que no es la esperada, se dice que posee errores. Los errores son uno de los caballos de batalla de los programadores ya que a veces son muy difíciles de encontrar (de ahí que hoy en día en muchas aplicaciones se distribuyan *parches* para subsanar errores no encontrados en la creación de la aplicación).

tipos de errores

- ◆ **Error del usuario.** Errores que se producen cuando el usuario realiza algo inesperado y el programa no reacciona apropiadamente.
- ◆ **Error del programador.** Son errores que ha cometido el programador al generar el código. La mayoría de errores son de este tipo.
- ◆ **Errores de documentación.** Ocurren cuando la documentación del programa no es correcta y provoca fallos en el manejo
- ◆ **Error de interfaz.** Ocurre si la interfaz de usuario de la aplicación es enrevesada para el usuario impidiendo su manejo normal. También se llaman así los errores de protocolo entre dispositivos.
- ◆ **Error de entrada / salida o de comunicaciones.** Ocurre cuando falla la comunicación entre el programa y un dispositivo (se desea imprimir y no hay papel, falla el teclado,...)
- ◆ **Error fatal.** Ocurre cuando el hardware produce una situación inesperada que el software no puede controlar (el ordenador se cuelga, errores en la grabación de datos,...)

- ◆ **Error de ejecución.** Ocurren cuando la ejecución del programa es más lenta de lo previsto.

La labor del programador es predecir, encontrar y subsanar (si es posible) o al menos controlar los errores. Una mala gestión de errores causa experiencias poco gratas al usuario de la aplicación.

(1.7) lenguajes de programación

(1.7.1) breve historia de los lenguajes de programación

inicios de la programación

Charles Babbage definió a mediados del siglo XIX lo que él llamó la **máquina analítica**. Se considera a esta máquina el diseño del primer ordenador. La realidad es que no se pudo construir hasta el siglo siguiente. El caso es que su colaboradora **Ada Lovelace** escribió en tarjetas perforadas una serie de instrucciones que la máquina iba a ser capaz de ejecutar. Se dice que eso significó el inicio de la ciencia de la programación de ordenadores.

En la segunda guerra mundial debido a las necesidades militares, la ciencia de la computación prospera y con ella aparece el famoso **ENIAC** (*Electronic Numerical Integrator And Calculator*), que se programaba cambiando su circuitería. Esa es la primera forma de programar (que aún se usa en numerosas máquinas) que sólo vale para **máquinas de único propósito**. Si se cambia el propósito, hay que modificar la máquina.

código máquina. primera generación de lenguajes (IGL)

No mucho más tarde apareció la idea de que las máquinas fueran capaces de realizar más de una aplicación. Para lo cual se ideó el hecho de que hubiera una memoria donde se almacenaban esas instrucciones. Esa memoria se podía llenar con datos procedentes del exterior. Inicialmente se utilizaron tarjetas perforadas para introducir las instrucciones.

Durante mucho tiempo esa fue la forma de programar, que teniendo en cuenta que las máquinas ya entendían sólo código binario, consistía en introducir la programación de la máquina mediante unos y ceros. El llamado **código máquina**. Todavía los ordenadores es el único código que entienden, por lo que cualquier forma de programar debe de ser convertida a código máquina.

Sólo se ha utilizado por los programadores en los inicios de la informática. Su incomodidad de trabajo hace que sea impensable para ser utilizado hoy en día. Pero cualquier programa de ordenador debe, finalmente, ser convertido a este código para que un ordenador puede ejecutar las instrucciones de dicho programa.

Un detalle a tener en cuenta es que el código máquina es distinto para cada tipo de procesador. Lo que hace que los programas en código máquina no sean portables entre distintas máquinas.

lenguaje ensamblador. segunda generación de lenguajes (2GL)

En los años 40 se intentó concebir un lenguaje más simbólico que permitiera no tener que programar utilizando código máquina. Poco más tarde se ideó el **lenguaje ensamblador**, que es la traducción del código máquina a una forma más textual. Cada tipo de instrucción se asocia a una palabra mnemotécnica (como SUM para sumar por ejemplo), de forma que cada palabra tiene traducción directa en el código máquina.

Tras escribir el programa en código ensamblador, un programa (llamado también ensamblador) se encargará de traducir el código ensamblador a código máquina. Esta traducción es rápida puesto que cada línea en ensamblador tiene equivalente directo en código máquina (en los lenguajes modernos no ocurre esto).

La idea es la siguiente: si en el código máquina, el número binario 0000 significa sumar, y el número 0001 significa restar. Una instrucción máquina que sumara el número 8 (00001000 en binario) al número 16 (00010000 en binario) sería:

```
0000 00001000 00010000
```

Realmente no habría espacios en blanco, el ordenador entendería que los primeros cuatro BITS representan la instrucción y los 8 siguientes el primer número y los ocho siguientes el segundo número (suponiendo que los números ocupan 8 bits). Lógicamente trabajar de esta forma es muy complicado. Por eso se podría utilizar la siguiente traducción en ensamblador:

```
SUM 8 16
```

Que ya se entiende mucho mejor.

Ejemplo¹ (programa que saca el texto “Hola mundo” por pantalla):

```
DATOS SEGMENT
    saludo db "Hola mundo!!!", "$"
DATOS ENDS
CODE SEGMENT
    assume cs:code,ds:datos
START PROC
    mov ax,datos
    mov ds,ax
    mov dx,offset saludo
    mov ah,9
    int 21h
    mov ax,4C00h
    int 21h
START ENDP
CODE ENDS
END START
```

¹ Ejemplo tomado de la página <http://www.victorsanchez2.net>

Puesto que el ensamblador es una representación textual pero exacta del código máquina; cada programa sólo funcionará para la máquina en la que fue concebido el programa; es decir, no es **portable**.

La ventaja de este lenguaje es que se puede controlar absolutamente el funcionamiento de la máquina, lo que permite crear programas muy eficientes. Lo malo es precisamente que hay que conocer muy bien el funcionamiento de la computadora para crear programas con esta técnica. Además las líneas requeridas para realizar una tarea se disparan ya que las instrucciones de la máquina son excesivamente simples.

lenguajes de alto nivel. lenguajes de tercera generación (3GL)

Aunque el ensamblador significó una notable mejora sobre el código máquina, seguía siendo excesivamente críptico. De hecho para hacer un programa sencillo requiere miles y miles de líneas de código.

Para evitar los problemas del ensamblador apareció la tercera generación de lenguajes de programación, la de los lenguajes de alto nivel. En este caso el código vale para cualquier máquina pero deberá ser traducido mediante software especial que adaptará el código de alto nivel al código máquina correspondiente. Esta traducción es necesaria ya que el código en un lenguaje de alto nivel no se parece en absoluto al código máquina.

Tras varios intentos de representar lenguajes, en 1957 aparece el que se considera el primer lenguaje de alto nivel, el **FORTRAN** (**FO**rmat **TR**anslation), lenguaje orientado a resolver fórmulas matemáticos. Por ejemplo la forma en FORTRAN de escribir el texto **Hola mundo** por pantalla es:

```
PROGRAM HOLA  
PRINT *, '¡Hola, mundo!'  
END
```

Poco a poco fueron evolucionando los lenguajes formando lenguajes cada vez mejores (ver). Así en 1958 se crea **LISP** como lenguaje declarativo para expresiones matemáticas.

Programa que escribe **Hola mundo** en lenguaje LISP:

```
(format t "¡Hola, mundo!")
```

En 1960 la conferencia **CODASYL** se creó el **COBOL** como lenguaje de gestión en 1960. En 1963 se creó **PL/I** el primer lenguaje que admitía la multitarea y la programación modular. En COBOL el programa **Hola mundo** sería éste (como se ve es un lenguaje más declarativo):

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. HELLO.  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
PROCEDURE DIVISION.  
MAIN SECTION.  
DISPLAY "Hola mundo"  
STOP RUN.
```

BASIC se creó en el año 1964 como lenguaje de programación sencillo de aprender en 1964 y ha sido, y es, uno de los lenguajes más populares. En 1968 se crea **LOGO** para enseñar a programar a los niños. **Pascal** se creó con la misma idea académica pero siendo ejemplo de lenguaje estructurado para programadores avanzados. El creador del Pascal (**Niklaus Wirth**) creó **Modula** en 1977 siendo un lenguaje estructurado para la programación de sistemas (intentando sustituir al **C**).

Programa que escribe por pantalla **Hola mundo** en lenguaje Pascal):

```
PROGRAM HolaMundo;  
BEGIN  
  Writeln('¡Hola, mundo!');  
END.
```

lenguajes de cuarta generación (4GL)

En los años 70 se empezó a utilizar éste término para hablar de lenguajes en los que apenas hay código y en su lugar aparecen indicaciones sobre qué es lo que el programa debe de obtener. Se consideraba que el lenguaje SQL (muy utilizado en las bases de datos) y sus derivados eran de cuarta generación. Los lenguajes de consulta de datos, creación de formularios, informes,... son lenguajes de cuarto nivel. Aparecieron con los sistemas de base de datos

Actualmente se consideran lenguajes de éste tipo a aquellos lenguajes que se programan sin escribir casi código (lenguajes visuales), mientras que también se propone que éste nombre se reserve a los lenguajes orientados a objetos.

lenguajes orientados a objetos

En los 80 llegan los lenguajes preparados para la programación orientada a objetos todos procedentes de **Simula** (1964) considerado el primer lenguaje con facilidades de uso de objetos. De estos destacó inmediatamente **C++**.

A partir de **C++** aparecieron numerosos lenguajes que convirtieron los lenguajes clásicos en lenguajes orientados a objetos (y además con mejoras en el entorno de programación, son los llamados lenguajes **visuales**): **Visual Basic**, **Delphi** (versión orientada a objetos de Pascal), **Visual C++**,...

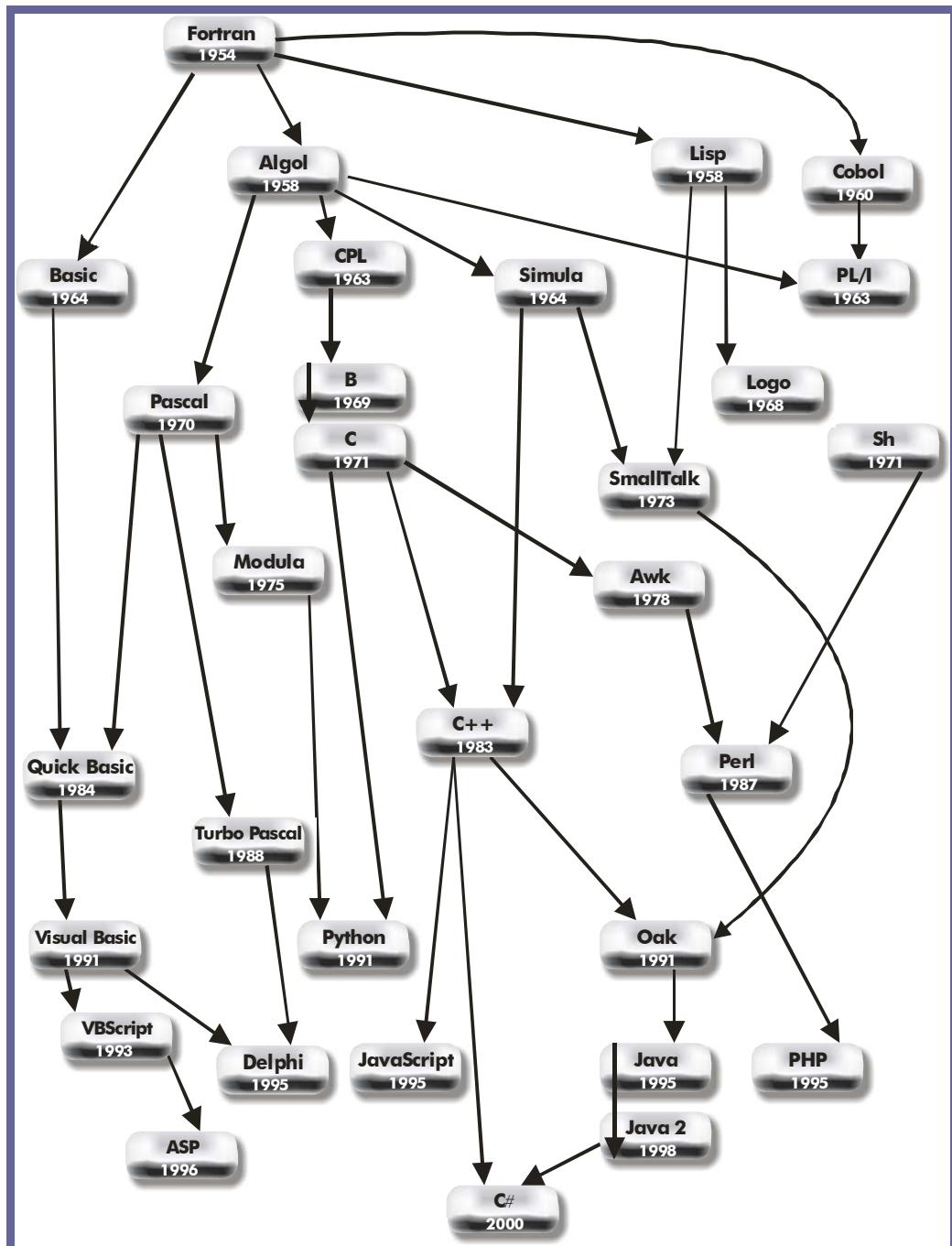


Ilustración 5, Evolución de algunos lenguajes de programación

En 1995 aparece Java como lenguaje totalmente orientado a objetos y en el año 2000 aparece C# un lenguaje que toma la forma de trabajar de C++ y del propio Java.

El programa **Hola mundo** en C# sería:

```
using System;

class MainClass
{
    public static void Main()
    {
        Console.WriteLine("¡Hola, mundo!");
    }
}
```

lenguajes para la web

La popularidad de Internet ha producido lenguajes híbridos que se mezclan con el código **HTML** con el que se crean las páginas web. HTML no es un lenguaje en sí sino un formato de texto pensado para crear páginas web. Éstos lenguajes se usan para poder realizar páginas web más potentes.

Son lenguajes interpretados como **JavaScript** o **VB Script**, o lenguajes especiales para uso en servidores como **ASP**, **JSP** o **PHP**. Todos ellos permiten crear páginas web usando código mezcla de página web y lenguajes de programación sencillos.

Ejemplo, página web escrita en lenguaje HTML y JavaScript que escribe en pantalla “Hola mundo” (de color rojo aparece el código en JavaScript):

```
<html>
<head>
    <title>Hola Mundo</title>
</head>
<body>
    <script type="text/javascript">
        document.write("¡Hola mundo!");
    </script>
</body>
</html>
```

(1.7.2) tipos de lenguajes

Según el estilo de programación se puede hacer esta división:

- ◆ **Lenguajes imperativos.** Son lenguajes donde las instrucciones se ejecutan secuencialmente y van modificando la memoria del ordenador para producir las salidas requeridas. La mayoría de lenguajes (**C**, **Pascal**, **Basic**, **Cobol**, ...) son de este tipo. Dentro de estos lenguajes están también los lenguajes orientados a objetos (**C++**, **Java**, **C#**, ...)
- ◆ **Lenguajes declarativos.** Son lenguajes que se concentran más en el qué, que en el cómo (cómo resolver el problema es la pregunta a realizarse cuando se usan lenguajes imperativos). Los lenguajes que se programan usando la pregunta ¿qué queremos? son los declarativos. El

más conocido de ellos es el lenguaje de consulta de Bases de datos, **SQL**.

- ◆ **Lenguajes funcionales.** Definen funciones, expresiones que nos responden a través de una serie de argumentos. Son lenguajes que usan expresiones matemáticas, absolutamente diferentes del lenguaje usado por las máquinas. El más conocido de ellos es el **LISP**.
- ◆ **Lenguajes lógicos.** Lenguajes utilizados para resolver expresiones lógicas. Utilizan la lógica para producir resultados. El más conocido es el **PROLOG**.

(1.7.3) intérpretes

A la hora de convertir un programa en código máquina, se pueden utilizar dos tipos de software: **intérpretes** y **compiladores**.

En el caso de los intérpretes se convierte cada línea a código máquina y se ejecuta ese código máquina antes de convertir la siguiente línea. De esa forma si las dos primeras líneas son correctas y la tercera tiene un fallo de sintaxis, veríamos el resultado de las dos primeras líneas y al llegar a la tercera se nos notificaría el fallo y finalizaría la ejecución.

El intérprete hace una simulación de modo que parece que la máquina entiende directamente las instrucciones del lenguaje, pareciendo que ejecuta cada instrucción (como si fuese código máquina directo).

El **BASIC** era un lenguaje interpretado, se traducía línea a línea. Hoy en día la mayoría de los lenguajes integrados en páginas web son interpretados, la razón es que como la descarga de Internet es lenta, es mejor que las instrucciones se vayan traduciendo según van llegando en lugar de cargar todas en el ordenador. Por eso lenguajes como **JavaScript** (o incluso, en parte, **Java**) son interpretados.

proceso

Un programa que se convierte a código máquina mediante un intérprete sigue estos pasos:

- (1) Lee la primera instrucción
- (2) Comprueba si es correcta
- (3) Convierte esa instrucción al código máquina equivalente
- (4) Lee la siguiente instrucción
- (5) Vuelve al paso 2 hasta terminar con todas las instrucciones

ventajas

- ◆ Se tarda menos en crear el primer código máquina. El programa se ejecuta antes.
- ◆ No hace falta cargar todas las líneas para empezar a ver resultados (lo que hace que sea una técnica idónea para programas que se cargan desde Internet)

desventajas

- ◆ El código máquina producido es peor ya que no se optimiza al valorar una sola línea cada vez. El código optimizado permite estudiar varias líneas a la vez para producir el mejor código máquina posible, por ello no es posible mediante el uso de intérpretes.
- ◆ Todos los errores son errores en tiempo de ejecución, no se pueden detectar antes de lanzar el programa. Esto hace que la depuración de los errores sea más compleja.
- ◆ El código máquina resultante gasta más espacio.
- ◆ Hay errores difícilmente detectables, ya que para que los errores se produzcan, las líneas de errores hay que ejecutarlas. Si la línea es condicional hasta que no probemos todas las posibilidades del programa, no sabremos todos los errores de sintaxis cometidos.

(1.7.4) compiladores

Se trata de software que traduce las instrucciones de un lenguaje de programación de alto nivel a código máquina. La diferencia con los intérpretes reside en que se analizan todas las líneas antes de empezar la traducción.

Durante muchos años, los lenguajes potentes han sido compilados. El uso masivo de Internet ha propiciado que esta técnica a veces no sea adecuada y haya lenguajes modernos interpretados o semi-interpretados, mitad se compila hacia un código intermedio y luego se interpreta línea a línea (esta técnica la siguen **Java** y los lenguajes de la plataforma **.NET** de Microsoft).

ventajas

- ◆ Se detectan errores antes de ejecutar el programa (errores de compilación)
- ◆ El código máquina generado es más rápido (ya que se optimiza)
- ◆ Es más fácil hacer procesos de depuración de código

desventajas

- ◆ El proceso de compilación del código es lento.
- ◆ No es útil para ejecutar programas desde Internet ya que hay que descargar todo el programa antes de traducirlo, lo que ralentiza mucho su uso.

(1.8) programación

(1.8.1) introducción

La programación consiste en pasar algoritmos a algún lenguaje de ordenador a fin de que pueda ser entendido por el ordenador. La programación de ordenadores comienza en los años 50 y su evolución ha pasado por diversos pasos.

La programación se puede realizar empleando diversas **técnicas** o métodos. Esas técnicas definen los distintos tipos de programaciones.

(1.8.2) programación desordenada

Se llama así a la programación que se realizaba en los albores de la informática (aunque desgraciadamente en la actualidad muchos programadores siguen empleándola). En este estilo de programación, predomina el **instinto** del programador por encima del uso de cualquier método lo que provoca que la corrección y entendimiento de este tipo de programas sea casi ininteligible.

Ejemplo de uso de esta programación (listado en Basic clásico):

```
10 X=RANDOM()*100+1;
20 PRINT "escribe el número que crees que guardo"
30 INPUT N
40 IF N>X THEN PRINT "mi numero es menor" GOTO 20
50 IF N<X THEN PRINT "mi numero es mayor" GOTO 20
60 PRINT "¡Acertaste!"
```

El código anterior crea un pequeño juego que permite intentar adivinar un número del 1 al 100.

(1.8.3) programación estructurada

En esta programación se utiliza una técnica que genera programas que sólo permiten utilizar tres estructuras de control:

- ◆ **Secuencias** (instrucciones que se generan secuencialmente)
- ◆ **Alternativas** (sentencias **if**)
- ◆ **Iterativas** (bucles condicionales)

El listado anterior en un lenguaje estructurado sería (listado en Pascal):

```
PROGRAM ADIVINANUM;  
USES CRT;  
VAR x,n:INTEGER;  
BEGIN  
    X=RANDOM()*100+1;  
    REPEAT  
        WRITE("Escribe el número que crees que guardo");  
        READ(n);  
        IF (n>x) THEN WRITE("Mi número es menor");  
        IF (n>x) THEN WRITE("Mi número es mayor");  
    UNTIL n=x;  
    WRITE("Acertaste");
```

La ventaja de esta programación está en que es más legible (aunque en este caso el código es casi más sencillo en su versión desordenada). **Todo programador debería escribir código de forma estructurada.**

(1.8.4) programación modular

Completa la programación anterior permitiendo la definición de módulos independientes cada uno de los cuales se encargará de una tarea del programa. De este forma el programador se concentra en la codificación de cada módulo haciendo más sencilla esta tarea. Al final se deben integrar los módulos para dar lugar a la aplicación final.

El código de los módulos puede ser invocado en cualquier parte del código. Realmente cada módulo se comporta como un subprograma que, partir de unas determinadas entradas obtienen unas salidas concretas. Su funcionamiento no depende del resto del programa por lo que es más fácil encontrar los errores y realizar el mantenimiento.

(1.8.5) programación orientada a objetos

Es la más novedosa, se basa en intentar que el código de los programas se parezca lo más posible a la forma de pensar de las personas. Las aplicaciones se representan en esta programación como una serie de objetos independientes que se comunican entre sí.

Cada objeto posee datos y métodos propios, por lo que los programadores se concentran en programar independientemente cada objeto y luego generar el código que inicia la comunicación entre ellos.

Es la programación que ha revolucionado las técnicas últimas de programación ya que han resultado un importante éxito gracias a la facilidad que poseen de encontrar fallos, de reutilizar el código y de documentar fácilmente el código.

Ejemplo (código Java):

```
/**  
 *Calcula los primos del 1 al 1000  
 */  
public class primos {  
    /** Función principal */  
    public static void main(String args[]){  
        int nPrimos=10000;  
        boolean primo[]=new boolean[nPrimos+1];  
        short i;  
        for (i=1;i<=nPrimos;i++){ primo[i]=true; }  
        for (i=2;i<=nPrimos;i++){  
            if (primo[i]){  
                for (int j=2*i;j<=nPrimos;j+=i){ primo[j]=false; }  
            }  
        }  
        for (i=1;i<=nPrimos;i++){  
            System.out.print(" "+i);  
        }  
    }  
}
```

(1.9) Índice de ilustraciones

| | |
|---|----|
| Ilustración 1, Modelo de Von Newman | 6 |
| Ilustración 2, arquitectura de los ordenadores actuales | 7 |
| Ilustración 3, Histórico de versiones de Windows | 10 |
| Ilustración 4, Ciclo de vida de una aplicación | 19 |
| Ilustración 5, Evolución de algunos lenguajes de programación | 25 |

Unidad 2:

Metodología de la programación

Fundamentos de Programación. 1º de ASI



Esta obra está bajo una licencia de Creative Commons.

Autor: Jorge Sánchez Asenjo (año 2008) <http://www.jorgesanchez.net>
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons

Para ver una copia de esta licencia, visite:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

o envíe una carta a:

Creative Commons, 559 Nathan Abbot

(2)

metodología de la programación

(1.2) esquema de la unidad

| | |
|---|----|
| (2.1) esquema de la unidad | 5 |
| (2.2) metodologías | 6 |
| (2.2.1) introducción | 6 |
| (2.2.2) fases de las metodologías | 6 |
| (2.2.3) prototipos | 7 |
| (2.3) fase de análisis | 9 |
| (2.4) diseño | 9 |
| (2.5) notaciones para el diseño de algoritmos | 11 |
| (2.5.1) diagramas de flujo | 11 |
| (2.5.2) pseudocódigo | 12 |
| (2.6) UML | 30 |
| (2.6.1) introducción | 30 |
| (2.6.2) diagramas UML | 31 |
| (2.6.3) diagrama de casos de uso | 32 |
| (2.6.4) diagramas de actividad | 33 |
| (2.7) índice de ilustraciones | 36 |

(1.3) metodologías

(1.3.1) introducción

Como ya se comentó en el tema anterior, la aplicación pasa por una serie de pasos relacionados con el ciclo de vida de la aplicación. En el tema anterior se indicaron los siguientes pasos:

- (1) Análisis**
- (2) Diseño**
- (3) Codificación o implementación**
- (4) Prueba**
- (5) Mantenimiento**

Esos pasos imponen un método de trabajo, lo que comúnmente se conoce como metodología. Las metodologías definen los pasos y tareas que hay que realizar para realizar un determinado proyecto. Aplicadas a la informática definen la forma exacta de desarrollar el ciclo de vida de una aplicación.

Una metodología marca las forma de realizar todas las fases de creación de un proyecto informático; en especial las relacionadas con el análisis y diseño.

Las metodologías marcan los siguientes elementos:

- ◆ **Pasos exactos a realizar en la creación de la aplicación.** Se indica de forma exacta qué fases y en qué momento se realiza cada una.
- ◆ **Protocolo.** Normas y reglas a seguir escrupulosamente en la realización de la documentación y comunicación en cada fase.
- ◆ **Recursos humanos.** Personas encargadas de cada fase y responsables de las mismas.

La metodología a elegir por parte de los desarrolladores es tan importante que algunas metodologías son de pago, pertenecen a empresas que poseen sus derechos. En muchos casos dichas empresas fabrican las herramientas (**CASE**) que permiten gestionar de forma informática el seguimiento del proyecto mediante la metodología empleada.

(1.3.2) fases en las metodologías

Todas las metodologías imponen una serie de pasos o fases a realizar en el proceso de creación de aplicaciones; lo que ocurre es que estos pasos varían de unas a otras. En general los pasos siguen el esquema inicial planteado en el punto anterior, pero varían los pasos para recoger aspectos que se consideran que mejoran el proceso. La mayoría de metodologías se diferencian fundamentalmente en la parte del diseño. De hecho algunas se refieren sólo a esa fase (como las famosas metodologías de **Jackson** o **Warnier** por ejemplo).

Por ejemplo la **metodología OMT** de Rumbaugh propone estas fases:

- ◆ Análisis
- ◆ Diseño del sistema
- ◆ Diseño de objetos
- ◆ Implementación

Naturalmente con esto no se completa el ciclo de vida, para el resto sería necesario completar el ciclo con otro método.

En la metodología **MERISE** desarrollada para el gobierno francés (y muy utilizada en las universidades españolas), se realizan estas fases:

- ◆ Esquema Director
- ◆ Estudio Previo
- ◆ Estudio Detallado
- ◆ Estudio Técnico
- ◆ Realización
- ◆ Mantenimiento

Otra propuesta de método indicada en varios libros:

- ◆ Investigación Preliminar
- ◆ Determinación de Requerimientos.
- ◆ Diseño del Sistema
- ◆ Desarrollo del Software
- ◆ Prueba del Sistema
- ◆ Implantación y Evaluación

En definitiva las fases son distintas pero a la vez muy similares ya que el proceso de creación de aplicaciones siempre tiene pasos por los que todas las metodologías han de pasar.

(1.3.3) prototipos

Independientemente de la metodología utilizada, siempre ha habido un problema al utilizar metodologías. El problema consiste en que desde la fase de análisis, el cliente ve la aplicación hasta que esté terminada, independientemente de si el fallo está en una fase temprana o no.

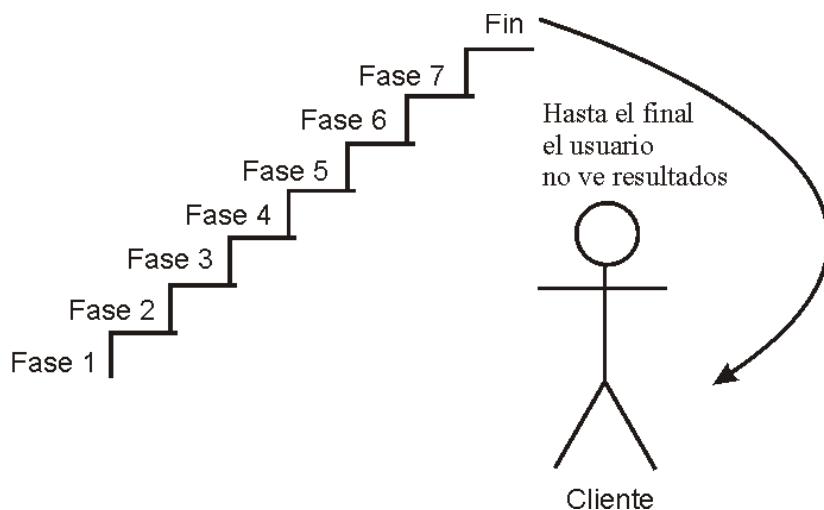


Ilustración 1, Fases clásicas de desarrollo de una aplicación

Esta forma de trabajo tiene el problema de que al usuario le puede llegar una aplicación distinta de la que deseaba de forma bastante habitual. El arreglo es complicado porque no sabemos en qué fase está el problema.

De ahí que apareciera el concepto de **prototipo**. Un prototipo es una forma de ver cómo va a quedar la aplicación antes de terminarla (como la maqueta de un edificio). Se suelen realizar varios prototipos; cuánto más alta sea la fase en la que se realiza, más parecido tendrá con la aplicación final.

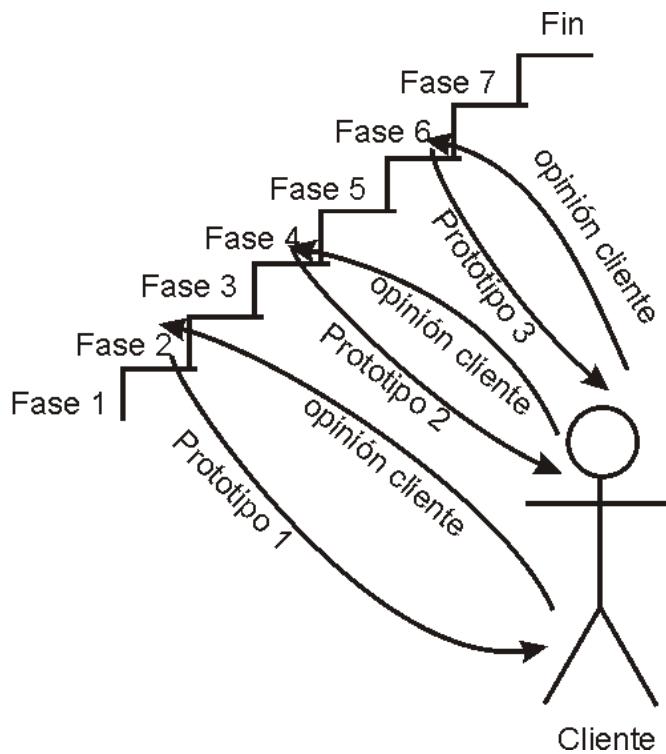


Ilustración 2, Prototipos en las metodologías modernas

(1.4) fase de análisis

Al programar aplicaciones siempre se debe realizar un análisis. El análisis estudia los requisitos que ha de cumplir la aplicación. El resultado del análisis es una **hoja de requerimientos** en la que aparecen las necesidades a cubrir por la aplicación. La habilidad para obtener correctamente la información necesaria de esta fase la realizan los/as **analistas**. La fase se cumple tras ser aprobada por el o la responsable del proyecto (normalmente un **jefe o jefa de proyecto**).

El análisis se suele dividir en estas subfases:

- (1) Estudio preliminar.** En el que se estudia la viabilidad del proyecto. Parte de una información general (la indicada por el cliente inicialmente). En ella se estiman los recursos (tanto humanos como materiales) necesarios y los costes.
- (2) Obtención de información.** Una vez aprobado el estudio preliminar, los y las analistas realizan entrevistas a los futuros usuarios para recabar la información imprescindible para realizar el proyecto.
- (3) Definición del problema.** Las entrevistas dan lugar a un primer estudio detallado del problema en el que se afina más el estudio preliminar a fin de determinar más exactamente los recursos y costes necesarios y las fases que se necesitarán.
- (4) Determinación de los requerimientos.** Se estudian los requerimientos de la aplicación a partir de los datos obtenidos de las fases anteriores.
- (5) Redacción de las hojas de requerimientos.** Se redactan las hojas de requerimientos en base al protocolo a utilizar.
- (6) Aprobación de las hojas.** Los jefes y jefas de proyecto se encargan de revisar y aprobar las hojas, a fin de proceder con las siguientes fases. En caso de no aprobar, habrá que corregir los errores o volver a alguna de las fases anteriores.
- (7) Selección y aprobación del modelo funcional.** O lo que es lo mismo, selección del modelo a utilizar en la fase de diseño. Se elegirá el idóneo en base al problema concreto que tengamos.

Como se comentó en el punto anterior, es habitual (y desde luego muy recomendable) el uso de prototipos. En esta fase bastaría con revisar los requerimientos con el usuario antes de aprobar las hojas de requerimientos.

(1.5) diseño

En esta fase se crean esquemas que simbolizan a la aplicación. Estos esquemas los elaboran analistas. Gracias a estos esquemas se facilita la implementación de la aplicación. Estos esquemas en definitiva se convierte en la documentación fundamental para plasmar en papel lo que el programador debe hacer.

En estos esquemas se pueden simbolizar: la organización de los datos de la aplicación, el orden de los procesos que tiene que realizar la aplicación, la estructura física (en cuanto a archivos y carpetas) que utilizará la aplicación, etc.

Cuanto más potentes sean los esquemas utilizados (que dependerán del modelo funcional utilizado), más mecánica y fácil será la fase de implementación.

La creación de estos esquemas se puede hacer en papel (raramente), o utilizar una **herramienta CASE** para hacerlo. Las herramientas CASE (*Computer Aided Software Engineering, Ingeniería de Software Asistida por Ordenador*) son herramientas software pensadas para facilitar la realización de la fase de diseño en la creación de una aplicación. Con ellas se realizan todos los esquemas necesarios en la fase de diseño e incluso son capaces de escribir el código equivalente al esquema diseñado.

En el caso de la creación de algoritmos, conviene en esta fase usar el llamado **diseño descendente**. Mediante este diseño el problema se divide en módulos, que, a su vez, se vuelven a dividir a fin de solucionar problemas más concretos. Al diseño descendente se le llama también **top-down**. Gracias a esta técnica un problema complicado se divide en pequeños problemas que son más fácilmente solucionables. Es el caso de las metodologías de **Warnier** y la de **Jackson**.

Hoy en día en esta fase se suelen utilizar modelos orientados a objetos como la metodología **OMT** o las basadas en **UML**.

Durante el diseño se suelen realizar estas fases:

- (1) Elaborar el modelo funcional.** Se trata de diseñar el conjunto de esquemas que representan el funcionamiento de la aplicación.
- (2) Elaborar el modelo de datos.** Se diseñan los esquemas que representan la organización de los datos. Pueden ser esquemas puros de datos (sin incluir las operaciones a realizar sobre los datos) o bien esquemas de objetos (que incluyen datos y operaciones).
- (3) Creación del prototipo del sistema.** Creación de un prototipo que simbolice de la forma más exacta posible la aplicación final. A veces incluso en esta fase se realiza más de un prototipo en diferentes fases del diseño. Los últimos serán cada vez más parecidos a la aplicación final.
- (4) Aprobación del sistema propuesto.** Antes de pasar a la implementación del sistema, se debe aprobar la fase de diseño (como ocurre con la de análisis).

(1.6) notaciones para el diseño de algoritmos

En el caso de la creación de algoritmos. Se utilizan esquemas sencillos que permiten simbolizar el funcionamiento del algoritmo. Pasar de estos esquemas a escribir el algoritmo en código de un lenguaje de programación es muy sencillo.

(1.6.1) diagramas de flujo

introducción

Es el esquema más viejo de la informática. Se trata de una notación que pretende facilitar la escritura o la comprensión de algoritmos. Gracias a ella se esquematiza el flujo del algoritmo. Fue muy útil al principio y todavía se usa como apoyo para explicar el funcionamiento de algunos algoritmos.

No obstante cuando el problema se complica, resulta muy complejo de realizar y de entender. De ahí que actualmente, sólo se use con fines educativos y no en la práctica. Desde luego no son adecuados para representar la fase de diseño de aplicaciones. Pero sus símbolos son tan conocidos que la mayoría de las metodologías actuales utilizan esquemas con símbolos que proceden en realidad de los diagramas de flujo.

Los símbolos de los diagramas están normalizados por organismos de estandarización como ANSI e ISO.

símbolos principales

La lista de símbolos que generalmente se utiliza en los diagramas de flujo es:

| | |
|--|--|
| | T e r m i n a l Sirve para indicar el inicio o el fin del algoritmo |
| | P r o c e s o Contiene una operación normal (cambio de valor, operaciones matemáticas,...) |
| | D i r e c c i ó n Indica la dirección que sigue el flujo del algoritmo |
| | C o n e x i ó n Permite enlazar el flujo del programa entre dos partes distantes del algoritmo que están en la misma página. Para conectar se pone un número en la conexión de salida y el mismo número en el conector de entrada |
| | S u b p r o g r a m a Llamada a un módulo independiente que recibirá una entrada de este algoritmo y con ella obtendrá una salida |
| | Entrada/Salida Scontiene una instrucción de lectura o de escritura de datos |
| | D e c i s i ó n Contiene una condición, si se cumple el programa continúa por la flecha del Sí. De otro modo sigue por la flecha del No |
| | Conexión externa Permite enlazar el flujo del programa entre dos partes distantes del algoritmo que están en distintas páginas. Para conectar se pone un número en la conexión de salida y el mismo número en el conector de entrada |
| | C o m e n t a r i o Texto explicativo sobre como funciona el algoritmo. No se tiene en cuenta este texto para la ejecución del algoritmo. |

Ejemplo:

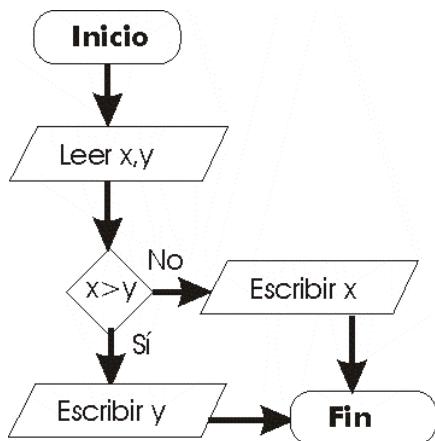


Ilustración 3, Diagrama de flujo que escribe el menor de dos números leídos

desventajas de los diagramas de flujo

Los diagramas de flujo son interesantes como primer acercamiento a la programación ya que son fáciles de entender. De hecho se utilizan fuera de la programación como esquema para ilustrar el funcionamiento de algoritmos sencillos.

Sin embargo cuando el algoritmo se complica, el diagrama de flujo se convierte en ininteligible. Además los diagramas de flujo no facilitan el aprendizaje de la programación estructurada, con lo que no se aconseja su uso a los programadores para diseñar algoritmos.

(1.6.2) pseudocódigo

introducción

Las bases de la programación estructurada fueron enunciadas por **Niklaus Wirth**. Según este científico cualquier problema algorítmico podía resolverse con el uso de estos tres tipos de instrucciones:

- ◆ **Secuenciales.** Instrucciones que se ejecutan en orden normal. El flujo del programa ejecuta la instrucción y pasa a ejecutar la siguiente.
- ◆ **Alternativas.** Instrucciones en las que se evalúa una condición y dependiendo si el resultado es verdadero o no, el flujo del programa se dirigirá a una instrucción o a otra.
- ◆ **Iterativas.** Instrucciones que se repiten continuamente hasta que se cumple una determinada condición.

El tiempo le ha dado la razón y ha generado un estilo universal que insta a todo programador a utilizar sólo instrucciones de los tres tipos indicados anteriormente. Es lo que se conoce como **programación estructurada**.

El propio Niklaus Wirth diseñó el lenguaje **Pascal** como el primer lenguaje estructurado. Lo malo es que el Pascal al ser un lenguaje completo y por tanto

adaptado al ordenador, incluye instrucciones que se saltan la regla de que los algoritmos han de fabricarse independientes del ordenador.

Por ello se aconseja para el diseño de algoritmos estructurados el uso de un lenguaje especial llamado **pseudocódigo**, que además se puede traducir a cualquier idioma (Pascal está basado en el pseudocódigo en inglés).

El pseudocódigo además permite el **diseño modular** de programas y el diseño descendente (técnica que permite solucionar un problema descomponiéndole en problemas pequeños) gracias a esta posibilidad

Hay que tener en cuenta que existen multitud de **pseudocódigos**, es decir **no hay un pseudocódigo 100% estándar**. Pero sí hay gran cantidad de detalles aceptados por todos los pseudocódigos. Aquí se comenta el pseudocódigo que parece más aceptado en España.

El pseudocódigo son instrucciones escritas en un lenguaje orientado a ser entendible por un ordenador (con la ayuda de un **compilador** o un **intérprete**). Por ello en pseudocódigo sólo se pueden utilizar ciertas instrucciones. La escritura de las instrucciones debe cumplir reglas muy estrictas. Las únicas permitidas son:

- ◆ **De Entrada /Salida.** Para leer o escribir datos desde el programa hacia el usuario.
- ◆ **De proceso.** Operaciones que realiza el algoritmo (suma, resta, cambio de valor,...)
- ◆ **De control de flujo.** Instrucciones alternativas o iterativas (bucles y condiciones).
- ◆ **De declaración.** Mediante las que se crean variables y subprogramas.
- ◆ **Llamadas a subprogramas.**
- ◆ **Comentarios.** Notas que se escriben junto al pseudocódigo para explicar mejor su funcionamiento.

escritura en pseudocódigo

Las instrucciones que resuelven el algoritmo en pseudocódigo deben de estar encabezadas por la palabra **inicio** (en inglés **begin**) y cerradas por la palabra **fin** (en inglés **end**). Entre medias de estas palabras se sitúan el resto de instrucciones. Opcionalmente se puede poner delante del inicio la palabra **programa** seguida del nombre que queramos dar al algoritmo.

En definitiva la estructura de un algoritmo en pseudocódigo es:

```
programa nombreDelPrograma
  inicio
    instrucciones
    ....
  fin
```

Hay que tener en cuenta estos detalles:

- ◆ Aunque no importan las mayúsculas y minúsculas en pseudocódigo, se aconsejan las minúsculas porque su lectura es más clara y además porque hay muchos lenguajes en los que sí importa el hecho de haber escrito en mayúsculas o minúsculas (C, Java, ...).
- ◆ Se aconseja que las instrucciones dejen un espacio (sangría) a la izquierda para que se vea más claro que están entre el inicio y el fin. Esta forma de escribir algoritmos permite leerlos mucho mejor.

comentarios

En pseudocódigo los comentarios que se deseen poner (y esto es una práctica muy aconsejable) se ponen con los símbolos *//* al principio de la línea de comentario (en algunas notaciones se escribe ****). Un comentario es una línea que sirve para ayudar a entender el código. Cada línea de comentario debe comenzar con esos símbolos:

```
inicio
  instrucciones
  //comentario
  //comentario
  instrucciones
fin
```

instrucciones

Dentro del pseudocódigo se pueden utilizar los siguientes tipos de instrucción:

- ◆ **Declaraciones.** Sirven para declarar variables. Las variables son nombres identificativos a los que se les asigna un determinado valor y son la base de la programación. Un ejemplo sería la variable *salario* que podría servir por ejemplo para almacenar el salario de una persona en un determinado algoritmo. La instrucción de pseudocódigo es la sección **var**, que se coloca detrás del nombre del programa.
- ◆ **Constantes.** Es un tipo especial de variable llamada constante. Se utiliza para valores que no van a variar en ningún momento. Si el algoritmo utiliza valores constantes, éstos se declaran mediante una sección (que se coloca delante de la sección **var**) llamada **const** (de constante).

Ejemplo:

```
programa ejemplo1
const
    PI=3.141592
    NOMBRE="Jose"
var
    edad: entero
    sueldo: real
inicio
....
```

Ilustración 4, Ejemplo de pseudocódigo con variables y constantes

- ◆ **Instrucciones primitivas.** Son las instrucciones que se realizan en el algoritmo. Se escriben entre las palabras inicio y fin. Sólo pueden contener estos posibles comandos:
 - Asignaciones (\leftarrow)
 - Operaciones (+, -, *, /,...)
 - Identificadores (nombres de variables o constantes)
 - Valores (números o texto encerrado entre comillas)
 - Llamadas a subprogramas
- ◆ **Instrucciones alternativas.** Permiten ejecutar una o más instrucciones en función de una determinada condición. Se trata de la instrucción **sí**.
- ◆ **Instrucciones iterativas.** Son una o más instrucciones cuya ejecución se realiza continuamente hasta que una determinada condición se cumpla. Son las instrucciones **mientras** (*while*), **repetir** (*repeat*) y **desde** (*for*).

(1.7) escritura de algoritmos usando pseudocódigo y diagramas de flujo

(1.7.1) instrucciones

Independientemente de la notación que utilicemos para escribir algoritmos, éstos contienen instrucciones, acciones a realizar por el ordenador. Lógicamente la escritura de estas instrucciones sigue unas normas muy estrictas. Las instrucciones pueden ser de estos tipos:

- ◆ **Primitivas.** Son acciones sobre los datos del programa. Son:
 - Asignación
 - Instrucciones de Entrada/Salida
- ◆ **Declaraciones.** Obligatorias en el pseudocódigo, opcionales en otros esquemas. Sirven para advertir y documentar el uso de variables y subprogramas en el algoritmo.

- ◆ **Control.** Sirven para alterar el orden de ejecución del algoritmo. En general el algoritmo se ejecuta secuencialmente. Gracias a estas instrucciones el flujo del algoritmo depende de ciertas condiciones que nosotros mismos indicamos.

(1.7.2) instrucciones de declaración

Sólo se utilizan en el pseudocódigo (en los diagramas de flujo se sobreentienden) aunque son muy importantes para adquirir buenos hábitos de programación. Indican el nombre y las características de las **variables** que se utilizan en el algoritmo. Las variables son nombres a los que se les asigna un determinado valor y son la base de la programación. Al nombre de las variables se le llama **identificador**.

identificadores

Los algoritmos necesitan utilizar datos. Los datos se identifican con un determinado **identificador** (nombre que se le da al dato). Este nombre:

- ◆ Sólo puede contener letras, números y el carácter _
- ◆ Debe comenzar por una letra
- ◆ No puede estar repetido en el mismo algoritmo. No puede haber dos elementos del algoritmo (dos datos por ejemplo) con el mismo identificador.
- ◆ Conviene que sea aclarativo, es decir que represente lo mejor posible los datos que contiene. **x** no es un nombre aclarativo, **saldo_mensual** sí lo es.

Los valores posibles de un identificador deben de ser siempre del mismo tipo (lo cual es lógico puesto que un identificador almacena un dato). Es decir no puede almacenar primero texto y luego números.

declaración de variables

Es aconsejable al escribir pseudocódigo indicar las variables que se van a utilizar (e incluso con un comentario indicar para qué se van a usar). En el caso de los otros esquemas (diagramas de flujo y tablas de decisión) no se utilizan (lo que fomenta malos hábitos).

Esto se hace mediante la sección del pseudocódigo llamada **var**, en esta sección se colocan las variables que se van a utilizar. Esta sección se coloca antes del **inicio** del algoritmo. Y se utiliza de esta forma:

```
programa nombreDePrograma
var
    identificador1: tipoDeDatos
    identificador2: tipoDeDatos
    ...
    inicio
        instrucciones
    fin
```

El tipo de datos de la variable puede ser especificado de muchas formas, pero tiene que ser un tipo compatible con los que utilizan los lenguajes informáticos. Se suelen utilizar los siguientes tipos:

- ◆ **entero.** Permite almacenar valores enteros (sin decimales).
- ◆ **real.** Permite almacenar valores decimales.
- ◆ **carácter.** Almacenan un carácter alfanumérico.
- ◆ **lógico** (o **booleano**). Sólo permiten almacenar los valores **verdadero** o **falso**.
- ◆ **texto.** A veces indicando su tamaño (**texto(20)** indicaría un texto de hasta 20 caracteres) permite almacenar texto. Normalmente en cualquier lenguaje de programación se considera un tipo compuesto.

También se pueden utilizar datos más complejos, pero su utilización queda fuera de este tema.

Ejemplo de declaración:

```
var
    numero_cliente: entero // código único de cada cliente
    valor_compra: real //lo que ha comprado el cliente
    descuento: real //valor de descuento aplicable al cliente
```

También se pueden declarar de esta forma:

```
var
    numero_cliente: entero // código único de cada cliente
    valor_compra, //lo que ha comprado el cliente
    descuento :real //valor de descuento aplicable al cliente
```

La coma tras **valor_compra** permite declarar otra variable real.

constantes

Hay un tipo especial de variable llamada constante. Se utiliza para valores que no van a variar en ningún momento. Si el algoritmo utiliza valores constantes, éstos se declaran mediante una sección (que se coloca delante de la sección **var**) llamada **const** (de constante). Ejemplo:

```
programa ejemplo1
const
    PI=3.141592
    NOMBRE="Jose"
var
    edad: entero
    sueldo: real
inicio
....
```

A las constantes se les asigna un valor mediante el símbolo `=`. Ese valor permanece constante (pi siempre vale 3.141592). Es conveniente (aunque en absoluto obligatorio) utilizar letras mayúsculas para declarar variables.

(1.7.3) instrucciones primitivas

Son instrucciones que se ejecutan en cuanto son leídas por el ordenador. En ellas sólo puede haber:

- ◆ Asignaciones (`←`)
- ◆ Operaciones (+, -, *, /,...)
- ◆ Identificadores (nombres de variables o constantes)
- ◆ Valores (números o texto encerrado entre comillas)
- ◆ Llamadas a subprogramas

En el pseudocódigo se escriben entre el inicio y el fin. En los diagramas de flujo y tablas de decisión se escriben dentro de un rectángulo

instrucción de asignación

Permite almacenar un valor en una variable. Para asignar el valor se escribe el símbolo `←`, de modo que:

identificador`←`valor

El identificador toma el valor indicado. Ejemplo:

`x←8`

Ahora `x` vale 8. Se puede utilizar otra variable en lugar de un valor. Ejemplo:

`y←9`
`x←y`

`x` vale ahora lo que vale `y`, es decir `x` vale 9.

Los valores pueden ser:

- ◆ **Números.** Se escriben tal cual, el separador decimal suele ser el punto (aunque hay quien utiliza la coma).
- ◆ **Caracteres simples.** Los caracteres simples (un solo carácter) se escriben entre comillas simples: ‘a’, ‘c’,etc.
- ◆ **Textos.** Se escriben entre comillas doble “Hola”
- ◆ **Lógicos.** Sólo pueden valer **verdadero** o **falso** (se escriben tal cual)
- ◆ **Identificadores.** En cuyo caso se almacena el valor de la variable con dicho identificador.

Ejemplo:

x←9
y←x //**y valdrá nueve**

En las instrucciones de asignación se pueden utilizar expresiones más complejas con ayuda de los operadores.

Ejemplo:

x←(y*3)/2

Es decir x vale el resultado de multiplicar el valor de **y** por tres y dividirlo entre dos. Los operadores permitidos son:

| | |
|------------|--|
| + | Suma |
| - | Resta o cambio de signo |
| * | Producto |
| / | División |
| mod | Resto. Por ejemplo 9 mod 2 da como resultado 1 |
| div | División entera. 9 div 2 da como resultado 4 (y no 4,5) |
| ↑ | Exponente. 9 \uparrow 2 es 9 elevado a la 2 |

Hay que tener en cuenta la prioridad del operador. Por ejemplo la multiplicación y la división tienen más prioridad que la suma o la resta. Si 9+6/3 da como resultado 5 y no 11. Para modificar la prioridad de la instrucción se utilizan paréntesis. Por ejemplo 9+(6/3)

(1.7.4) instrucciones de entrada y salida

lectura de datos

Es la instrucción que simula una lectura de datos desde el teclado. Se hace mediante la orden **leer** en la que entre paréntesis se indica el identificador de la variable que almacenará lo que se lea. Ejemplo (pseudocódigo):

leer x

El mismo ejemplo en un diagrama de flujo sería:

leer x

En ambos casos **x** contendrá el valor leído desde el teclado. Se pueden leer varias variables a la vez:

leer x,y,z

leer x, y, z

escritura de datos

Funciona como la anterior pero usando la palabra **escribir**. Simula la salida de datos del algoritmo por pantalla.

escribir x,y,z

escribir x, y, z

ejemplo de algoritmo

El algoritmo completo que escribe el resultado de multiplicar dos números leídos por teclado sería (en pseudocódigo)

```
programa mayorDe2
var
    x,y: entero
inicio
    leer x,y
    escribir x*y
fin
```

En un diagrama de flujo:

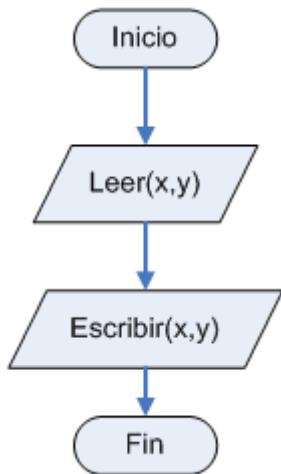


Ilustración 5, Diagrama de flujo con lectura y escritura

(1.7.5) instrucciones de control

Con lo visto anteriormente sólo se pueden escribir algoritmos donde la ejecución de las instrucciones sea secuencial. Pero la programación estructurada permite el uso de decisiones y de iteraciones.

Estas instrucciones permiten que haya instrucciones que se pueden ejecutar o no según una condición (instrucciones alternativas), e incluso que se ejecuten repetidamente hasta que se cumpla una condición (instrucciones iterativas). En definitiva son instrucciones que permiten variar el flujo normal del programa.

expresiones lógicas

Todas las instrucciones de este apartado utilizan expresiones lógicas. Son expresiones que dan como resultado un valor lógico (verdadero o falso). Suelen ser siempre comparaciones entre datos. Por ejemplo $x>8$ da como resultado verdadero si x vale más que 8. Los operadores de relación (de comparación) que se pueden utilizar son:

- | | |
|---|---------------|
| > | Mayor que |
| < | Menor que |
| ≥ | Mayor o igual |
| ≤ | Menor o igual |
| ≠ | Distinto |
| = | Igual |

También se pueden unir expresiones utilizando los operadores **Y** (en inglés **AND**), el operador **O** (en inglés **OR**) o el operador **NO** (en inglés **NOT**). Estos operadores permiten unir expresiones lógicas. Por ejemplo:

| Expresión | Resultado verdadero si... |
|--------------------------|---|
| $a>8 \text{ Y } b<12$ | La variable a es mayor que 8 y (a la vez) la variable b es menor que 12 |
| $a>8 \text{ Y } b<12$ | O la variable a es mayor que 8 o la variable b es menor que 12. Basta que se cumpla una de las dos expresiones. |
| $a>30 \text{ Y } a<45$ | La variable a está entre 31 y 44 |
| $a<30 \text{ O } a>45$ | La variable a no está entre 30 y 45 |
| NO $a=45$ | La variable a no vale 45 |
| $a >8 \text{ Y NO } b<7$ | La variable a es mayor que 8 y b es menor o igual que 7 |
| $a>45 \text{ Y } a<30$ | Nunca es verdadero |

instrucción de alternativa simple

La alternativa simple se crea con la instrucción **si** (en inglés **if**). Esta instrucción evalúa una determinada expresión lógica y dependiendo de si esa expresión es verdadera o no se ejecutan las instrucciones siguientes. Funcionamiento:

```
si expresión_lógica entonces
    instrucciones
fin_si
```

Esto es equivalente al siguiente diagrama de flujo:

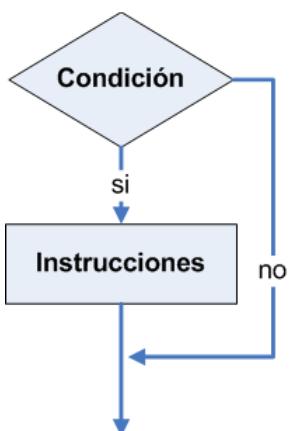


Ilustración 6, Diagrama de flujo con una alternativa simple

Las instrucciones sólo se ejecutarán si la expresión evaluada es verdadera.

instrucción de alternativa doble

Se trata de una variante de la alternativa en la que se ejecutan unas instrucciones si la expresión evaluada es verdadera y otras si es falsa.
Funcionamiento:

```
si expresión_lógica entonces
    instrucciones //se ejecutan si la expresión es verdadera
    si_no
        instrucciones //se ejecutan si la expresión es falsa
    fin_si
```

Sólo se ejecuta unas instrucciones dependiendo de si la expresión es verdadera.
El diagrama de flujo equivalente es:

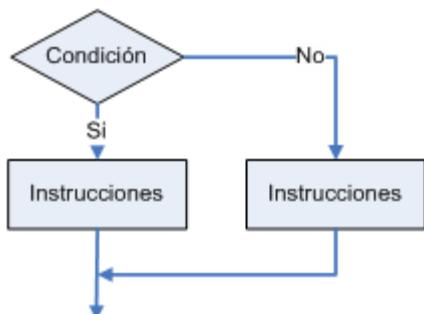


Ilustración 7, Diagrama de flujo con una alternativa doble

Hay que tener en cuenta que se puede meter una instrucción si dentro de otro si. A eso se le llama alternativas anidadas.

Ejemplo:

```
si a≥5 entonces
    escribe "apto"
    si a≥5 y a<7 entonces
        escribe "nota:aprobado"
    fin_si
    si a≥7 y a<9 entonces
        escribe "notable"
    si_no
        escribe "sobresaliente"
    fin_si
    si_no
        escribe "suspenso"
    fin_si
```

Al anidar estas instrucciones hay que tener en cuenta que hay que cerrar las instrucciones **si** interiores antes que las exteriores.

Eso es una regla básica de la programación estructurada:

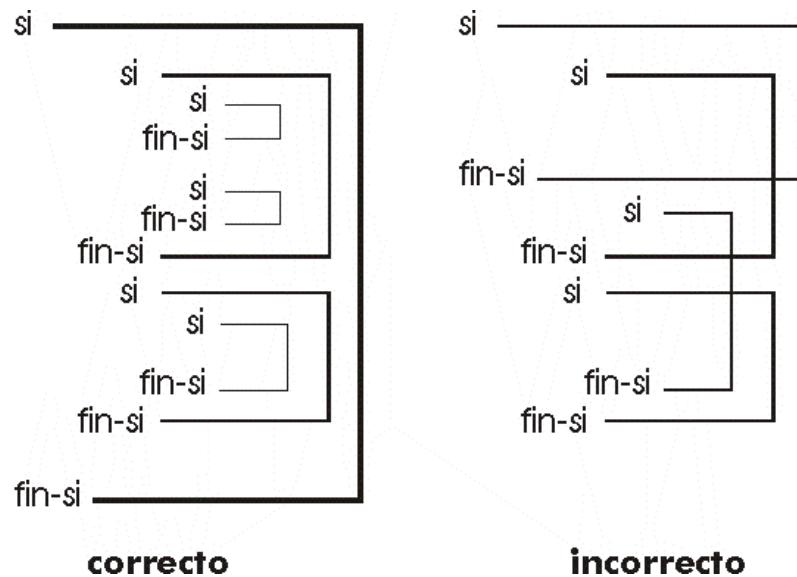


Ilustración 8, Escritura de condiciones estructuradas

alternativa compuesta

En muchas ocasiones se requieren condiciones que poseen más de una alternativa. En ese caso existe una instrucción evalúa una expresión y según los diferentes valores que tome se ejecutan unas u otras instrucciones.

Ejemplo:

```
según_sea expresión hacer
valor1:
    instrucciones del valor1
valor2:
    instrucciones del valor2
...
si_no
    instrucciones del si_no
fin_según
```

Casi todos los lenguajes de programación poseen esta instrucción que suele ser un **case** (aunque C, C++, Java y C# usan **switch**). Se evalúa la expresión y si es igual que uno de los valores interiores se ejecutan las instrucciones de ese valor. Si no cumple ningún valor se ejecutan las instrucciones del **si_no**.

Ejemplo:

```
programa pruebaSelMultiple
var
    x: entero
inicio
    escribe "Escribe un número del 1 al 4 y te diré si es par o impar"
    leer x
    según_sea x hacer
        1:
            escribe "impar"
        2:
            escribe "par"
        3:
            escribe "impar"
        4:
            escribe "par"
        si_no
            escribe "error eso no es un número de 1 a 4"
    fin_según
fin
```

El según sea se puede escribir también:

```
según_sea x hacer
    1,3:
        escribe "impar"
    2,4:
        escribe "par"
    si_no
        escribe "error eso no es un número de 1 a 4"
fin_según
```

Es decir el valor en realidad puede ser una lista de valores. Para indicar esa lista se pueden utilizar expresiones como:

| | |
|-----------|--|
| 1..3 | De uno a tres (1,2 o 3) |
| >4 | Mayor que 4 |
| >5 Y <8 | Mayor que 5 y menor que 8 |
| 7,9,11,12 | 7,9,11 y 12. Sólo esos valores (no el 10 o el 8 por ejemplo) |

Sin embargo estas últimas expresiones no son válidas en todos los lenguajes (por ejemplo el C no las admite).

En el caso de los diagramas de flujo, el formato es:

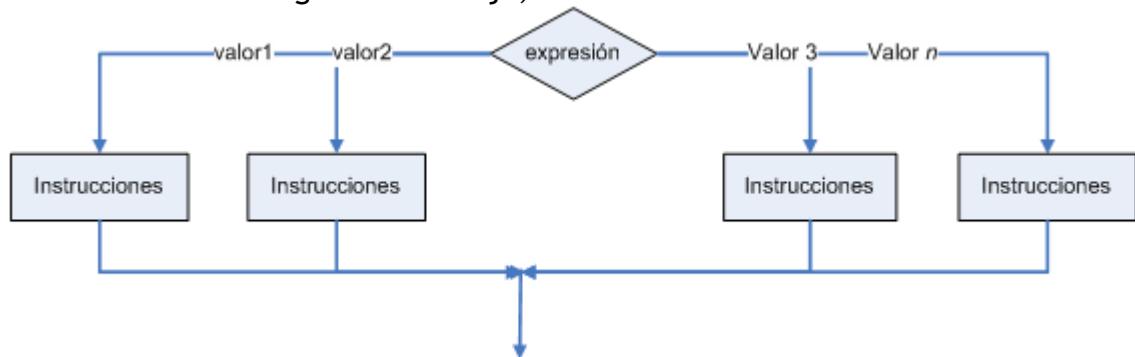


Ilustración 9, Diagrama de flujo con condiciones múltiples

instrucciones iterativas de tipo mientras

El pseudocódigo admite instrucciones iterativas. Las fundamentales se crean con una instrucción llamada **mientras** (en inglés **while**). Su estructura es:

```
mientras condición hacer  
    instrucciones  
fin_mientras
```

Significa que las instrucciones del interior se ejecutan una y otra vez mientras la condición sea verdadera. Si la condición es falsa, las instrucciones se dejan de ejecutar. El diagrama de flujo equivalente es:

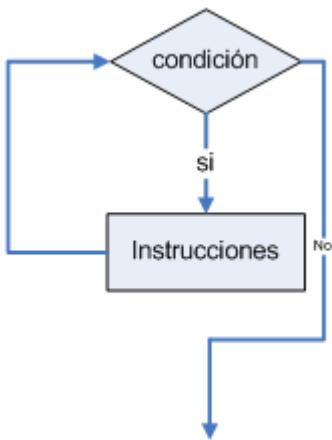


Ilustración 10, Diagrama de flujo correspondiente a la iteración *mientras*

Ejemplo (escribir números del 1 al 10):

```
x←1  
mientras x<=10  
    escribir x  
    x←x+1  
fin_mientras
```

Las instrucciones interiores a la palabra mientras podrían incluso no ejecutarse si la condición es falsa inicialmente.

instrucciones iterativas de tipo *repetir*

La diferencia con la anterior está en que se evalúa la condición al final (en lugar de al principio). Consiste en una serie de instrucciones que repiten continuamente su ejecución hasta que la condición sea verdadera (funciona por tanto al revés que el **mientras** ya que si la condición es falsa, las instrucciones se siguen ejecutando. Estructura

```
repetir  
    instrucciones  
hasta que condición
```

El diagrama de flujo equivalente es:

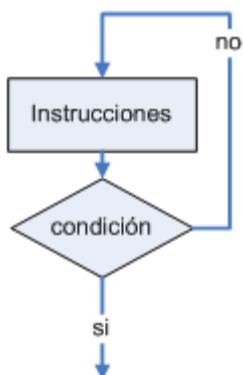


Ilustración 11, Diagrama de flujo correspondiente a la iteración *repetir*

Ejemplo (escribir números del 1 al 10):

```
x←1  
repetir  
    escribir x  
    x←x+1  
hasta que x>10
```

instrucciones iterativas de tipo *hacer...mientras*

Se trata de una iteración que mezcla las dos anteriores. Ejecuta una serie de instrucciones mientras se cumpla una condición. Esta condición se evalúa tras la ejecución de las instrucciones. Es decir es un bucle de tipo ***mientras*** donde las instrucciones al menos se ejecutan una vez (se puede decir que es lo mismo que un bucle *repetir* salvo que la condición se evalúa al revés). Estructura:

```
hacer  
    instrucciones  
mientras condición
```

Este formato está presente en el lenguaje C y derivados (C++, Java, C#), mientras que el formato de *repetir* está presente en el lenguaje Java.

El diagrama de flujo equivalente es:

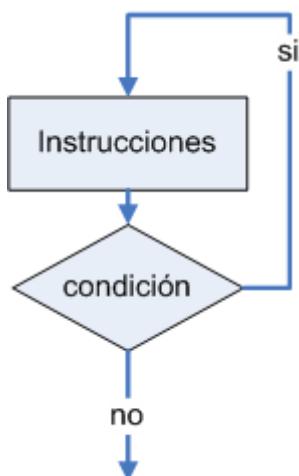


Ilustración 12, Diagrama de flujo correspondiente a la iteración *hacer..mientras*

Ejemplo (escribir números del 1 al 10):

```
x←1
hacer
  escribir x
  x←x+1
mientras x≤10
```

instrucciones iterativas para

Existe otro tipo de estructura iterativa. En realidad no sería necesaria ya que lo que hace esta instrucción lo puede hacer una instrucción **mientras**, pero facilita el uso de bucles con contador. Es decir son instrucciones que se repiten continuamente según los valores de un contador al que se le pone un valor de inicio, un valor final y el incremento que realiza en cada iteración (el incremento es opcional, si no se indica se entiende que es de uno). Estructura:

```
para variable←valorInicial hasta valorfinal hacer
  instrucciones
fin_para
```

Si se usa el incremento sería:

```
para variable←vInicial hasta vFinal incremento valor hacer
    instrucciones
fin_para
```

El diagrama de flujo equivalente a una estructura **para** sería:

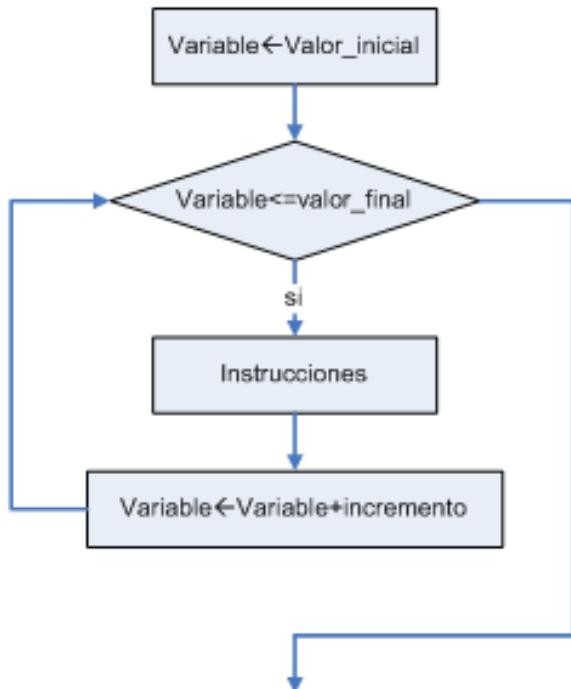


Ilustración 13, Diagrama de flujo correspondiente a la iteración *para*

También se puede utilizar este formato de diagrama:

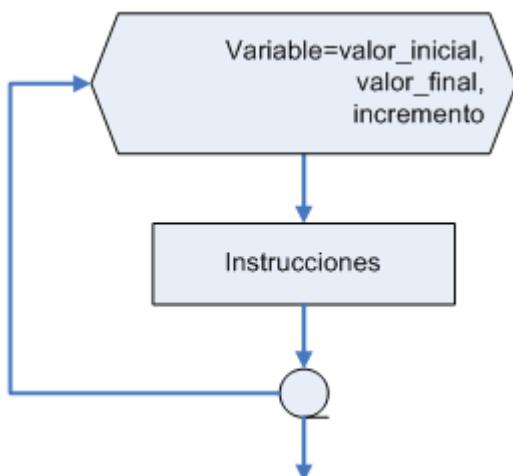


Ilustración 14, Diagrama de flujo con otra versión de la iteración *para*

Otros formatos de pseudocódigo utilizan la palabra **desde** en lugar de la palabra **para** (que es la traducción de **for**, nombre que se da en el original inglés a este tipo de instrucción).

estructuras iterativas anidadas

Al igual que ocurría con las instrucciones **si**, también se puede insertar una estructura iterativa dentro de otra; pero en las mismas condiciones que la instrucción **si**. Cuando una estructura iterativa esta dentro de otra se debe cerrar la iteración interior antes de cerrar la exterior (véase la instrucción **si**).

(1.8) UML

(1.8.1) introducción

UML es la abreviatura de **Universal Modelling Language** (*Lenguaje de Modelado Universal*). No es una metodología, sino una forma de escribir esquemas con pretensiones de universalidad (que ciertamente ha conseguido).

La idea es que todos los analistas y diseñadores utilizaran los mismos esquemas para representar aspectos de la fase de diseño. UML no es una metodología sino una forma de diseñar (de ahí lo de lenguaje de modelado) el modelo de una aplicación.

Su vocación de estándar está respaldada por ser la propuesta de **OMG** (*Object Management Group*) la entidad encargada de desarrollar estándares para la programación orientada a objetos. Lo cierto es que sí se ha convertido en un estándar debido a que une las ideas de las principales metodologías orientadas a objetos: **OMT** de **Rumbaugh**, **OOD** de **Grady Booch** y **Objectory** de **Jacobson**.

Los objetivos de UML son:

- (1)** Poder incluir en sus esquemas todos los aspectos necesarios en la fase de diseño.
- (2)** Facilidad de utilización.
- (3)** Que no se preste a ninguna ambigüedad, es decir que la interpretación de sus esquemas sea una y sólo una.
- (4)** Que sea soportado por multitud de herramientas CASE. Objetivo perfectamente conseguido, por cierto.
- (5)** Utilizable independientemente de la metodología utilizada. Evidentemente las metodologías deben de ser orientadas a objetos.

(1.8.2) diagramas UML

Lo que realmente define UML es la forma de realizar diagramas que representen los diferentes aspectos a identificar en la fase de diseño. Actualmente estamos en la versión 2.1.1 de UML, aunque el UML que se utiliza mayoritariamente hoy en día sigue siendo el primero. Los diagramas a realizar con esta notación son:

- ◆ **Diagramas que modelan los datos.**
 - **Diagrama de clases.** Representa las clases del sistema y sus relaciones.
 - **Diagrama de objetos.** Representa los objetos del sistema.
 - **Diagrama de componentes.** Representan la parte física en la que se guardan los datos.
 - **Diagrama de despliegue.** Modela el hardware utilizado en el sistema
 - **Diagrama de paquetes.** Representa la forma de agrupar en paquetes las clases y objetos del sistema.
- ◆ **Diagramas que modelan comportamiento** (para el modelo funcional del sistema).
 - **Diagrama de casos de uso.** Muestra la relación entre los usuarios (actores) y el sistema en función de las posibles situaciones (casos) de uso que los usuarios hacen.
 - **Diagrama de actividades.** Representa los flujos de trabajo del programa.
 - **Diagrama de estados.** Representa los diferentes estados por los que pasa el sistema.
 - **Diagrama de colaboración.** Muestra la interacción que se produce en el sistema entre las distintas clases.
 - **Diagramas de secuencia.** Muestran la actividad temporal que se produce en el sistema. Representa como se relacionan las clases, pero atendiendo al instante en el que lo hacen

Cabe decir que esto no significa que al modelar un sistema necesitemos todos los tipos de esquema.

A continuación veremos dos tipos de diagramas UML a fin de entender lo que aporta UML para el diseño de sistemas. Sin duda el diagrama más importante y utilizado es el de clases (también el de objetos). De hecho en la última fase del curso veremos en detalle como crear dichos diagramas.

(1.8.3) diagrama de casos de uso

Este diagrama es sencillo y se suele utilizar para representar la primera fase del diseño. Con él se detalla la utilización prevista del sistema por parte del usuario, es decir, los casos de uso del sistema.

En estos diagramas intervienen dos protagonistas:

- ◆ **Actores.** Siempre son humanos (no interesa el teclado, sino quién le golpea) y representan usuarios del sistema. La forma de representarlos en el diagrama es ésta:

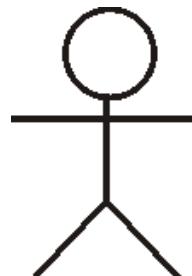


Ilustración 15, Actor

- ◆ **Casos de uso.** Representan tareas que tiene que realizar el sistema. A dichas tareas se les da un nombre y se las coloca en una elipse.

El diagrama de casos de uso representa la relación entre los casos de uso y los actores del sistema. Ejemplo:

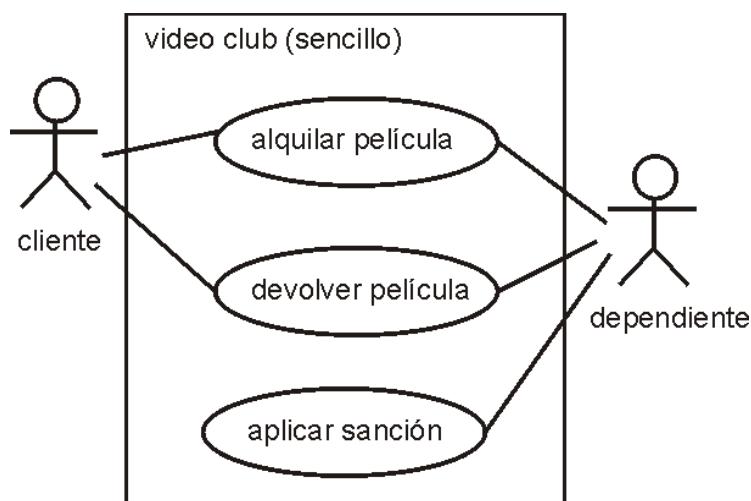


Ilustración 16, Ejemplo sencillo del funcionamiento de un videoclub con un diagrama de casos de uso.

En el dibujo el rectángulo representa el sistema a crear (video club). Fuera del sistema los actores y dentro los casos de uso que el sistema tiene que proporcionar.

Este tipo de diagramas se suele crear en la toma de requisitos (fase de análisis) con el fin de detectar todos los casos de uso del sistema.

Después cada caso de uso se puede subdividir (diseño descendente) en esquemas más simplificados. También se debe describir el caso de uso como

texto en hojas aparte para explicar exactamente su funcionamiento. En un formato como éste:

| | |
|----------------------------|--|
| Nombre del caso de uso: | Alquilar película |
| Autor: | Jorge Sánchez |
| Fecha: | 14/10/2007 |
| Descripción: | Permite realizar el alquiler de la película |
| Actores: | Cliente, dependiente |
| Precondiciones: | El cliente debe de ser socio |
| Funcionamiento normal: | <ol style="list-style-type: none">1) El cliente deja la película en el mostrador junto con su carnet2) El dependiente anota en el sistema el número de carnet para comprobar sanciones3) Sin sanciones, se anota la película y el alquiler se lleva a cabo4) El sistema calcula la fecha de devolución y el dependiente se lo indica al cliente |
| Funcionamiento alternativo | Si hay sanción, no se puede alquilar la película |
| Postcondiciones | El sistema anota el alquiler |

(1.8.4) diagramas de actividad

Hay que tener en cuenta que todos los diagramas UML se basan en el uso de clases y objetos. Puesto que manejar clases y objetos no es el propósito de este tema, no se utilizan al explicar los diagramas de estado y actividad.

Se trata de un tipo de diagrama muy habitual en cualquier metodología. Representa el funcionamiento de una determinada tarea del sistema (normalmente un caso de uso). En realidad el diagrama de estado UML y el de actividad son muy parecidos, salvo que cada uno se encarga de representar un aspecto del sistema. Por ello si se conoce la notación de los diagramas de actividad es fácil entender los diagramas de estado.

Elementos de los diagramas de actividad

| | |
|--|---|
| Actividad:  | Representada con un rectángulo de esquinas redondeadas dentro del cual se pone el nombre de la actividad. Cada actividad puede representar varios pasos y puede iniciarse tras la finalización de otra actividad. |
| Transición:  | Representada por una flecha, indican el flujo de desarrollo de la tarea. Es decir unen la finalización de una actividad con el inicio de otra (es decir, indican qué actividad se inicia tras la finalización de la otra) |
| Barra de sincronización:  | Barra gruesa de color negro. Sirve para coordinar actividades. Es decir si una actividad debe de iniciarse tras la finalización de otras, la transición de las actividades a finalizar irán hacia la barra y de la barra saldrá una flecha a la otra actividad. |
| Diamante de decisión:  | Representado por un rombo blanco, se utiliza para indicar alternativas en el flujo de desarrollo de la tarea según una determinada condición |
| Creación:  | Indican el punto en el que comienza la tarea |
| Destrucción:  | Indica el punto en el que finaliza el desarrollo del diagrama |

Ejemplo de diagrama de actividad (para representar el funcionamiento del alquiler de una película del videoclub):

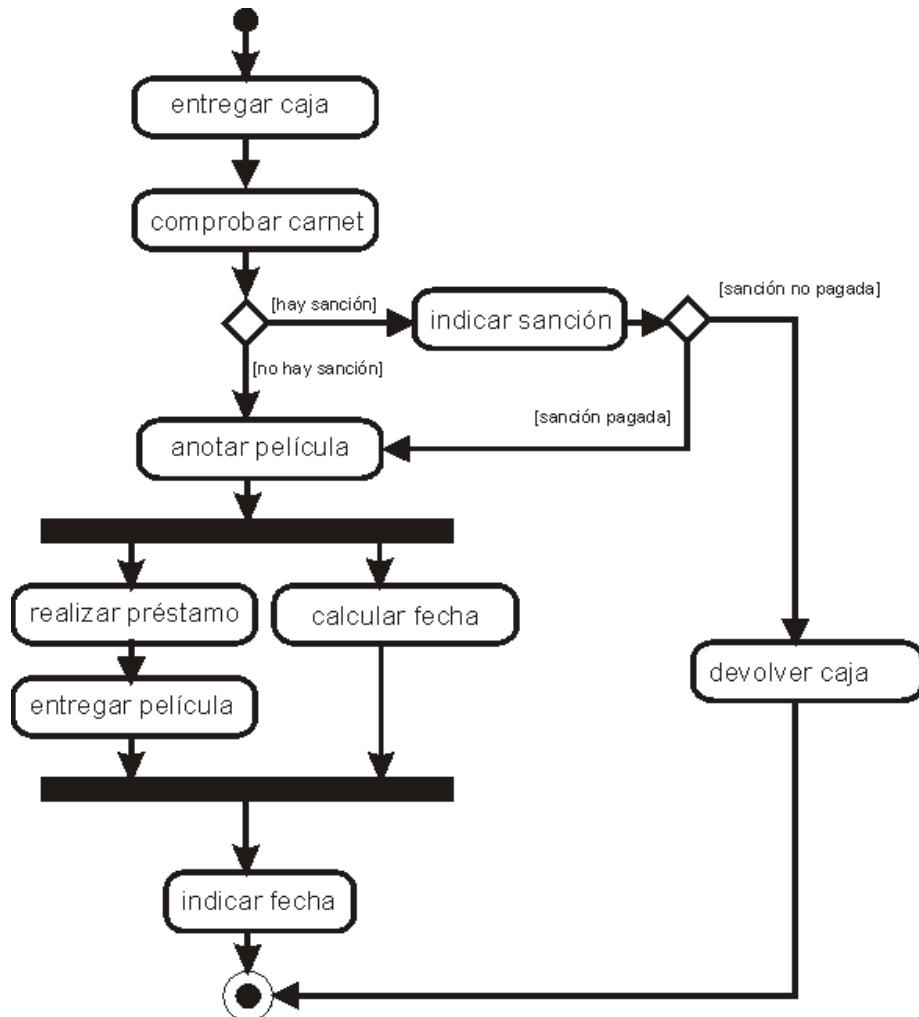


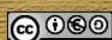
Ilustración 17, El diagrama de actividad del préstamo de una película

(1.9) Índice de ilustraciones

| | |
|--|----|
| Ilustración 1, Fases clásicas de desarrollo de una aplicación | 8 |
| Ilustración 2, Prototipos en las metodologías modernas | 8 |
| Ilustración 3, Diagrama de flujo que escribe el menor de dos números leídos | 12 |
| Ilustración 4, Ejemplo de pseudocódigo con variables y constantes | 15 |
| Ilustración 5, Diagrama de flujo con lectura y escritura | 20 |
| Ilustración 6, Diagrama de flujo con una alternativa simple | 22 |
| Ilustración 7, Diagrama de flujo con una alternativa doble | 22 |
| Ilustración 8, Escritura de condiciones estructuradas | 23 |
| Ilustración 9, Diagrama de flujo con condiciones múltiples | 25 |
| Ilustración 10, Diagrama de flujo correspondiente a la iteración <i>mientras</i> | 26 |
| Ilustración 11, Diagrama de flujo correspondiente a la iteración <i>repetir</i> | 27 |
| Ilustración 12, Diagrama de flujo correspondiente a la iteración <i>hacer..mientras</i> | 28 |
| Ilustración 13, Diagrama de flujo correspondiente a la iteración <i>para</i> | 29 |
| Ilustración 14, Diagrama de flujo con otra versión de la iteración <i>para</i> | 29 |
| Ilustración 15, Actor | 32 |
| Ilustración 16, Ejemplo sencillo del funcionamiento de un videoclub con un diagrama de casos de uso..... | 32 |
| Ilustración 17, El diagrama de actividad del préstamo de una película | 35 |

Unidad 3: Programación básica en Lenguaje C

Fundamentos de Programación. 1º de ASI



Esta obra está bajo una licencia de Creative Commons.

Autor: Jorge Sánchez Asenjo (año 2008) <http://www.jorgesanchez.net>
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons

Para ver una copia de esta licencia, visite:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

o envíe una carta a:

Creative Commons, 559 Nathan Abbot

(3) programación básica en lenguaje C

esquema de la unidad

| | |
|--|-----------|
| (3.1) historia del lenguaje C | 6 |
| (3.1.1) el nacimiento de C..... | 6 |
| (3.1.2) C y C++..... | 6 |
| (3.1.3) características del C | 7 |
| (3.2) compilación de programas en C | 8 |
| (3.3) fundamentos de C | 9 |
| (3.3.1) estructura de un programa C | 9 |
| (3.4) compilación de programas C | 10 |
| (3.4.1) generación básica de programas en C | 10 |
| (3.4.2) Entornos de desarrollo integrado (IDEs)..... | 11 |
| (3.4.3) creación de un programa C en DevCpp | 13 |
| (3.5) elementos de un programa en C | 17 |
| (3.5.1) sentencias | 17 |
| (3.5.2) comentarios | 17 |
| (3.5.3) palabras reservadas..... | 18 |
| (3.5.4) identificadores..... | 18 |
| (3.5.5) líneas de preprocesador..... | 18 |
| (3.6) variables | 19 |
| (3.6.1) introducción..... | 19 |
| (3.6.2) declaración de variables | 19 |
| (3.6.3) tipos de datos..... | 19 |
| (3.6.4) asignación de valores..... | 21 |
| (3.6.5) literales..... | 22 |
| (3.6.6) ámbito de las variables | 23 |
| (3.6.7) conversión de tipos..... | 25 |
| (3.6.8) modificadores de acceso | 26 |
| (3.7) entrada y salida por consola | 27 |
| (3.7.1) función <i>printf</i> | 27 |
| (3.7.2) función <i>scanf</i> | 29 |

| | |
|--|-----------|
| (3.8) operadores | 30 |
| (3.8.1) operadores aritméticos..... | 30 |
| (3.8.2) operadores relacionales..... | 31 |
| (3.8.3) operadores lógicos | 31 |
| (3.8.4) operadores de bits | 32 |
| (3.8.5) operador de asignación | 32 |
| (3.8.6) operador ? | 33 |
| (3.8.7) operadores de puntero & y * | 33 |
| (3.8.8) operador sizeof | 33 |
| (3.8.9) operador coma | 33 |
| (3.8.10) operadores especiales | 34 |
| (3.8.11) orden de los operadores | 34 |
| (3.9) expresiones y conversión de tipos | 35 |
| (3.9.1) introducción | 35 |
| (3.9.2) conversión..... | 35 |
| (3.9.3) operador de molde o cast..... | 36 |
| (3.10) índice de ilustraciones | 36 |

(3.1) historia del lenguaje C

(3.1.1) el nacimiento de C

Fue **Dennis Ritchie** quien en 1969 creo el lenguaje C a partir de las ideas diseñadas por otro lenguaje llamado **B** inventado por **Ken Thompson**.

Ritchie lo inventó para programar la computadora **PDP-11** que utilizaba el sistema **UNIX** (el propio Ritchie creo también Unix). De hecho La historia de C está muy ligada a la de UNIX, este sistema siempre ha incorporado compiladores para trabajar en C. El lenguaje C se diseño como lenguaje pensado para programar sistemas operativos, debido a sus claras posibilidades para ello.

Pero su éxito inmediato hizo que miles de programadores en todo el mundo lo utilizaran para crear todo tipo de aplicaciones (hojas de cálculo, bases de datos,...), aunque siempre ha tenido una clara relación con las aplicaciones de gestión de sistemas.

Debido a la proliferación de diferentes versiones de C, en 1983 el organismo **ANSI** empezó a producir un C estándar para normalizar su situación. En 1989 aparece el considerado como **C estándar** que fue aceptado por **ISO**, organismo internacional de estándares. Actualmente éste C es el universalmente aceptado (aunque han aparecido nuevos estándares de ISO en estos últimos años).

Actualmente se sigue utilizando enormemente para la creación de aplicaciones de sistemas (casi todas las distribuciones de Linux están principalmente creadas en C) y en educación se considera el lenguaje fundamental para aprender a programar.

(3.1.2) C y C++

Debido al crecimiento durante los años 80 de la **programación orientada a objetos**, en 1986 **Bjarne Stroustrup** creó un lenguaje inspirado en Simula pero utilizando la sintaxis del lenguaje C.

C y C++ pues, comparten instrucciones casi idénticas. Pero la forma de programar es absolutamente diferente. Saber programar en C no implica saber programar en C++

(3.1.3) características del C

Se dice que el lenguaje C es un lenguaje de nivel **medio**. La razón de esta indicación está en que en C se pueden crear programas que manipulan la máquina casi como lo hace el lenguaje **Ensamblador**, pero utilizando una sintaxis que se asemeja más a los lenguajes de alto nivel. De los lenguajes de alto nivel toma las estructuras de control que permiten programar de forma estructurada.

Al tener características de los lenguajes de bajo nivel se puede tomar el control absoluto del ordenador. Además tiene atajos que gustan mucho a los programadores al tener una sintaxis menos restrictiva que lenguajes como Pascal por ejemplo), lo que le convierte en el lenguaje idóneo para crear cualquier tipo de aplicación.

Sus características básicas son:

- ◆ Es un lenguaje que incorpora manejo de estructuras de bajo nivel (punteros, bits), lo que le acerca a los lenguajes de segunda generación
- ◆ Es un lenguaje estructurado y modular. Lo que facilita su compresión y escritura. No obstante al tratarse de un lenguaje muy libre y cercano a la máquina (hay quien dice que es un lenguaje de **segunda generación y media**, haciendo referencia a su cercanía al lenguaje ensamblador) es posible escribir código muy poco estructurado (algo que no es recomendable y que estos apuntes no contemplan).
- ◆ Permite utilizar todo tipo de estructuras de datos complejas (arrays, pilas, colas, textos,...) lo que permite utilizar el lenguaje en todo tipo de situaciones.
- ◆ Es un lenguaje compilado.
- ◆ El código que se obtiene en C es muy rápido (gracias a ser compilado y a que se cercanía a la máquina permite escribir código muy eficiente).
- ◆ Permite crear todo tipo de aplicaciones

(3.2) compilación de programas en C

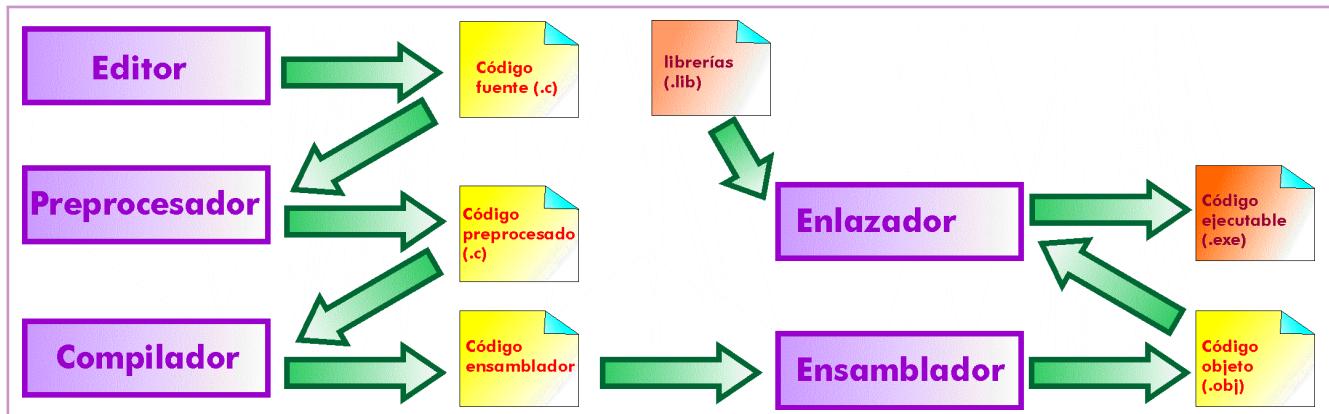


Ilustración 1, proceso de compilación de un programa C

Ya se ha explicado en unidades anteriores que la conversión del código escrito en un lenguaje de programación (**código fuente**) puede ser traducido a código máquina mediante un compilador o mediante un intérprete.

En el caso del lenguaje C se utiliza siempre un compilador (es un **lenguaje compilado**). El proceso completo de conversión del código fuente en código ejecutable sigue los siguientes pasos (ver Ilustración 1):

- (1) Edición.** El código se escribe en un editor de texto o en un editor de código preparado para esta acción. El archivo se suele guardar con extensión **.c**
- (2) Preprocesado.** Antes de compilar el código, el preprocesador lee las instrucciones de preprocesador y las convierte al código fuente equivalente.
- (3) Compilación.** El código fuente resultante en lenguaje C se compila mediante el software apropiado, obteniendo el código en ensamblador equivalente.
- (4) Ensamblado.** El código anterior se ensambla (utilizando software ensamblador, ya que el código que este paso requiere es código en ensamblador) para obtener código máquina. Éste código aún no es ejecutable pues necesita incluir el código de las librerías utilizadas. Éste es el código objeto (con extensión **obj**)
- (5) Enlazado.** El código objeto se une al código máquina de las librerías y módulos invocados por el código anterior. El resultado es un archivo ejecutable (extensión **.exe** en Windows)

El código enlazado es el que se prueba para comprobar si el programa funciona. La mayoría de entornos de programación tienen un **cargador** o **lanzador** que permite probar el programa sin abandonar el entorno. Desde el punto de vista del usuario de estos entornos, sólo hay dos pasos: **compilar** (conseguir el código

ejecutable) y **ejecutar** (cargar el código máquina para que el ordenador lo ejecute)

(3.3) fundamentos de C

(3.3.1) estructura de un programa C

Un programa en C consta de una o más **funciones**, las cuales están compuestas de diversas **sentencias** o **instrucciones**. Una sentencia indica una acción a realizar por parte del programa. Una función no es más que (por ahora) un nombre con el que englobamos a las sentencias que posee a fin de poder invocarlas mediante dicho nombre. La idea es:

```
nombreDeFunción(parámetros) {  
    sentencias  
}
```

Los símbolos **{** y **}** indican el inicio y el final de la función. Esos símbolos permiten delimitar bloques en el código.

El nombre de la función puede ser invocado desde otras sentencias simplemente poniendo como sentencia el nombre de la función. Como a veces las funciones se almacenan en archivos externos, necesitamos incluir esos archivos en nuestro código mediante una sentencia especial **include**, que en realidad es una **directiva de preprocesador**. Una directiva de preprocesador es una instrucción para el compilador con el que trabajamos. El uso es:

```
#include <cabeceraDeArchivoExterno>
```

La directiva **include** permite indicar un archivo de cabecera en el que estará incluida la función que utilizamos. En el lenguaje C estándar los archivos de cabecera tienen extensión **h**. Los archivos de cabecera son los que permiten utilizar funciones externas (o librerías) en nuestro programa.

Una de las librerías más utilizadas en los programas, es la que permite leer y escribir en la consola del Sistema Operativo. En el caso de C esta librería está disponible al incluir la cabecera **stdio.h**

el primer programa en C

En todos los lenguajes de programación, el primer programa a realizar es el famoso **Hola mundo**, un programa que escribe este texto en pantalla. En C++ el código de este programa es:

```
#include <stdio.h>  
int main() {  
    printf("Hola mundo");  
    return 0;  
}
```

En el programa anterior ocurre lo siguiente:

- (1)** La línea con el texto `#include<stdio.h>` sirve para poder utilizar en el programa funciones de lectura y escritura en el programa. En concreto en este caso es necesario para poder utilizar la función `printf`.
- (2)** La función `main` es la función cuyas instrucciones se ejecutan en cuanto el programa inicia su proceso. Esa línea por tanto indica que comienza la función principal del programa.
- (3)** La instrucción `printf("Hola mundo")` es la encargada de escribir el texto `"Hola mundo"` por pantalla
- (4)** La instrucción `return 0` finaliza el programa e indica (con el valor cero) al Sistema Operativo que la finalización ha sido correcta.

(3.4) compilación de programas; C

(3.4.1) generación básica de programas; en C

Existen muchas posibilidades de generar código ejecutable a partir de código C. Al ser C un lenguaje tan estándar, además la mayoría de las soluciones son de código abierto y gratuitas.

En el caso de **Linux** se incluye un compilador llamado **gcc**. Es uno de los compiladores más populares y de hecho los compiladores gratuitos de Windows están basados en él. Para compilar un programa en C desde Linux basta:

- ◆ Escribir el programa en un editor de texto (desde los incómodos **vi** o **emacs** a los más cómodos **gedit** o los entornos **Anjuta** o **Geany**).
- ◆ Después se puede ir a la línea de comandos, acceder al directorio en el que se creó el programa y ejecutar la orden:

```
gcc nombreArchivo.c -o nombreArchivoEjecutable
```

- ◆ Para probar el programa sería (desde la línea de comandos):

```
./nombreArchivoEjecutable
```

En el caso de **Windows** necesitamos instalar las herramientas necesarias para poder desarrollar programas en C al modo de Linux. Eso implica instalar software que incluya el compilador, ensamblador, etc. de C. La opción más interesante (y utilizada) para crear programas en C estándar es el software **Mingw** (www.mingw.org).

Dicho software se puede utilizar para generar programas libremente ya que se está hecho bajo licencia **GNU General Public License** (la misma que utiliza el sistema Linux) lo que significa que el software se puede utilizar sin restricciones e incluso distribuir como parte de una aplicación nuestra (sin embargo dicha aplicación se distribuya con licencia GNU).

MigW es un compilador compatible con **gcc** y su uso en Windows es idéntico. Los pasos para utilizarle en Windows serían:

- ◆ Descargar e instalar MingW desde la página www.mingw.org
- ◆ Añadir la carpeta de ejecutables de MingW al **path** de Windows (normalmente los archivos ejecutables se instalan en la ruta **c:\Mingw\bin**) lo cual se hace así:
 - Pulsar el botón derecho sobre Mi PC y elegir **Propiedades**
 - Ir al apartado **Opciones avanzadas**
 - Hacer clic sobre el botón **Variables de entorno**
 - Añadir a la lista de la variable **Path** la ruta a la carpeta de ejecutables de MingW
- ◆ Escribir el programa en C en cualquier editor de código (como el propio editor de notas) y guardarle con extensión **.c**
- ◆ Ir a la línea de comandos de Windows y entrar en la carpeta en la que creamos el programa.
- ◆ Escribir:

```
gcc nombreArchivo.c -o nombreArchivoEjecutable
```

La sintaxis de MingW es la misma que la de **gcc** bajo Linux.

(3.4.2) Entornos de desarrollo integrado (IDEs)

La programación de ordenadores requiere el uso de una serie de herramientas que faciliten el trabajo al programador. Esas herramientas (software) son:

- ◆ **Editor de código.** Programa utilizado para escribir el código. Normalmente basta un editor de texto (el código en los lenguajes de programación se guarda en formato de texto normal), pero es conveniente que incorpore:
 - **Coloreado inteligente de instrucciones.** Para conseguir ver mejor el código se colorean las palabras claves de una forma, los identificadores de otra, el texto de otra, los números de otra,....
 - **Corrección instantánea de errores.** Los mejores editores corrigen el código a medida que se va escribiendo.
 - **Ayuda en línea.** Para documentarse al instante sobre el uso del lenguaje y del propio editor.
- ◆ **Compilador.** El software que convierte el lenguaje en código máquina.
- ◆ **Lanzador.** Prueba la ejecución de la aplicación
- ◆ **Depurador.** Programa que se utiliza para realizar pruebas sobre el programa.

- ◆ **Organizador de ficheros.** Para administrar todos los ficheros manejados por las aplicaciones que hemos programado.

Normalmente todas esas herramientas se integran en un único software conocido como entorno de programación o **IDE** (*Integrate Development Environment*, Entorno de desarrollo integrado), que hace más cómodos todos los procesos relacionados con la programación.

Algunos de los de uso más habitual:

- ◆ Para Linux (todos ellos con licencia GNU):
 - **KDevelop.** Integrada en el gestor de ventanas **KDE**
 - **Anjuta.** Uno de los más populares, por su facilidad de manejo y versatilidad.
 - **Geany.** Quizá el sucesor del anterior. Es un entorno muy sencillo y potente para programar C ó C++ en Linux
 - **Eclipse.** En realidad funciona también para Windows y está más orientado al lenguaje Java, pero permite programar en C (y C++) y dada su impresionante popularidad merece la pena mencionarlo
- ◆ Para Windows:
 - **Devcpp.** Quizá el entorno de programación ligero más popular de la actualidad. Con licencia GNU. Utiliza el compilador MingW lo que permite programar fácilmente en C estándar (es en el que se han programado todos los ejemplos de los apuntes y el que se usa como ejemplo).
 - **WxDevCpp.** Software basado en el anterior utilizando WxWidgets
 - **Code::blocks.** Con la misma filosofía del anterior, una opción similar que gana adeptos gana día. Está programado en C y C++ (devcpp está programado en **Delphi**) lo que le permite ejecutarse para cualquier plataforma.
 - **Visual Studio (.NET).** Se trata del entorno de programación desarrollado por Microsoft para su plataforma .NET, aunque permite crear cualquier tipo de aplicación C/C++ (aunque el lenguaje no es 100% estándar). Hay versiones Express que no incorporan las funcionales de la versión profesional (que evidentemente es de pago) pero que es gratuita. Dicha versión funciona durante 30 días y luego pide registrarse al usuario para poder seguir utilizándola.
 - **Borland C++ Builder.** Otro de los grandes entornos profesionales de programación. Con el anterior quizás los mejores (aunque éste es más respetuoso con el estándar).

(3.4.3) creación de un programa C en DevCpp

Como ejemplo de uso de un entorno de desarrollo utilizaremos DevCpp. Este software es de código abierto (bajo licencia GNU) y esté disponible en <http://www.bloodshed.net/devcpp.html>. La única pega es que sólo funciona para Windows (aunque para Linux disponemos de herramientas de código abierto aún mejores como **Anjuta** y **Geany**).

Una vez descargado, se instala en el ordenador. Al abrir por primera vez nos pide crear una caché para acelerar la función de completar el código al escribir (que no es imprescindible, pero sí es interesante). Tras esto tendremos delante la pantalla de inicio.

Para crear el primer programa en C con el entorno hay que elegir (suponemos que el menú está en castellano):

- ◆ Archivo-Nuevo-Código fuente
- ◆ Escribir el programa y guardarlo con extensión .c
- ◆ Pulsar la tecla F9

El resultado aparece en una ventana. Una cosa que nos ocurrirá por ejemplo al crear el programa **Hola Mundo**, es que no le veremos porque en la ventana en la que se muestra el resultado, no se hace una pausa para poder examinar dicho resultado.

Un truco es hacer esta versión del **Hola mundo**:

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

int main(void){
    printf("Hola mundo\n");
    system("pause"); /* para la pantalla. truco no estándar,
                      pero que funciona en Windows*/
    return 0;
}
```

La librería **<stdlib.h>** es necesaria para utilizar la instrucción **system("pause")** que permite hacer una pausa al final.

La instrucción **system** en realidad permite ejecutar instrucciones del sistema operativo. Por ejemplo **system("dir")** mostraría el directorio actual. Desgraciadamente esta instrucción no es estándar.

opciones del entorno Dev Cpp

Una de las primeras labores con nuestro editor es personalizar el funcionamiento. Algunas opciones interesantes son:

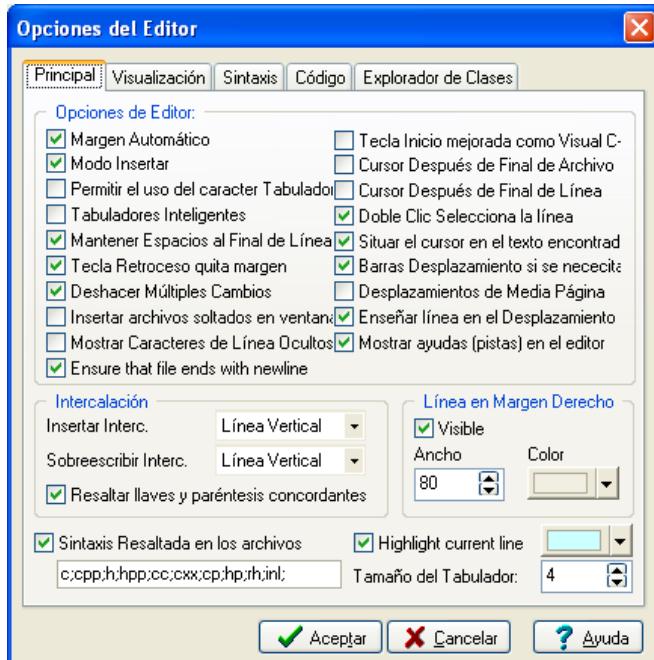


Ilustración 2, Opciones principales del editor

- ◆ **Opciones principales de edición.** Se encuentran en Herramientas-Opciones del editor-Principal. Algunas opciones elegibles en esa pantalla son:
 - **Tamaño del tabulador.** Espacio que dejamos cada vez que pulsamos la tecla tabulador
 - **Resaltar llaves y paréntesis concordantes.** Para facilitar no olvidar cerrar llaves ni paréntesis
 - **Sintaxis resaltada en los archivos.** Permite ver el código con diferentes colores según el tipo de palabra que tengamos
 - **Tabuladores inteligentes.** Permite que el programa deje espacios detectando si es lo apropiado para el código (puede no funcionar muy bien a veces)
- ◆ **Opciones de visualización.** Se trata de la segunda solapa del cuadro anterior. Permite:
 - **Cambiar el tipo de letra y tamaño del código.**
 - **Mostrar los números de línea** (y la letra de éstos)



Ilustración 3, Opciones de visualización

- ◆ **Opciones de sintaxis.** Permite elegir la forma de colorear el código del editor en función del tipo de palabra que sea (palabra clave, texto literal, función, etc.). Está en la tercera solapa

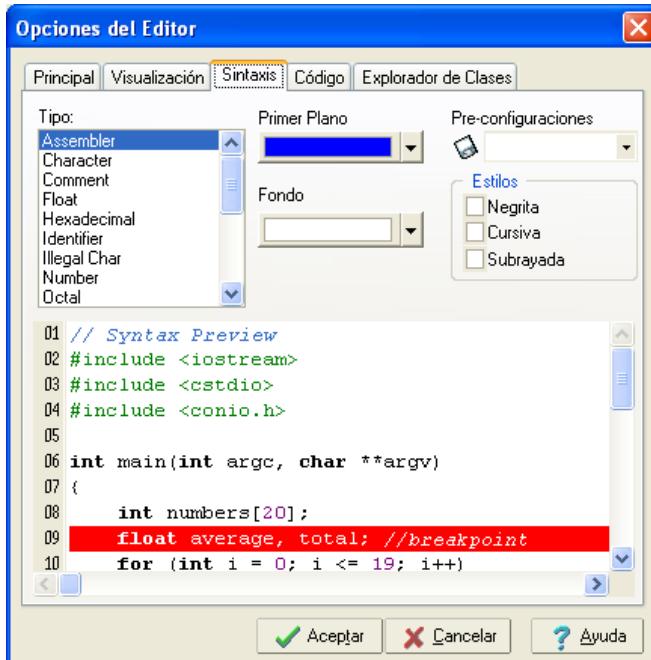


Ilustración 4, Opciones de sintaxis, coloreado del código

- ◆ **Opciones de código.** Tienes dos posibilidades muy interesantes:

- **Elementos del menú Insertar.** Permite colocar una entrada en el menú **Insertar** (que aparece al pulsar el botón derecho sobre la ventana de código) y de ese modo hacer que al elegir una entrada (15)

del menú se escriba automáticamente el trozo de código que deseemos. Es una de las opciones más interesantes del programa. En el código deseado podemos escribir el símbolo | para indicar donde quedará el cursor tras pegar el código.

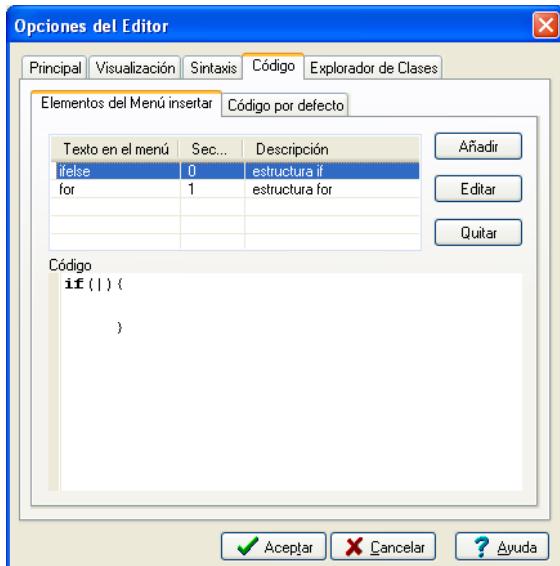


Ilustración 5, Opciones del menú Insertar

- **Código por defecto.** Se trata de un apartado del cuadro que nos permite escribir el código que deseemos para que sea el que utiliza el programa para colocarle al hacer un nuevo código fuente. Es decir, cuando creamos un nuevo programa aparecerá ese código (deberemos asegurar que no tenga errores y colocar todo aquello que sea común a cualquier programa).

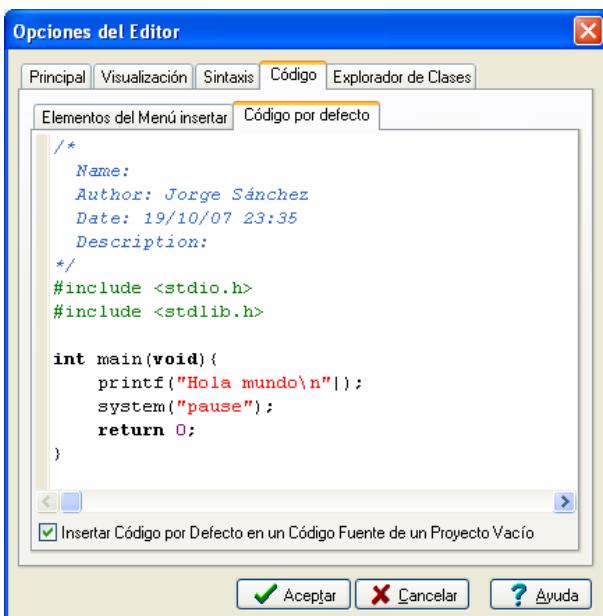


Ilustración 6, Ejemplo de código por defecto

- ◆ **Opciones de compilación.** En el menú Herramientas se pueden elegir las opciones de compilación, para controlar la forma en la que se compilará el programa. Permite cambiar el entorno utilizado para compilar (en lugar de MingW utilizar otro compilador, por ejemplo).
- ◆ **Opciones del entorno.** Se refiere a opciones generales de funcionamiento del programa. Se obtienen desde el menú Herramientas. Podemos elegir el idioma del programa, el máximo número de ventanas que permite abrir, las carpetas en las que se guarda el programa, las extensiones de archivos asociadas al programa,...
- ◆ **Atajos de teclado.** También en el menú de herramientas, permite seleccionar una tecla para asociarla a una tarea del programa.

(3.5) elementos de un programa en C

(3.5.1) sentencias

Los programas en C se basan en sentencias las cuales siempre se incluyen dentro de una función. En el caso de crear un programa ejecutable, esas sentencias están dentro de la función **main**. A esta función le precede la palabra **int**.

Ahora bien al escribir sentencias hay que tener en cuenta las siguientes normas:

- (1) Toda sentencia en C termina con el símbolo "punto y coma" (;
- (2) Los bloques de sentencia empiezan y terminan delimitados con el símbolo de llave. Así { significa inicio y } significa fin
- (3) En C hay distinción entre mayúsculas y minúsculas. No es lo mismo **main** que **MAIN**. Todas las palabras claves de C están en minúsculas. Los nombres que pongamos nosotros también conviene ponerles en minúsculas ya que el código es mucho más legible así (y se es más coherente con el lenguaje)

(3.5.2) comentarios

Se trata de texto que es ignorado por el compilador al traducir el código. Esas líneas se utilizan para documentar el programa.

Esta labor de documentación es fundamental. De otro modo el código se convierte en ilegible incluso para el programador que lo diseñó. Tan importante como saber escribir sentencias es utilizar los comentarios. Todavía es más importante cuando el código va a ser tratado por otras personas, de otro modo una persona que modifique el código de otra lo tendría muy complicado. En C los comentarios se delimitan entre los símbolos /* y */

/* Esto es un comentario
el compilador hará caso omiso de este texto */

Como se observa en el ejemplo, el comentario puede ocupar más de una línea.

(3.5.3) palabras reservadas:

Se llaman así a palabras que en C tienen un significado concreto para los compiladores. No se pueden por tanto usar esas palabras para poner nombre a variables o a funciones.

La lista de palabras reservadas del lenguaje C es ésta:

| | | | |
|----------|--------|----------|----------|
| 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 |

A estas palabras a veces los compiladores añaden una lista propia. Como C distingue entre mayúsculas y minúsculas, el texto **GoTo** no es una palabra reservada.

(3.5.4) identificadores:

Son los nombres que damos a las variables y a las funciones de C. Lógicamente no pueden coincidir con las palabras reservadas. Además puesto que C distingue entre las mayúsculas y las minúsculas, hay que tener cuidado de usar siempre las minúsculas y mayúsculas de la misma forma (es decir, *nombre*, *Nombre* y *NOMBRE* son tres identificadores distintos).

El límite de tamaño de un identificador es de 32 caracteres (aunque algunos compiladores permiten más tamaño). Además hay que tener en cuenta que los identificadores deben de cumplir estas reglas:

- ◆ Deben comenzar por una letra o por el signo de subrayado (aunque comenzar por subrayado se suele reservar para identificadores de funciones especiales del sistema).
- ◆ Sólo se admiten letras del abecedario inglés (no se admite ni la ñ ni la tilde ni la diéresis por ejemplo), números y el carácter de subrayado (el guión bajo _)

(3.5.5) líneas de preprocesador

Las sentencias que comienzan con el símbolo “#” y que no finalizan con el punto y coma son líneas de preprocesador (o directivas de compilación). Son indicaciones para el preprocesador, pero que en el lenguaje C son sumamente importantes. La más utilizada es

#include

Que permite añadir utilizar funciones externas en nuestro código, para lo que se indica el nombre del archivo de cabecera que incluye a la función (o funciones) que queremos utilizar.

(3.6) variables

(3.6.1) introducción

Las variables sirven para identificar un determinado valor que se utiliza en el programa. En C es importante el hecho de tener en cuenta que una variable se almacena en la memoria interna del ordenador (normalmente en la **memoria RAM**) y por lo tanto ocupará una determinada posición en esa memoria. Es decir, en realidad identifican una serie de bytes en memoria en los que se almacenará un valor que necesita el programa.

Es decir si **saldo** es un identificador que se refiere a una variable numérica que en este instante vale **8**; la realidad interna es que **saldo** realmente es una dirección a una posición de la memoria en la que ahora se encuentra el número **8**.

(3.6.2) declaración de variables

En C hay que declarar las variables antes de poder usarlas. Al declarar lo que ocurre es que se reserva en memoria el espacio necesario para almacenar el contenido de la variable. No se puede utilizar una variable sin declarar. Para declarar una variable se usa esta sintaxis:

tipo identificador;

Por ejemplo:

int x;

Se declarara **x** como variable que podrá almacenar valores enteros.

En C se puede declarar una variable en cualquier parte del código, basta con declararla antes de utilizarla por primera vez. Pero es muy buena práctica hacer la declaración al principio del bloque o función en la que se utilizará la variable. Esto facilita la comprensión del código.

También es buena práctica poner un pequeño comentario a cada variable para indicar para qué sirve. A veces basta con poner un identificador claro a la variable. Identificadores como **a**, **b** o **c** por ejemplo, no aclaran la utilidad de la variable; no indican nada. Identificadores como **saldo**, **gastos**, **nota**,... son mucho más significativos.

(3.6.3) tipos de datos

Al declarar variables se necesita indicar cuál es el tipo de datos de las variables los tipos básicos permitidos por el lenguaje C son:

| tipo de datos | rango de valores posibles | tamaño en bytes |
|---------------|---|-----------------|
| char | 0 a 255 (o caracteres simples del código ASCII) | 1 |

| tipo de datos | rango de valores posibles | tamaño en bytes |
|---------------|--|-----------------|
| int | -32768 a 32767 (algunos compiladores consideran este otro rango: -2.147.483.648 a -2.147.483.647) | 2 (algunos 4) |
| float | 3.4E-38 a 3.3E+38 | 4 |
| double | 1.7E-308 a 1.7E+308 | 8 |
| void | sin valor | 0 |

Hay que tener en cuenta que esos rangos son los clásicos, pero en la práctica los rangos (sobre todo el de los enteros) depende del computador, procesador o compilador empleado.

tipos enteros

Los tipos **char** e **int** sirven para almacenar enteros y también valen para almacenar caracteres. Normalmente los números se almacenan en el tipo **int** y los caracteres en el tipo **char**; pero la realidad es que cualquier carácter puede ser representado como número (ese número indica el código en la tabla **ASCII**). Así el carácter '**A**' y el número **65** significan exactamente lo mismo en lenguaje C.

tipos decimales

En C los números decimales se representan con los tipos **float** y **double**. La diferencia no solo está en que en el **double** quepan números más altos, sino en la precisión.

Ambos tipos son de **coma flotante**. Se conoce como coma flotante a una técnica de almacenamiento de números decimales en notación binaria, muy utilizada en todo tipo de máquinas electrónicas. En este estilo de almacenar números decimales, los números decimales no se almacenan de forma exacta.

La razón de esta **inexactitud** reside en la incompatibilidad del formato binario con el decimal. Por ello cuantos más bits se destinen para almacenar los números, más precisión tendrá el número. Por eso es más conveniente usar el tipo **double** aunque ocupe más memoria.

tipos lógicos

En C estándar el uso de valores lógicos se hace mediante los tipos enteros. Cualquier valor distinto de cero (normalmente se usa el uno) significa verdadero y el valor cero significa falso. Es decir en el código:

```
int x=1;
```

Se puede entender que **x** vale **1** o que **x** es **verdadera**. Ejemplo:

```
int x=(18>6);
printf("%d",x); /*Escribe 1, es decir verdadero */
```

En C++ se añadió el tipo **bool** (booleano o lógico) para representar valores lógicos, las variables **bool** pueden asociarse a los números cero y uno, o mejor a las palabras **true** (verdadero) y **false** (falso). Ejemplo:

```
bool b=true; //Ese es el equivalente C++ del C int b=1;
```

En C para conseguir el mismo efecto se utilizan trucos mediante la directiva **#define** (cuyo uso se explica más adelante)

modificadores de tipos

A los tipos anteriores se les puede añadir una serie de modificadores para que esos tipos varíen su funcionamiento. Esos modificadores se colocan por delante del tipo en cuestión. Son:

- ◆ **signed**. Permite que el tipo modificado admita números negativos. En la práctica se utiliza sólo para el tipo **char**, que de esta forma puede tomar un rango de valores de -128 a 127. En los demás no se usan ya que todos admiten números negativos
- ◆ **unsigned**. Contrario al anterior, hace que el tipo al que se refiere use el rango de negativos para incrementar los positivos. Por ejemplo el **unsigned int** tomaría el rango 0 a 65535 en lugar de -32768 a 32767
- ◆ **long**. Aumenta el rango del tipo al que se refiere.
- ◆ **short**. El contrario del anterior. La realidad es que no se utiliza.

Las combinaciones que más se suelen utilizar son:

| tipo de datos | rango de valores posibles | tamaño en bytes |
|---|---|-----------------|
| signed char | -128 a 127 | 1 |
| unsigned int | 0 a 65335 | 2 |
| long int (o long a secas) | -2147483648 a 2147483647 | 4 |
| long long | -9.223.372.036.854.775.809 a 9.223.372.036.854.775.808 (casi ningún compilador lo utiliza, casi todos usan el mismo que el long) | 8 |
| long double | 3.37E-4932 a 3,37E+4932 | 10 |

Una vez más hay que recordar que los rangos y tamaños depende del procesador y compilador. Por ejemplo, el tipo **long double** no suele incorporarlo casi ningún compilador (aunque sí la mayoría de los modernos).

Aún así conviene utilizar esas combinaciones por un tema de mayor claridad en el código.

(3.6.4) asignación de valores

Además de declarar una variable. A las variables se las pueden asignar valores. El operador de asignación en C es el signo “=”. Ejemplo:

```
x=3;
```

Si se utiliza una variable antes de haberla asignado un valor, ocurre un error. Pero es un error que no es detectado por un compilador. Por ejemplo el código:

```
#include <stdio.h>
int main(){
    int a;
    printf("%d",a);
    return 0;
}
```

Este código no produce error, pero como a la variable `a` no se le ha asignado ningún valor, el resultado del `printf` es un número sin sentido. Ese número representa el contenido de la zona de la memoria asociada a la variable `a`, pero el contenido de esa zona no le hemos preparado, por ello el resultado será absolutamente aleatorio.

Hay que destacar que se puede declarar e inicializar valores a la vez:

```
int a=5;
```

E incluso inicializar y declarar varias variables a la vez:

```
int a=8, b=7, c;
```

También es válida esta instrucción:

```
int a=8, b=a, c;
```

La asignación `b=a` funciona si la variable `a` ya ha sido declarada (como es el caso)

(3.6.5) literales

Cuando una variable se asigna a valores literales (17, 2.3,etc.) hay que tener en cuenta lo siguiente:

- ◆ Los números se escriben tal cual ([17](#), [34](#), [39](#))
- ◆ El separador de decimales es el punto ([18.4](#) se interpreta como [18 coma 4](#))
- ◆ Si un número entero se escribe anteponiendo un cero, se entiende que está en notación octal. Si el número es [010](#), C entiende que es el [8](#) decimal
- ◆ Si un número entero se escribe anteponiendo el texto `0x` ([cero equis](#)), se entiende que es un número hexadecimal. El número [0x10](#) significa 16.
- ◆ En los números decimales se admite usar notación científica. Así [1.23e+23](#) significa [1,23 · 1023](#)
- ◆ Los caracteres simples van encerrados entre comillas simples, por ejemplo '[a](#)'
- ◆ Los textos ([strings](#)) van encerrados entre comillas dobles. Por ejemplo "[Hola](#)"

- ◆ Los enteros se pueden almacenar como caracteres ‘A’ o como enteros cortos. Es más ‘A’ es lo mismo que 65 (ya que 65 es el código ASCII de la letra A). Eso vale tanto para los tipos **char** como para los **int**.

secuencias de escape

En el caso de los caracteres, hay que tener en cuenta que hay una serie de caracteres que son especiales. Por ejemplo si queremos almacenar en una variable **char** el símbolo de la comilla simple, no podríamos ya que ese símbolo está reservado para delimitar caracteres. Es decir:

```
char a='';/*Error*/
```

Eso no se puede hacer ya que el compilador entiende que hay una mala delimitación de caracteres. Para resolver este problema (y otros) se usan los caracteres de escape, que representan caracteres especiales.

Todos comienzan con el signo “\” (**backslash**) seguido de una letra minúscula, son:

| código | significado |
|--------|--|
| \a | Alarma (beep del ordenador) |
| \b | Retroceso |
| \n | Nueva línea |
| \r | Retorno de carro |
| \t | Tabulador |
| \' | Comilla simple |
| \“ | Comilla doble |
| \\\ | Barra inclinada invertida o backslash |

(3.6.6) ámbito de las variables

Toda variable tiene un **ámbito**. Esto es la parte del código en la que una variable se puede utilizar. De hecho las variables tienen un ciclo de vida:

- (1) En la declaración se reserva el espacio necesario para que se puedan comenzar a utilizar (digamos que se avisa de su futura existencia)
- (2) Se la asigna su primer valor (la variable **nace**)
- (3) Se la utiliza en diversas sentencias (no se puede utilizar su contenido sin haberla asignado ese primer valor).
- (4) Cuando finaliza el bloque en el que fue declarada, la variable **muere**. Es decir, se libera el espacio que ocupa esa variable en memoria. No se la podrá volver a utilizar.

variables locales

Son variables que se crean dentro de un bloque (se entiende por bloque, el código que está entre { y }). Con el fin del bloque la variable es eliminada. La

mayoría son locales a una determinada función, es decir sólo se pueden utilizar dentro de esa función. Ejemplo:

```
void func1(){  
    int x;  
    x=5;  
}  
void func2(){  
    int x;  
    x=300;  
}
```

Aunque en ambas funciones se usa `x` como nombre de una variable local. En realidad son dos variables distintas, pero con el mismo nombre. Y no podríamos usar `x` dentro de la función `func2` porque estamos fuera de su ámbito. Otro ejemplo:

```
void func0{  
    int a;  
    a=13;  
    {  
        int b;  
        b=8;  
    }/*Aquí muere b*/  
    a=b; /*Error! b está muerta*/  
}/*Aquí muere a*/
```

En la línea `a=b` ocurre un error de tipo “**Variable no declarada**”, el compilador ya no reconoce a la variable `b` porque estamos fuera de su ámbito.

variables globales

Son variables que se pueden utilizar en cualquier parte del código. Para que una variable sea global basta con declararla fuera de cualquier bloque. Normalmente se declaran antes de que aparezca la primera función:

```
#include <stdio.h>  
int a=3; //La variable "a" es global  
int main(){  
    printf("%d",a);  
    return 0;  
}
```

En C no se permite declarar en el mismo bloque dos variables con el mismo nombre. Pero sí es posible tener dos variables con el mismo nombre si están en bloques distintos. Esto plantea un problema, ya que cuando se utiliza la variable surge una duda: ¿qué variable utilizará el programa, la más local o la más global? La respuesta es que siempre se toma la variable declarada más localmente.

Ejemplo:

```
#include <stdio.h>
int a=3;
int main(){
    int a=5;
    {
        int a=8;
        printf("%d",a); //escribe 8. No hay error
    }
    return 0;
}
```

En el código anterior se han declarado tres variables con el mismo nombre (*a*). Cuando se utiliza la instrucción *printf* para escribir el valor de *a*, la primera vez escribe 8, la segunda vez escribe 5 (ya que ese *printf* está fuera del bloque más interior).

Es imposible acceder a las variables globales si disponemos de variables locales con el mismo nombre. Por eso no es buena práctica repetir el nombre de las variables.

(3.6.7) conversión de tipos

En numerosos lenguajes no se pueden asignar valores entre variables que sean de distinto tipo. Esto implicaría que no se podría asignar a una variable **char** valores de tipo **int**.

En C no existe esta restricción, al ser un lenguaje más libre. Lo que significa que los valores se toman directamente (en realidad el valor es el mismo, es la forma de almacenarse en el ordenador lo que cambia). Esto provoca problemas porque la copia de los valores que hace el lenguaje C es literal, sin embargo los enteros **int** normalmente utilizan 2 o 4 bytes para almacenar valores mientras que las variables **char** 1 o 2. Por ello cuando se asignan valores de una variable entera a una char puede haber problemas. Ejemplo:

```
#include <stdio.h>
int main(){
    char a;
    int b=300;
    a=b;
    printf("%d %d",a,b);
}
/* Escribe el contenido de a y de b. Escribe 44 y 300 */
```

En ese programa el contenido de *a* debería ser 300, pero como 300 sobrepasa el rango de las variables **char**, el resultado es 44. Es decir, no tiene sentido, esa salida está provocada por el hecho de perder ciertos bits en esa asignación.

En la conversión de **double** a **float** lo que ocurre normalmente es un redondeo de los valores para ajustarlos a la precisión de los **float**.

(3.6.8) modificadores de acceso

Los modificadores son palabras que se colocan delante del tipo de datos en la declaración de las variables para variar su funcionamiento (al estilo de **unsigned**, **short** o **long**)

modificador extern

Se utiliza al declarar variables globales e indica que la variable global declarada, en realidad se inicializa y declara en otro archivo. Ejemplo

| Archivo 1 | Archivo 2 |
|--|---|
| <pre>int x,y; int main(){ x=12; y=6; } void funcion1(void){ x=7; }</pre> | <pre>extern int x,y; void func2(void){ x=2*y; }</pre> |

El segundo archivo utiliza las variables declaradas en el primero

modificador auto

En realidad las variables toman por defecto este valor (luego no hace falta utilizarle). Significa que las variables se eliminan al final del bloque en el que fueron creadas.

modificador static

Se trata de variables que no se eliminan cuando el bloque en el que fueron creadas finaliza. Así que si ese bloque (normalmente una función), vuelve a invocarse desde el código, la variable mantendrá el último valor anterior.

Si se utiliza este modificador con variables globales, indica que esas variables sólo pueden utilizarse desde el archivo en el que fueron creadas.

modificador register

Todos los ordenadores posee una serie de memorias de pequeño tamaño en el propio procesador llamadas **registros**. Estas memorias son mucho más rápidas pero con capacidad para almacenar muy pocos datos.

Este modificador solicita que una variable sea almacenada en esos registros para acelerar el acceso a la misma. Se utiliza en variables **char** o **int** a las que se va a acceder muy frecuentemente en el programa (por ejemplo las variables contadoras en los bucles). Sólo vale para variables locales.

```
register int cont;
for (cont=1;cont<=300000;cont++){
    ...
}
```

modificador **const**, Constantes

Las variables declaradas con la palabra **const** delante del tipo de datos, indican que son sólo de lectura. Es decir, constantes. Las constantes no pueden cambiar de valor, el valor que se asigne en la declaración será el que permanezca (es obligatorio asignar un valor en la declaración). Ejemplo:

```
const float PI=3.141592;
```

Inicialmente no existía en C este modificador, fue de hecho añadido por ANSI en versiones más modernas de C. En su lugar para las constantes, los programadores en C utilizaban la directiva **#define**.

De hecho aún sigue siendo una directiva muy empleada. Su funcionamiento es el siguiente:

```
#define identificador valor
```

Al identificador se le asigna un valor. Este directiva (que se suele poner después de las directivas **#include**), hace que en la fase de preprocessado, se sustituya el identificador indicado por el valor correspondiente. Normalmente el identificador se pone en mayúsculas (es el uso más recomendado) para indicar que es una constante.

Ejemplo:

```
#define PI 3.141592
```

Desde ese momento podremos utilizar el nombre PI, refiriéndonos al valor 3.141592 (observar que no se utiliza ningún signo de igualdad, =), en la fase de preprocessado (antes de compilar) todas las constantes definidas con la directiva **#define**, serán traducidas por el valor correspondiente (de ahí que no se consideren verdaderas constantes).

modificador **volatile**

Se usa para variables que son modificadas externamente al programa (por ejemplo una variable que almacene el reloj del sistema).

(3.7) entrada y salida por consola

Aunque este tema será tratado con detalle más adelante. Es conveniente conocer al menos las funciones **printf** y **scanf** que permiten entradas y salidas de datos de forma muy interesante.

(3.7.1) función **printf**

La función **printf** permite escribir datos en la consola de salida (en la pantalla). Si lo que se desea sacar es un texto literal se utiliza de esta manera:

```
printf("Hola");
```

Lógicamente a veces se necesitan sacar el contenido de las variables en la salida. Para ello dentro del texto que se desea mostrar se utilizan unas indicaciones de formato, las cuales se indican con el signo "%" seguido de una letra minúscula. Por ejemplo:

```
int edad=18;  
...  
printf("Tengo %d años",edad);  
/*escribe tengo 18 años*/
```

El código **%d** sirve para recoger el valor de la variable que se indique entendiendo que esa variable tiene valores enteros. Esa variable va separada del texto por una coma. Si se usan más variables o valores, éstos se separan con comas entre sí. Ejemplo:

```
int edad=18, edadFutura=19;  
printf("Tengo %d años, el año que viene %d",edad,edadFutura);  
/*Escribe:  
Tengo 18 años, el año que viene 19  
*/
```

Además del código **%d**, hay otros que se pueden utilizar:

| código | significado |
|------------------|---|
| %d | Entero |
| %i | Entero |
| %c | Un carácter |
| %f | Punto flotante (para <i>double</i> o <i>float</i>) |
| %e | Escribe en notación científica |
| %g | Usa %e o %f, el más corto de los dos |
| %o | Escribe números enteros en notación octal |
| %x | Escribe números enteros en notación hexadecimal |
| %s | Cadena de caracteres |
| %u | Enteros sin signo |
| %li o %ld | Enteros largos (<i>long</i>) |
| %p | Escribe un puntero |
| %% | Escribe el signo % |

Ejemplo:

```
char c='H';  
int n=28;  
printf("Me gusta la letra %c y el número %d, %s",c,n,"adiós");  
/* Escribe: Me gusta la letra H y el número 28, adiós */
```

Mediante **printf**, también se pueden especificar anchuras para el texto. Es una de sus características más potentes. Para ello se coloca un número entre el signo % y el código numérico (i, d, f,...). Ese número indica anchura mínima.

Por ejemplo:

```
int a=127, b=8, c=76;
printf("%4d\n",a);
printf("%4d\n",b);
printf("%4d\n",c);
/* Sale:
   127
   8
   76
*/
```

En el ejemplo se usan cuatro caracteres para mostrar los números. Los números quedan alineados a la derecha en ese espacio, si se desean alinear a la izquierda, entonces el número se pone en negativo (**%-4d**).

Si se coloca un cero delante del número, el espacio sobrante se rellena con ceros en lugar de con espacios.

Ejemplo:

```
int a=127, b=8, c=76;
printf("%04d\n",a);
printf("%04d\n",b);
printf("%04d\n",c);
/* Sale:
   0127
   0008
   0076
*/
```

También se pueden especificar los decimales requeridos (para los códigos decimales, como f por ejemplo). De esta forma el código **%10.4f** indica que se utiliza un tamaño mínimo de 10 caracteres de los que 4 se dejan para los decimales.

(3.7.2) función **scanf**

Esta función es similar a la anterior. sólo que ésta sirve para leer datos. Posee al menos dos parámetros, el primero es una cadena que indica el formato de la lectura (que se usa del mismo modo que en la función **printf**), el segundo (y siguientes) son las zonas de memoria en las que se almacenará el resultado.

Por ejemplo, para leer un número entero por el teclado y guardarla en una variable de tipo **int** llamada **a**, se haría:

```
scanf("%d",&a);
```

El símbolo **&**, sirve para tomar la dirección de memoria de la variable **a**. Esto significa que **a** contendrá el valor leído por el teclado. Más adelante se explicará

el significado de dicho símbolo, por ahora sólo indicar que es absolutamente necesario su uso con la función scanf.

Este otro ejemplo:

```
scanf("%d %f",&a,&b);
```

Lee dos números por teclado. El usuario tendrá que colocar un espacio entre cada número. El primero se almacenará en *a* y el segundo (decimal) en *b*.

Otro ejemplo:

```
scanf("%d,%d",&a,&b);
```

No obstante a la hora de leer por teclado es muy aconsejable leer sólo un valor cada vez.

(3.8) operadores

Se trata de uno de los puntos fuertes de este lenguaje que permite especificar expresiones muy complejas gracias a estos operadores.

(3.8.1) operadores aritméticos

Permiten realizar cálculos matemáticos. Son:

| operador | significado |
|----------|----------------|
| + | Suma |
| - | Resta |
| * | Producto |
| / | División |
| % | resto (módulo) |
| ++ | Incremento |
| -- | Decremento |

La suma, resta, multiplicación y división son las operaciones normales. Sólo hay que tener cuidado en que el resultado de aplicar estas operaciones puede ser un número que tenga un tipo diferente de la variable en la que se pretende almacenar el resultado.

El signo “-“ también sirve para cambiar de signo (*-a* es el resultado de multiplicar a la variable *a* por -1).

El incremento (++), sirve para añadir uno a la variable a la que se aplica. Es decir *x++* es lo mismo que *x=x+1*. El decremento funciona igual pero restando uno. Se puede utilizar por delante (**preincremento**) o por detrás (**postincremento**) de la variable a la que se aplica (*x++* ó *++x*). Esto último tiene connotaciones.

Por ejemplo:

```
int x1=9,x2=9;
int y1,y2;
y1=x1++;
y2=++x2;
printf("%i\n",x1); /*Escribe 10*/
printf("%i\n",x2); /*Escribe 10*/
printf("%i\n",y1); /*iiiEscribe 9!!! */
printf("%i\n",y2); /*Escribe 10*/
```

La razón de que $y1$ valga 9, está en que la expresión $y1=x1++$, funciona de esta forma:

```
y1=x1;
x1=x1+1;
```

Mientras que en $y2=++x2$, el funcionamiento es:

```
x2=x2+1;
y2=x2;
```

Es decir en $x2++$ primero se asigna y luego se incrementa. Si el incremento va antes se realiza al contrario.

(3.8.2) operadores relacionales

Son operadores que sirven para realizar comparaciones. El resultado de estos operadores es verdadero o falso (uno o cero). Los operadores son:

| operador | significado |
|----------|-------------------|
| > | Mayor que |
| >= | Mayor o igual que |
| < | Menor que |
| <= | Menor o igual que |
| == | Igual que |
| != | Distinto de |

(3.8.3) operadores lógicos

Permiten agrupar **expresiones lógicas**. Las expresiones lógicas son todas aquellas expresiones que obtienen como resultado verdadero o falso. Estos operadores unen estas expresiones devolviendo también verdadero o falso. Son:

| operador | significado |
|----------|-------------|
| && | Y (AND) |
| | O (OR) |
| ! | NO (NOT) |

Por ejemplo: $(18>6) \&\& (20<30)$ devuelve verdadero (1) ya que la primera expresión $(18>6)$ es verdadera y la segunda $(20<30)$ también. El operador Y ($\&\&$)

devuelve verdadero cuando las dos expresiones son verdaderas. El operador O (||) devuelve verdadero cuando cualquiera de las dos es verdadera.

Finalmente el operador NO (!) invierte la lógica de la expresión que le sigue; si la expresión siguiente es verdadera devuelve falso y viceversa. Por ejemplo !(18>15) devuelve falso (0).

(3.8.4) operadores de bits:

Permiten realizar operaciones sobre los bits del número, o números, sobre los que operan. Es decir si el número es un **char** y vale **17**, **17** en binario es **00010001**. Estos operadores operan sobre ese código binario. En estos apuntes simplemente se indican estos operadores:

| operador | significado |
|----------|------------------------------------|
| & | AND de bits |
| | OR de bits |
| ~ | NOT de bits |
| ^ | XOR de bits |
| >> | Desplazamiento derecho de los bits |
| << | Desplazamiento izquierdo de bits |

(3.8.5) operador de asignación

Ya se ha comentado que el signo “=” sirve para asignar valores. Se entiende que es un operador debido a la complejidad de expresiones de C. Por ejemplo:

```
int x=5,y=6,z=7;  
x=(z=y++)*8;  
printf("%d",x); //Escribe 48
```

En C existen estas formas abreviadas de asignación. Esto sirve como abreviaturas para escribir código. Así la expresión:

```
x=x+10;
```

Se puede escribir como:

```
x+=10;
```

Se permiten estas abreviaturas:

| operador | significado |
|----------|---------------------------|
| += | Suma y asigna |
| -= | Resta y asigna |
| *= | Multiplica y asigna |
| /= | Divide y asigna |
| %= | Calcula el resto y asigna |

Además también se permiten abreviar las expresiones de bit: &=, |=, ^=, >>=, <<=

(3.8.6) operador ?

Permite escribir expresiones condicionales. Su uso es el siguiente:

Expresión_a_valorar?Si_verdadera:Si_falsa

Ejemplo:

`x=(y>5?'A':'B');`

Significa que si la variable `y` es mayor de `5`, entonces a `x` se le asigna el carácter '`A`', sino se le asignará el carácter '`B`'.

Otro ejemplo:

`printf("%s",nota>=5?"Aprobado":"Suspensos");`

En cualquier caso hay que utilizar este operador con cautela. Su dominio exige mucha práctica.

(3.8.7) operadores de puntero & y *

Aunque ya se le explicará más adelante con detalle, conviene conocerle un poco. El operador `&` sirve para obtener la dirección de memoria de una determinada variable. No tiene sentido querer obtener esa dirección salvo para utilizar punteros o para utilizar esa dirección para almacenar valores (como en el caso de la función `scanf`).

El operador `*` también se utiliza con punteros. Sobre una variable de puntero, permite obtener el contenido de la dirección a la que apunta dicho puntero.

(3.8.8) operador sizeof

Este operador sirve para devolver el tamaño en bytes que ocupa en memoria una determinada variable. Por ejemplo:

`int x=18;
printf("%i",sizeof x); /*Escribe 2 (o 4 en algunos compiladores)*/`

Devuelve 2 o 4, dependiendo del gasto en bytes que hacen los enteros en la máquina y compilador en que nos encontremos.

(3.8.9) operador coma

La coma `,` sirve para poder realizar más de una instrucción en la misma línea. Por ejemplo:

`y=(x=3,x++);`

La coma siempre ejecuta la instrucción que está más a la izquierda. Con lo que en la línea anterior primero la `x` se pone a 3; y luego se incrementa la `x` tras haber asignado su valor a la variable `y` (ya que es un postincremento).

No es un operador muy importante y puede dar lugar a fallos; especialmente habría que evitarlo cuando se está empezando a programar.

(3.8.10) operadores especiales

Todos ellos se verán con más detalle en otros temas. Son:

- ◆ **El operador “.” (punto).** Este operador permite hacer referencia a campos de un registro. Un registro es una estructura de datos avanzada que permite agrupar datos de diferente tipo.
- ◆ **El operador flecha “->”** que permite acceder a un campo de registro cuando es un puntero el que señala a dicho registro.
- ◆ **Los corchetes “[]”,** que sirven para acceder a un elemento de un array. Un array es una estructura de datos que agrupa datos del mismo tipo
- ◆ **(Molde) o (cast).** Que se explica más adelante y que sirve para conversión de tipos.

(3.8.11) orden de los operadores

En expresiones como:

`x=6+9/3;`

Podría haber una duda. ¿Qué vale x? Valdría 5 si primero se ejecuta la suma y 9 si primero se ejecuta la división. La realidad es que valdría 9 porque la división tiene preferencia sobre la suma. Es decir hay operadores con mayor y menor preferencia.

Lógicamente el orden de ejecución de los operadores se puede modificar con paréntesis.

Por ejemplo:

`x=(6+9/3;
y=(x*3)/((z*2)/8);`

Como se observa en el ejemplo los paréntesis se pueden anidar.

Sin paréntesis el orden de precedencia de los operadores es orden de mayor a menor precedencia, forma 16 niveles. Los operadores que estén en el mismo nivel significa que tienen la misma precedencia. En ese caso se ejecutan primero los operadores que estén más a la izquierda.

El orden es (de mayor a menor precedencia):

- (1) () [] . ->
- (2) Lo forman los siguientes:
 - ♦ NOT de expresiones lógicas: !
 - ♦ NOT de bits: ~
 - ♦ Operadores de punteros: * &
 - ♦ Cambio de signo: -
 - ♦ sizeof
 - ♦ (cast)
 - ♦ Decremento e incremento: ++ --
- (3) Aritméticos prioritarios: / * %
- (4) Aritméticos no prioritarios (suma y resta): + -
- (5) Desplazamientos: >> <<
- (6) Relacionales sin igualdad: > < >= <=
- (7) Relacionales de igualdad: == !=
- (8) &
- (9) ^
- (10) |
- (11) &&
- (12) ||
- (13) ?:
- (14) = *= /* += -= %= >>= <<= |= &= ^=
- (15) , (coma)

(3.9) expresiones y conversión de tipos

(3.9.1) introducción

Operadores, variables, constantes y funciones son los elementos que permiten construir expresiones. Una expresión es pues un código en C que obtiene un determinado valor (del tipo que sea).

(3.9.2) conversión

Cuando una expresión utiliza valores de diferentes tipos, C convierte la expresión al mismo tipo. La cuestión es, qué criterio sigue para esa conversión. El criterio, en general, es que C toma siempre el tipo con rango más grande. En

ese sentido si hay un dato **long double**, toda la expresión se convierte a long double, ya que ese es el tipo más grande. Si no aparece un long double entonces el tipo más grande en el que quepan los datos.

El orden de tamaños es:

- (1) long double**
- (2) double**
- (3) float**
- (4) unsigned long**
- (5) long**
- (6) int**

Es decir si se suma un **int** y un **float** el resultado será **float**.

En una expresión como:

```
int x=9.5*2;
```

El valor 9.5 es **double** mientras que el valor 2 es **int** por lo que el resultado (19) será **double**. Pero como la variable **x** es entera, el valor deberá ser convertido a entero finalmente.

(3.9.3) operador de molde o cast

A veces se necesita hacer conversiones explícitas de tipos. Para eso está el operador **cast**. Este operador sirve para convertir datos. Su uso es el siguiente, se pone el tipo deseado entre paréntesis y a la derecha el valor a convertir. Por ejemplo:

```
x=(int) 8.3;
```

x valdrá **8** independientemente del tipo que tenga, ya que al convertir datos se pierden decimales.

Este ejemplo (comparar con el utilizado en el apartado anterior):

```
int x=(int)9.5*2;
```

Hace que **x** valga **18**, ya que al convertir a entero el **9.5** se pierden los decimales.

(3.10) índice de ilustraciones

| | |
|---|----|
| Ilustración 1, proceso de compilación de un programa C | 8 |
| Ilustración 2, Opciones principales del editor | 14 |
| Ilustración 3, Opciones de visualización..... | 15 |
| Ilustración 4, Opciones de sintaxis, coloreado del código | 15 |
| Ilustración 5, Opciones del menú Insertar | 16 |
| Ilustración 6, Ejemplo de código por defecto..... | 16 |

Unidad 4: Programación estructurada en Lenguaje C

Fundamentos de Programación. 1º de ASI



Esta obra está bajo una licencia de Creative Commons.

Autor: Jorge Sánchez Asenjo (año 2008) <http://www.jorgesanchez.net>
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons

Para ver una copia de esta licencia, visite:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

o envíe una carta a:

Creative Commons, 559 Nathan Abbot

(4)

programación

estructurada en

lenguaje C

(4.1) esquema de la unidad

| | |
|---|----|
| (1.1) esquema de la unidad | 5 |
| (1.2) expresiones lógicas | 6 |
| (1.3) sentencia if | 6 |
| (1.3.1) sentencia condicional simple | 6 |
| (1.3.2) sentencia condicional compuesta | 7 |
| (1.3.3) anidación | 8 |
| (1.4) sentencia switch | 9 |
| (1.5) bucles | 11 |
| (1.5.1) sentencia while | 11 |
| (1.5.2) sentencia do..while | 14 |
| (1.5.3) sentencia for | 14 |
| (1.6) sentencias de ruptura de flujo | 15 |
| (1.6.1) sentencia break | 15 |
| (1.6.2) sentencia continue | 16 |
| (1.7) índice de ilustraciones | 16 |

(4.2) expresiones lógicas

Hasta este momento con lo visto en los temas anteriores, nuestros programas en C apenas pueden realizar programas que simulen, como mucho, una calculadora. Lógicamente necesitamos poder elegir qué cosas se ejecutan según unas determinada circunstancias y eso va a permitir realizar programas más complejos y útiles.

Las sentencias que permiten tomar decisiones en un programa son las llamadas **sentencias de control de flujo**. Todas ellas se basan en evaluar una **expresión lógica**. Una expresión lógica es cualquier expresión que al ser evaluada por el compilador devuelve **verdadero** o **falso**. En C (o C++) se considera verdadera cualquier expresión distinta de 0 (en especial el uno, valor **verdadero**) y falsa el cero (**falso**).

(4.3) sentencia if

(4.3.1) sentencia condicional simple

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de sentencias en caso de que la expresión lógica sea verdadera. Su sintaxis es:

```
if(expresión lógica){  
    sentencias  
    ...  
}
```

Si sólo se va a ejecutar una sentencia, no hace falta usar las llaves:

```
if(expresión lógica) sentencia;
```

Ejemplo:

```
if(nota>=5){  
    printf("Aprobado");  
    aprobados++;  
}
```

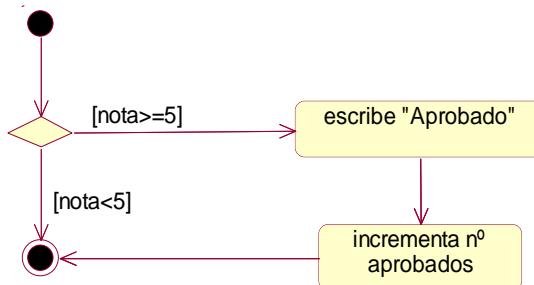


Ilustración 1, Diagrama de actividad del if simple

(4.3.2) sentencia condicional compuesta

Es igual que la anterior, sólo que se añade un apartado **else** que contiene instrucciones que se ejecutarán si la expresión evaluada por el **if** es falsa. Sintaxis:

```
if(expresión lógica){  
    sentencias  
    ....  
}  
else {  
    sentencias  
    ...  
}
```

Las llaves son necesarias sólo si se ejecuta más de una sentencia. Ejemplo:

```
if(nota>=5){  
    printf("Aprobado");  
    aprobados++;  
}  
else {  
    printf("Suspensos");  
    suspensos++;  
}
```

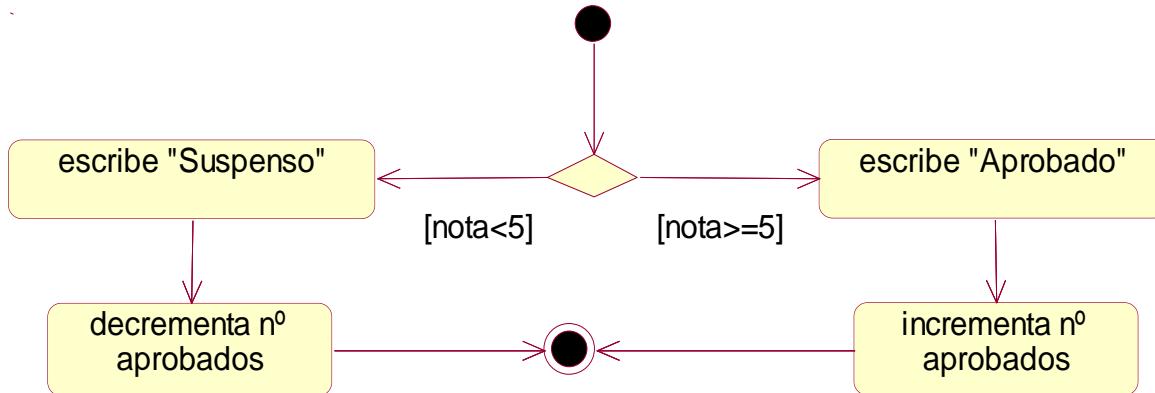


Ilustración 2, diagrama UML de actividad del if-else

(4.3.3) anidación

Dentro de una sentencia **if** se puede colocar otra sentencia **if**. A esto se le llama **anidación** y permite crear programas donde se valoren expresiones complejas.

Por ejemplo en un programa donde se realice una determinada operación dependiendo de los valores de una variable.

El código podría ser:

```
if (x==1) {
    sentencias
    ...
}
else {
    if(x==2) {
        sentencias
        ...
    }
    else {
        if(x==3) {
            sentencias
            ...
        }
    }
}
```

Pero si cada **else** tiene dentro sólo una instrucción **if** entonces se podría escribir de esta forma (que es más legible), llamada **if-else-if**:

```
if (x==1) {  
    instrucciones  
    ...  
}  
else if (x==2) {  
    instrucciones  
    ...  
}  
else if (x==3) {  
    instrucciones  
    ...  
}
```

(4.4) sentencia switch

Se trata de una sentencia que permite construir alternativas múltiples. Pero que en el lenguaje C está muy limitada. Sólo sirve para evaluar el valor de una variable entera (o de carácter, **char**).

Tras indicar la expresión entera que se evalúa, a continuación se compara con cada valor agrupado por una sentencia **case**. Cuando el programa encuentra un **case** que encaja con el valor de la expresión se ejecutan todos los **case** siguientes. Por eso se utiliza la sentencias **break** para hacer que el programa abandone el bloque **switch**. Sintaxis:

```
switch(expresión entera){  
    case valor1:  
        sentencias  
        break; /* break, sirve para que programa salte fuera del  
                switch de otro modo atraviesa todos los  
                demás case */  
    case valor2:  
        sentencias  
    ...  
    default:  
        sentencias  
}
```

Ejemplo:

```
switch (diasemana) {  
    case 1:  
        printf("Lunes");  
        break;  
    case 2:  
        printf("Martes");  
        break;  
    case 3:  
        printf("Miércoles");  
        break;  
    case 4:  
        printf("Jueves");  
        break;  
    case 5:  
        printf("Viernes");  
        break;  
  
    case 6:  
        printf("Sábado");  
        break;  
    case 7:  
        printf("Domingo");  
        break;  
    default:  
        std::cout<<"Error";  
}
```

Sólo se pueden evaluar expresiones con valores concretos (no hay una `case >3` por ejemplo). Aunque sí se pueden agrupar varias expresiones aprovechando el hecho de que al entrar en un case se ejecutan las expresiones de los siguientes.

Ejemplo:

```
switch (diasemana) {  
    case 1:  
    case 2:  
    case 3:  
    case 4:  
    case 5:  
        printf("Laborable");  
        break;  
    case 6:  
    case 7:  
        printf("Fin de semana");  
        break;  
    default:  
        printf("Error");  
}
```

(4.5) bucles

A continuación se presentan las instrucciones C que permiten realizar instrucciones repetitivas (**bucles**). Los bucles permiten repetir una serie de instrucciones hasta que se cumpla una determinada condición. Dicha condición debe variar en el bucle, de otro modo el bucle sería infinito.

(4.5.1) sentencia while

Es una de las sentencias fundamentales para poder programar. Se trata de una serie de instrucciones que se ejecutan continuamente mientras una expresión lógica sea cierta.

Sintaxis:

```
while (expresión lógica) {  
    sentencias  
}
```

El programa se ejecuta siguiendo estos pasos:

- (1) Se evalúa la expresión lógica
- (2) Si la expresión es verdadera ejecuta las sentencias, sino el programa abandona la sentencia **while**
- (3) Tras ejecutar las sentencias, volvemos al paso 1

Ejemplo (escribir números del 1 al 100):

```
int i=1;  
while (i<=100){  
    printf("%d ",i);  
    i++;  
}
```

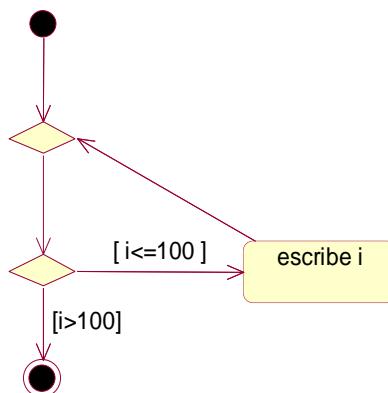


Ilustración 3, diagrama UML de actividad del bucle while

bucles con contador

Se llaman así a los bucles que se repiten una serie determinada de veces. Dichos bucles están controlados por un (puede que incluso más de un) contador. El contador es una variable que va contando (de uno en uno, de dos en dos,... o como queramos) en cada vuelta del bucle. Cuando el contador alcanza un límite, entonces el bucle termina.

En todos los bucles de contador necesitamos saber:

- (1)** Lo que vale la variable contadora al principio. Antes de entrar en el bucle
- (2)** Lo que varía (lo que se incrementa o decrementa) el contador en cada vuelta del bucle.
- (3)** El valor final del contador. En cuanto se rebasa el bucle termina. Dicho valor se pone como condición del bucle, pero a la inversa. Es decir, la condición mide el valor que tiene que tener el contador para que el bucle se repita.

Ejemplo:

```
i=10; /*Valor inicial del contador, empieza valiendo 10  
       (por supuesto i debería estar declarada como entera, int) */  
  
while (i<=200){ /* condición del bucle, mientras i sea menor de  
                   200, el bucle se repetirá, cuando i rebasa este  
                   valor, el bucle termina */  
  
    printf("%d\n",i); /*acciones que ocurren en cada vuelta del bucle  
                      en este caso simplemente escribe el valor  
                      del contador */  
  
    i+=10; /* Variación del contador, en este caso cuenta de 10 en 10*/  
}  
/* Al final el bucle escribe:  
10  
20  
30  
...  
y así hasta 200  
*/
```

Bucles de centinela

Es el segundo tipo de bucle básico. Una condición lógica llamada **centinela**, que puede ser desde una simple variable booleana hasta una expresión lógica más compleja, sirve para decidir si el bucle se repite o no. De modo que cuando la condición lógica se incumpla, el bucle termina.

Ejemplo:

```
int salir=0; /* En este caso el centinela es una variable booleana
               que inicialmente vale falso, o */

while(salir==0){ /* Condición de repetición: que salir siga siendo
                   falso. Ese es el centinela.
                   También se podía haber escrito simplemente:
                   while(!salir)

               */
    scanf("%c",&caracter); /* Lo que se repite en el bucle: leer carácter*/
    salir=(caracter=='S'); /* El centinela vale verdadero si el carácter
                           leído es la S mayúscula, de no ser
                           así, el centinela seguirá siendo falso
                           y el bucle se repite
               */
}
```

Comparando los bucles de centinela con los de contador, podemos señalar estos puntos:

- ◆ Los bucles de contador se repiten un número concreto de veces, los bucles de centinela no
- ◆ Un bucle de contador podemos considerar que es seguro que finalice, el de centinela puede no finalizar si el centinela jamás varía su valor (aunque seguro que alguna vez lo alcanza)
- ◆ Un bucle de contador está relacionado con la programación de algoritmos basados en series.

Un bucle podría ser incluso mixto, de centinela y de contador. Por ejemplo imaginar un programa que se repita indefinidamente hasta que llegue una determinada contraseña (que podría ser el número 12345), pero que como mucho se repite tres veces (al tercer fallo de contraseña, el programa termina). Sería:

```
int i=1; /*Contador*/
int salir=0; /*Centinela*/
char caracter;
while(salir==0 && i<=3){
    scanf("%c",&caracter);
    salir=(caracter=='S');
    i++;
}
```

(4.5.2) sentencia do..while

La única diferencia respecto a la anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir el bucle al menos se ejecuta una vez. Es decir los pasos son:

- (1) Ejecutar sentencias
- (2) Evaluar expresión lógica
- (3) Si la expresión es verdadera volver al paso 1, sino continuar fuera del while

Sintaxis:

```
do {  
    sentencias  
    ...  
} while (expresión lógica)
```

Ejemplo (contar del 1 al 1000):

```
int i=0;  
do {  
    i++;  
    printf("%d",i);  
} while (i<=1000);
```

Se utiliza cuando sabemos al menos que las sentencias del bucle se van a repetir una vez (en un bucle **while** puede que incluso no se ejecuten las sentencias que hay dentro del bucle si la condición fuera falsa, ya desde un inicio).

De hecho cualquier sentencia **do..while** se puede convertir en while. El ejemplo anterior se puede escribir usando la instrucción while, así:

```
int i=0;  
  
i++;  
printf("%d",i);  
while (i<=1000){  
    i++;  
    printf("%d",i);  
}
```

(4.5.3) sentencia for

Se trata de una sentencia pensada para realizar bucles contadores. Su formato es:

```
for(inicialización;condición;incremento){  
    sentencias  
}
```

Las sentencias se ejecutan mientras la condición sea verdadera. Además antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir el funcionamiento es:

- (1) Se ejecuta la instrucción de inicialización
- (2) Se comprueba la condición
- (3) Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque **for**
- (4) Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2

Ejemplo (contar números del 1 al 1000):

```
for(int i=1;i<=1000;i++){  
    printf("%d",i);  
}
```

La ventaja que tiene es que el código se reduce. La desventaja es que el código es menos comprensible. El bucle anterior es equivalente al siguiente bucle **while**:

```
i=1; /*sentencia de inicialización*/  
while(i<=1000){ /*condición*/  
    printf("%d",i);  
    i++; /*incremento*/  
}
```

(4.6) sentencias de ruptura de flujo

No es aconsejable su uso ya que son instrucciones que rompen el paradigma de la programación estructurada. Cualquier programa que las use ya no es estructurado. Se comentan aquí porque puede ser útil conocerlas, especialmente para interpretar código de terceros.

(4.6.1) sentencia break

Se trata de una sentencia que hace que el flujo del programa abandone el bloque en el que se encuentra.

```
for(int i=1;i<=1000;i++){  
    printf("%d",i);  
    if(i==300) break;  
}
```

En el listado anterior el contador no llega a 1000, en cuanto llega a 300 sale del **for**

(4.6.2) sentencia continue

Es parecida a la anterior, sólo que en este caso en lugar de abandonar el bucle, lo que ocurre es que no se ejecutan el resto de sentencias del bucle y se vuelve a la condición del mismo:

```
for(int i=1;i<=1000;i++){  
    if(i%3==0) continue;  
    printf("%d",i);  
}
```

En ese listado aparecen los números del 1 al 1000, menos los múltiplos de 3 (en los múltiplos de 3 se ejecuta la instrucción **continue** que salta el resto de instrucciones del bucle y vuelve a la siguiente iteración.

El uso de esta sentencia genera malos hábitos, siempre es mejor resolver los problemas sin usar **continue**. El ejemplo anterior sin usar esta instrucción quedaría:

```
for(int i=1;i<=1000;i++){  
    if(i%3!=0) printf("%d",i);  
}
```

La programación estructurada prohíbe utilizar las sentencias **break** y **continue**

(4.7) índice de ilustraciones

| | |
|---|----|
| Ilustración 1, Diagrama de actividad del if simple..... | 7 |
| Ilustración 2, diagrama UML de actividad del if-else..... | 8 |
| Ilustración 3, diagrama UML de actividad del bucle while..... | 11 |

Unidad 5: Programación modular en Lenguaje C

Fundamentos de Programación. 1º de ASI



Esta obra está bajo una licencia de Creative Commons.

Autor: Jorge Sánchez Asenjo (año 2008) <http://www.jorgesanchez.net>
e-mail:info@jorgesanchez.net

Esta obra está bajo una licencia de Reconocimiento-NoComercial-CompartirIgual de Creative Commons

Para ver una copia de esta licencia, visite:

<http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es>

o envíe una carta a:

Creative Commons, 559 Nathan Abbot

(5)

programación

estructurada en

lenguaje C

esquema de la unidad

| | |
|--|-----------|
| (5.1) funciones y programación modular | 6 |
| (5.1.1) introducción | 6 |
| (5.1.2) uso de las funciones | 6 |
| (5.1.3) crear funciones | 7 |
| (5.1.4) funciones void | 9 |
| (5.2) parámetros y variables | 10 |
| (5.2.1) introducción | 10 |
| (5.2.2) paso de parámetros por valor y por referencia | 10 |
| (5.2.3) variables static | 11 |
| (5.2.4) ámbito de las variables y las funciones | 12 |
| (5.3) definición de macro funciones con #define | 12 |
| (5.4) recursividad | 14 |
| (5.4.1) introducción | 14 |
| (5.4.2) ¿recursividad o iteración? | 15 |
| (5.4.3) recursividad cruzada | 16 |
| (5.5) bibliotecas | 16 |
| (5.5.1) introducción | 16 |
| (5.5.2) ejemplo de librería estándar. math | 17 |
| (5.5.3) números aleatorios | 17 |
| (5.5.4) crear nuestras propias librerías | 18 |
| (5.6) índice de ilustraciones | 19 |

(5.1) funciones y programación modular

(5.1.1) introducción

Uno de los problemas habituales del programador ocurre cuando los programas alcanzan un tamaño considerable en cuanto a líneas de código. El problema se puede volver tan complejo que fuera inabordable.

Para mitigar este problema apareció la **programación modular**. En ella el programa se divide en módulos de tamaño manejable. Cada módulo realiza una función muy concreta y se pueden programar de forma independiente. Se basa en concentrar los esfuerzos en resolver problemas sencillos y una vez resueltos, el conjunto de esos problemas soluciona el problema original.

En definitiva la programación modular implementa el paradigma **divide y vencerás**, tan importante en la programación. El programa se descompone en módulos. Los módulos se puede entender que son pequeños programas. Reciben datos y a partir de ellos realizan un calculo o una determinada tarea. Una vez el módulo es probado y validado se puede utilizar las veces que haga falta en el programa sin necesidad de tener que volver a programar.

En C la programación modular se implementa mediante **funciones**. De hecho en los temas anteriores hemos usado algunas funciones implementadas en las librerías del lenguaje C (**printf** y **scanf** por ejemplo). El lenguaje C proporciona diversas funciones ya preparadas (se las llama **funciones estándar**) para evitar que el programador tenga que codificar absolutamente todo el funcionamiento del programa.

Pero también el propio programador puede crear sus propias funciones, que podrán ser utilizadas en un programa e incluso en distintos programas; de esa forma se aprovecha al máximo todo el código escrito.

Las funciones son **invocadas** desde el código utilizando su nombre. Cuando desde el código se invoca a una función, entonces el flujo del programa salta hasta el código de la función invocada. Despues de ejecutar el código de la función, el flujo del programa regresa al código siguiente a la invocación.

(5.1.2) uso de las funciones

Toda aplicación de consola creada en lenguaje C se basa en una función llamada **main** que contiene el código que se ejecuta en primer lugar en el programa. Dentro de ese **main** habrá llamadas a funciones ya creadas, bien por el propio programador o bien funciones que forman parte de las bibliotecas de C o de bibliotecas privadas (una **biblioteca** o **librería** no es más que una colección de funciones).

Así por ejemplo:

```
int main(){
    printf("%lf", pow(3,4));
}
```

Ese código utiliza dos funciones, la función `printf` que permite escribir en la pantalla y la función `pow` que permite elevar un número a un exponente (en el ejemplo calcula: 3^4).

Para poder utilizarlas, las funciones tienen que estar definidas en el código del programa o en un archivo externo. Si están en un archivo externo (como ocurre en el ejemplo) habrá que incluir la cabecera de ese archivo para que durante el proceso de compilación se incluya el código de las librerías dentro de nuestro propio código.

Es decir, hay que indicar en qué archivos se definen esas funciones. En el ejemplo, habría que incluir al principio estas líneas:

```
#include <stdio.h> /* Para la función printf */  
#include <math.h> /* Para la función pow */
```

(5.1.3) crear funciones

Si creamos funciones, éstas deben definirse en el código. Los pasos para definir una función son:

- ◆ El **tipo de datos** que devuelve dicha función (si no devuelve nada, se indica con la palabra **void**)
- ◆ El **nombre** de la función, se indica justo después del tipo de datos (en la misma línea).
- ◆ Los **parámetros** que acepta la función, se colocan en la misma línea y se encierran entre paréntesis. Los parámetros son los datos de entrada que necesita la función para trabajar. Por ejemplo una función que sirva para dibujar el cuadrado seguramente tendrá como parámetro el tamaño del cuadrado. Una función que no usa parámetros se suele indicar colocando la palabra **void** en el que normalmente irían los parámetros (hoy en día se puede obviar esta palabra, pero no los paréntesis)
- ◆ Indicar las variables locales a la función
- ◆ Indicar las instrucciones de la función
- ◆ Indicar el valor que devuelve mediante la palabra **return** (salvo que la función sea **void**)

Sintaxis:

```
tipo nombreDeLaFunción(parámetros){  
    definiciones  
    instrucciones  
}
```

La primera línea de la sintaxis anterior es lo que se llama el **prototipo de la función**. Es la definición formal de la función. Desglosamos más su contenido:

- ◆ **tipo**. Sirve para elegir el tipo de datos que devuelve la función. Toda función puede obtener un resultado. Eso se realiza mediante la instrucción **return**. El tipo puede ser: **int, char, long, float, double,....**

y también **void**. Éste último se utiliza si la función no devuelve ningún valor (a estas funciones se las suele llamar **procedimientos**).

- ◆ **nombreDeLaFunción**. El identificador de la función. Debe cumplir las reglas ya comentadas en temas anteriores correspondientes al nombre de los identificadores.
- ◆ **parámetros**. Su uso es opcional, hay funciones sin parámetros (funciones con parámetros **void**). Los parámetros son una serie de valores que la función puede requerir para poder ejecutar su trabajo. En realidad es una lista de variables y los tipos de las mismas. Son variables cuya existencia está ligada a la función

Ejemplo de función:

```
double potencia(int base, int exponente){  
    int contador;  
    int negativo; /* Indica si el exponente es negativo */  
    double resultado=1.0;  
  
    if (exponente<0) {  
        exponente=-exponente;  
        negativo=1;  
    }  
  
    for(contador=1;contador<=exponente;contador++)  
        resultado*=base;  
  
    if (negativo) resultado=1/resultado;  
    return resultado;  
}
```

En los parámetros hay que indicar el tipo de cada parámetro (en el ejemplo se pone **int base, int exponente** y no **int base, exponente**), si no se pone el tipo se suele tomar el tipo **int** para el parámetro (aunque esto puede no ser así en algunos compiladores).

prototipos de las funciones

En realidad fue el lenguaje **C++** el que ideó los prototipos, pero **C** los ha aceptado para verificar mejor el tipo de los datos de las funciones.

Gracias a estos prototipos el compilador reconoce desde el primer momento las funciones que se van a utilizar en el programa. Los prototipos se ponen al principio del código (tras las instrucciones **#include**) delante del **main**.

Ejemplo (programa completo):

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

/*Prototipo de la función*/
double potencia(int base, int exponente);

int main(){
    int b, e;
    do{
        printf("Escriba la base de la potencia (o cero para salir");
        scanf("%d",&b);
        if(b!=0){
            printf("Escriba el exponente");
            scanf("%d",&e);
            printf("Resultado: %lf", potencia(b,e));
        }
    }while(b!=0);
    getch();
}

double potencia(int base, int exponente){
    int contador;
    int negativo=0;
    double resultado=1.0;
    if(exponente<0){
        exponente=-exponente;
        negativo=1;
    }
    for(contador=1;contador<=exponente;contador++)
        resultado*=base;
    if(negativo) resultado=1/resultado;
    return resultado;
}
```

Aunque no es obligatorio el uso de prototipos, sí es muy recomendable ya que permite detectar errores en compilación (por errores en el tipo de datos) que serían muy difíciles de detectar en caso de no especificar el prototipo.

(5.1.4) funciones void

A algunas funciones se les pone como tipo de datos el tipo **void**. Son funciones que no devuelven valores. Sirven para realizar una determinada tarea pero esa tarea no implica que retornen un determinado valor. Es decir son funciones sin instrucción **return**.

A estas funciones también se las llama **procedimientos**. Funcionan igual salvo por la cuestión de que al no retornar valores, no tienen porque tener instrucción **return**. Si se usa esa instrucción dentro de la función, lo único que ocurrirá es

que el flujo del programa abandonará la función (la función finaliza). No es muy recomendable usar así el `return` en funciones de tipo void ya que propicia la adquisición de muy malos hábitos al programar (programación no estructurada).

(5.2) parámetros y variables

(5.2.1) introducción

Ya se ha comentado antes que las funciones pueden utilizar parámetros para almacenar valores necesarios para que la función pueda trabajar.

La cuestión es que los parámetros de la función son variables locales que recogen valores enviados durante la llamada a la función y que esas variables mueren tras finalizar el código de la función. En el ejemplo anterior, `base` y `exponente` son los parámetros de la función `potencia`. Esas variables almacenan los valores utilizados en la llamada a la función (por ejemplo si se llama con `potencia(7,2)`, `base` tomará el valor 7 y `exponente` el valor 2).

(5.2.2) paso de parámetros por valor y por referencia

Se explicará con detalle más adelante. En el ejemplo:

```
int x=3, y=6;  
printf("%lf",potencia(x,y));
```

Se llama a la función `potencia` utilizando como parámetros los valores de `x` e `y`. Estos valores se asocian a los parámetros `base` y `exponente`. La duda está en qué ocurrirá con `x` si en la función se modifica el valor de `base`. La respuesta es, nada.

Cuando se modifica un parámetro dentro de una función no le ocurre nada a la variable que se utilizó para pasar valores a la función. En el ejemplo, `base` tomará una copia de `x`. Por lo que `x` no se verá afectada por el código de la función. Cuando ocurre esta situación se dice que el parámetro se pasa por **valor**.

Sin embargo en la función `scanf` ocurre lo contrario:

```
scanf("%d",&x);
```

La variable `x` sí cambia de valor tras llamar a `scanf`. Esto significa que `x` se pasa por **referencia**. En realidad en C todos los argumentos se pasan por valor, otros lenguajes (como Pascal por ejemplo) admiten distinción. En C para usar el paso por referencia hay que pasar la dirección de la variable (`&x`), o lo que es lo mismo un **puntero** a la variable.

Es decir la llamada `scanf("%d",&x)` envía la dirección de `x` a la función `printf`, esa función utiliza dicha función para almacenar el número entero que el usuario escriba por teclado.

En temas posteriores se profundizará sobre este concepto (en concreto en el tema dedicado a los punteros).

(5.2.3) variables static

Las variables *static* son variables que permanecen siempre en la memoria del ordenador. Su uso fundamental es el de variables locales a una función cuyo valor se desea que permanezca entre llamadas.

Para entender mejor su uso, veamos este ejemplo:

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

void prueba();

int main(){
    int i;
    for(i=1;i<=10;i++) prueba();
    system("pause");
}

void prueba(){
    int var=0;
    var++;
    printf("%d\n",var);
}
```

En el ejemplo, se llama 10 veces seguidas a la función *prueba()*, esta función escribe siempre lo mismo, el número 1. La razón está en que la variable *var* se crea e inicializa de nuevo en cada llamada a la función y muere cuando el código de la función finaliza. Con lo que siempre vale 1.

Si modificamos las declaración e la variable *var* para que ahora sea estática:

```
void prueba(){
    static int var=0;
    var++;
    printf("%d\n",var);
}
```

Ahora el resultado es absolutamente distinto. al ser estática la variable *var* se crea en la primera llamada y permanece en memoria, es decir no se elimina al finalizar la función. En la segunda llamada no se crea de nuevo la variable, es más la instrucción de creación se ignora, ya que la función ya está creada. El resultado ahora es absolutamente distinto. Aparecerán en pantalla los números del 1 al 10.

(5.2.4) ámbito de las variables y las funciones

Ya se comentó en temas precedentes. Pero se vuelve a comentar ahora debido a que en las funciones es donde mejor se aplica este concepto.

Toda variable tiene un ciclo de vida que hace que la variable sea declarada, inicializada, utilizada las veces necesarias y finalmente eliminada. En la declaración además de declarar el tipo, implícitamente estamos declarando el ámbito, de modo que una variable que se declara al inicio se considera global, significa pues que se puede utilizar en cualquier parte del archivo actual.

Si una variable se declara dentro de una función, ésta muere en cuanto la función termina. Lo mismo ocurre con los parámetros, en realidad los parámetros son variables locales que copian el valor de los valores enviados en la llamada a la función, pero su comportamiento es el mismo que el de las variables locales.

Las variables **static** son variables locales que mantienen su valor entre las llamadas, es decir no se eliminan pero sólo se pueden utilizar en la función en la que se crearon.

El ámbito más pequeño lo cubren las variables declaradas dentro de un bloque ({}) de una función. Estas variables sólo se pueden utilizar dentro de las llaves en las que se definió la variables, tras la llave de cierre del bloque {}, la variable muere.

Es importante tener en cuenta que **todo lo comentado anteriormente es aplicable a las funciones**. Normalmente una función es global, es decir su prototipo se coloca al principio del archivo y eso hace que la función se pueda utilizar en cualquier parte del archivo, pero lo cierto es que **se puede definir una función dentro de otra función**. En ese último caso dicha función sólo se puede utilizar dentro de la función en la que se declara.

(5.3) definición de macro funciones con #define

En el tema anterior hemos conocido la directiva de procesador **#define**, que permite declarar constantes, en realidad macros. Se trata de definiciones que son sustituidas cuando se preprocesa el código. Así:

```
#define MAX_COLUMNAS 15
```

Hará que cuando aparezca en el código el texto MAX_COLUMNAS, el preprocesador lo sustituirá por 15.

La realidad es que #define permite macro-sustituciones más elaboradas, ya que permite que se usen parámetros en la definición. Por ejemplo:

```
#define CUADRADO(x) x*x
```

Cuando el preprocesador se encuentra con esta definición en el código, sustituirá esta función por su forma expandida, por ejemplo:

```
areaTotal=CUADRADO(base);
```

el preprocesador lo convertirá en:

```
areaTotal=base*base;
```

Hay que tener cuidado con el uso prioritario de los operadores, de hecho conviene siempre definir el resultado de la función entre paréntesis, de otro modo puede haber problemas, por ejemplo:

```
#define PI 3.141592
#define CIRCUNFERENCIA(x) 2*x*PI
```

Fallaría en la expresión:

```
longitudCircunf=CIRCUNFERENCIA(h-3);
```

Ya que se expande como:

```
longitudCircunf = 2*h-3*PI;
```

Y eso no calcula lo que pretendemos. Por lo que la definición correcta sería:

```
#define CIRCUNFERENCIA(x) (2*(x)*PI)
```

Y el ejemplo anterior quedaría expandido como:

```
longitudCircunf = (2*(h-3)*PI)
```

La ventaja de usar macros en lugar de funciones es que se evita usar llamadas a funciones (lo que acelera la ejecución del programa). Pero tiene dos desventajas a tener en cuenta:

- ◆ El tamaño del código aumenta. Ya que se expande más código
- ◆ El uso de macros queda fuera del normal planteamiento estructurado de creación de programas. No son realmente funciones, por lo que su uso tiene que ser limitado.

Hay otra desventaja a comentar:

- ◆ Los lenguajes modernos (como Java) no utilizan macros. Por lo que si nos acostumbramos en exceso a su uso, adquiriremos hábitos que no son compatibles con esos lenguajes.

(5.4) recursividad

(5.4.1) introducción

La recursividad es una técnica de creación de programas, pensada para soluciones a problemas complejos. La idea es que las funciones se pueden llamar o invocar a sí mismas.

Esta técnica es peligrosa ya que se pueden generar fácilmente llamadas infinitas (la función se llama a sí misma, tras la llamada se vuelve a llamar a sí misma,...). Hay que ser muy cauteloso con ella (incluso evitarla si no es necesario su uso); pero permite soluciones muy originales y abre la posibilidad de solucionar problemas muy complejos. De hecho ciertos problemas (como el de las torres de Hanoi, por ejemplo) serían casi imposibles de resolver sin esta técnica.

La idea es que la función resuelva parte del problema y se llame a sí misma para resolver la parte que queda, y así sucesivamente. En cada llamada el problema debe ser cada vez más sencillo hasta llegar a una llamada en la que la función devuelve un único valor. Tras esa llamada los resultados se devuelven en cadena hasta llegar al código que realizó la primera llamada y pasarse la solución.

Es fundamental tener en cuenta cuándo la función debe dejar de llamarse a sí misma en algún momento, es decir es importante decidir cuándo acabar. De otra forma se corre el riesgo de generar infinitas llamadas, lo que bloquearía de forma grave el PC en el que trabajamos. Un bucle infinito utilizando recursividad causa graves inestabilidades en el ordenador de trabajo--.

Como ejemplo vamos a ver la versión recursiva del factorial.

```
double factorial(int n){  
    if(n<=1) return 1;  
    else return n*factorial(n-1);  
}
```

La última instrucción (`return n*factorial(n-1)`) es la que realmente aplica la recursividad. La idea (por otro lado más humana) es considerar que el factorial de nueve es nueve multiplicado por el factorial de ocho; a su vez el de ocho es ocho por el factorial de siete y así sucesivamente hasta llegar al uno, que devuelve uno.

Para la instrucción `factorial(4)`; usando el ejemplo anterior, la ejecución del programa generaría los siguientes pasos:

- (1) Se llama a la función factorial usando como parámetro el número 4 que será copiado en la variable-parámetro `n`
- (2) Como `n>1`, entonces se devuelve 4 multiplicado por el resultado de la llamada `factorial(3)`
- (3) La llamada anterior hace que el nuevo `n` (variable distinta de la anterior) valga 3, por lo que esta llamada devolverá 3 multiplicado por el resultado de la llamada `factorial(2)`

- (4) La llamada anterior devuelve 2 multiplicado por el resultado de la llamada **factorial(1)**
- (5) Esa llamada devuelve 1
- (6) Eso hace que la llamada **factorial(2)** devuelva $2 * 1$, es decir 2
- (7) Eso hace que la llamada **factorial(3)** devuelva $3 * 2$, es decir 6
- (8) Por lo que la llamada **factorial(4)** devuelve $4 * 6$, es decir **24** Y ese es ya el resultado final

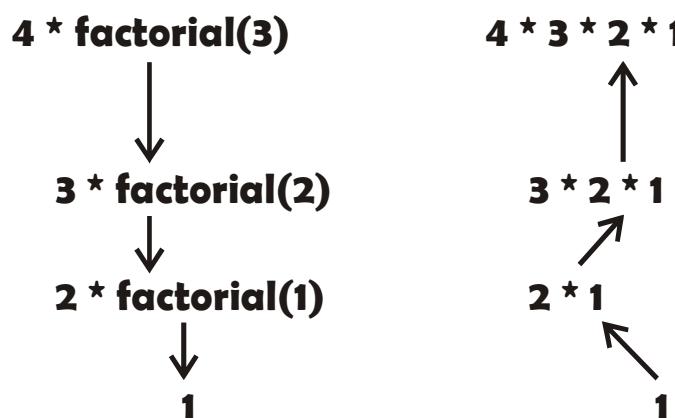


Ilustración 1, Pasos para realizar la recursividad

(5.4.2) ¿recursividad o iteración?

Hay otra versión de la función factorial resuelta mediante un bucle **for** (solución iterativa) en lugar de utilizar la recursividad. La cuestión es ¿cuál es mejor?

Ambas implican sentencias repetitivas hasta llegar a una determinada condición. Por lo que ambas pueden generar programas que no finalizan si la condición nunca se cumple. En el caso de la iteración es un contador o un centinela el que permite determinar el final, la recursividad lo que hace es ir simplificando el problema hasta generar una llamada a la función que devuelva un único valor.

Para un ordenador es más costosa la recursividad ya que implica realizar muchas llamadas a funciones en cada cual se genera una copia del código de la misma, lo que sobrecarga la memoria del ordenador. Es decir, **es más rápida y menos voluminosa la solución iterativa de un problema recursivo**.

¿Por qué elegir recursividad? De hecho si poseemos la solución iterativa, no deberíamos utilizar la recursividad. **La recursividad se utiliza sólo si:**

- ◆ No encontramos la solución iterativa a un problema
- ◆ El código es mucho más claro en su versión recursiva

(5.4.3) recursividad cruzada

Hay que tener en cuenta la facilidad con la que la recursividad genera bucles infinitos. Por ello una función nunca ha de llamarse a sí misma si no estamos empleando la recursividad. Pero a veces estos problemas de recursividad no son tan obvios. Este código también es infinito en su ejecución:

```
int a(){
    int x=1;
    x*=b();
    return x;
}

int b(){
    int y=19;
    y-=a();
    return y;
}
```

Cualquier llamada a la función *a* o a la función *b* generaría código infinito ya que ambas se llaman entre sí sin parar jamás. A eso también se le llama recursividad, pero recursividad cruzada.

(5.5) bibliotecas

(5.5.1) introducción

En cualquier lenguaje de programación se pueden utilizar bibliotecas (también llamadas **librerías**) para facilitar la generación de programas. Estas bibliotecas en realidad son una serie de funciones organizadas que permiten realizar todo tipo de tareas.

Cualquier entorno de programación en C permite el uso de las llamadas bibliotecas estándar, que son las que incorpora el propio lenguaje de programación.

Hay varias librerías estándar en C. Para poder utilizarlas se debe incluir su archivo de cabecera en nuestro código, avisando de esa forma al compilador de que se debe enlazar el código de dicha librería al nuestro para que el programa funcione.

Eso se realizar mediante la directiva de procesador **#include** seguida del nombre del archivo de cabecera de la librería. Ese archivo suele tener extensión **.h** y contiene las declaraciones de las funciones de dicha librería.

(5.5.2) ejemplo de librería estándar. math

Math es el nombre de una librería que incorpora funciones matemáticas. Para poder utilizar dichas funciones hay que añadir la instrucción

```
#include <math.h>
```

al principio del código de nuestro archivo. Cada directiva **include** se coloca en una línea separada y no requiere punto y coma al ser una indicación al compilador y no una instrucción de verdad.

Esta librería contiene las siguientes funciones (se indican los prototipos y uso):

| Prototipo | Descripción |
|--|--|
| int abs(int n) | Devuelve el valor absoluto del número entero indicado |
| int ceil(double n) | Redondea el número decimal <i>n</i> obteniendo el menor entero mayor o igual que <i>n</i> . Es decir <i>ceil(2.3)</i> devuelve 3 y <i>ceil(2.7)</i> también devuelve 3 |
| double cos(double n) | Obtiene el seno de <i>n</i> (<i>n</i> se expresa en radianes) |
| double exp(double n) | Obtiene <i>e</i> ^{<i>n</i>} |
| double fabs(double n) | Obtiene el valor absoluto de <i>n</i> (siendo <i>n</i> un número <i>double</i>) |
| int floor(double n) | Redondea el número decimal <i>n</i> obteniendo el mayor entero menor o igual que <i>n</i> . Es decir <i>floor(2.3)</i> devuelve 2 y <i>floor(2.7)</i> también devuelve 2 |
| double fmod(double x, double y) | Obtiene el resto de la división entera de <i>x/y</i> |
| double log(double n) | Devuelve el logaritmo neperiano de <i>n</i> |
| double log10(double n) | Devuelve el logaritmo decimal de <i>n</i> |
| double pow(double base, double exponente) | Obtiene <i>base</i> ^{<i>exponente</i>} |
| double sin(double n) | Obtiene el seno de <i>n</i> (<i>n</i> se expresa en radianes) |
| double sqrt(double n) | Obtiene la raíz cuadrada de <i>n</i> |
| double tan(double n) | Obtiene la tangente de <i>n</i> (<i>n</i> se expresa en radianes) |

(5.5.3) números aleatorios

Otras de las funciones estándar más usadas son las de manejo de números aleatorios. Están incluidas en **stdlib.h**.

La función **rand()** genera números aleatorios entre 0 y una constante llamada **RAND_MAX** que está definida en la propia stdlib.h (normalmente vale 32767). Así el código:

```
for(i=1;i<=100;i++)  
    printf("%d\n",rand());
```

Genera 100 números aleatorios entre 0 y RAND_MAX. Para controlar los números que salen se usa el operador del módulo (%) que permite obtener en este caso números aleatorios del en un rango más concreto.

Por ejemplo:

```
for(i=1;i<=100;i++)
    printf("%d\n",rand()%10);
```

Obtiene 100 números aleatorios del 0 al 9. Pero hay un problema, los números son realmente **pseudoaleatorios**, es decir la sucesión que se obtiene siempre es la misma (se basa en una fórmula matemática). Aunque los números circulan del 0 al 9 con apariencia de falta de orden o sentido, lo cierto es que la serie es la misma siempre.

La razón de este problema reside en lo que se conoce como semilla. La semilla es el número inicial desde el que parte el programa. Es un primer número que se pasa a la función de generación de números. Si pasamos siempre la misma semilla, la sucesión de números generados será la misma.

La semilla se puede generar mediante la función **srand**. La función srand recibe un número entero que será la semilla en la que se inicie la sucesión. Si usamos semillas distintas en cada ejecución, tendremos sucesiones de números distintas.

Para que sea impredecible conocer la semilla se usa la expresión:

```
srand(time(NULL));
```

Esa expresión toma un número entero a partir de la fecha y hora actual, por lo que dicho número es impredecible (varía en cada centésima de segundo). Así realmente el código:

```
srand(time(NULL));
for(i=1;i<=100;i++)
    printf("%d\n",rand()%10);
```

genera cien números del 0 al 9 que variarán en cada ejecución del programa, siendo considerados números aleatorios debido a que la semilla usa el reloj del sistema (cuyo valor no se puede controlar).

(5.5.4) crear nuestras propias librerías

Bajo el paradigma de reutilizar el código, normalmente las funciones interesantes que crean los usuarios, es conveniente agruparlas en archivos que luego se pueden reutilizar.

Para ello se hacen los siguientes pasos:

- (1)** Se crea en un archivo (generalmente con extensión .h) las funciones. En esos archivos se agrupan las funciones según temas. Esos archivos **no tendrán ninguna función main**, sólo se incluyen las funciones de nuestra librería
- (2)** Se crea el archivo principal con su función **main** correspondiente.
- (3)** En dicho archivo se utiliza la instrucción **include** seguida de la ruta al archivo que contiene alguna de las funciones que deseamos usar. Esa ruta debe ir entre comillas dobles (de otra forma se busca el archivo en

la carpeta de librerías estándar). Cada archivo de funciones requiere una instrucción **include** distinta.

- (4) Cada vez que un archivo requiera usar funciones de una librería habrá que incluir dicho código. Eso puede provocar que se incluya más de una vez el código de las funciones, lo que provocaría errores.

Debido a este último punto es conveniente que un archivo que es susceptible de ser incluido más de una vez utilice la directiva **#ifndef**. Se trata de una directiva que hace que él código se compile si existe la constante a la que hace referencia. El uso es:

```
#ifndef constante
    código que se ejecuta si la constante no ha sido definida
#endif
```

El código tras la línea **#ifndef**, sólo se compila si la constante no ha sido definida con un **#define**. Por ello en un archivo que corre el riesgo de ser incluido más de una vez (lo que podemos entender que le podría ocurrir a cualquier archivo de librería), se pone el siguiente código (supongamos que se trata del archivo matematicas2.h):

```
#ifndef MATEMATICAS2_H
    ...
    #define MATEMATICAS2_H
#endif
```

Cuando se incluya este código por primera vez, la constante MATEMATICAS2_H no estará definida, por lo que se incluirá el código del archivo. La línea del **#define** se traducirá y por lo tanto se definirá la constante. Si otro archivo vuelve a incluir a éste, el código no se compilará (por lo tanto se evita el error) ya que ahora sí aparece como definida MATEMATICAS2_H.

(5.6) Índice de ilustraciones

| | |
|--|----|
| Ilustración 1, Pasos para realizar la recursividad | 15 |
|--|----|