[Unidad 1] algoritmos y programas

[1.1] computadora y sistema operativo

[1.1.1] 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*.

Sin duda esta máquina es la responsable de toda una revolución que está cambiando el panorama económico, social e incluso cultural. Debido a la importancia y al difícil manejo de estas máquinas, aparece la **informática** como la ciencia orientada al proceso de información mediante el uso de computadoras.

Una computadora consta de diversos componentes entre los que sobresale el procesador, el componente que es capaz de realizar las tareas que se requieren al ordenador o computadora.

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**

Algunos de los componentes destacables de un ordenador son:

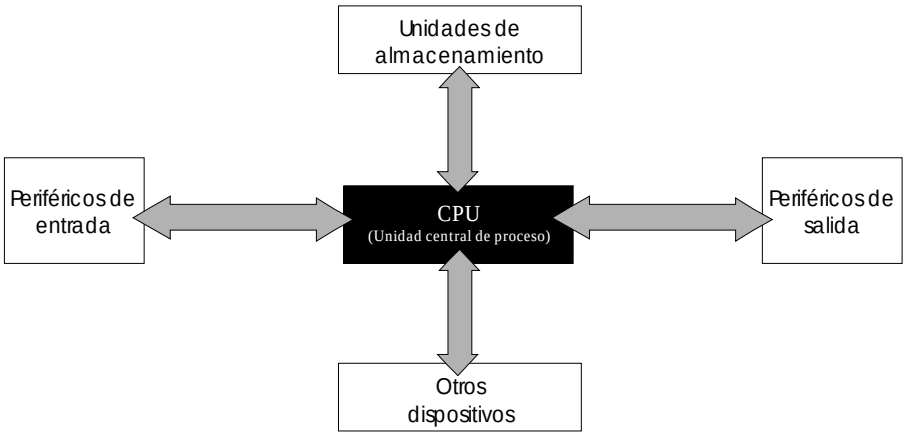


Ilustración 1, componentes de un ordenador desde un punto de vista lógico

Este desglose de los componentes del ordenador es el que interesa a los programadores. Pero desde un punto de vista más físico, hay otros componentes a señalar:

- **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 interna formada por un circuito digital que está conectado mediante tarjetas a la placa base. Su contenido se evapora cuando se desconecta al ordenador. Lo que se almacena no es permanente.
- **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.
- **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.**

[1.1.2] hardware y software

hardware

Se trata de todos los componentes físicos que forman parte de un ordenador: procesador, RAM, impresora, teclado, ratón,...

software

Se trata de la parte conceptual del ordenador. Es decir los datos y aplicaciones que maneja y que permiten un grado de abstracción mayor. Cualquier cosa que se pueda almacenar en una unidad de almacenamiento es software (la propia unidad sería hardware).

[1.1.3] 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.

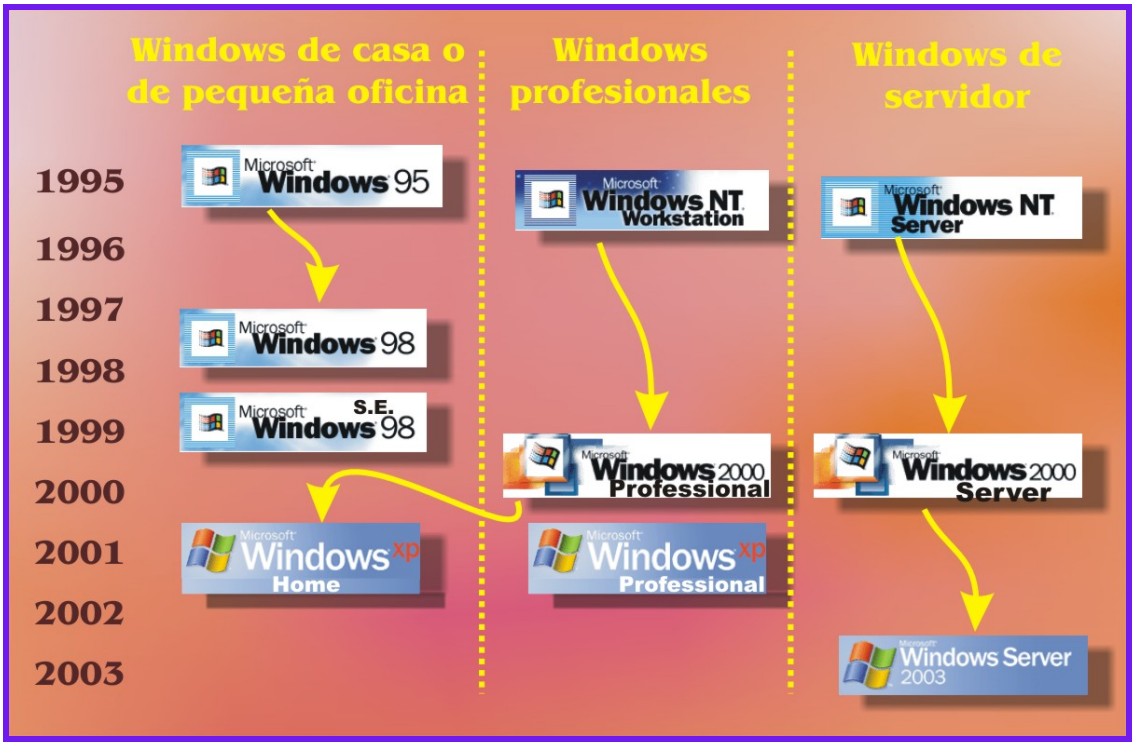


Ilustración 2, Versiones actuales de Windows

- **Linux.** Sistema operativo de código abierto. Posee numerosas distribuciones (muchas de ellas gratuitas) y software adaptado para él (aunque sólo el 15% de ordenadores posee Linux).
- **MacOs.** Sistema operativo de los ordenadores MacIntosh.
- **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).
- **Solaris.** Versión de Unix para sistemas Sun.

[1.2] codificación de la información

[1.2.1] introducción

Sin duda una de las informaciones que más a menudo un ordenador tiene que manipular son los números. Pero también el ordenador necesita codificar otro tipo de información, como por ejemplo caracteres, imágenes, sonidos,...

EL problema es que para el ordenador toda la información debe estar en formato binario (de unos y ceros). Por ello se necesita pasar todos los datos a ese formato.

[1.2.2] sistemas numéricos

A lo largo de la historia han existido numerosos sistemas de numeración. Para simplificar los dividiremos 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.

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. Lo cual se hace con la fórmula:

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

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 e sistema octal (base ocho), el paso a decimal se haría:

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

[1.2.3] sistema binario

introducción

Los números binarios son los que utilizan las computadoras para almacenar información. Debido a ello hay términos informáticos que se refieren al sistema binario y que se utilizan continuamente. Son:

- **BIT (de *Binary diGIT*)**. Se trata de un dígito binario, el número binario 1001 tiene cuatro BITS.
- **Byte**. Es el conjunto de 8 BITS.
- **Kilobyte**. Son 1024 bytes.

- **Megabyte**. Son 1024 Kilobytes.
- **Gigabyte**. Son 1024 Megabytes.
- **Terabyte**. Son 1024 Gigabytes.
- **Petabyte**. Son 1024 Terabytes.

conversión binario a decimal

Utilizando el teorema fundamental de la numeración, por ejemplo para el número binario 10011011011 el paso sería (los ceros se han ignorado):

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

conversión decimal a binario

El método más utilizado es ir haciendo divisiones sucesivas entre dos. Los restos son las cifras binarias. Por ejemplo para pasar el 39:

```
39:2 = 19 resto 1
19:2 = 9 resto 1
9:2 = 4 resto 1
4:2 = 2 resto 0
2:2 = 1 resto 0
1:2 = 0 resto 1
```

El número 111001 es el equivalente en binario de 39.

operaciones aritméticas binarias

suma

Se efectúa igual que las sumas decimales, sólo que cuando se suma un uno y otro uno, ese dice que tenemos un acarreo de uno y se suma a la siguiente cifra. Ejemplo (suma de 31, en binario 10011, y 28, en binario, 11100)

Acarreo	**1**	**1**			
	1	1	1	1	1
	1	1	1	0	0
1	**1**	**1**	**0**	**1**	**1**

El resultado es 111011, 59 en decimal.

resta

El concepto es parecido sólo que en el caso de la resta es importante tener en cuenta el signo. No se explica en el presente manual ya que se pretende sólo una introducción a

los números binarios. En la actualidad la resta se hace sumando números en **complemento a 2**[1].

operaciones lógicas

Se trata de operaciones que manipulan BITS de forma lógica, son muy utilizadas en la informática. Se basan en una interpretación muy utilizada con los números binarios en la cual el dígito **1** se interpreta como **verdadero** y el dígito **0** se interpreta como falso.

operación AND

La operación AND (en español *Y*), sirve para unir expresiones lógicas, se entiende que el resultado de la operación es verdadero si alguna de las dos expresiones es verdadero (por ejemplo la expresión *ahora llueve y hace sol* sólo es verdadera si ocurren ambas cosas).

En el caso de los dígitos binarios, la operación AND opera con dos BITS de modo que el resultado será uno si ambos bits valen uno.

AND	0	1
0	0	0
1	0	1

La tabla superior se llama **tabla de la verdad** y sirve para mostrar resultados de operaciones lógicas, el resultado está en la parte blanca, en la otra parte se representan los operadores . El resultado será 1 si ambos operadores valen 1

operación OR

OR (*O* en español) devuelve verdadero si cualquiera de los operandos es verdadero (es decir, si valen 1). La tabla es esta:

OR	0	1
0	0	1
1	1	1

operación NOT

Esta operación actúa sobre un solo BIT y lo que hace es invertirle; es decir, si vale uno valdrá cero, y si vale cero valdrá uno.

NOT	0	1
	1	0

[1] Se trata de una forma avanzada de codificar números que utiliza el primer BIT como signo y utiliza el resto de forma normal para los números positivos y cambiando los unos por los ceros para los números negativos.

codificación de otros tipos de datos a binario

texto

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).

Una ampliación de este método 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. Poco a poco es el código que se va extendiendo.

otros datos

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 es la que graba cada **píxel** (cada punto distinguible 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.

[1.2.4] sistema hexadecimal

Es un sistema que se utiliza mucho para representar números binarios. Un problema (entre otros) de los números binarios es que ocupan mucho espacio para representar información. El sistema hexadecimal es la forma de representar números en base 16. de modo que en los dígitos del 0 al 9 se utilizan los mismos símbolos que en el sistema decimal y a partir del 10 se utiliza la letra A y así hasta la letra F que simboliza el 15.

Así el número hexadecimal CA3 sería:

$$C \cdot 16^2 + A \cdot 16^1 + 3 \cdot 16^0 = 12 \cdot 256 + 10 \cdot 16 + 3 = 3235$$

Como se observa pasar de hexadecimal a decimal es complejo. La razón del uso de este sistema es porque tiene una equivalencia directa con el sistema binario. De hecho en una cifra hexadecimal caben exactamente 4 bits. Por ello la traducción de hexadecimal a binario se basa en esta tabla:

Hexadecimal	Binario
0	0000
1	0001

Hexadecimal	Binario
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Así el número hexadecimal C3D4 sería el binario 1100 0011 1101 0100. Y el binario 0111 1011 1100 0011 sería el hexadecimal 7BC3

[1.3] algoritmos

[1.3.1] 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 efectuarse. 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:

[1]Comprobar si hay bombillas de repuesto

[1.1] Abrir el cajón de las bombillas

[1.2] Observar si hay bombillas

[2]Si hay bombillas:

[2.1] Coger la bombilla

[2.2] Coger una silla

[2.3] Subirse a la silla

[2.4] Poner la bombilla en la lámpara

[3]Si no hay bombillas

[3.1] Abrir la puerta

[3.2] 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.

[1.3.2] 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.

[1.3.3] 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.

[1.3.4] fases en la creación de algoritmos

Hay tres fases en la elaboración de un algoritmo:

[1] **Análisis.** En esta se determina cuál es exactamente el problema a resolver. Qué datos forman la entrada del algoritmo y cuáles deberán obtenerse como salida.

[2] **Diseño**. Elaboración del algoritmo.

[3] **Prueba**. Comprobación del resultado. Se observa si el algoritmo obtiene la salida esperada para todas las entradas.

[1.4] aplicaciones

[1.4.1] 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.

[1.4.2] 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 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.

[1.4.3] el 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:

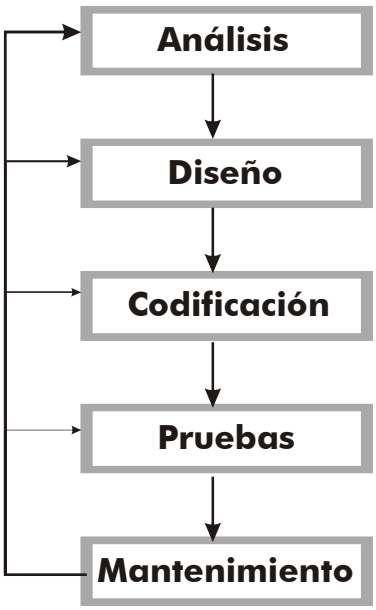


Ilustración 3, Ciclo de vida de una aplicación

[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 le realiza un analista

[4] **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.

[5] **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

[6] **Pruebas.** Se trata de comprobar que el funcionamiento de la aplicación es la adecuada. Se realiza en varias fases:

- [1.1] **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.
- [1.2] **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.
- [1.3] **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 está versión acaba siendo la definitiva (como ocurre con muchos de los programas distribuidos libremente por Internet).

[7] **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.

[1.5] 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 inesperado 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.

[1.6] programación

[1.6.1] 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 a pasado por diversos pasos.

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

[1.6.2] 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.

[1.6.3] 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.**

[1.6.4] 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.

[1.6.5] 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.

[1.6.6] lenguajes de programación

introducción

Los lenguajes de programación permiten codificar algoritmos en el ordenador. Son una serie de símbolos e instrucciones de acuerdo con una sintaxis que facilita su posterior traducción a código entendible por el ordenador,

En realidad los ordenadores sólo reconocen un lenguaje formado por los unos y ceros del código binario. Según la disposición de esos unos y ceros entenderá unas instrucciones u otras. De esa forma hay lenguajes más próximos al lenguaje de las computadores (**lenguajes de bajo nivel**) y lenguajes más próximos al lenguaje humando (**lenguajes de alto nivel**)

código máquina (lenguaje de primera generación)

Está formado por una sucesión de unos y ceros que el procesador del ordenador reconoce como instrucciones. Es el lenguaje que reconoce directamente el ordenador por lo tanto es el que está a más bajo nivel.

Sólo se ha utilizado por los programadores en los inicios de la informática. Su incomodidad de trabajo hace que sea impensable para ser utilizado. 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 cuente 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.

ensamblador (lenguajes de segunda generación)

Se trata de un lenguaje que representa el código máquina pero escrito con una serie de términos mnemotécnicos que facilitan su escritura. Después un software especial se encargará de traducir las instrucciones a código máquina. Ejemplo[2] (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
```

[2] Ejemplo tomado de la página http://www.victorsanchez2.net

```
     mov  ax,4C00h
     int  21h
START ENDP
CODE ENDS
     END START
```

Este lenguaje tiene traducción exacta al código máquina, por lo que es un lenguaje diferente para cada procesador; 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 en estos lenguajes. 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)

Se aproximan más al lenguaje de los humanos. Los programas se diseñan en un lenguaje estricto pero independiente de la máquina, lo que permite que la escritura del código cree programas ejecutables en cualquier máquina.

Hace falta software que transforme el código en el lenguaje de alto nivel en código entendible por el ordenador en un proceso conocido como **interpretación** o **compilación** (dependiendo del lenguaje).

El código es menos eficiente que en el caso anterior, pero es más entendible y mucho más fácilmente corregible. Hoy en día casi todos los lenguajes son de alto nivel (C, Basic, Cobol, Fortran, Pascal,...).

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);
            }
        }
}
```

A veces se habla de lenguajes de medio nivel para referirse a lenguajes que utilizan una codificación que está entre el ensamblador y el lenguaje de alto nivel. El C se considera a menudo un lenguaje de nivel medio ya que su codificación puede resultar tan críptica como un código en ensamblador. De hecho las posibilidades del C son similares a las del ensamblador, pero su estructura es la de un lenguaje de alto nivel.

lenguajes de cuarta generación o 4GL (*fourth generation languages)*

Son lenguajes en los que apenas hay código y en su lugar aparecen indicaciones sobre qué es lo que el programa debe de obtener. En estos lenguajes hay herramientas de tipo más visual mediante las que se diseña el funcionamiento del programa.

Los lenguajes de consulta de datos, creación de formularios, informes,... son lenguajes de cuarto nivel. Aparecieron con los sistemas de base de datos.

[Unidad 2] metodología de la programación

[2.1] metodologías

[2.1.1] introducción

Se entiende por metodología el conjunto de reglas y pasos estrictos que se siguen para desarrollar una aplicación informática completa. Hay diversas metodologías, algunas incluso registradas (hay que pagar por utilizarlas).

Independientemente de la metodología utilizada suele haber una serie de pasos comunes a todas ellas (relacionados con el ciclo de vida de la aplicación):

[1] Análisis

[2] Diseño

[3] Codificación

[4] Ejecución

[5] Prueba

[6] Mantenimiento

[2.1.2] 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 especificaciones** en la que aparece los requerimientos de la aplicación. Esta hoja es redactada por el o la analista, la persona responsable del proceso de creación de la aplicación.

En la creación de algoritmos sencillos, el análisis consistiría únicamente en:

- **Determinar las entradas**. Es decir, los datos que posee el algoritmo cuando comienza su ejecución. Esos datos permiten obtener el resultado.
- **Determinar las salidas**. Es decir, los datos que obtiene el algoritmo como resultado. Lo que el algoritmo devuelve al usuario.
- **Determinar el proceso**. Se estudia cuál es el proceso que hay que realizar.

[2.1.3] diseño

En esta fase se crean esquemas que simbolizan a la aplicación. Estos esquemas los elaboran analistas. Gracias a estos esquemas se simboliza 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.

La creación de estos esquemas se puede hacer en papel, o utilizar una **herramienta CASE** para hacerlo.

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.

Siempre existe en el diseño la **zona principal** que es el programa principal que se ejecutará cuando el programa esté codificado en un lenguaje de programación.

En la construcción de aplicaciones complejas en esta fase se utilizan gran cantidad de esquemas para describir la organización de los datos y los procedimientos que ha de seguir el programa. En pequeños algoritmos se utilizan esquemas más sencillos.

[2.1.4] codificación

Escritura de la aplicación utilizando un lenguaje de programación (C, Pascal, C++, Java,...). Normalmente la herramienta utilizada en el diseño debe ser compatible con el lenguaje que se utilizará para codificar. Es decir si se utiliza un lenguaje orientado a objetos, la herramienta de diseño debe ser una herramienta que permita utilizar objetos.

[2.1.5] ejecución

Tras la escritura del código, mediante un software especial se traduce a código interpretable por el ordenador (código máquina). En este proceso pueden detectarse errores en el código que impiden su transformación. En ese caso el software encargado de la traducción (normalmente un **compilador** o un **intérprete**) avisa de esos errores para que el programador los pueda corregir.

[2.1.6] prueba

Se trata de testear la aplicación para verificar que su funcionamiento es el correcto. Para ello se comprueban todas las entradas posibles, comprobando que las salidas son las correspondientes.

[2.1.7] mantenimiento

En esta fase se crea la documentación del programa (paso fundamental en la creación de aplicaciones). Gracias a esa documentación se pueden corregir futuros errores o renovar el programa para incluir mejoras detectadas, operaciones que también se realizan en esta fase.

[2.2] notaciones para el diseño de algoritmos

[2.2.1] diagramas de flujo

introducción

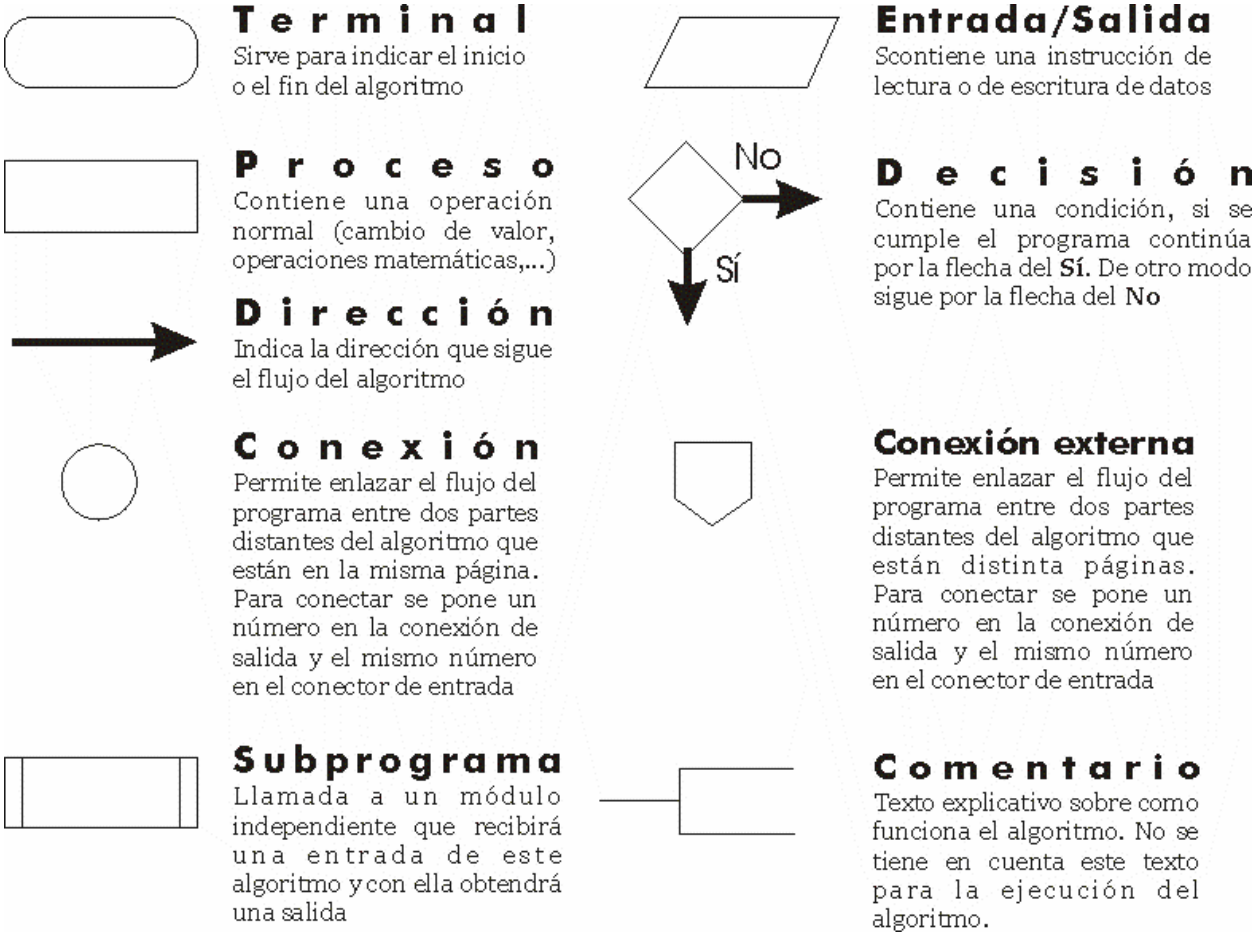
Es el esquema más viejo de la informática. Se trata de una notación que pretende facilitar la escritura o la comprensión de algoritmos. Gracias a ella se esquematiza el flujo del algoritmo. Fue muy útil al principio y todavía se usa como apoyo para explicar ciertos algoritmos. Si los algoritmos son complejos, este tipo de esquemas no son adecuados.

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. Pero sigue siendo interesante en el aprendizaje de la creación de algoritmos.

Los diagramas utilizan símbolos especiales que ya 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:



Ejemplo:

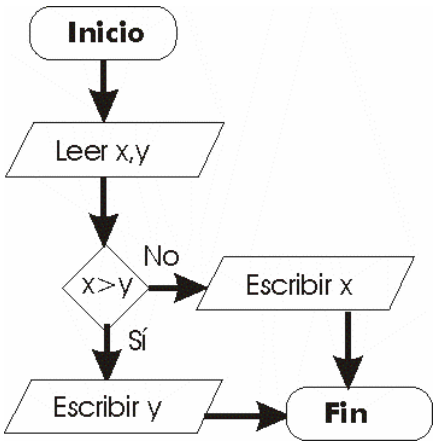


Ilustración 4, Diagrama de flujo que escribe el mayor de dos números leídos

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.

[2.2.2] pseudocódigo

introducción

Las bases de la programación estructurada fueron enunciadas por **Niklaus Wirdth**. 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 una programación que insta a todo programador a utilizar sólo instrucciones de esos tres tipos. Es lo que se conoce como programación estructurada.

El propio Niklaus Wirdth diseñó el lenguaje **Pascal** como el primer lenguaje estructurado. Lo malo es que el Pascal al ser lenguaje completo incluye instrucciones excesivamente orientadas al 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 inglés).

El pseudocódigo además permite el diseño modular de programas y el diseño descendente 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 que escriben pseudocódigos. Aquí se comenta el pseudocódigo que parece más aceptado en España. Hay que tener en cuenta que el pseudocódigo se basa en Pascal, por lo que la traducción a Pascal es casi directa.

El pseudocódigo son instrucciones escritas en un lenguaje orientado a ser entendido por un ordenador. 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 hecho escribir 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 **). Cada línea de comentario debe comenzar con esos símbolos:

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

[2.3] creación de algoritmos

[2.3.1] 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.

[2.3.2] instrucciones de declaración

Sólo se utilizan en el pseudocódigo. 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 las 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.

[2.3.3] 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↑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. Sí 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)

[2.3.4] instrucciones de entrada y salida

lectura de datos

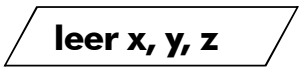
Es la instrucción que simula una lectura de datos desde el teclado. Se hace mediante la orden **leer** en la que entre paréntesis se indica el identificador de la variable que almacenará lo que se lea. Ejemplo (pseudocódigo):

```
leer(x)
```

El mismo ejemplo en un diagrama de flujo sería:

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

```
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.

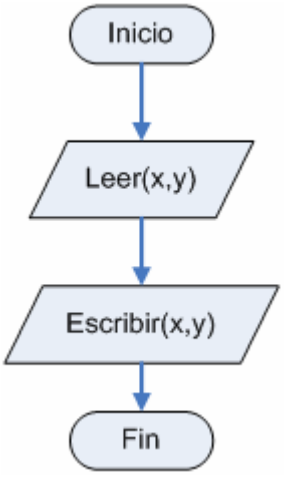
```
escribir(x,y,z)
```

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:



[2.3.5] 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:

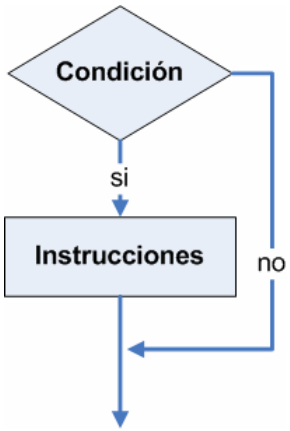
Expresión	Resultado verdadero si...
a>8 **Y** b<12	La variable *a* es mayor que 8 y (a la vez) la variable *b* es menor que 12
a>8 **Y** b<12	O la variable *a* es mayor que 8 o la variable *b* es menor que 12. Basta que se cumpla una de las dos expresiones.
a>30 **Y** a<45	La variable *a* está entre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



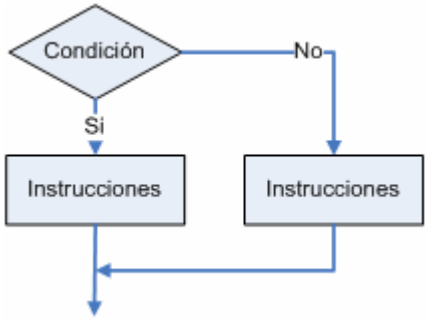
Las instrucciones sólo se ejecutarán si la expresión evaluada es verdadera.

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:



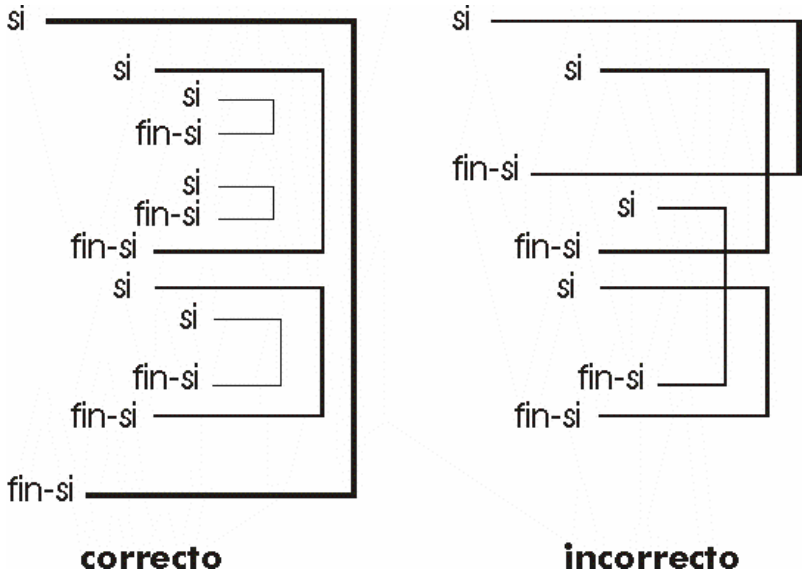
Hay que tener en cuenta que se puede meter una instrucción si dentro de otro si. A eso se le llama alternativas anidadas.

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:



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")
   lee(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:

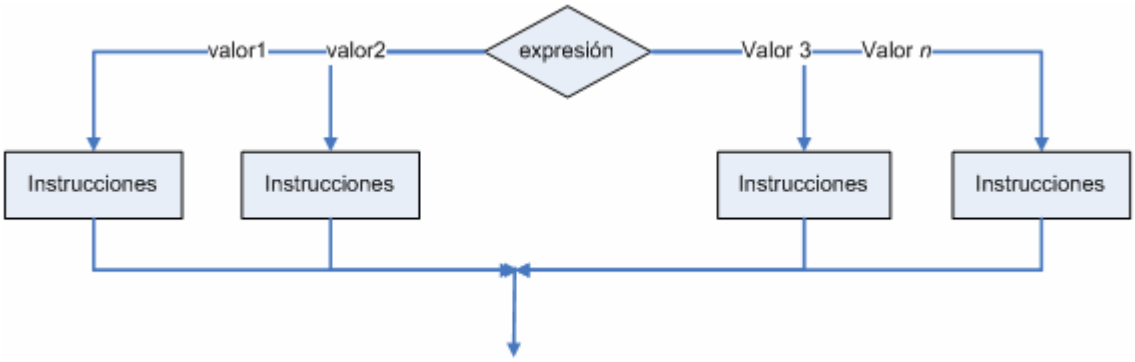
```
   según_sea x hacer
         1,3:
               escribe("impar")
         2,4:
               escribe("par")
         si_no
               escribe("error eso no es un número de 1 a 4")
   fin_según
```

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:

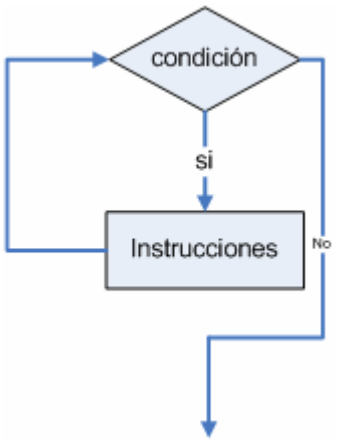


instrucciones iterativas de tipo *mientras*

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:



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:

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:

Ejemplo (escribir números del 1 al 10):

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

instrucciones iterativas *para*

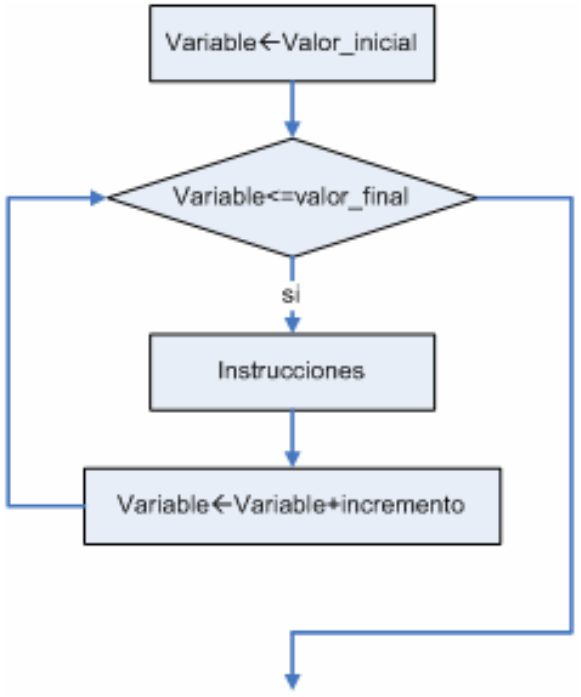
Existe otro tipo de estructura iterativa. En realidad no sería necesaria ya que lo que hace esta instrucción lo puede hacer una instrucción *mientras*, pero facilita el uso de bucles con contador. Es decir son instrucciones que se repiten continuamente según los valores de un contador al que se le pone 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
```

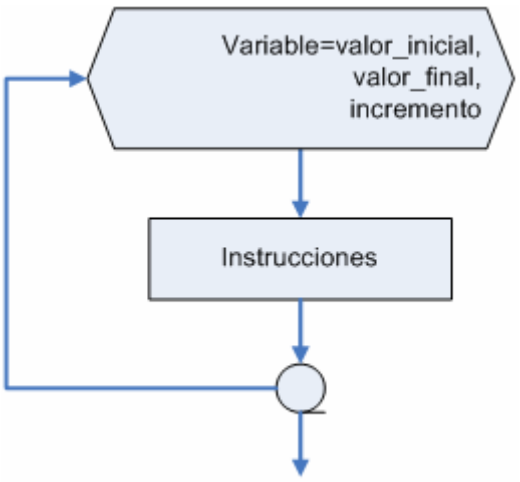
Si se usa el incremento sería:

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

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



También se puede utilizar este formato de diagrama:



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*).

[2.4] otros tipos de diagramas

La programación estructurada hoy en día no se suele representar en ningún otro tipo de diagrama. Pero para expresar determinadas situaciones a tener en cuenta o cuando se utilizan otras programaciones se utilizan otros diagramas

[2.4.1] diagramas entidad/relación

Se utilizan muchísimo para representar datos que se almacenan en una base de datos. Son imprescindibles para el diseño de las bases de datos que es una de las aplicaciones más importantes de la informática.

[2.4.2] diagramas modulares

Representan los módulos que utiliza un programa y la relación que hay entre ellos. Suelen utilizarse conjuntamente con los diagramas descritos anteriormente.

[2.4.3] diagramas de estados

Representan los estados por los que pasa una aplicación, son muy importantes para planificar la aplicación.

[2.4.4] diagramas de secuencia

Indica tiempo transcurrido en la ejecución de la aplicación. Son muy útiles para diseñar aplicaciones donde el control del tiempo es crítico.

[2.4.5] diagramas de clases

Representan los elementos que componen una aplicación orientada a objetos y la relación que poseen éstos. Hoy en día el más popular es el diagrama de clases de la notación UML.

[2.4.6] UML

UML es el nombre que recibe el lenguaje de modelado universal. UML es un estándar en la construcción de diagramas para la creación de aplicaciones orientadas a objetos. Utiliza diversos tipos de esquemas para representar aspectos a tener en cuenta en la aplicación (organización de los datos, flujo de las instrucciones, estados de la aplicación, almacenamiento del código, etc.). Este tipo de diagramas serán explicados en temas posteriores.

[Unidad 3]
el lenguaje C++

[3.1]
lenguajes de programación

[3.1.1] introducción

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.

primeros lenguajes

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 rellenar 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.

En los años 40 se intentó concebir un lenguaje más simbólico. 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 mnemónica (como SUM para sumar por ejemplo), de forma que esas palabras tienen traducción directa en el código máquina. Después habrá que traducir el código ensamblador a código máquina, tarea que la realiza un software especial llamado también **ensamblador.**

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. La cuestión es que este código (todavía de bajo nivel) se escribirá en un editor de texto y después un software especial (que también se le llama ensamblador) lo traducirá al código máquina equivalente. La ventaja es que la traducción es literal, tenemos toda la potencia de la máquina ya que podemos utilizar cualquier instrucción de la misma. La desventaja es que tenemos que conocer muy bien la máquina y que el código sólo vale para máquinas totalmente compatibles

El problema es que como el ensamblador es traducción absoluta del código máquina, sólo vale para máquinas compatibles. No vale para cualquiera. Además de lo pesado y lento que es programar en ensamblador.

lenguajes de alto nivel

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** (*FORmula TRANslation)*, lenguaje orientado a resolver fórmulas matemáticos. 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. En 1960 la conferencia **CODASYL** se creo el **COBOL** como lenguaje de gestión en 1960. En 1963 se creo **PL/I** el primer lenguaje que admitía la multitarea y la programación modular.

BASIC se creo 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 creo con la misma idea académica pero siendo ejemplo de lenguaje estructurado para programadores avanzados. El creador del Pascal (**Niklaus Wirdth**) creo **Modula** en 1977 siendo un lenguaje estructurado para la programación de sistemas (intentando sustituir al **C**).

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++,...**

En 1995 aparece Java como lenguaje totalmente orientado a objetos y en el año 200 aparece C# un lenguaje que procede de **C++** y del propio Java.

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. Todos ellos son de finales de los años 90.

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.

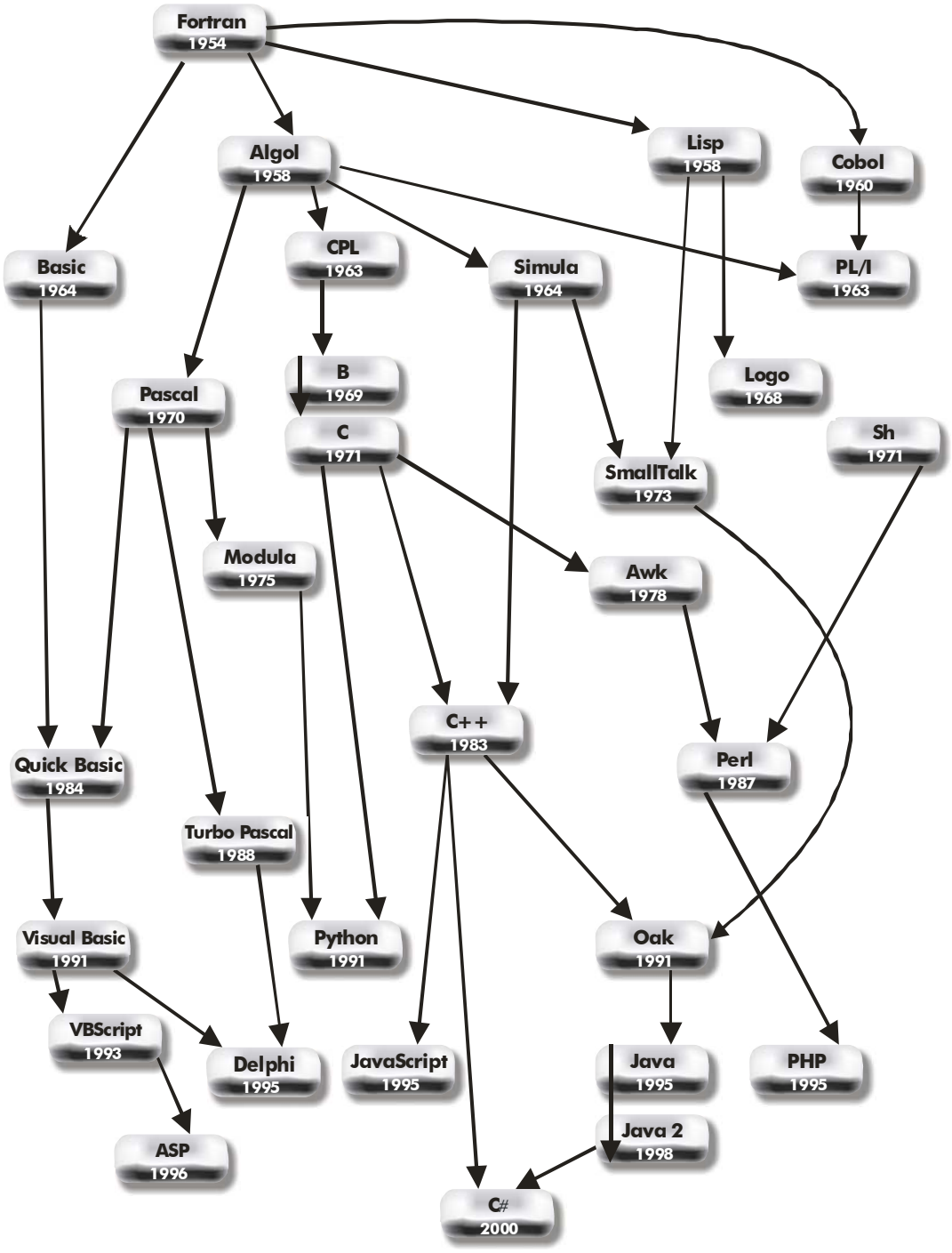


Ilustración 5, Evolución de algunos lenguajes de programación

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**.

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.

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 traducirle, lo que ralentiza mucho su uso.

[3.2] historia del C

[3.2.1] 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**, quien en los años 70 fue el encargado de desarrollar el **lenguaje C.**

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 diseñó como lenguaje pensado para programar sistemas operativos.

Sin embargo tuvo un éxito inmediato y curiosamente ocurrió que sus programas eran muy compatibles con todo tipo de sistemas por lo que se transportó a todo tipo de máquinas y utilidades (proceso de textos, bases de datos, aplicaciones gráficas).

En 1983 el organismo **ANSI** (*Instituto de Estándares de Estados Unidos*) empezó a producir un C estándar para normalizar su situación. En 1989 aparece el considerado como C estándar que fue aceptado por **ISO** (*Instituto Mundial de Estándares*), organismo internacional de estándares. Actualmente este C es universalmente aceptado como el C estándar.

[3.2.2] C y C++

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

Hay que entender que C++ no es un C mejorado, sino que utiliza la sintaxis de C para crear un lenguaje mucho más potente y sobre todo preparado para utilizar la programación orientada a objetos, que está ya considerada como el método de programación más eficiente. No obstante todos los compiladores de C++ admiten crear programas en 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++

[3.2.3] el éxito de C++

Desde su nacimiento C++ ha estado rodeado del éxito. Fue adoptado por una miríada de programadores que veían en él, el lenguaje ideal para generar aplicaciones. Ha sido durante muchos años el lenguaje más utilizado para programar y, de hecho, sigue siéndolo.

Actualmente se habla de la futura extinción de este lenguaje, pero es difícil creer en ello viendo las enormes cantidades de código escritas en este lenguaje y las que se siguen añadiendo.

[3.2.4] ventajas de C++

- **Es un lenguaje que permite programar de forma estructurada, modular y orientado a objetos. Por lo que permite la asimilación de todas estas técnicas.**
- **Es un lenguaje que permite el manejo de todo tipo de estructuras de datos (arrays, pilas, colas, textos, objetos,...) por lo que es capaza de resolver todo tipo de problemas**
- **Es un lenguaje compilado, lo que le hace muy rápido (quizá el más rápido, descontando el ensamblador)**
- **Permite crear todo tipo de aplicaciones (aunque no está pensado para crear aplicaciones para ejecutar desde Internet)**
- **Gracias a su capacidad de uso de objetos y clases, facilita la reutilización del código**
- **Permite incorporar las librerías más utilizadas para hacer aplicaciones**
- **Es el lenguaje que sirve de base para los lenguajes modernos como Java, C#, Perl,...**

[3.3] entornos de programación

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.
- **Intérprete o compilador.** El software que convierte el lenguaje en código máquina.
- **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. En el caso de C++ hay multitud de entornos (a veces llamados simplemente compiladores), algunos interesantes son:

- **Turbo C/C++.** Ahora gratis desde el enlace
  http://community.borland.com/article/images/21751/tcpp101.zip
- **Borland C/C++.** Más orientado a C++, versión más moderna del anterior (y de pago).
- **Microsoft C/C++.** Compilador C de Microsoft que ya dejó de fabricar
- **Visual Studio.** Ahora llamado Visual Studio .NET por su compatibilidad con esa famosa plataforma
- **GCC.** Compilador C++ con licencia GNU (de distribución gratuita y código abierto) http://gcc.gnu.org/
- **KDevelop.** Entorno de programación para desarrollar aplicaciones gráficas en Linux (permite crear aplicaciones KDE y GNOME)
- **Dev C/C++.** Uno de los mejores entornos de programación en C o C++. De libre distribución y que posee un amigable entorno gráfico.
  http://www.bloodshed.net/devcpp.html

[3.4] compilación

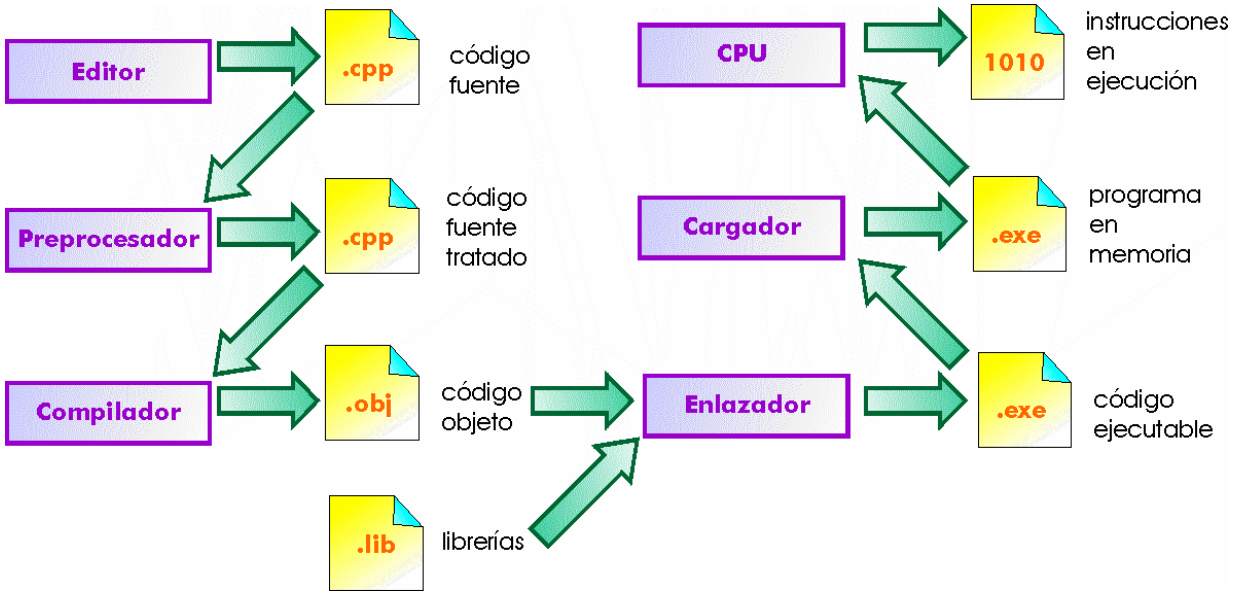


Ilustración 6, proceso de ejecución de un programa C++

Ya se ha explicado anteriormente 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. El código C++ no se puede traducir línea a línea, se compila en bloque para obtener el código ejecutable más óptimo posible

El proceso completo de conversión del código fuente en código ejecutable sigue los siguientes pasos (ver Ilustración 6):

[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 **.cpp** (también en **cxx, c++ o cc**)

[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 un código máquina llamado archivo objeto (cuya extensión suele ser **.obj**). Este código aún no es ejecutable ya que necesita incluir el código máquina relativo a las funciones y módulos que se utilizaban en nuestro código. Estas funciones están incluidas en archivos externos (**librerías**).

[4] **Enlazado.** El código objeto se une al código compilado de las librerías y módulos invocados por el código anterior. El resultado es un archivo ejecutable (extensión **.exe** en Windows)

[5] **Cargado.** Normalmente lo realiza el propio entorno de ejecución. El archivo ejecutable se lanza en el Sistemas Operativo

[6] **Ejecución de las instrucciones por la CPU.** Es entonces cuando el procesador accede al código máquina del programa ya convertido y ejecuta las acciones. Será entonces cuando veamos los resultados.

Estas dos últimas fases cubren lo que se conoce como **ejecución del programa**

[3.5] fundamentos de C++

[3.5.1] estructura de un programa C++

Un programa en C++ (o 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() {
   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 compilación.** Una directiva de compilación 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 tenían extensión **h**, en C++ estándar (ANSI) no se indica la extensión de éstos; aunque se suele incluir la extensión por compatibilidad.

Los archivos de cabecera son los que permiten utilizar funciones externas (o librerías) en nuestro programa. Hay que tener en cuenta que las cabeceras son distintas en C y C++, pero que se pueden utilizar los archivos de cabecera clásicos del C en C++.

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 en la cabecera **iostream** (en C se utiliza la cabecera **stdio.h).**

Para poder utilizar esta función hay que incluir su archivo de cabecera que es **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 <iostream>
int main()
{
      std::cout << "Hola mundo";
      return 0;
}
```

En el programa anterior ocurre lo siguiente:

[1] La línea **include** permite utilizar funciones de la librería *iostream* que es la que permite leer y escribir datos por la consola del Sistema Operativo

[2] La función **main** es la función cuyas instrucciones se ejecutan en cuanto el programa inicia su proceso.

[3] La instrucción **std::cout << "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) que la finalización ha sido correcta.

[3.5.2] elementos de un programa en C++

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 está función le precede la palabra **void** o la palabra **int** (en realidad es más correcta ésta última como se verá más adelante).

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 ({ y }). 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 (aunque esto último no es obligatorio, deberíamos tomárnoslo como tal, debido a su importancia).

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 ya que sino en cuanto pasa un tiempo el código no se comprende bien (ni siquiera por el autor del mismo). 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+`hay dos tipos de comentarios:

- **Comentarios delimitados entre los símbolos /* y */** (procedentes del lenguaje C estándar)

```
/* 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.

- **Comentarios iniciados con los símbolos //** En ellos se ignora todo el texto hasta el final de línea. Ejemplo:

```
int x=5; //Comentario en la misma línea
double a=8.0;
```

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, constantes, funciones o clases definidas por el programador. La lista de palabras reservadas proceden tanto del C, como del C++

palabras procedentes del lenguaje C (afectan también al C++)

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

palabras reservadas propias sólo del lenguaje C++

asm	bool	catch	class
const_cast	delete	dynamic_cast	explicit
false	friend	inline	mutable
namespace	new	operator	private
protected	public	reinterpret_cast	static_cast
template	this	throw	true
try	typeid	typename	using
virtual	wchar_t		

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), números y el carácter de subrayado** (en algunas versiones modernas de compiladores sí que se permiten estos caracteres, pero conviene evitarlos para escribir acordes a la mayoría de compiladores)

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 instrucciones que serán traducidas a código fuente antes de iniciar el proceso de compilado. La más utilizada es

```
#include
```

Que ya se comentó anteriormente

[Unidad 4] programación básica en C++

[4.1] variables

[4.1.1] introducción

Las variables sirven para identificar un determinado valor. Es importante 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.

Desde el punto de vista del programador, la memoria RAM es una sucesión de celdillas, cada una de las cuales posee una dirección. Cada celdilla ocupa un espacio que, en general, es de un byte. Por lo tanto un valor que ocupe cuatro bytes, ocupará cuatro celdas.

Es decir si *saldo* es un identificador que se refiere a un valor numérico 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.

[4.1.2] declaración de variables

En C++ (y en casi todos los lenguajes) hay que declarar las variables antes de poder utilizarlas. 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 entera (*int*).

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 código. Esto facilita la comprensión del código.

También es buena práctica poner un pequeño comentario a cada variable para indicar para qué sirve. Finalmente el nombre de la variable (el identificador) conviene que sea descriptivo. Nombres como *a, b* o *c*; no indican nada. Nombre como *saldo, gastos, nota,...* son mucho más significativos.

Se pueden declarar más una variable a la vez:

```
int x,y,z;
```

E incluso asignar valores al declarar:

```
int x=19;
```

[4.1.3] 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	tamaño en bytes	rango de valores posibles
char	1	0 a 255 (o caracteres simples del código ASCII)
int	2 (algunos 4)	con 2 bytes: -32768 a 32767 con 4 bytes: -2.147.483.648 a -2.147.483.647
float	4	3.4E-38 a 3.3E+38
double	8	1.7E-308 a 1.7E+308
bool	1	**true** o **false**
void	0	sin valor

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 y procesador empleados.

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**, la realidad es que cualquier carácter puede ser representado como número (ese número indica el código en la tabla **ASCII** del carácter 'A' y 65 es lo mismo desde el punto de vista del 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**. En este estilo de almacenar números decimales, la precisión es limitada. Cuántos más bits se destinen a la precisión, más preciso será el número. Por eso es más conveniente usar el tipo **double** aunque ocupe más memoria.

tipos lógicos

En el lenguaje las variables lógicas eran variables enteras que utilizaban el valor **0** para referirse al valor falso, y cualquier otro valor para el verdadero (en general para el verdadero se empleaba el **1**)

```
int x=1; //En C estándar x es verdadera
```

En C++ estándar se debería utilizar el tipo **bool**, que fue creado para dar un mayor sentido a las variables lógicas. Las variables de este tipo pueden tomar los valores **true** (1) o **false** (0)

```
bool x=true;
std::cout<<x; //Escribe 1, es decir verdadero
```

modificadores de tipos

A los tipos anteriores (excepto a **bool**) 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	tamaño en bytes	rango de valores posibles
signed char	1	-128 a 127
unsigned int	2	0 a 65335
long int (o **long** a secas)	4	-2147483648 a 2147483647
long long	8	-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**)
long double	12	3.37E-4932 a 3,37E+4932

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.

[4.1.4] typedef

A veces conviene definir nuestros propios tipos de datos para ser utilizados como sinónimos. Eso lo realiza la instrucción **typedef.** La idea es crear tipos que se manejen de forma más cómoda. En general esos tipos (aunque no es obligatorio se escriben en mayúsculas):

```
typedef unsigned short int USHORT;
...
USHORT x=13;
```

Para el caso, USHORT es un nuevo tipo de datos, equivalente al *unsigned short int*.

[4.1.5] 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 <iostream>
int main(){
    int a;
    std::cout<<a;
    return 0;}
```

Este código no produce error, pero como a la variable *a* no se le ha asignado ningún valor, el resultado del *cout* 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 tiene ningún sentido.

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;
```

La variables *a* y *b* están inicializadas. 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)

[4.1.6] 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: 1.23e+23 (eso significa $(1{,}23 \cdot 10^{23})$
- Los caracteres simples van encerrados entre comillas simples, 'a'
- Los textos (*strings*) van encerrados entre comillas dobles "Hola"
- Los enteros se pueden almacenar como caracteres 'A' o como enteros cortos. Es más 'A' es lo mismo que 65. 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 como almacenamos en una variable *char* el símbolo de la comilla simple, si la propia comilla simple sirve para delimitar, 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 comienza 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*

[4.1.7] 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 le asigna su primer valor (la variable *nace*)

[3] Se la utiliza en diversas sentencias (no se debe leer 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 func(){
   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

Debido a que C++ es un lenguaje orientado a objetos, no es muy correcto el uso de variables globales; usarlas causa malos hábitos.

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 <iostream>
int a=3; //La variable "a" es global
int main(){
    std::cout<<a;
}
```

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 <iostream>
int a=3;
int main(){
    int a=5;
    {
        int a=8;
        std::cout<<a; //escribe 8. No hay error
    }
    std::cout<<a; //escribe 5. No hay error
}
//cout<<a; //escribe 5. No hay error
```

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

[4.1.8] conversión de tipos

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

En C++ no existe esta comprobación. Lo que significa que los valores se convierten automáticamente. Pero eso también significa que puede haber problemas indetectables, por ejemplo este programa:

```
#include <iostream>
int main(){
    char a;
    int b=300;
    a=b;
    std::cout<<int(a)<<'\t'<<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 ajustarles a la precisión de los *float*.

El uso de los operadores de **cast** sirve para convertir de un valor a otro.

Los operadores de conversión (cast) se usan así:

```
(tipo) variable
```

Ejemplo:

```
x=char(z); //z se convierte en un carácter
```

[4.1.9] 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

```
int x,y;
int main(){
      x=12;
      y=6;
}
void funcion1(void){
      x=7;
}
```

Archivo 2

```
extern int x,y;
void func2(void){
      x=2*y;
}
```

El segundo archivo utiliza las variables declaradas en el primero

modificador *auto*

En realidad las variables toman por defecto este valor (por lo tanto 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*

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;
```

Otra posibilidad (hoy en día menos recomendable es utilizar la directiva **#define** esta directiva sustituye en todo el programa el valor de la constante definida:

```
#define PI 3.141592
```

Esta directiva debe colocarse antes del inicio de las funciones del programa (antes de *main*). Aunque la funcionalidad es la misma, el significado es muy distinto. De hecho *define* permite opciones de sustitución, realmente no crea constantes, es una indicación para que antes de compilar se sustituyan todas las apariciones en el programa del texto *PI* por el valor *3.141592*.

modificador *volatile*

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

[4.2]
entrada y salida por consola

Aunque este tema será tratado con detalle más adelante. Es conveniente conocer al menos las instrucciones procedentes de la librería *iostream* que son las encargadas en C++ de gestionar la entrada y salida del programa. Para poder utilizar esas operaciones, hemos de incluir la directiva siguiente al inicio del nuestro programa:

```
#include <iostream>
```

objeto *cout*

El objeto **cout** es el encargado de producir la salida de datos en C++. La forma de utilizarle es la siguiente:

```
std::cout<<"Hola";
```

a razón de utilizar *std::cout* y no *cout* sin más se debe, a que la palabra *cout* pertenece al espacio de nombres de *std* (algo que será explicado en temas muy posteriores). Baste explicar ahora que el operador de flujo **<<** es el encargado de pasar los datos a este objeto que se encarga de mostrarles por pantalla en el formato más adecuado (es capaz de saber si lo que llega es texto o números).

La ventaja de este objeto es que puede recibir distintos tipos de datos en la misma línea, basta con encadenarles utilizando sucesivos "<<":

```
int x=65;
char l='A';
std::cout<<"El código de "<<l<<" es "<<x<<"\n";
//Escribe: El código de A es 65
```

El objeto *cout* es capaz de obtener adecuadamente el contenido de las variables puesto que comprende perfectamente su tipo. Pero a veces conviene convertir el formato para que sea entendido en la forma deseada, por ejemplo:

```
char l='A';
std::cout<<"El código de "<<l<<" es "<<int(l)<<"\n";
//Escribe: El código de A es 65
```

En el ejemplo, se escribe la letra *A* al principio porque la variable *l* se interpreta como carácter. Gracias a la instrucción **int(l)** se interpreta la segunda vez como número entero.

objeto *cin*

Se trata del objeto que permite leer del teclado y almacenar lo leído en una variable. Por ejemplo:

```
std::cin >> x;
```

almacena en *x* el valor leído por teclado. La lectura utiliza el tipo de datos adecuado según el tipo de la variable (almacena texto si es *char* y números si es *int, float, long, double,...* los valores *bool* se manejan como si fueran números).

objeto *endl*

Representa el fin de línea (el carácter *\n*) para hacer saltos de línea se usaría de esta forma:

```
std::cout<<"Hola"<<std::endl<<"mundo";
```

```
//Escribe Hola en la primera línea y mundo en la segunda
```

Para no tener que escribir el texto *std::* continuamente, se puede colocar la instrucción **using** antes del **main.**

```
using std::cout;
using std::cin;
using std::endl;
int main(){
   cout<<"Hola"<< endl<<"mundo";}
```

[4.3] operadores

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

[4.3.1] 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;
```

```
std::cout<<x1<<'\n'; //Escribe 10
std::cout<<x2<<'\n'; //Escribe 10
std::cout<<y1<<'\n'; //¡¡¡Escribe 9!!!
std::cout<<y2<<'\n'; //Escribe 10
```

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

[1] y1=x1

[2] x1=x1+1

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

x2=x2+1

y2=x2

Es decir en *++x2* primero se incrementó y luego se asignó el valor incremento a la variable *y2*.

[4.3.2] 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

Un detalle importante es el hecho de que en C++ el signo **=** sirve para asignar valores, mientras que **==** sirve para comparar dos valores

[4.3.3] 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).

[4.3.4] 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 este manual simplemente se indican estos operadores:

operador	significado
&	AND de bits
|	OR de bits
~	NOT de bits
^	XOR de bits

[4.3.5] 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;
std::cout<<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

operador	significado
%=	Calcula el resto y asigna

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

[4.3.6] 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:

```
int nota;
std::cin>>nota;
std::cout<<(nota>=5?"Aprobado":"Suspenso");
```

En este ejemplo si la nota leída es superior a 5 se escribe *Aprobado* y si no *Suspenso*

[4.3.7] operadores de puntero & y *

Aunque ya se lea explicarán más adelante con detalle, conviene irles conociendo. 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.

[4.3.8] operador sizeof

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

```
int x=18;
std::cout<<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.

[4.3.9] 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). Hay que tener cuidado en su uso

[4.3.10] 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.

[4.3.11] 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 en 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] Relacionales sin igualdad: > < >= <=

[6] Relacionales de igualdad: == !=

[7] &

[8] ^

[9] |

[10] &&

[11] ||

[12] ?:

[13] = *= /* += -= %= >>= <<= |= &= ^=

[14] , (coma)

[4.4] expresiones y conversión de tipos

[4.4.1] 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).

[4.4.2] 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.

[4.4.3] 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 (comprar 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.

[4.5] control del flujo del programa

[4.5.1] expresiones lógicas

Hasta este momento 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.

Todas las sentencias de control de flujo se basan en evaluar una expresión lógica. Una expresión lógica es cualquier expresión que pueda ser evaluada con verdadero o falso. En C (o C++) se considera verdadera cualquier expresión distinta de 0 (en especial el 1, valor **true**) y falsa el cero (**false**).

[4.5.2] sentencia *if*

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){
   std::cout<<"Aprobado";
   aprobados++;
}
```

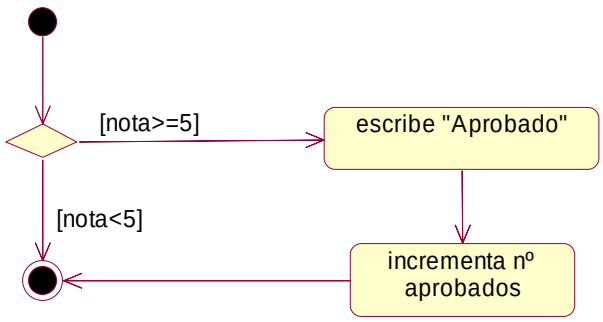


Ilustración 7, Diagrama de actividad del ejemplo anterior

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){
   std:cout<<"Aprobado";
   aprobados++;
}
else {
   std:cout<< "Suspensos";
   suspensos++;
}
```

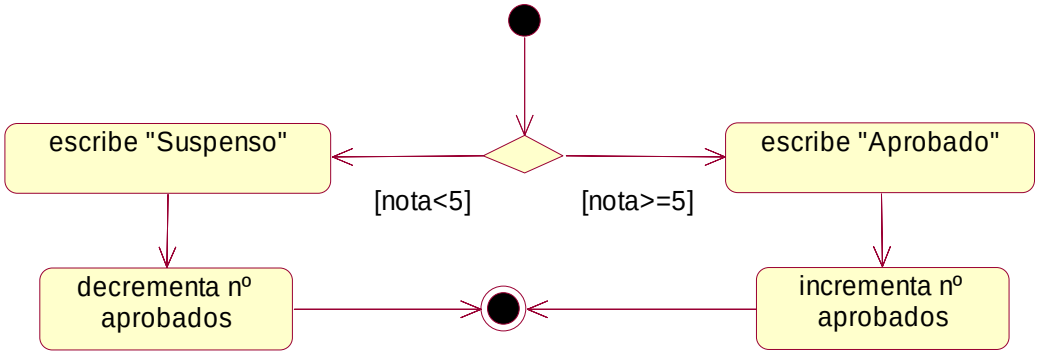


Ilustración 8, diagrama UML de actividad del ejemplo anterior

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 quedar:

```
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
```

```
}
```

[4.5.3] 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; /*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:
         std::cout<<"Lunes";
         break;
   case 2:
         std::cout<<"Martes";
         break;
   case 3:
         std::cout<<"Miércoles";
         break;
   case 4:
         std::cout<<"Jueves";
         break;
   case 5:
         std::cout<<"Viernes";
         break;
```

```
  case 6:
        std::cout<<"Sábado";
        break;
  case 7:
        std::cout<<"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:
        std::cout<<"Laborable";
        break;
  case 6:
  case 7:
        std::cout<<"Fin de semana";
        break;
  default:
        std::cout<<"Error";
}
```

[4.5.4] bucles

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){
   std::cout<<i<<" ";
   i++;
}
```

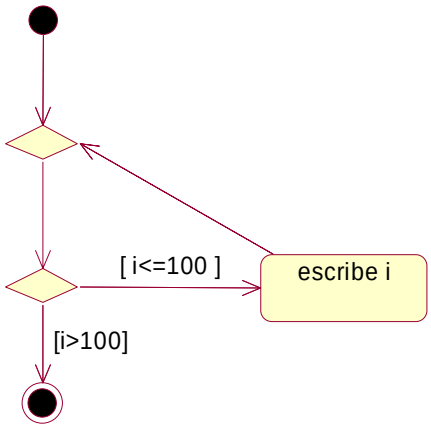


Ilustración 9, diagrama UML de actividad del bucle anterior

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++;
   std::cout<<i<<" ";
} while (i<=1000);
```

sentencia *for*

Se trata de un bucle especialmente útil para utilizar 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++){
   std::cout<<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
   std::cout<<i<<" ";
   i++; //incremento
}
```

[4.5.5] 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 en algunos listados de código puede ser útil conocerlas.

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++){
   std::cout<<i<<" ";
   if(i==300) break;
}
```

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

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;
   std::cout<<i<<" ";
}
```

En ese listado aparecen los números del 1 al 1000ç, 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) std::cout<<i<<" ";
}
```

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

[Unidad 5]
funciones

[5.1]
funciones y programación modular

[5.1.1] introducción

Al realizar programas, a medida que pretendamos resolver problemas más complejos cada vez, el tamaño de nuestro código empieza a desbordarnos. Para mitigar este problema apareció la **programación modular.** En ella el programa se divide en distintos módulos, de manera que cada módulo contiene código de tamaño más manejable. Cada módulo realiza una función muy concreta y se puede programar de forma independiente.

En definitiva la programación modular implementa el paradigma *divide y vencerás*, tan importante en la programación. El programa se descompone en esos módulos, lo que permite concentrarse en problemas pequeños para resolver los problemas grandes. Cada módulo tiene que ser probado y verificado para cualquier caso. De eso modo aseguramos su fiabilidad y nos concentramos en otro módulo.

Así una aplicación se puede entender (en la programación modular) como un conjunto de módulos que se comunican entre sí. Cuando este paradigma se domina, se programan módulos utilizables en más de un programa, a esto se le llama **reutilizar el código**.

En C++ los módulos se programan utilizando **funciones.** las funciones están compuestas de código fuente que responde a una finalidad. Este código generalmente devuelve un determinado valor o bien realiza algún tipo de acción aunque no devuelva

ningún valor concreto (a las funciones que no devuelven ningún valor se las suele llamar **procedimientos**)

Las funciones son **invocadas** desde el código utilizando su nombre después del cual se colocan paréntesis y dentro de los paréntesis los datos que la función necesita para su realización. Por ejemplo:

```
std::cout<<factorial(8);
```

Este código invoca a una función llamada *factorial* enviando a la misma el valor 8, por lo que se supone que aparecerá en la pantalla el factorial de 8

Una instrucción que contiene el nombre de la función hace que el flujo del programa cambie a la función y ejecute su contenido.

C++ incorpora una serie de funciones estándar agrupadas en librerías. Por ejemplo la función *system* que sirve para ejecutar comandos del sistema operativo está contenida en la librería (procedente en realidad del lenguaje C *stdlib.h*)

[5.1.2] uso de las funciones

Todo programa 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 (invocaciones) a funciones ya creadas, bien por el propio programador o bien que forman parte de las librerías estándar de C++ o de C (una **biblioteca** o **librería** no es más que una colección de funciones).

Así por ejemplo:

```
int main(){
   std::cout<<pow(3,4);
}
```

Ese código utiliza la función *pow* que permite elevar un número a un exponente (en el ejemplo 3^4).

Para poder utilizarlas las funciones tienen que estar definidas en el código del programa o en un archivo externo que uniremos a nuestro código durante la fase de enlazado (*link*) del ejecutable. Si están en un archivo externo (como ocurre en el ejemplo) habrá que incluir la cabecera de ese archivo para que al crear el ejecutable de nuestro programa se enlacen los archivos compilados de las funciones externas que utilice el programa.

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

```
#include <iostream> //para el objeto cout
#include <math.h>
```

[5.1.3] crear funciones

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

[1] Crear una línea en la que se indica el nombre de la función, el tipo de datos que devuelve dicha función y los parámetros que acepta. A esto se le llama la **cabecera de la función**. Tras la cabecera se abre la llave que permite ver el código propio de la función

[2] Indicar las variables locales a la función. Aquí se declaran las variables que la función utiliza. Estas variables sólo podrán ser utilizadas desde la propia función.

[3] Indicar las instrucciones de la función

[4] Si es preciso indicar el valor que devuelve mediante la sentencia return. Si la función no devuelve nada, no habrá **return** y obligatoriamente en la cabecera el tipo de datos de la función será **void**

Sintaxis:

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

- *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.
- *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. 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.

La cabecera sirve también para definir el prototipo de la función. Este prototipo se debe colocar al principio del código (delante de **main**) a fin de avisar al compilador del posterior uso de la función. Hoy en día la definición de las funciones se suele hacer al final.

Ejemplo de función y su uso (cálculo de la potencia de un número):

```
/*
   Programa: Potencia
   Autor: Jorge Sánchez
   Versión: 1.0
   Descripción: Ejemplo de uso de funciones. En
   este caso la función calcula la potencia de un número
*/
#include <iostream>
#include <stdlib.h>

using namespace std;
```

```
// Prototipo
double potencia(double base, int exponente);
```

```
int main(){
    double b;
    int e;
    cout<<"Escriba la base de la potencia";
    cin>>b;
    cout<<"Escriba el exponente";
    cin>>e;
    cout<<"resultado: "<<potencia(b,e)<<endl;
    system("pause");
    return 0;
}
```

```
// Definición de la función
double potencia(double base, int exponente){
  int contador;
  /* Indica si el exponente es negativo*/
  bool negativo=false;
  double resultado=1.0;

  if (exponente<0) {
        exponente=-exponente;
        negativo=true;
  }
```

```
  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 (no vale **int a,b** sino **int a, int b**), si no se pone el tipo se suele tomar el tipo **int** para el parámetro (aunque esto puede no ser así en algunos compiladores).

[5.1.4] prototipos de funciones

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.

En los prototipos realmente el compilador sólo tienen en cuenta el tipo de retorno, el nombre de la función y el tipo (no el nombre) de los parámetros. Así el prototipo del ejemplo anterior podría haberse escrito así:

```
double potencia(double, int);
```

En un programa no puede haber dos prototipos de funciones iguales.

[5.1.5] 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* ya que fomenta la creación de programas no estructurados.

[5.1.6] archivos *include*

Los archivos a los que hace referencia la directiva **incluye** son en realidad **archivos de cabecera**. Estos archivos suelen contener constantes, prototipos de funciones y otras directivas include para usar funciones de otras librerías. El código real de las librerías se almacena aparte (normalmente ya compilado) y es unido al código ejecutable de nuestro programa durante la fase de enlazado.

[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 valor2).

[5.2.2] problemas de alcance de las variables

Ser comentó en el tema anterior que las variables sólo persisten en el bloque en el que fueron creadas. Lo cual significa que un parámetro y una variable definida dentro de una función, son variables locales a la función; es decir, se eliminan cuando la ejecución de la función termina.

Eso tiene estas connotaciones:

```
void prueba();
const double X=5;

int main(){
    prueba();
    system("pause");
    return 0;
}


void prueba(){
    int X=8;
    std::cout<<X; //Escribe 8
}
```

Dentro de la función prueba se utiliza una variable llamada *x*, lo que sugiere cierta ambigüedad ya que hay una variable global llamada *x*. Lógicamente la función entiende que la variable a la que se hace referencia es a la local.

Pero si empleamos los operadores **::**, entonces indicaríamos que queremos la variable global:

```
void prueba();
const double X=5;
```

```
int main(){
    prueba();
    system("pause");
    return 0;
}


void prueba(){
    int X=8;
    std::cout<<::X; //Escribe 5
}
```

[5.2.3] paso de parámetros por valor y por referencia

Se explicará con detalle más adelante. En el ejemplo:

```
int x=3, y=6;
cout<<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 ocurre nada. 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.**

No obstante hay veces en los que es necesario utilizar parámetros en los que el valor de la variable se modifica dentro de la función. Aunque esto tiene que ver con **punteros**, tema que será debidamente tratado más adelante, es necesario entender ahora cómo se maneja y en qué casos es necesario.

Normalmente una función sencilla que calcule el cuadrado de un número, sería así:

```
double cuadrado(double n){
  return n*n;
}
```

Pero imaginemos que deseamos hacer esta llamada:

```
int x=3;
cuadrado(x);
std::cout<<x;
```

Saldría 3 por pantalla, ya que el valor de x no varía. Si lo que queremos es cambiar el valor de x, tendríamos que hacer:

```
int x=3;
x=cuadrado(x);
std::cout<<x;
```

No obstante, hay una forma de variar el valor de x. Se le llama pasar por **referencia** (en lugar de pasar por valor). Esa forma de pasar implica pasar la dirección y no el valor de la variable. En la declaración de la función, hay que añadir tras el tipo del parámetro el signo **&**, que indica que operaremos con la dirección de la variable. Dicho de otro modo, utilizaremos paso por referencia.

En el ejemplo la función cuadrado definida por referencia sería:

```
void cuadrado(double &n){
    n*=n;
}
```

La función es de tipo **void**, por que ahora no devuelve valores, sino que directamente toma al parámetro *a* por referencia y le calcula el cuadrado. La llamada ahora sí podría ser:

```
int x=3;
cuadrado(x);
std::cout<<x; //escribe 9
```

El manejo de referencias en lenguaje C (no en C++) es un poco más complejo ya que la llamada *cuadrado(x)* no sería válida. Habría que pasar realmente la dirección de *x*, es decir: *cuadrado(&x)*

Eso significa que la definición de la función cuadrado también sería distinta. En C++ es más sencillo ya que los punteros son un poco más sencillos.

[5.2.4] parámetros por defecto

A veces interesa en el prototipo de la función indicar un valor predeterminado en alguno de los parámetros. Si hacemos esto, entonces al llamar la función podremos no indicar dichos parámetros, por lo que tomarán el valor por defecto.

Ejemplo:

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

using namespace std;

// Prototipo
double potencia(double base=1.0, int exponente=2);
```

```
int main(){
    double b;
    int e;
    cout<<potencia(3,2)<<endl;//Escribe 9
    cout<<potencia(5)<<endl;//Escribe 25
    cout<<potencia()<<endl;//Escribe 1
    system("pause");
    return 0;
}


double potencia(double base, int exponente){
… //código de la función
```

Cuando en las llamadas no se pasan todos los parámetros. Éstos toman el valor por defecto. Si no tuvieran valores por defecto, éstas llamadas generarían errores de compilación.

[5.2.5] sobrecarga de funciones

Es una de las grandes mejoras del lenguaje C++ (en C no es posible realizar ésta técnica). Consiste en definir dos veces la misma función, pero utilizando diferentes parámetros. Ésta técnica permite que la función se adapte a todo tipo de llamadas. Una razón para ello es este ejemplo:

```
using namespace std;
double cuadrado(int);
int main(){
    double x=5.5;
    int y=7;
    cout<<cuadrado(y)<<endl; //Escribe 49
    cout<<cuadrado(x)<<endl; //Escribe 25
    system("pause");
    return 0;
}

double cuadrado(int n){
    return n*n;
}
```

La llamada con el valor 5,5 falla por que se convierte a **int**, ya que la función así lo pide. Lógicamente un arreglo rápido es definir el parámetro de la función como **double**,

ya que ese tipo permite usar los enteros. Sin embargo otra posibilidad es definir dos veces el cuadrado:

```
using namespace std;

double cuadrado(int);
double cuadrado(double); //Segunda versión

int main(){
    double x=5.5;
    int y=7;
    cout<<cuadrado(y)<<endl; //Escribe 49. Llama a la 1ª versión
    cout<<cuadrado(x)<<endl; //Escribe 30.25 Segunda versión
    system("pause");
    return 0;
}

double cuadrado(int n){
    return n*n;
}

double cuadrado(double n){
    return n*n;
}
```

Aunque ahora pueda parecer absurda esta doble definición, en realidad es extremadamente útil, ya que prepara las funciones antes todas las pasibilidades de uso.

[5.2.6] parámetros constantes

En la lista de parámetros se puede poner el indicador **const** para señalar que el valor del parámetro ha de permanecer constantes a lo largo de la función. En este ejemplo:

```
double prueba(const double x){
    x=4;
    return x; //error
}
```

Ocurre un error ya que el valor de *x* no se puede cambiar dentro de la función al haber sido definido como constante.

[5.2.7] funciones *inline*

Retrata de una característica propia sólo de este lenguaje. Son funcione que tratan de evitar el problema de las llamadas a funciones. Esas llamadas suelen generar un ejecutable lento. Las funciones **inline**, lo que hace es hacer que las llamadas a estas funciones se traduzcan por el código de la función.

Eso hace que el código fuente sea más grande, ya que el compilador traducirá la llamada, por lo que sólo son útiles en funciones simples; de hecho los compiladores modernos ignoran el indicador *inline* si la función es compleja.

Para definir funciones de este tipo basta con poner la palabra **inline** antes del nombre de la función. Ejemplo:

```
inline double cuadrado(const double n){
    return n*n;
}
```

[5.3] recursividad

[5.3.1] introducción

La recursividad es una técnica de creación de funciones en la que una función se llama a sí misma. Hay que ser muy cauteloso con ella (incluso evitarla si no es necesario su uso), pero permite soluciones muy originales, a veces, muy claras a ciertos problemas. 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.

Es fundamental tener en cuenta cuándo la función debe dejar de llamarse a sí misma, es decir cuando acabar. De otra forma se corre el riesgo de generar infinitas llamadas, lo que bloquearía de forma grave el PC en el que trabajamos.

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

[5.3.2] ¿recursividad o iteración?

Ya conocíamos otra versión de la función factorial resuelta por un bucle **for** en lugar de por 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 pero no pesa demasiado en la ejecución del ordenador**

[5.3.3] 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.

[5.3.4] la pila de funciones

Cada vez que desde un código se llama a una función, esta llamada se coloca en una zona de la memoria llamada pila de funciones. Cada vez que desde una función llamamos a otra, esta llamada se coloca en la pila y así sucesivamente. A medida que las funciones finalizan, su llamada se retira de la pila.

Esta pila tiene un tamaño limitado. Por lo que si la recursividad genera un bucle infinito entonces el sistema operativo generará un error de **desbordamiento de pila**, referido a que ya no caben más llamadas a funciones.

[5.4] plantillas de funciones

Como se comentó en el apartado sobrecarga de funciones, página 91 a veces es necesario definir dos o más veces la misma función para que sea aplicable a distintos tipos de datos. Por ejemplo tendríamos que definir las funciones:

```
int maximo(int a, int b);
double maximo(double a, double b);
char maximo(char a, char b);
```

Pero podríamos observar que el código es siempre el mismo. Por ello en C++ se utilizan las plantillas de función (llamadas **templates**). Con ellas el programador escribe una sola versión de la función aplicable a cualquier tipo de datos (o a cualquier clase de

datos en el caso de que se trate de objetos). Lo que hará el compilador será generar todas las versiones necesarias de la función para aplicarlas a cualquier tipo de datos.

Las plantillas se definen delante de la definición de la función poniendo **<template** seguida de los nombres que daremos a los tipos genéricos. Se pueden definir tipos de datos utilizando la palabra **typename** o **class** (aunque son sinónimos, se recomienda más usar la segunda debido a la facilidad de encajar esta palabra con la programación orientada a objetos).

La idea es definir nombres de tipos de datos que representan a cualquier tipo con el que se llame a la función. Quedará más clara la idea viendo el código de la función *max* usando plantillas:

```
template <class T> //T representa un tipo cualquiera
inline T maximo(T a,T b){
            //a y b son variables del mismo tipo de datos
            // y ese tipo es T, es decir cualquiera
  return (a>b?a:b);
}
```

En el código anterior se define la palabra T como un tipo de datos genérico, es decir T representa cualquier tipo de datos (**int, bool, char**,….)

Nota: El prototipo de la función incluiría el apartado *template*, es decir el prototipo de la función anterior es:

```
template <class T>
inline T maximo(T a,T b);
```

[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 que se debe enlazar el código de dicha librería al nuestro para que nuestro programa funcione.

Eso se realizar mediante la directiva de procesador **#include** seguida del nombre del archivo de cabecera de la librería. Es posible incluso utilizar librerías del lenguaje C en casi cualquier compilador de C++, en ese caso la cabecera suele tener extensión **.h**.

[5.5.2] ejemplo de librería estándar *cmath*

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

```
#include <cmath>
```

al principio del código de nuestro archivo.

Esta librería contiene las funciones que se comentan en la siguiente tabla. Se indican los prototipos y uso, teniendo en cuenta que todas las funciones tienen versiones para **long double**, **float** y **double**, en la tabla sólo se expone la versión **double**. Además hay que tener en cuenta que casi todas las funciones son de tipo **inline** (aceleran la ejecución pero extienden el tamaño del código)

Prototipo	Descripción
double abs(double *n***)**	Devuelve el valor absoluto del número entero indicado
double ceil(double *n***)**	Redondea el número decimal *n* obteniendo el menor entero mayor o igual que *n.* Es decir *ceil(2.3)* devuelve 3 y *ceil(2.7)* también devuelve 3
double cos(double *n***)**	Obtiene el coseno de *n* (*n* se expresa en radianes)
double exp(double *n***)**	Obtiene e^n
double floor(double *n***)**	Redondea el número decimal *n* obteniendo el mayor entero menor o igual que *n.* Es decir *floor(2.3)* devuelve 2 y *floor(2.7)* también devuelve 2
double fmod(double *x***, double** *y***)**	Obtiene el resto de la división de *x/y* dividiéndoles como decimales
double log(double *n***)**	Devuelve el logaritmo neperiano de *n*
double log10(double *n***)**	Devuelve el logaritmo decimal de *n*
double pow(double *base***, double** *exponente***)**	Obtiene $base^{exponente}$
double sin(double *n***)**	Obtiene el seno de *n* (*n* se expresa en radianes)
double sqrt(double *n***)**	Obtiene la raíz cuadrada de *n*
double sin(double *n***)**	Obtiene la tangente de *n* (*n* 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 **cstdlib**. Esta librería sustituye a *stdlib.h*

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

```
for(i=1;i<=100;i++)
  std::cout<<rand()<<std::endl;
```

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++)
    std::cout<<rand()%10<<std::endl;
```

Obtiene 100 números aleatorios del 0 al 9. Pero ocurre un problema, los números son realmente *pseudoaleatorios*, es decir la sucesión que se obtiene siempre es la misma. Aunque los números circulan del 0 al 9 sin orden ni 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. La semilla se puede generar mediante la función **srand** a la que se le pasa como parámetro un número entero que será la semilla. Según dicha semilla la serie será distinta.

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

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

Esa expresión toma un número 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++)
    std::cout<<rand()%10<<std::endl;
```

genera cien números del 0 al 9 siendo casi impredecible saber qué números son, ya que la semilla se toma del reloj del sistema.

[Unidad 6] estructuras estáticas de datos

[6.1] estructuras estáticas

En programación se llaman estructuras estáticas a datos compuestos de datos simples (enteros, reales, caracteres,...) que se manejan como si fueran un único dato y que ocupan un espacio concreto en memoria que no varía durante la ejecución del programa.

Las estructuras estáticas de C++ son:

- **Arrays.** También llamadas **listas estáticas**, **matrices** y **arreglos** (aunque quizá lo más apropiado es no traducir la palabra *array*). Son una colección de datos del mismo tipo.
- **Cadenas de caracteres**. También llamadas **Strings**. Se trata de un conjunto de caracteres que es tratado como un texto completo.

- **Punteros**. Permiten definir variables que contienen posiciones de memoria; son variables que se utilizan para apuntar a otras variables, se las llama también **apuntadores.**

[6.2] arrays

[6.2.1] introducción

Imaginemos que deseamos leer las notas de una clase de 25 alumnos. Desearemos por tanto almacenarlas y para ello, utilizando variables clásicas no nos quedará más remedio que declarar 25 variables.

Eso es tremendamente pesado de programar. Manejar las notas significaría manejar continuamente 25 variables. Este tipo de problemas es el que se encargan de solucionar los arrays.

Los arrays son una colección de datos del mismo tipo al que se le pone un nombre (por ejemplo *nota*). Para acceder a un dato de la colección hay que utilizar su índice (por ejemplo *nota[4]* leerá la cuarta nota).

Tras leer las 25 notas, el resultado se almacena en la variable *nota* y se podrá acceder a cada valor individual usando *nota[i],* donde *i* es el elemento al que queremos acceder. Hay que tener en cuenta que en los arrays el primer elemento es el 0; es decir *nota[4]* en realidad es el quinto elemento.

En definitiva un array es una colección de elementos que se disponen en memoria consecutivamente y que poseen el mismo tipo y el mismo nombre y que permite acceder a cada elemento individual de la colección a través de un número entero que se le llama índice.

[6.2.2] declaración de arrays

Una array ocupa un determinado espacio fijo en memoria. Para que el ordenador asigne la cantidad exacta de memoria que necesita el array, hay que declararle. En la declaración se indica el tipo de datos que contendrá el array y la cantidad de elementos. Por ejemplo:

```
int a[7];
```

Esa instrucción declara un array llamado *a* de siete elementos. Lo que significa que en memoria se reserva el espacio de siete enteros (normalmente 2 o 4 bytes por cada entero). Los elementos del array irán de *a[0]* a *a[6]* (hay que recordar que en C el primer elemento de un array es el cero).

Hay que tener en cuenta que **los arrays contienen una serie finita de elementos**, es decir, de antemano debemos conocer el tamaño que tendrá el array. El valor inicial de los elementos del array será indeterminado (dependerá de lo que contenga la memoria que se ha asignado al array, ese valor no tiene ningún sentido).

Es posible declarar varios arrays a la vez haciendo:

```
int a[100], b[25];
```

También es válida esta declaración:

```
int x=10;
int a[x]; //Declara un array de 10 enteros
```

[6.2.3] utilización de arrays

asignación de valores

El acceso a cada uno de los valores se realiza utilizando un **índice** que permite indicar qué elemento del array estamos utilizando.

Por ejemplo:

```
a[2]=17;
std::cout<<a[2]<<std::endl;
```

a[2] es el tercer elemento del array, se le asigna el valor 17 y ese valor se muestra luego en la pantalla.

Para acceder a todos los datos de un array se suele utilizar un bucle (normalmente de tipo **for**). Por ejemplo en el caso de tener que leer 21 notas, la comodidad de los arrays reside en que sin arrays necesitamos 21 instrucciones, con los arrays basta un bucle. Ejemplo (lectura de 21 notas):

```
/* Array de 21 elementos */
int nota[21];
/* Contador para recorrer el array */
int i;

for(i=0;i<21;i++) {
   cout<<"Escriba la nota "<<i<<":  ";
   cin>>nota[i];
}
```

Al final del bucle anterior, las notas estarán rellenadas con los valores que el usuario escriba por teclado. Por ejemplo con estos valores:

componente	nota[0]	nota[1]	nota[2]	…	nota[19]	nota[20]
valor	5	6	4	…	7	4

Otro ejemplo (array que almacena y muestra los 20 primeros factoriales);

```
double factorial[21];
int i;
```

```
factorial[0]=1;
for(i=1;i<21;i++){
   factorial[i]=i*factorial[i-1];
}
/*Mostrar el resultado*/
for(i=0;i<21;i++){
   cout<<"El factorial de "<<i<<" es "<<factorial[i]<<endl;
}
```

En C++ hay que tener un cuidado especial con los índices, por ejemplo este código es erróneo:

```
int a[4];
a[4]=13;
```

No hay elemento *a[4]* por lo que ese código no tiene sentido. Sin embargo ningún compilador C++ nos advertirá del fallo. Lo que ocurrirá es que se almacenará el número 13 en la posición del que sería el quinto elemento del array. Sin embargo al haber reservado sólo espacio para cuatro elementos, esto significará que invadiremos el espacio de la memoria contiguo a nuestro array. Es un error grave ya que esta instrucción podría modificar datos de otras variables generando errores muy difíciles de controlar.

Una buena práctica de programación consiste en utilizar constantes en lugar de números para indicar el tamaño del array. En el ejemplo de las notas ocurre que si nos hemos equivocado y en lugar de *21* notas había *30*, tendremos que cambiar todos los *21* por *30*. En su lugar es más conveniente:

```
const int TAMANIO=20;
int nota[TAMANIO];
/* Contador para recorrer el array */
int i;

for(i=0;i<TAMANIO;i++) {
   std::cout<<Escriba la nota "<<i<<":";
   std::cin>>nota[i];
}
```

iniciar arrays en la declaración

C posee otra manera de asignar valores iniciales al array. Consiste en poner los valores del array entre llaves en la declaración del mismo.

Ejemplo:

```
int nota[10]={3,5,6,7,3,2,1,7,4,5};
```

nota[0] valdrá 3, *nota[1]* valdrá 5, *nota[2]* valdrá 6 y así sucesivamente. Las llaves sólo se pueden utilizar en la declaración. Si hay menos elementos en las llaves de los que posee el array entonces se rellenan los elementos que faltan con ceros, ejemplo:

```
int nota[10]={3,5};
int i=0;
for(i=0;i<10;i++) std::cout<<nota[i]<<"   ";
```

El resultado es:

```
3  5  0  0  0  0  0  0  0  0
```

es decir, *nota[0]* valdrá 3, *nota[1]* valdrá 5, y el resto se rellena con ceros. En el caso de que haya más elementos en las llaves de los que declaramos en el array, por ejemplo:

```
int a[2]={3,5,4,3,7};
```

Esa instrucción produce un error normalmente del tipo *demasiados inicializadores.*

Otra posibilidad es declarar el array sin indicar el tamaño y asignarle directamente valores.

El tamaño del array se adaptará al tamaño de los valores:

```
int a[]={3,4,5,2,3,4,1}; /* a es un array int[7] */
```

[6.2.4] pasar un array como parámetro de una función

Los arrays son un elemento muy interesante para utilizar conjuntamente con las funciones. Al igual que las funciones pueden utilizar enteros, números reales o caracteres como argumentos, también pueden utilizar arrays; pero su tratamiento es diferente.

En el tema anterior (dedicado a las funciones), ya se comentó la que significaba pasar como valor o pasar por referencia una variable. Cuando se pasa por valor, se toma una copia del valor del parámetro; por referencia se toma la dirección del dato. Esto último implica que cuando se modifica una variable pasada por referencia, en realidad se está modificando la variable original y no el parámetro.

En el caso de los arrays, la dirección del primer elemento del array, marca el comienzo del mismo. Un array se pasa por referencia ya que se pasa sólo el nombre del array.

Imaginemos que tenemos una función llamada *doblaArray* que sirve para doblar el valor de cada elemento que forma parte de un array. Supongamos también que la función trabaja con arrays de enteros. Si tenemos un array llamado *numero* que contiene 20 números enteros, la llamada a la función que dobla cada elemento sería:

```
int numero[20]={…….}; //se supone que entre las llaves se
                      //colocarían los valores del array
doblaArray(numero,20);
```

En la llamada se usan dos parámetros, el primero es el nombre del array. El segundo es el tamaño del array. Las funciones que trabajan con array suelen tener este segundo parámetro, precisamente para saber hasta dónde hay que recorrer el array. Hay que observar que en el primer parámetro se pasa el nombre del array sin usar corchetes.

La realidad es que la función recibe la dirección del array (se pasa por referencia) y que esta dirección será la que utilice el primer parámetro.

La función tendría este código:

```
void doblaArray(int a[], int tamanio){
  int i;
  for(i=0;i<tamanio;i++){
        a[i]*=2;
  }
}
```

Esta función modifica el contenido del array, no recibe una copia, sino que al recibir la dirección del array, realmente trabaja con su dirección. Es decir tras la llamada *doblaArray(numero,20)* el array *a* al que se refiere el código de la función es el mismo array *numero*. Modificar *a* es lo mismo que modificar *numero*.

A veces no se requiere modificar el array. Por ejemplo imaginemos que queremos crear una función llamada *escribeArray* que sirva para escribir el contenido de un array de enteros. El prototipo podría ser:

```
void escribeArray(const int a[], int tamanio);
```

La palabra **const** indica que no se modificará el contenido del array dentro de la función (de hacerlo ocurriría un error de compilación). La razón de utilizar este modificar en esta función es que está función sólo lee el contenido del array.

[6.2.5] algoritmos de búsqueda y ordenación en arrays

búsqueda en arrays

En muchas ocasiones se necesita comprobar si un valor está en un array. Para ello se utilizan funciones que indican si el valor está o no en el array. Algunas devuelven verdadero o falso, dependiendo de si el valor está o no; y otras (mejores aún) devuelven la posición del elemento buscado en el array (o un valor clave si no se encuentra, por ejemplo el valor -1).

En algunos casos las funciones trabajan sobre arrays que deben de estar ordenados.

búsqueda lineal

Trabajan con arrays desordenados. La función de búsqueda busca un valor por cada elemento del array. Si le encuentra devuelve la posición del mismo y si no devuelve -1.

```
/*
 devuelve la posición del elemento buscado, si no lo
 encuentra devuelve -1
 parámetros:
        vector: Array que contiene todos los valores
        tamanio: Tamaño del mismo
        valor: valor a buscar
*/
template <class T>
int busquedaLineal(T vector[], int tamanio, T valor){
  int i=0;
  bool encontrado=false;
  while (encontrado==false && i< tamanio){
        if(vector[i]==valor) encontrado=true;
        else i++;
  }
  if(encontrado==true) return i;
  else return -1;
}
```

búsqueda binaria

Se basa en que el array que contiene los valores está ordenado. En ese caso se puede acelerar la búsqueda. Se basa en buscar desde el centro del array y mediante comparaciones ir descartando cada parte del array.

Este algoritmo consiste en comparar el valor que buscamos con el valor que está en el centro del array; si ese valor es menor que el que buscamos, ahora buscaremos en la mitad derecha; si es mayor buscaremos en la mitad derecha. Y así hasta que lo encontremos.

Una variable (en el código es la variable *central*) se encarga de controlar el centro del array que nos queda por examinar, las variables *bajo* y *alto* controlan los límites (inferior y superior respectivamente) que nos quedan por examinar.

En caso de que el valor no se encuentre, entonces los límites *bajo* y *alto* se cruzan. Entonces el algoritmo devuelve -1 para indicar que el valor no se encontró. Esta búsqueda es rapidísima ya que elimina muchas comparaciones al ir quitando la mitad de lo que nos queda por examinar.

Código de búsqueda binaria:

```
/* parámetros:
   vector: Array que contiene todos los valores
   tamanio: Tamaño del array
   valor: valor a buscar
*/
template <class T>
int busquedaBinaria(T vector[], int tamanio, T valor){
   int central,bajo=0,alto=tamanio-1;

   while(bajo<=alto){
      central=(bajo+alto)/2;
      if(valor==vector[central]) return central;
      else if (valor<vector[central])
            /* Se busca en la mitad izquierda */
            alto=central-1;
      else
         /* Se busca en la mitad derecha */
         bajo=central + 1;
   }/* Fin del while */

   return -1; /* No se encontró */
}
```

ordenación de arrays

Se trata de una de las operaciones más típicas de la programación. Un array contiene datos sin ordenar, y tenemos que ordenarle. Es, por supuesto, una de las tareas fundamentales de la programación.

método por inserción directa

Consiste en que el array está en todo momento ordenado y cada vez que se añade un elemento al array, éste se coloca en la posición que corresponda.

Para ello habrá que desplazar elementos. Si el array es:

2 3 4 6 8 9 *vacío* *vacío* *vacío* *vacío* *vacío* *vacío*

y ahora añadimos el número 5, el resultado será:

2 3 4 5 6 8 9 *vacío* *vacío* *vacío* *vacío* *vacío*

se han tenido que desplazar los valores 6,8 y 9 para hacer hueco al 5 en su sitio.

Para un array de enteros la función que inserta un nuevo valor en dicho array sería:

```
/* Parámetros de la función:
   vector: Array que contiene los datos ya ordenados
   tamanio: Tamaño actual del array, es el tamaño de los
         datos ordenados. cero significa que aún no se ha
         insertado ningún valor
   valor: El nuevo valor a insertar en el array
*/
template <class T>
void insercion(T vector[], int  tamanio, T valor){
   /* valor es el valor que se inserta */
   int i,pos=0;
   bool enc=true; /*indica si hemos encontrado la posicion del
              elemento a insertar*/

   if (tamanio==0) /*Si no hay datos añadimos este y salimos*/
         vector[0]=valor;
   else{ /* Hay datos */
         /*Localizar la posición en la que irá el nuevo valor*/
         while(enc==false && pos<tamanio ){
           if(valor<vector[pos]) enc=true;
           else pos++;
         }
         /* Desplazar los elementos necesarios para hacer hueco
               al nuevo */
        for(i=tamanio-1;i>=pos;i--)
              vector[i+1]=vector[i];

        /*Se coloca el elemento en su posición*/
        if(pos<=tamanio) vector[pos]=valor;  }
}
```

método burbuja

Para ordenar un array cuando ya contiene valores y éstos no están ordenados, no sirve el método anterior.

Para resolver esa situación, el algoritmo de ordenación de la **burbuja** o de **intercambio directo** es uno de los más populares. Consiste en recorrer el array *n* veces, donde *n* es el tamaño del array. De modo que en cada recorrido vamos empujando hacia el principio del array los valores pequeños (si ordenamos en ascendente). Al final, el array está ordenado. La función que ordenaría mediante el método de la burbuja un array de enteros de forma ascendente.

Código:

```
template <class T>
void burbuja(T vector[], int tamanio){
  int i,j;
  T aux; /* Para intercambiar datos del array */
  for(i=0;i<tamanio;i++){
        for(j=0;j<tamanio-1;j++){
              if(vector[j]>vector[j+1]){
                    aux=vector[j];
                    vector[j]=vector[j+1];
                    vector[j+1]=aux;
              }
        }
  }
}
```

método de ordenación rápida o *quicksort*

Es uno de los métodos más rápidos de ordenación. Se trata de uno de los algoritmos más famosos de la historia. Fue desarrollado por **Sir Charles Anthony Richard Hoare** en 1960 utilizando recursividad, técnica de la que sin duda es uno de los ejemplos más sobresalientes. Hay versiones iterativas para mejorar su eficiencia (siempre son más rápidos los algoritmos iterativos).

El funcionamiento se basa en estos pasos:

[1] Elegir un elemento del array llamado *pivote*

[2] Organizar los elementos del pivote de modo que los menores queden a su izquierda y los mayores a su derecha.

[3] Llamar recursivamente a la función para cada lado del pivote

Lógicamente la elección del pivote puede acelerar o retrasar el algoritmo. Un mal pivote causa un algoritmo lento. Por ello se usan algunas técnicas para elegir el pivote. Pero en general se suele coger cualquiera porque las posibilidades de un mal pivote son pequeñas y eso simplifica la codificación del algoritmo.

El algoritmo es:

```
/* Ordena el array lista.
   Parámetros:
        lista: el array que contiene los valores
        inicio: índice inicial del array
        fin: índice final del array
*/
template <class T>
void quicksort(T lista[],int inicio,int fin){
    T pivote=lista[fin]; //Elección del pivote.
  //no responde a ningún planteamiento de optimización, se
  //coge el último elemento del array

    T aux; //Auxiliar para hacer el intercambio de elementos
    int i=inicio, j=fin-1; //Índices auxiliares inicializados
    bool continuar=true; //Centinela

    if(inicio<=fin) { //Sólo si los índices no se cruzan continuamos
        //Clasificación de la lista
        while(continuar){
             //colocación del índice i en el primer
             //elemento igual o mayor que el pivote
            while (lista[i]<pivote) i++;
             //colocación del índice j en el primer
             //elemento igual o menor que el pivote
            while (lista[j]>pivote) j--;
            if(i<j){//Si los índices no se cruzan
                aux=lista[i];
                lista[i]=lista[j];
                lista[j]=aux;
            }
            else continuar=false;
        }
//Colocación del pivote en el sitio que le corresponde en el array
        lista[fin]=lista[i];
        lista[i]=pivote;

//Ordenación mediante recursividad de los trozos que falta ordenar
        quicksort(lista, inicio,i-1);
        quicksort(lista, i+1,fin);
    } }
```

Para entender mejor el funcionamiento del algoritmo, supongamos que tenemos este array:

3	8	9	2	7	5	6	**4**

El pivote es el 4. El índice *i* avanzaría hasta el 8 y el *j* hasta el 2.

3	**8**	9	**2**	7	5	6	**4**

Se intercambiarían ambos elementos:

3	**2**	9	**8**	7	5	6	**4**

Después el índice *i* avanzaría hasta el 9 y el *j* hasta el 3. Como en este caso los índices se cruzan, se sale del bucle principal. Al salir se coloca el pivote en su sitio (que será el punto señalado por el índice i. Es decir se intercambian el 4 y el 9:

3	2	**4**	8	7	5	6	**9**

Tras este paso se llamaría a la función usando el primer trozo:

3	2

Se empezaría el algoritmo. Y se colocaría el índice *i* sobre el elemento 3 (el j se saldría del array). Habría un intercambio entre el 2 y el 3. En la siguiente llamada recursiva, no ocurriría nada ya que se cruzan los índices. La otra llamada que falta es para el trozo:

8	7	5	6	**9**

Con el que se seguiría el proceso (el pivote es malo ya que es el 9, pero funcionaría).

Hay versiones iterativas del algoritmo que mejoran su rendimiento, pero no se expondrán en estos apuntes.

arrays multidimensionales

En muchas ocasiones se necesitan almacenar series de datos que se analizan de forma tabular. Es decir en forma de tabla, con su fila y su columna. Por ejemplo:

	columna 0	columna 1	columna 2	columna 3	columna 4	columna 5
fila 0	4	5	6	8	9	3
fila 1	5	4	2	**8**	5	8
fila 2	6	3	5	7	8	9

Si esos fueran los elementos de un array de dos dimensiones (por ejemplo el array *a*) el elemento resaltado sería el **a[3][1]** (cuyo valor es 8)

La declaración de arrays de dos dimensiones se realiza así:

```
int a[3][6];
```

a es un array de tres filas y seis columnas. Otra forma de declarar es inicializar los valores:

```
int a[][]={{2,3,4},{4,5,2},{8,3,4},{3,5,4}};
```

En este caso *a* es un array de cuatro filas y tres columnas.

Al igual que se utilizan arrays de dos dimensiones se pueden declarar arrays de más dimensiones. Por ejemplo imaginemos que queremos almacenar las notas de 6 aulas que tienen cada una 20 alumnos y 6 asignaturas. Eso sería un array de una dimensión de 720 elementos, pero es más claro el acceso si hay tres dimensiones: la primera para el aula, la segunda para el alumno y la tercera para la asignatura.

En ese caso se declararía el array así:

```
int a[6][20][6];
```

Y para colocar la nota en el aula cuarta al alumno cinco de la asignatura 3 (un ocho es la nota):

```
a[3][4][2]=8;
```

Cuando un array multidimensional se utiliza como parámetro de una función, entonces se debe especificar el tamaño de todos los índices excepto del primero. La razón está en que de otra forma no se pueden calcular los índices correspondientes por parte de la función. Es decir:

```
void funcion(int a[][]);
```

Eso es incorrecto, en cuanto hiciéramos uso de un acceso a *a[2][3],* por ejemplo, no habría manera de saber en qué posición de memoria está ese valor ya que sin saber cuántas columnas forman parte del array, es imposible determinar esa posición. Lo correcto es:

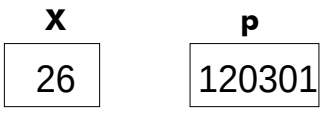
```
void funcion(int a[][5]);
```

[6.3] punteros

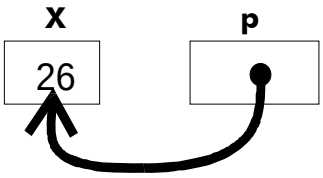
[6.3.1] introducción

Se trata de una de las herramientas más importantes de C++. Los punteros (o *apuntadores*) son variables cuyo contenido es la dirección de otra variable. Es decir son variables que señalan al contenido de otras variables.

En general una variable contiene un valor que es con el que se opera, en el caso de los punteros no es un valor directo sino que es la dirección de memoria en la que se encuentra el valor.



Si x es una variable normal y p es un puntero, 120301 será una dirección de memoria. Supongamos que esa dirección de memoria es la de la variable x. Entonces p apunta a x:



Para esta labor se usan dos operadores:

- `*` Devuelve el contenido de un puntero
- `&` Devuelve la dirección de una variable

[6.3.2] declaración de punteros

Un puntero señala a una dirección de memoria. Esa dirección contendrá valores de un determinado tipo. Por ello al declarar un puntero hay que indicar de qué tipo es el puntero; o, lo que es lo mismo, el tipo de valores a los que apunta. La sintaxis es:

```
tipo *nombrePuntero;
```

El asterisco es el que indica que lo que tenemos es un puntero. El *tipo* es el tipo de valores a los que señala el puntero. Ejemplo:

```
int *ptrSuma;
```

ptrSuma es un puntero a valores enteros.

[6.3.3] operaciones con punteros

Para asignar valores a un puntero muchas veces se utiliza el operador **&** (ya comentado en los temas anteriores) que sirve para obtener la dirección de una variable. Ejemplo:

```
int *ptrSuma, suma=18;
ptrSuma=&suma;
```

Desde la última instrucción *ptrSuma* contendrá la dirección de la variable *suma*; o dicho de otra forma: *ptrSuma* apunta a la variable *suma*.

Cuando un puntero tiene un determinado valor (apunta a una determinada dirección), a menudo desearemos acceder al contenido de la dirección a la que apunta. Para ello se utiliza el operador "*****", que permite acceder al contenido del puntero.

Ejemplo:

```
int *ptrSuma, suma=18;
ptrSuma=&suma;
cout<<*ptrSuma; //Escribe 18
```

Ese mismo operador se utiliza para cambiar el valor de la variable a la que apunta el puntero.

Ejemplo:

```
int *ptrSuma, suma=18;
ptrSuma=&suma;
*ptrSuma=11;
cout<<suma; // Escribe 11, ahora suma vale 11
```

Esto es lo interesante de los punteros, a través de los punteros se modifican otras variables. Incluso sería válido:

```
cin>>*ptrSuma; //Lo que lea se almacena en suma
```

Cuando se desea que un puntero no señale a nada en particular, se puede asignar al valor **NULL** o al valor **0**. El significado es el mismo. Por cierto, en C++ el valor 0 es el único entero que se puede asignar a un puntero (sin embargo en C sí era posible asignar a una dirección en concreto).

[6.3.4] punteros como argumento

Además del paso por referencia y por valor, una tercera forma en C++ de pasar parámetros a una función, es mediante el uso de punteros. Esa forma es la misma que la que posee el lenguaje C (aunque no es la más aconsejable, hay veces en las que es muy interesante).

Gracias a ello tendríamos tres versiones de una función que sirva para calcular el cubo de un número. La primera usando el paso por valor)en este caso sería la idónea:

```
double cubo(double n){
   return n*n*n;
}
```

Para utilizar la función la llamada sería:

```
int a=cubo(13); //a vale 13³
```

La segunda usando el paso por referencia de C++:

```
void cubo(double &n){
   n=n*n*n;
}
```

La llamada sería:

```
int a=13;
cubo(a);
```

La tercera forma usando punteros (o paso por referencia del lenguaje C) es:

```
void cubo(double *ptrX){
   *ptrX=(*ptrX) * (*ptrX) * (*ptrX);
}
```

A esta función se la pasa un puntero a un número *double* y el contenido de ese puntero es multiplicado por si mismo tres veces (los paréntesis no son necesarios, pero si aclaran mejor la función. La llamada ala función podría ser:

```
double a=8.0;
cubo(&a);
```

en esta llamada *a* pasa su dirección a la función *cubo*. De esta forma cubo tiene un puntero a *a* con el que modifica su contenido.

[6.3.5] punteros y constantes

El uso de la palabra **const** ya se ha discutido en temas anteriores. Se coloca esta palabra delante de declaraciones de variables, para indicar que no pueden cambiar su valor. Si es delante de un parámetro, indican que el parámetro no es modificable dentro de la función en la que se declaran.

En el caso de los punteros podemos tener estos casos relacionados con constantes:

- **Puntero normal que señala a una constante.** Es perfectamente posible, lo único que ocurre es que mientras el puntero señale a la constante, no podrá

modificar el contenido. Es decir, la instrucción *ptr=7* no sería válida. Sin embargo en cualquier momento puede señalar a otra variable (o constante).

- **Puntero constante señalando a una variable.** Si el puntero fue definido anteponiendo la palabra **const** en la declaración, significará que el puntero no puede modificar su valor. El puntero señalará siempre a la misma variable, sobre la que sí podrá modificar su valor haciendo uso del operador *.
- **Puntero constante señalando a una variable.** Realmente es el mismo caso, sólo que al señalar a una constante no podrá modificar el valor de la misma (como es lógico).

[6.3.6] aritmética de punteros

A los punteros se les puede aplicar operaciones aritméticas sin que ello produzca errores de ningún tipo. Pero hay que tener en cuenta que esta aritmética es un tanto peculiar.

De hecho en este programa:

```
int *p;
p=0; // p apunta a la dirección 0
cout<<p; // Escribe 0
cout<<p+2; // Escribe 0x8
```

Es toda una sorpresa la última línea: lo esperable sería que *p+2* sumara dos al resultado, pero suma 8, aunque en muchos compiladores sumaría 4. De hecho lo que suma son dos veces el tamaño de los enteros (se puede saber esta cifra usando **sizeof(int)**). El hecho de que el número salga en hexadecimal es labor del objeto **cout** que muestra los punteros de esa forma (si lo convirtiéramos a enteros no saldría bien el resultado).

Es decir si *p* apunta a un entero, *p+1* apunta al siguiente entero de la memoria. De hecho, la expresión *p++* hace que se incremente *p* para que apunte al siguiente entero en memoria. Lo mismo vale para punteros de cualquier tipo de datos que no sean enteros.

También se pueden restar punteros. El resultado es el número de bytes que separan aun puntero de otro. Por ejemplo:

```
int *p1, *p2, x=8,y=7;
p1=&x;
p2=&y;
cout<<"Enteros que separan a p1 de p2="<<p2-p1;
```

Esa diferencia es la correspondiente al número de posiciones en memoria para enteros que separan a *p1* de *p2.* Es decir, si la diferencia es de 2 significará que hay 8 bytes entre los dos punteros (suponiendo que un entero ocupe 4 bytes).

[6.3.7] punteros y arrays

La relación entre punteros y arrays es muy directa, ya que los arrays son referencias a direcciones de memoria en la que se organizan los datos. Tanto es así, que **se puede hacer que un apuntador señale a un array**. Por ejemplo:

```
int A[]={2,3,4,5,6,7,8};
int *ptr;
ptr=A; //Es perfectamente válido
```

Esa última instrucción hace que el puntero señale al primer elemento del array. Desde ese momento se pueden utilizar los índices del array o la aritmética de punteros para señalar a determinadas posiciones del array. Es decir: es lo mismo *A[3]* que *ptr+3*. De esta forma también son equivalentes estas instrucciones:

```
x=A[3]; x=*(ptr+3);
```

Pero la analogía entre punteros y arrays es tal, que incluso se pueden utilizar índices en el puntero:

```
x=ptr[3];
```

También sería válido (con lo cual se observa ya la gran equivalencia entre arrays y punteros):

```
cout<<*A; //mostraría el contenido del primer elemento del array
cout<<*(A+2); //mostraría el tercer elemento del array
```

Una de las grandes utilidades de esta capacidad es utilizar esta equivalencia para pasar los arrays a las funciones. Sin embargo no es válido:

```
A++; //Error
```

El array no se puede mover de sitio (debe señalar siempre a la misma dirección).

[6.3.8] arrays de punteros

Su uso no es habitual pero permiten solucionar problemas complejos. Como se verá en el siguiente apartado el caso clásico es el uso de arrays de strings. Por ejemplo la declaración:

```
int *a[4];
```

Declara un array de cuatro punteros a enteros. Así, estas instrucciones son válidas:

```
int x=9, y=12, z=19;
int *ptr;
a[0]=&x;
```

```
cout<<*a[0]<<endl; //Escribe 9
cout<<*a<<endl; //escribe la dirección de x
cout<<**a<<endl; //Escribe 9
a[1]=&y;
ptr=a[0];
ptr++; //Señala al segundo elemento del array
cout<<*ptr<<endl; //Escribe 12
cout<<*a[1]<<endl; //Escribe 12
```

[6.4] cadenas de caracteres

[6.4.1] introducción

Las cadenas de caracteres son los textos formados por varios caracteres. En C++ las cadenas (o **strings**) son arrays de caracteres, pero que tienen como particularidad que su último elemento es el carácter con código cero (es decir **'\0'**).

En sentido estricto una cadena de C++ es un puntero al primer carácter de un array de caracteres que señala al primer carácter. Sólo que esa lista de caracteres finaliza en el código **0**.

[6.4.2] declarar e inicializar cadenas

Las cadenas de caracteres no son exactamente arrays de caracteres (es importante tener en cuenta que las cadenas terminan con el carácter nulo).

Por ello su declaración puede ser una de las siguientes:

```
char nombre1[]={'J','u','á','n','\0'};
char nombre2[]="Juán";
char *nombre3="Juán";
```

Las dos primeras declaraciones, declaran un array cuyo contenido es en ambos casos el texto *Juán* finalizando con el carácter 0. La segunda definición creará el texto *Juán* (con su terminación en nulo) en algún lugar de memoria, a cuyo primer elemento señala el puntero *nombre3.*

La tercera forma de declarar e inicializar cadenas, puede tener problemas en algunos compiladores (como **Dev C++** por ejemplo), ya que la cadena se coloca en zonas de la memoria en las que no se pueden modificar los datos (es decir lo graba como literal. En esos compiladores es mejor inicializar cadenas utilizando siempre un array (usando la segunda forma en el ejemplo).

Al utilizar arrays para guardar cadenas, convienen que éstos sean lo suficientemente grandes, de otro modo tendríamos problemas ya que al sobrepasar el array invadiríamos el espacio de otras variables. Si no queremos dar valores iniciales a la cadena, conviene declararla en forma de array con tamaño suficiente. Por ejemplo:

```
char cadena[80];
```

Lo cual hace que se reserven en memoria 80 caracteres para esa variable.

Los punteros que señalan a cadenas se utilizan mucho ya que, puesto que las cadenas finalizan con el delimitador cero, en casi todos los algoritmos con textos, son los punteros los que recorren y operan con los caracteres.

```
char cadena[80]="Esto es una prueba";
char *p=cadena; /* El puntero señala al primer carácter de la
                   cadena */
cout<<p; //escribe Esto es una prueba
```

[6.4.3] leer cadenas desde teclado

En C++ no es problemático. Se pueden leer los strings igual que se leen el resto de variables. EL objeto *cin* es el ideal para realizar esta lectura:

```
char texto[20];
cin>>texto;
```

El problema es si leemos más caracteres de los que la variable puede almacenar. Para ello se utiliza el modificador **setw** (que pertenece también al espacio *std*). Se haría:

```
cin>>setw(20)<<texto;
```

Como mucho lee 19 caracteres (el veinte es para almacenar el final de la cadena), el resto de los caracteres escritos, serán ignorados.

Pero hay un problema con la lectura de textos y es que el objeto **cin** cuando lee un texto con espacios en blanco cree que estamos leyendo dos variables, por ello es más interesante usar la función **getline** del propio objeto **cin**. Esta función requiere la variable **string** en la que almacenaremos el texto y el tamaño máximo de la lectura (lo que comúnmente se llama *búfer de lectura*). Ejemplo:

```
char str[80];
//Lectura con cin
cin>>str;
//Lectura con getline
cin.getline(str,80);
```

Si con ese código escribimos el texto *Hola amigos*. En el primer caso sólo guardaría la palabra *Hola*, el resto se lo llevaría el siguiente *cin*. En el segundo caso en *str* tendremos todo el texto almacenado.

[6.4.4] comparar cadenas

La comparación entre cadenas es problemática (como lo es también la comparación entre arrays). La razón está en que aunque dos cadenas tengan el mismo contenido, como sus direcciones serán distintas, la comparación con el operador "==" saldrá falsa (ya que lo que hace es comparar las direcciones).

La comparación de cadenas necesita comparar carácter a carácter o utilizar la función **strcmp** de la librería **cstring** (hay que incluirla en el código para poder utilizar esta función) que devuelve 0 si dos cadenas tienen el mismo contenido, -1 si en el orden de la tabla ASCII la primera es menor que la segunda y 1 si la segunda es menor que la primera.

Hay que tener en cuenta que **strcmp** no comprueba las mayúsculas y que las letras como la eñe o las acentuadas causan problemas al comparar el orden alfabético (lo que no la hace muy apta para el idioma español.

[6.4.5] funciones y cadenas

En el caso de las funciones que reciben como parámetro cadenas, siempre utilizan punteros (**char ***) ya que permiten un mejor recorrido de los mismos. Por ejemplo esta función se encargaría de realizar una copia de una cadena en otra:

```
void copiaCadena(char *cadena1, const char *cadena2){
  while(*cadena2!=0){
        *cadena1=*cadena2;
        cadena1++;
        cadena2++;
  }
  *cadena1=0;
}
```

La palabra **const** sirve para indicar al compilador que la *cadena2* no puede ser modificada en el interior de la función. Es importante poner el delimitador cero al final de la primera cadena (tras haber copia el texto, es la instrucción ***cadena1=0**).

La llamada a esa función podría ser:

```
char s1[50]; /*s1 reserva 50 caracteres, eso es importante*/
char s2[]="Prueba";
copiaCadena(s1,s2);
puts(s1); /* escribe Prueba */
```

Una función puede devolver un puntero que señale a un carácter (una función que devuelve **char ***), pero si una función crea una cadena, al salir de la función, esta cadena se elimina. Esto significa que una **función no debe devolver la dirección de una cadena creada dentro de la función.**

[6.4.6] arrays de cadenas

Ya se ha comentado que una cadena es un array de caracteres en el que hay delimitador de fin de cadena que es el código cero . Luego todo lo comentado para los arrays y los punteros vale también para las cadenas.

La declaración:

```
char *s[10];
```

Declara un array de 10 punteros a caracteres (10 cadenas). Para inicializar con valores:

```
char *s[4]={"Ana","Pablo","Julián","Roberto"};
cout<<s[2]; /* Escribe Julián */
```

No obstante esta declaración también genera problemas en numerosos compiladores (otra vez en **Dev C++** por ejemplo) ya que al modificar esas cadenas, como se almacenan en la zona de constantes, tendríamos problemas. Es mejor (si se van a modificar los textos) declarar reservando memoria, por ejemplo:

```
char s[4][50]={"Ana","Pablo","Julián","Roberto"};
cout<<s[2]; /* Escribe Julián */
```

La declaración en este caso crea un array bidimensional de 4 x 50 caracteres. Eso se puede interpretar como cuatro arrays de caracteres de 50 caracteres como máximo cada uno. Así los textos utilizados en el ejemplo se copian en el área reservada por el array y sí se podrían modificar.

Si deseamos un puntero señalando a la segunda cadena del array:

```
char *p=s[2];
cout<<p; /*También escribe Julián */
```

Eso es válido ya que *s[2]* es el tercer elemento del array *s*, es un array de 50 caracteres. Es decir, *s[2]* es la tercera cadena del array de cadenas *s.*

[6.4.7] funciones de uso con cadenas

funciones de manipulación de caracteres

Están en la librería **cctype** y trabajan con caracteres, pero son muy útiles para usar en algoritmos para cadenas. Las funciones son:

prototipo de la función	descripción
int isdigit(**int** carácter)	Devuelve verdadero (un valor distinto de 0) si el carácter es un dígito de 0 a 9
int isalpha(**int** carácter)	Devuelve verdadero (un valor distinto de 0) si el carácter es una letra

prototipo de la función	descripción
int isalnum(**int** carácter)	Devuelve verdadero si el carácter es una letra o un número.
int islower(**int** carácter)	Devuelve verdadero si el carácter es una letra minúscula (según el alfabeto inglés).
int isupper(**int** carácter)	Devuelve verdadero si el carácter es una letra mayúscula (según el alfabeto inglés).
int tolower(**int** carácter)	Convierte el carácter a minúsculas
int toupper(**int** carácter)	Convierte el carácter a mayúsculas
int isspace(**int** carácter)	Devuelve verdadero si el carácter es el espacio en blanco
int iscntrl(**int** carácter)	Devuelve verdadero si el carácter es de control
int ispunct(**int** carácter)	Devuelve verdadero si el carácter es de puntuación
int isprint(**int** carácter)	Devuelve verdadero si el carácter no es de control
int isgraph(**int** carácter)	Devuelve verdadero si el carácter no es de control y no es el espacio

Todas estas funciones son muy interesantes, pero sólo funcionan correctamente con el lenguaje inglés. En el caso del español hay problemas en todas ellas por no considerar correctamente símbolos como la ñ, los acentos,…

funciones de conversión de cadenas

La librería **cstdlib** contiene varias funciones muy interesantes para procesar cadenas son:

prototipo de la función	descripción
double atof(char *s)	Convierte la cadena *s* a formato **double**. Si la cadena no contiene datos que permitan la conversión (por ejemplo si contiene texto), su comportamiento es impredecible.
int atoi(char *s)	Convierte la cadena *s* a formato **int**. Si la cadena no contiene datos que permitan la conversión (por ejemplo si contiene texto), su comportamiento es impredecible.
long atol(char *s)	Convierte la cadena *s* a formato **long**. Si la cadena no contiene datos que permitan la conversión (por ejemplo si contiene texto), su comportamiento es impredecible.

funciones de manipulación de cadenas

La librería **cstring** proporciona las funciones más interesantes para manipular cadenas:

prototipo de la función	descripción
char * strcat(char ***s1, **const char ***s2)**	Añade *s2* al final de la cadena *s1*. Devuelve la propia *s1*
char * strncat(char ***s1, **const char ***s2, **int n**)**	Añade los *n* primeros caracteres de *s2* al final de la cadena *s1*. Devuelve la propia *s1*
char * strcpy(char ***s1, **const char ***s2)**	Copia *s2* al final en *s1*. Devuelve la propia *s1*
char * strncpy(char ***s1, **const char ***s2, **int n**)**	Copia los *n* primeros caracteres de *s2* al final en *s1*. Devuelve la propia *s1*
int strcmp(const char ***s1, **const char ***s2)**	Compara *s1 con* s2. Si *s1* es mayor devuelve un valor positivo, si es menor un valor negativo y si son iguales devuelve 0 (según el código ASCII)
int strncmp(const char ***s1, **const char ***s2, **int n**)**	Compara *s1 con* s2 hasta llegar al carácter número n. Si *s1* es mayor devuelve un valor positivo, si es menor un valor negativo y si son iguales devuelve 0 (según el código ASCII)
char * strchr(char ***s1, **int carácter)	Busca la primera vez que aparece el *carácter* dentro de la cadena *s1* si le encuentra devuelve un **puntero** al mismo, sino devuelve el valor *NULL*.
char * strrchr(char ***s1, **int carácter)	Busca la última vez que aparece el carácter dentro de la cadena *s1* si le encuentra devuelve un puntero al mismo, sino devuelve el valor *NULL*.
char * strstr(char * s1**,const char *** s2)	Busca la primera vez que aparece el texto *s2* de la cadena *s1*. Si le encuentra devuelve un puntero al primer carácter de *s2* dentro de *s1*, sino devuelve el valor *NULL*.
int strlen(const char * s)	Devuelve el tamaño del texto *s*.
char * strpbrk(char * s1**, char ***s2**)**	Devuelve la primera aparición en el string apuntado por s1 de cualquier carácter contenido en s2

prototipo de la función	descripción
char * strtok(char * s,**const char *** tokens)	Divide la cadena en **tokens**, es decir, partes de la cadena delimitados por una determinada cadena. La primera llamada debe utilizar en el parámetro *tokens* el o los caracteres delimitadores. Por ejemplo en una llamada *strtok(s,",")* nos devolvería el texto de *s* que hay hasta llegar a la primera coma. En las siguientes llamadas se debe utilizar el valor **NULL**. Así si la siguiente es *strtok(NULL,",")* devolverá el trozo de texto desde la primera coma a la segunda. Y así sucesivamente. Si no hay más *tokens*, devuelve **NULL**. Es una función potentísima.

Ejemplo de uso de tokens:

```
char x[100]="Esto, es, una, funcion delimitada, con tokens";
char *aux;
aux=strtok(x,",");//La primera llamada usa x
while(aux!=NULL){
    cout<<aux<<endl;
    aux=strtok(NULL,","); //Las siguientes usan NULL
}
/* Sale:
  Esto
   es
   una
   funcion delimitada
   con tokens
*/
```

[Unidad 7] programación orientada a objetos con C++

[7.1] objetos y clases

[7.1.1] programación orientada a objetos

En los inicios de la programación ya se observaron los problemas que conlleva el hecho de programar sin un análisis previo. Los arreglos en el código costaban miles de horas y provocaban la generación de código casi ilegible.

La primera técnica de programación que intentaba paliar estos problemas fue la **programación estructurada**, que permite escribir programas con un código más claro y legible, lo que permite una fácil modificación del mismo. En esta programación todo el código se escribía utilizando las estructuras **if** y **while**.

Aunque mejoró la creación de código, el problema siguiente se planteó cuando los programas alcanzaron una extensión de miles o millones de líneas de código. Ante semejante cantidad de datos, la organización de los programas era casi imposible.

La respuesta fue la **programación modular** que permite dividir el código en diferentes módulos (o funciones) cada una dedicada a un fin. Eso permite que cada módulo se programa de forma independiente al resto facilitando el mantenimiento y funcionamiento del código.

Pero la informática moderna trajo consigo aplicaciones de gran envergadura con elementos gráficos interaccionando entre sí de forma independiente. Eso no lo pudo resolver la programación modular. Se necesitan módulos más integrados e independientes, para poder ser utilizados incluso en aplicaciones diferentes.

A esos nuevos módulos más independientes y eficientes es a lo que se llama objetos. Un objeto es un elemento distinguible en una aplicación, puede ser algo perfectamente distinguible en ella (un botón) o algo más abstracto (un error de programa).

[7.1.2] objetos

Un objeto es un elemento que agrupa datos y funciones. Es decir es una estructura que posee información estática (números, textos,…) e información dinámica (procedimientos). Por ejemplo en el caso de un objeto que represente un perro, dicho objeto posee datos (llamados atributos) como la edad, el nombre, el tamaño,… y métodos como ladrar, correr,…

En la POO una aplicación se entiende como una serie de objetos que se relacionan entre sí. Cada objeto responde a una programación previamente realizada; esta programación se realiza de forma independiente, eso nos asegura que el objeto es reutilizable (se puede usar en aplicaciones distintas).

En la programación estructurada y modular el diseño de programas se realiza pensando en una sucesión de instrucciones que van de un inicio a un fin.

En la programación orientada a objetos el diseño consiste en analizar qué objetos necesita una aplicación y cómo se relacionan entre sí. Después se pasa a realizar el código de cada objeto. El código de cada objeto se prepara sin tener en cuenta al resto de elementos del programa; eso permite que un objeto sea utilizado fácilmente en aplicaciones diferentes, gracias a la independencia del código.

En un programa los objetos se comunican entre sí lanzándose **mensajes**. Un mensaje es un método que permite que el objeto ejecute alguna acción. Por ejemplo, si tenemos el objeto de clase persona *pepe* y este quiere arrancar su coche, que es otro objeto de otra clase, podría hacerlo lanzando el mensaje *cocheDePepe.arranca().*

Cuando se crea un objeto se crea código en memoria; para que ese código sea accesible desde el programa, se utilizan **referencias** a dicho código. Las referencias en realidad son punteros al código del objeto. De modo que una referencia permitirá cambiar los atributos del objeto.

Incluso puede haber varias referencias al mismo objeto, de modo que si una referencia cambia el estado del objeto, el resto de referencias (como es lógico) mostrarán esos cambios.

Otra cuestión importante es que todo objeto en un sistema posee: una identidad, un estado y un comportamiento.

- El **estado** marca las condiciones de existencia del objeto dentro del programa. Lógicamente este estado puede cambiar. Un coche puede estar parado, en marcha, estropeado, funcionando, sin gasolina, etc.
- El **comportamiento** determina como responde el objeto ante peticiones de otros objetos. Por ejemplo un objeto conductor puede lanzar el mensaje arrancar a un coche. El comportamiento determina qué es lo que hará el objeto.
- La **identidad** indica que cada objeto es único; aunque encontremos dos objetos con los mismos contenidos. No existen dos objetos iguales. Lo que sí tenemos son referencias distintas al mismo objeto (las variables *a* y *b* pueden hacer referencia al mismo objeto).

[7.1.3] clases

Antes de poder utilizar un objeto, se debe definir su **clase.** La clase es la definición de un tipo de objeto. Al definir una clase lo que se hace es indicar como funciona un determinado tipo de objetos. Luego, a partir de la clase, podremos crear objetos de esa clase.

Por ejemplo, si quisiéramos crear el juego del parchís en Java, una clase sería la casilla, otra las fichas, otra el dado, etc., etc. En el caso de la casilla, se definiría la clase para indicar su funcionamiento y sus propiedades, y luego se crearía tantos objetos casilla como casillas tenga el juego.

Lo mismo ocurriría con las fichas, la clase **ficha** definiría las propiedades de la ficha (color y posición por ejemplo) y su funcionamiento mediante sus métodos (por ejemplo un método sería mover, otro llegar a la meta, etc., etc.,), luego se crearían tantos objetos ficha, como fichas tenga el juego.

Normalmente las clases representan cosas como:

- Cosas tangibles del mundo *real*, como sillas, libros, coches,…
- Roles de las personas como: socios, clientes, estudiantes,…
- Eventos: llegada, salida, requerimiento,…
- Interacciones: encuentro, intersección, transacción

[7.1.4] diferencias entre clase y objeto

Una clase define el funcionamiento de los objetos. Es decir, la clase es la plantilla que cumplen todos los objetos de esa clase. Es decir, la clase marca el tipo de objeto. Más exactamente marca la *idea* de objeto.

La clase *Coche* definiría la idea o modelo general de coche que tenemos en la cabeza. Esa idea abarca a todos los coches; en esa idea está el hecho de que los coches tienen cuatro ruedas, motor, consumen combustible, aceleran, frenan,… Sin embargo si miramos por la ventana y vemos un coche, ese coche no es ya una clase, **es un objeto.**

Es decir, cuando le preguntamos a una persona ¿qué es un coche? nos responderá indicándonos la clase. Pero si montamos en un coche, estaremos montados en un **ejemplar** (también se le llama **instancia**, por la traducción extraña de la palabra inglesa **instance**), es decir un objeto de clase coche.

[7.2] propiedades de la POO

- **Encapsulamiento**. Una clase se compone tanto de variables (**propiedades**) como de funciones y procedimientos (**métodos**). En la POO más estricta (y aconsejable) toda variable pertenece a una clase (es decir, no hay variables, sino propiedades).
- **Ocultación**. Hay una zona oculta al definir la clases (**zona privada**) que sólo es utilizada por esa clases y por alguna clase relacionada. Hay una **zona pública** (llamada también **interfaz** de la clase) que puede ser utilizada desde cualquier parte del código (desde cualquier otra clase).
- **Polimorfismo**. Cada método de una clase puede tener varias definiciones distintas. En el caso del parchís: *partida.empezar(4)* empieza una partida para cuatro jugadores, *partida.empezar(ROJO, AZUL)* empieza una partida de dos jugadores para los colores rojo y azul; estas son dos formas distintas de emplear el método empezar, que es polimórfico.
- **Herencia**. Una clase puede heredar propiedades y métodos de otra. Se estudiará con detalle en el tema siguiente.

[7.3] creación de clases

[7.3.1] introducción

Las clases son las plantillas para hacer objetos. Una clase sirve para definir una serie de objetos con propiedades (atributos), comportamientos (operaciones o métodos), y semántica comunes. Hay que pensar en una clase como un molde. A través de las clases se obtienen los objetos en sí.

Es decir antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición incluye:

- **Sus atributos.** Es decir, los datos miembros de esa clase. Los datos pueden ser públicos (accesibles desde otra clase) o privados (sólo accesibles por código de su propia clase. También se las llama campos.
- **Sus métodos.** Las funciones miembro de la clase. Son las acciones (u operaciones) que puede realizar la clase.
- **Código de inicialización**. Para crear una clase normalmente hace falta realizar operaciones previas (es lo que se conoce como el **constructor** de la clase).
- **Código de finalización**. Código a ejecutar cuando cada objeto de esa clase desaparezca de memoria. Es lo que se conoce como **destructor** de la clase.

[7.3.2] notación UML

UML es la abreviatura de **Universal Modelling Language** (*Lenguaje De Modelado Universal*) que define una serie de esquemas diseñados para facilitar la tarea de diseñar aplicaciones informáticas.

El organismo responsable de UML es el **OMG (Objects Management Group,** *Grupo de Administración de Objetos*) que es un organismo sin ánimo de lucro que pretende estandarizar la programación orientada a objetos.

El diagrama UML que permite representar clases se llama precisamente **diagrama de clases.** En este diagrama las clases se representan usando esta forma:

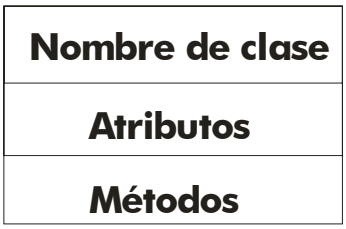


Ilustración 10, Clase en notación UML

Desde ese punto de vista una clase se compone de atributos y métodos (u **operaciones**). Los atributos marcan las propiedades de una clase. Para indicar los atributos, en las clases se pone el nombre del atributo seguido de dos puntos y el tipo de datos del atributo (el tipo de datos se suele indicar de forma conceptual: entero, cadena, lógico,…, pero es más habitual indicarlo en la forma de C++. Así se hará en esta guía).

En los métodos se indica el nombre, los parámetros entre paréntesis (indicado su tipo de datos) y tras el paréntesis se indica el tipo de datos que devuelve el método. Por ejemplo:

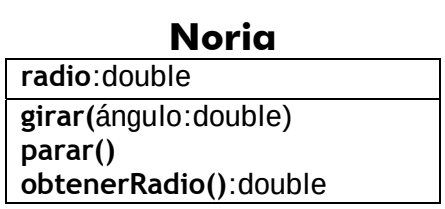


Ilustración 11, Clase Noria bajo notación UML

En los diagramas de clases a veces se representan las clases de forma simplificada. En esta forma sólo aparece el nombre de la clase en un cuadrado sin indicar:

Ilustración 12, representación simplificada de una clase en UML

En el caso de los objetos, se representan igual, sólo que en la parte dedicada al nombre se indica el nombre del objeto seguido de **::** y del nombre de la clase a la que pertenece el objeto. Los objetos casi siempre se representan de forma simplificada:

noria1::Noria

Ilustración 13, representación simplificada de un objeto en UML. *noria1* **es un objeto de clase** *Noria*

Incluso al representar objetos a veces se puede evitar el nombre e indicar sólo su clase:

::Noria

Ilustración 14, representación simplificada de un objeto de clase *Noria* **en UML**

Las clases se relacionan unas con otras mediante **asociaciones**, que indican la relación existente entre las clases. En este manual no se va a comentar como se representan dichas asociaciones, pero cualquier diagrama de clases las utiliza a fin de definir el funcionamiento del programa correctamente.

[7.3.3] creación de clases en C++

La creación de clases en C++ consta de dos partes:

- **Definición de la interfaz de la clase.** Que consiste en indicar las propiedades y métodos de la clase, sin especificar el código de los métodos, simplemente indicando la cabecera de los mismos.
- **Implementación de los métodos.** En esta parte se define el código de los métodos. Lo ideal es que esta fase se programe de forma independiente (incluso en un archivo distinto).

[7.3.4] definición de la interfaz de clase

Lo primer al crear la clase es escribir el código que declara la clase. Eso se hace de este modo:

```
class NombreDeClase {
   public:
        propiedades y cabeceras de métodos públicos
   private:
        propiedades y cabeceras de métodos privados
};
```

De las propiedades se indica el tipo y el nombre de la propiedad. De los métodos (que podemos entender que son funciones, pero funciones que forman parte de una clase, es decir: **funciones miembro**) se indica su cabecera.

Por ejemplo:

```
class Noria{
  private:
        double radio;
  public:
        void girar(double angulo);
        void parar();
        double obtenerRadio();
};
```

Nota: Es muy importante observar que la declaración de una clase en C++ se debe cerrar con el cierre de la llave y un punto y coma. Es decir con **};** el punto y coma aquí es obligatorio.

[7.3.5] alcance de las propiedades y métodos

Una clase está programada para ser usada en otros archivos de código. Desde este código se puede acceder a las propiedades y métodos de los objetos creados en la clase, sin embargo podemos evitar que algunas de éstas propiedades y métodos no sean accesible desde dicho código.

Para ello se utiliza la zona **private** de la interfaz de clase. Lo declarado dentro de **private** sólo es accesible dentro de la propia clase, pero no en código externo a la misma.

Una norma importante es que una clase debe ocultar la mayor parte de propiedades y métodos. Cuánto más se oculte mejor, así sólo dejaremos visible a otras clases lo fundamental. Esto facilita a otros programadores el uso de nuestras clases.

La zona **public** sirve para definir operaciones y propiedades visibles para cualquier clase.

En los diagramas UML las propiedades y métodos públicos se indican con un signo *más*. Los privados con un signo *menos*. Ejemplo:

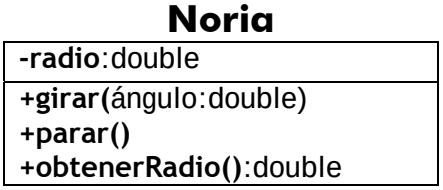


Ilustración 15, Clase Noria bajo notación UML

En C++ por defecto se entiende que los métodos y propiedades son privados por defecto, es decir que si no se indica nada se entenderá como **private**, por lo que la clase Noria comentada antes se puede escribir de esta forma (aunque es menos recomendable, por menor claridad):

```
class Noria{
        double radio;
  public:
        void girar(double angulo);
        void parar();
        double obtenerRadio();
};
```

El radio es privado, lo demás es público.

[7.3.6] implementación de los métodos

En la declaración de la interfaz sólo se indican los miembros de la clase. La parte de implementación de los métodos se realiza una vez cerrada la interfaz.

Los métodos se programan igual que las funciones; de hecho son funciones, sólo que pertenecientes a una clase concreta. Al implementar una función debemos indicar el prototipo de la misma, sólo que antes del nombre del método se pone el nombre de la clase, después los símbolos de resolución de ámbito **::** (dos veces dos puntos) y después el nombre del método. Tras el prototipo se escribe (entre llaves) el código del método.

El operador de resolución de ámbito (**::**) permite indicar a qué clase pertenece el método que estamos implementando. También (como se verás más adelante) sirve para acceder a variables y métodos estáticos.

Ejemplo:

```
//declaración
class Noria{
    private:
        double radio;
    public:
        void girar(double angulo);
        void parar();
        double obtenerRadio();
};
//implementación
void Noria::girar(double angulo){
    radio+=angulo;
}

void Noria::parar(){
```

```
    radio=0;
}

double Noria::obtenerRadio(){
    return radio;
}
```

Para facilitar aún más la modificación de los programas, la declaración y la implementación de una clase suelen ir en archivos separados. La cabecera no suele cambiar y la implementación, por ello es mucho mejor separar ambas partes.

La declaración de la clase se suele realizar en un archivo de cabecera. Los archivos de cabecera tienen extensión **.h**

Por el contrario la implementación de una clase se coloca en un archivo de extensión **.cpp** Este archivo carga la cabecera mediante el uso de una directiva **include.** Ejemplo**:**

```
//Archivo Noria.h
class Noria{
    private:
        double radio;
    public:
        void girar(double angulo);
        void parar();
        double obtenerRadio();
};
//Archivo Noria.cpp
#include "Noria.h"

void Noria::girar(double angulo){
    radio+=angulo;
}

void Noria::parar(){
    radio=0;
}
double Noria::obtenerRadio(){
    return radio;
}
```

En los programas en los que se deseen utilizar norias, tenemos que compilar el código, añadiendo en la compilación el código de *Noria.cpp* además en ese código también se colocará el *#include "Noria.h"*

[7.4] creación de objetos

[7.4.1] crear objetos

A la hora de utilizar objetos en un programa, tenemos que incluir en primer lugar el archivo de cabecera con la declaración de la clase y asegurarnos que se compila el código de la clase del objeto que queremos utilizar con nuestro código compilado.

Crear objetos es simplemente indicar el nombre de la clase del objeto seguido del nombre del objeto (es decir se crean objetos igual que se declaran variables). Ejemplo:

```
Noria noria1;
```

Hoy en día es fundamental tener en cuenta que los nombres de las clases se ponen con la primera inicial en mayúsculas, mientras que el nombre de los objetos, propiedades y métodos se ponen siempre con todas las letras en minúsculas. Aunque no es obligatorio es una norma aceptada por todos los programadores.

Si el nombre de una clase o de un objeto consta de varias palabras, se suele poner la primera inicial de cada palabra en mayúsculas. Por ejemplo: *NoriaGiratoria* o *miPrimeraNoria.* Aún en este caso hay que observar que el hecho de que la primera letra sea mayúscula indica **nombre de clase**.

[7.4.2] uso de las propiedades y los métodos de un objeto

Una vez creado un objeto podemos acceder a los datos y métodos públicos del mismo. Para ello se pone el nombre del objeto seguido de un punto y del nombre de la propiedad o método. Ejemplo:

```
noria1.parar();
noria1.girar(45.2);
cout<<noria1.obtenerRadio(); //Éscribirá 45.2
```

[7.4.3] punteros y arrays a objetos

Igual que se crean punteros a números enteros o a Strings, podemos crear punteros y arrays a objetos:

```
Noria noria[10], //array de 10 norias
  *pNoria; //Puntero a una noria
noria[3].parar(); //paramos la tercera noria del array
pNoria=&noria[2]; //el puntero señala a la tercera noria
pNoria->parar();
cout<<pNoria->obtenerRadio();
```

En el caso de los punteros a objetos, si queremos utilizar una propiedad o un método desde un puntero, tenemos que utilizar el operador **->** en lugar del punto. Ese operador evita utilizar expresiones del tipo:

```
cout<<(*pNoria).obtenerRadio();
```

Mucho más enrevesadas.

[7.5] conceptos avanzados sobre implementación

[7.5.1] evitar múltiples inclusiones de la cabecera de una clase

Supongamos que en un archivo implementamos una clase llamada *Feria* y que esa clase utiliza norias. En ese caso tendremos que añadir la directiva *#include noria.h*

Si en otro archivos queremos implementar otra clase que también utiliza norias, lógicamente haremos lo mismo. Esto puede provocar que se incluya dos veces el mismo código lo que daría lugar a un error del tipo "*clase redefinida".*

Para evitar esto se utiliza una directiva que permite definir una constante de modo que si la constante no está definida, se define la misma y se ejecuta el código del archivo. Si la constante ya está definida, entonces no lee el archivo.

Sintaxis:

```
#ifndef CONSTANTE
#define CONSTANTE
  //código de la clase
#endif
```

Ejemplo:

```
//Archivo Noria.h
#ifndef NORIA_H
#define NORIA_H
class Noria{
    public:
        void girar(double angulo);
        void parar();
        double obtenerRadio();
    private:
        double radio;
};
```

```
#endif
```

En el código anterior, si no se ha definido la constante *NORIA_H* entonces se ejecuta el código entre la instrucción **#ifndef** hasta el anterior a la instrucción **#endif.** En ese código la instrucción #define, define la constante. Así la siguiente vez que se incluya este código, dicha constante ya estará definida; lo que provocará que el compilador salte hasta la línea *#endif* ignorando el código entre medias (y evitando el error que produciría volver a compilar dicho código).

[7.5.2] sobrecarga de métodos

Una propiedad de la POO es el polimorfismo. C++ posee esa propiedad ya que admite sobrecargar los métodos. Esto significa crear distintas variantes del mismo método. En esas variantes el prototipo de la función debe de ser diferente. Supongamos que creamos una clase llamada *X*. En esa clase podríamos declarar:

```
class X{
  public:
        void escribirValores();
        void escribirValores(char separador);
//…
```

Esa clase define dos métodos con el mismo nombre, sólo que el segundo tiene un parámetro de tipo **char** y el segundo no. Lógicamente en la parte de implementación tendremos que implementar ambos métodos.

[7.5.3] métodos inline

Ya se comentó en temas anteriores lo que era una función **inline**. En resumen es una macro, es decir una función que el compilador en cada llamada a la misma cambia la llamada por su código. Son candidatas a ser *inline* las funciones que tengan muy poco código.

En el caso de las clases, se ha comentado que se separa la declaración de la implementación. Pero en el caso de que un método se desee programar de forma *inline*, entonces el cuerpo se define en la declaración (no hace falta poner la palabra inline). Ejemplo:

```
class Punto{
    private:
        double x;
        double y;
    public:
        double obtenerX() {return x;} //método inline
        double obtenerY() {return y;} //método inline
        void escribirCoordenadas();
};
```

Muchos autores consideran que incluso las funciones **inline** deben de ir definidas en el archivo de implementación, la razón es que de otra forma estaremos obligados a leer dos archivos para revisar el código de implementación. Las funciones **inline** pueden ir en otro archivo si utilizamos la palabra clave **inline,** en la declaración y luego las implementamos en otro archivo.

[7.5.4] constructores

Cuando se crea un objeto de una determinada clase, lo lógico es inicializar las propiedades de dicho objeto. Esta labor la realiza el constructor de una clase.

Un constructor es un método de clase que es invocado automáticamente cuando se crea el objeto.

Ejemplo:

```
class Punto{
    private:
        double x;
        double y;
    public:
        Punto(); //Constructor sin parámetros
        Punto(double dx, double dy); //Segundo constructor
        double obtenerX();
        double obtenerY();
        void escribirCoordenadas();
};
```

En el ejemplo anterior se crea la interfaz de la clase Punto. En esta clase se definen dos constructores. EL primer no tiene parámetros (a este tipo de constructor se le llama **constructor por defecto**) el segundo tiene dos parámetros.

Los constructores son métodos que tienen como nombre el nombre de la clase, además (muy importante) los constructores no indican ningún tipo de resultado (**void, double**,..).

Los constructores se implementan de esta forma:

```
Punto::Punto(){
    x=0;
    y=0;
}

Punto::Punto(double dx, double dy){
    x=dx;y=dy;
}
```

Como se ha indicado antes se llama a los constructores durante la creación de objetos.

De esta forma:

```
Punto p; //El punto p se crea con las coordeanadas x=0 e
         // y=0 ya que se llama al constructor por defecto
Punto q(4,5); //Esta instrucción llama al segundo
              //constructor. x valdrá 4 e y valdrá 5
```

El ejemplo comentado anteriormente podría haberse hecho de esta forma:

```
class Punto{
    private:
        double x;
        double y;
    public:
        Punto(double dx=0, double dy=0);
        double obtenerX();
        double obtenerY();
        void escribirCoordenadas();
};
```

La diferencia está en que ahora se utilizará un solo constructor, pero que tendrá valores por defecto. Bastará implementar el segundo constructor:

```
Punto::Punto(double dx, double dy){
    x=dx;
    y=dy;
}
```

El resultado es el mismo que antes. Por eso es mucho más habitual utilizar constructores con valores por defecto.

[7.5.5] constructor de copia y asignaciones

Hay un tipo de constructor que es muy utilizado para cualquier clase y que permite crear un objeto copiando otro objeto de la misma clase. Por ejemplo para la clase *Punto* ese constructor podría ser:

```
class Punto{
  ...
  public:
       Punto(const Punto &p);//constructor de copia
  ...
}
```

Siempre tiene que ser declarado de esa forma, como un método que recibe una referencia constante a otro punto.

Copiar objetos es una tarea habitual. A diferencia de otros lenguajes, en C++ se pueden copiar objetos mediante el operador de asignación. Ejemplo:

```
Punto p(2,3);
Punto q;
q=p; //q copiará el código de p
```

En muchas ocasiones se suele redefinir el operador de asignación para evitar problemas.

[7.5.6] destructores

Un destructor es una función que es invocada automáticamente cuando un objeto está a punto de ser eliminado. la forma de declarar un destructor es casi idéntica a la de un constructor. Se pone el nombre de la clase, pero en este caso se coloca el símbolo **~** (que se escribe en el teclado español pulsando la tecla **Alt** seguida del número 126 escrito en el teclado numérico).

Normalmente los destructores se utilizan para eliminar los recursos que la clase reservó en memoria. Toda clase tiene sólo un destructor. Los destructores no pueden tener parámetros. Ejemplo:

```
class Punto{
    private:
        double x;
        double y;
    public:
        Punto(double dx=0, double dy=0);
        ~Punto();//destructor
        double obtenerX();
        double obtenerY();
        void escribirCoordenadas();
};
```

La implementación del método sería:

```
Punto::~Punto(){
    cout<<"Me muero"<<endl;
}
```

En este caso lo único que hará es que cada vez que un objeto de tipo Punto se elimine, escribirá en la pantalla el texto *Me muero*.

[7.5.7] referencia *this*

Todos los objetos pueden obtener un puntero hacia sí mismos llamado **this**. Si estamos en la parte de implementación de una clase, por ejemplo la clase *Punto*, **this** sería un puntero constante a dicha clase (no puede cambiarse de dirección).

Un ejemplo de uso:

```
Punto::Punto(double x, double y){
  this->x=x; //El atributo x de la clase toma el valor del
             //parámetro x
  this->y=y;
}
```

Este ejemplo (que es muy sencillo y fácilmente evitable) permite diferenciar el parámetro *x* de la propiedad *x* de la clase (de otro modo sería imposible).

Su utilidad práctica, sin embargo está relacionada con otros casos de uso más sofisticados.

[7.5.8] objetos constantes

El modificador **const**, se ha tratado en capítulos anteriores. En el caso de ser utilizado en la creación de objetos, permite crear objetos constantes, es decir objetos que no pueden modificarse (intentar su modificación daría como resultado un error).

Los objetos constantes tampoco permiten que se utilicen sus métodos (aunque dichos métodos no modifiquen los atributos del objeto) a no ser que las funciones sean declaradas como const. En ese caso estos métodos se definen como constantes también:

```
void Punto::escribirCoordenadas() const{
    cout<<x<<","<<y<<endl;
}
```

Sería un error que un método const modificara el valor de un atributo. Por ello es muy interesante declarar como **const** aquellos métodos que no modifiquen ningún atributo del objeto.

[7.5.9] clases *amigas*

Normalmente una clase no tiene acceso a la parte privada de otra clase. Pero existe una posibilidad de que una clase acceda a esa parte si se utiliza la posibilidad de hacer que sea *amiga* (en inglés **friendly**) de la clase a la que quiere acceder.

Tenemos dos posibilidad a la hora de crear amigas:

[1] **Hacer que una clase completa sea amiga de otra**. En cuyo clase podemos acceder completamente a todos los elementos de la clase desde cualquier método de la clase amiga.

Para hacer esto basta con usar la sintaxis: **friend class** *NombreDeClase* en la declaración de la clase. La clase así declarada podrá acceder a todas las partes de la clase en la que se hizo esa declaración.

```
class Recta{
   friend class Punto;
   ……….
```

Eso significa que desde la clase *Recta* se puede acceder a todos los elementos de **Punto**, sean públicos o privados. En UML eso se suele representar así:

[2] Hacer amiga a una función. Dicha función podrá acceder a todos los miembros de la clase. Para conseguir que una función acceda a todos los miembros de una clase, se declara como amiga en la declaración de clase. Ejemplo:

```
class Punto{
   friend void doblaCoords(Punto &p);
   ……….
```

No obstante crear funciones amigas no es nada recomendable (en la POO más rígida se prohíbe el uso de funciones globales). También está muy denostada en la actualidad la técnica de las clases amigas (Java utiliza mecanismos más interesantes para diferenciar clases).

[7.5.10] miembros estáticos

atributos estáticos

En general los miembros de una clase (propiedades y métodos) no existen en memoria hasta que creamos objetos de esa clase. Además cada objeto tiene su propia copia de estos miembros. Es decir en el caso de la clase *Punto*, cada punto que creemos (como es lógico) tendrá sus propias coordenadas *x* e *y*. En el caso de los miembros de tipo estático, su característica es que son comunes a todos los objetos de esa clase.

Para definir un atributo como estático, basta anteponer la palabra **static.** Eso indica que la propiedad en cuestión es común a todas las clases. Por ejemplo imaginemos que en la clase *Punto* deseáramos utilizar una propiedad para que nos indique el número de puntos que llevamos creados. Ese atributo es estático porque el valor del mismo es común para todos los objetos.

Ejemplo:

```
class Punto{
    private:
        double x;
        double y;
    public:
        static int totalPuntos;
        Punto(double dx=0, double dy=0);
        ~Punto();//destructor
        double obtenerX();
        double obtenerY();
        void escribirCoordenadas();
};
```

En la parte de implementación podemos inicializar la propiedad estática (es el único sitio en el que podemos). Lo cual se hace de esta forma:

```
//Inicialización de la propiedad totalPuntos de
//la clase Puntos
int Punto::totalPuntos=0;

//Constructor
Punto::Punto(double dx=0, double dy=0){
  x=dx;
  y=dy;
  totalPuntos++; //se incrementa la variable estática
                 //para que vaya contando los puntos que
                 //vamos creando
}

//Aquí irían el resto de métodos
//...
```

Al inicializar el atributo estático, se debe hacer en la parte de implementación sin indicar **static**, pero sí el tipo de datos (**int**). Después dentro del código de implementación se utiliza el atributo estático como si fuera un atributo normal.

Aunque no se cree ningún objeto de clase Punto, el atributo estático ya está en memoria (con el valor 0). Lo cual nos permite utilizarlas aunque no haya objetos. Es decir las propiedades estáticas se manejan como si fueran variables globales (aunque no lo son). Ahora, en realidad ¿cómo se usan? Para utilizarlas hay que usar el operador de

ámbito (**::**), ya que hay que indicar a qué clase pertenece el atributo (podríamos tener el mismo nombre para clases distintas). Ejemplo:

```
int main(){
   std::cout<<Punto::totalPuntos; //Escribe 0, el valor
                    //inicial del atributo
   Punto p, q(2,4); //Al crear dos puntos, el constructor
              //incrementa el valor del atributo
   std::cout<<Punto::totalPuntos; //Escribe 2
   std::cout<<p.totalPuntos; //Es correcto, también escribe 2
              //El total es el mismo para todos los Puntos
}
```

La última línea del código, *p.totalPuntos* nos indica que se puede acceder al atributo estático a través de un objeto de tipo punto. Pero el resultado es el mismo y no es nada aconsejable hacerlo puesto que puede inducir a errores de concepto.

métodos estáticos

Al igual que hay propiedades estáticas, hay métodos estáticos. Estos métodos son comunes a todos los objetos de la clase. De hecho, los métodos estáticos no pueden utilizar atributos que no sea estáticos.

Se utilizan bien en sustitución de funciones globales o para habilitar funciones a las clases que permitan extraer o configurar características comunes a todos los objetos de dicha clase.

Supongamos por ejemplo que deseamos que el atributo estático anterior deseamos que sea privado y para obtener su valor queremos utilizar un método. Entonces dicho método lo lógico es que sea estático:

```
class Punto{
    private:
        double x;
        double y;
        static int totalPuntos;
public:
        static int obtenerTotal();
        Punto(double dx=0, double dy=0);
        ~Punto();//destructor
        double obtenerX();
        double obtenerY();
        void escribirCoordenadas();
};
```

En la parte de implementación ahora el código sería:

```
//Inicialización de la propiedad totalPuntos de
//la clase Puntos
int Punto::totalPuntos=0;

//Constructor
Punto::Punto(double dx=0, double dy=0){
   x=dx;
   y=dy;
   totalPuntos++; //se incrementa la variable estática
                  //para que vaya contando los puntos que
                  //vamos creando
}
```

```
int Punto::obtenerTotal(){
   return totalPuntos;
}
```

```
//Aquí irían el resto de métodos
//...
```

El uso, al igual que en el caso de los atributos estáticos, de los métodos estáticos consiste en utilizar el método desde el nombre de la clase (aunque se podría utilizar cualquier objeto para acceder al método, esta técnica no es recomendable). Ejemplo:

```
int main(){
   std::cout<<Punto::obtenerTotal(); //Escribe 0, el valor
                  //inicial del atributo
   Punto p, q(2,4); //Al crear dos puntos, el constructor
             //incrementa el valor del atributo
   std::cout<<Punto::obtenerTotal(); //Escribe 2
   std::cout<<p.obtenerTotal(); //No es recomendable, aunque
                  //funciona. Escribe 2
```

En los métodos estáticos:

- No se puede utilizar ninguna propiedad ni método que no sea estático
- No pueden usar el término **this** para referirse al objeto actual (no hay objeto actual)
- No permiten sobrecarga de operadores

- Son más recomendables que las funciones globales (mantienen la encapsulación del programa)

[7.6] uso de la memoria dinámica

[7.6.1] introducción

Hasta ahora en todos los temas todas las variables y objetos del tipo que fueran se almacenaban en la memoria estática. Lo que se almacena en dicha memoria, se elimina en cuanto se cierra la llave que da fin al bloque de código en el que se creo la variable u objeto.

El problema de esa memoria es que el tamaño se debe conocer de antemano. Es la causa de tengamos tantos problemas con los arrays, o de que tengamos dificultades para crear funciones que devuelvan objetos.

La memoria dinámica se utiliza para crear objetos en tiempo de ejecución. Lo que se crea de forma dinámica no se elimina de la memoria, es el programador el que lo debe de eliminar (esta es una responsabilidad fundamental del programador de C++, si no se realiza correctamente la memoria se llenará de *basura digital* datos dinámicos a los que ya no se puede acceder y que no se pue4den eliminar).

[7.6.2] operador new

Se trata de uno de los operadores fundamentales de C++, permite crear un objeto en memoria y devolver la dirección de memoria en la que se almacena. Ejemplo:

```
Punto *ptr1= new Punto(3,4);
```

El puntero *ptr1* señala a un nuevo objeto creado en memoria. Obsérvese que al usar **new** estamos creando un nuevo objeto, por lo que podemos invocar al constructor que más nos interese (en el ejemplo se llama al constructor que crea puntos con las coordenadas *3,4*). Ese puntero nos sirve también (como es lógico) para acceder a los métodos y propiedades del objeto.

```
ptr1->escribeCoordenadas();
ptr1->cambiarX(3);
```

También se pueden crear punteros dinámicos a variables sencillas (enteros, double, bool,…):

```
int *p1=new int;
*p1=4;
cout<<*p1<<endl;
```

También se puede iniciar la variable dinámica en la creación:

```
int *p1=new int(4);
cout<<*p1<<endl;
```

En el caso de arrays:

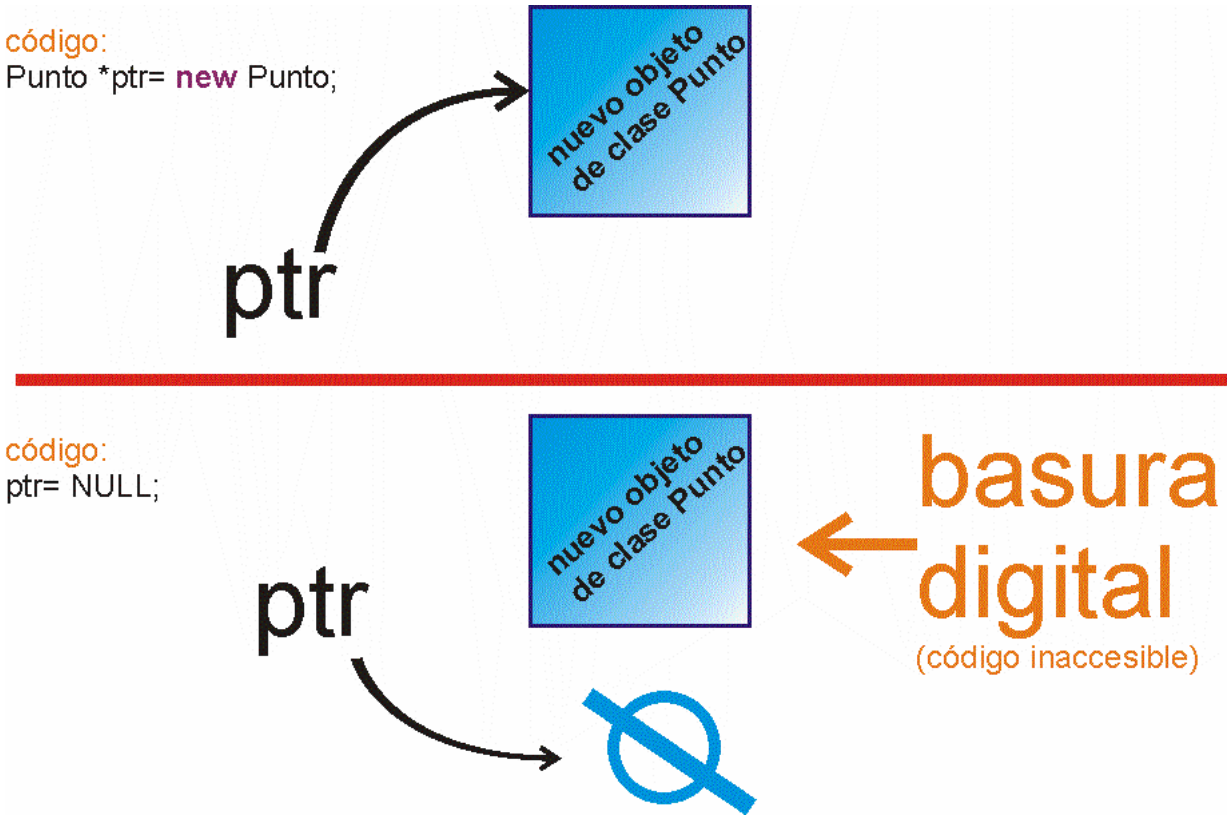
```
int *pArray=new int[10];
```

No obstante hay que tener en cuenta que todo lo creado con **new** debe de ser eliminado.

Por ejemplo en este código:

```
Punto *ptr=new Punto;
ptr=NULL;
```

El puntero *ptr* deja de apuntar al objeto que se crea en la primera línea tras hacer la segunda instrucción. Esto provoca que ya no se pueda acceder al punto creado como se explica en este diagrama:



Al asignar memoria dinámica podríamos tener el problema de que ya no hubiera más memoria libre, por lo que el objeto no sería creado. En C++ lo que ocurre en este caso es que el puntero que recibe el resultado del uso de **new**, apuntará a la dirección 0.

[7.6.3] operador *delete*

Es el encargado de eliminar la memoria creada por **new**. Su uso es muy sencillo, ejemplos (relacionados con los anteriores del apartado new):

```
delete ptr1;
delete p1;
delete [] pArray; //un array se debe eliminar con delete[]
```

El operador **delete** suele ser muy utilizados por los destructores de las clases, la razón estriba en que es muy habitual que muchas propiedades de los objetos sean creadas de forma dinámica, por lo que hay que liberar esa memoria antes de que nuestro objeto se elimine.

[7.7] sobrecarga de operadores

La propiedad de la programación orientada a objetos conocida como polimorfismo permite que un mismo método funciones con diferentes valores e incluso pueda tener diferentes significados. Los operadores también se sobrecargan, es el caso del operador ***** que sirve tanto para multiplicar como para obtener el contenido de un elemento señalado por un puntero.

En el caso de nuestras propias clase, podremos sobrecargar operadores para que funcionen con nuestras clases. La sobrecarga se realiza como si estuviéramos definiendo un método , sólo que el nombre siempre es la palabra **operator** seguida del operador que se sobrecarga (por ejemplo **operator+**).

Los operadores que se pueden sobrecargar son:

+	-	*	/	%	^	&	|
!	=	+=	-=	*=	/=	%=	^=
|=	<<	>>	>>=	<<=	==	!=	<=
>=	<	>	&&	||	++	->	[]
()	new	delete	new[]	delete[]			

Más fácil, no se pueden sobrecargar:

. .* :: ¿:

En cualquier caso esta sobrecarga lo que produce es que sumas, restas o asignaciones entre objetos tengan sentido. Además la sobrecarga debe cumplir estos requisitos:

- La sobrecarga de operadores funciona sólo cuando se aplica a objetos de una clase
- No se puede cambiar la preferencia de los operadores C++
- No se puede cambiar un operador binario para funcionar con un único objeto
- No se puede cambiar un operador unitario para funcionar con dos objetos
- No se puede sobrecargar un operador que funcione exclusivamente con punteros

[7.7.2] sobrecarga de operadores unarios

Los operadores unarios se utilizan con un solo operando que va a su derecha. Al definirlas en una clase hay que entender que es el objeto **this** el que se está modificando.

Por ejemplo el operador **!** significa NO de forma lógica y devuelve verdadero o falso. Estudiando ese operador, se trata de un operador que devuelve un valor lógico (verdadero o falso). En el caso de enteros devuelve verdadero si el entero es cero (NO falso vale verdadero).

En el caso de un objeto de tipo Punto supondremos vamos a definir el operador ! para que devuelva verdadero en caso de que ambas coordenadas sean cero (es un uso un poco raro del operador. En la declaración de la clase habría que definir el prototipo de la función de sobrecarga del operador, que sería:

```
bool operator!();
```

Entre los paréntesis no hay parámetros puesto que es un operador unario. En la fase implementación el código sería:

```
bool Punto::operator!(){
    return x==0 && y==0;
}
```

Uno de los habituales es la redefinición de los operadores ++ y -- Teniendo en cuenta que se pueden utilizar en la forma ++x o en la forma x++. Para la primera versión (conocida como **prefija**) **,** la declaración sería:

```
Punto & operator++();
```

Devuelve una referencia a un Punto. La versión **postfija**, es decir cuando el operador va en el lado derecho (*p++* por ejemplo) utiliza un parámetro ciego (sin nombre) de tipo **int**. Es decir:

```
Punto & operator++(int);
```

La implementación de ambas funciones podría ser:

```
//versión postfija, para p++ por ejemplo
Punto & Punto::operator++(int){
```

```
    Punto aux=*this; //Para almacenar los valores antiguos
    x++;y++;
    return aux; //devuelve los valores antiguos
}

//versión prefija, para ++p por ejemplo
Punto & Punto::operator++(){
    x++;y++;
    return *this;//Retorna el Punto con el valor ya cambiado
}
```

De esta forma este código funcionaría:

```
Punto p(3,5);
Punto q=p++; //Las coordenadas de q serán 3,5 las de p 4,6
```

[7.7.3] operadores binarios

En este caso los operadores tienen operandos a la izquierda y a la derecha. Se sobreentiende que el de la izquierda es el objeto **this**, el de la derecha se pasa como parámetro al método que sobrecarga al operador.

Por ejemplo en el caso de la clase Punto es fácil pensar en la suma de puntos como una operación que suma las coordenadas de dos puntos y devuelve otro punto que contiene esa suma. Por ello el prototipo sería:

```
Punto & operator+(const Punto &p);
```

Y el código:

```
Punto & Punto::operator+(const Punto &p){
    Punto aux(x+p.x,y+p.y);
    return aux;
}
```

[Unidad 8] relaciones entre clases

[8.1] relaciones entre clases

[8.1.1] introducción

A la hora de realizar programas complejos, ocurre que en dichos programas tendremos muchos objetos de distinta clases, comunicándose unos con otros. Esto significa que dichos objetos poseen relaciones. Durante todo este tema estudiaremos dichas relaciones, deteniéndonos más en las relaciones de herencia.

[8.1.2] asociaciones

La relación más simple entre clases es la asociación. Indica que dos clases tienen una relación que asocia a dos clases según el papel que cada una juega con respecto a otra. En un **diagrama de clases UML**, se indica mediante una línea continua que se dirige de un objeto a otro.

En dicha línea se anota el nombre de la relación (normalmente un verbo) y con un triángulo se indica la dirección en la que hay que leer dicho nombre. Opcionalmente se

pueden colocar un nombre de **rol** para cada clase, dicho rol es un nombre alternativo que representa el papel que toma cada clase en la relación:

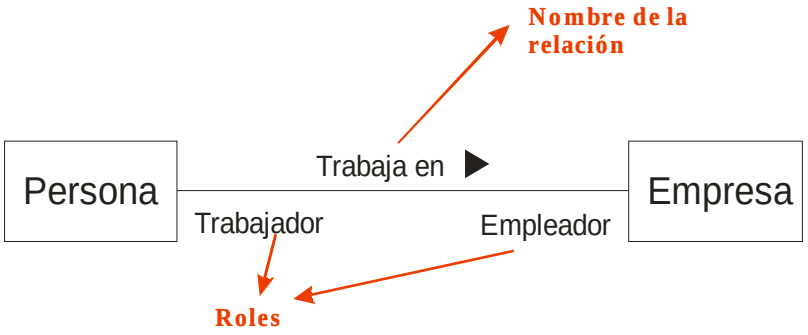


Ilustración 16, Ejemplo de asociación dibujada en UML

Hay que considerar que una asociación puede relacionar varios objetos con uno sólo. Por ejemplo, en el ejemplo anterior una misma empresa puede contratar a muchos trabajadores y la propia persona, podría haber pertenecido a varias empresas. A eso se le llama **multiplicidad** de la asociación y en un diagrama de clases se indica de esta manera:

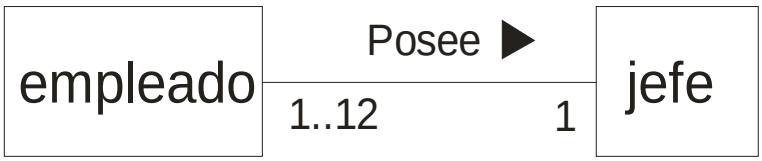
- En cada extremo de la línea de asociación aparecería uno o más números que indicarían las veces que participa cada clase en la relación. Por ejemplo, si colocamos el número 8 al lado de la clase *Persona* en el ejemplo anterior, significaría que cada empresa se relaciona con 8 personas como mucho y como poco. Si en empresa colocamos un uno (o no ponemos nada) significaría que cada Persona se asocia a una sola empresa.
- Si se indican dos números separados por dos puntos (..) indica un intervalo. Por ejemplo, indicar 1..3 en el lado de las empresas (en el ejemplo anterior) significa que cada persona puede estar relacionada como mínimo con una empresa y como máximo con 3. Si se pone un asterisco (por ejemplo 1..*) indica que la asociación puede estar relacionada con un número indeterminado de objetos. En el ejemplo anterior colocar 1..* en el lado de las personas indicaría que cada empresa está asociada como mínimo a una persona que puede estar relacionada con muchas personas (sin límite).

Hay que tener en cuenta que una clase se puede relacionar consigo misma en una relación reflexiva:

Ilustración 17, Asociación reflexiva

[8.1.3] implementación de asociaciones en C++

Hay diversas formas de implementar asociaciones en C++, pero las más utilizadas consisten en utilizar punteros. En el caso de que la multiplicidad sea uno, basta con usar un puntero a la clase relacionada. Ejemplo:



En ese ejemplo cada empleado o empleada posee un solo jefe o jefa. Mientras que los jefes y jefas pueden tener hasta doce empleados a su cargo. Eso se podía implementar de esta forma; creando un array de doce punteros a empleados en la clase jefes. Ejemplo:

```
#include <Empleado.h>
…
class Jefe{
   …
   Empleados *listaEmpleado[12];
   …
}
```

De esta forma sabremos la lista de los empleados de cada jefe. Además, cada empleado tendría que tener un puntero a su jefe:

```
#include “Jefe.h”
class Empleado{
   …
   Jefe *jefe;
   …
}
```

En el caso de que la relación fuera del tipo *, es decir que no tiene un máximo de elementos, en lugar de un array se utiliza una lista dinámica (estructura que se comentará más adelante).

[8.1.4] relación de agregación

Se trata de un tipo especial de relación que indica que un objeto forma parte de otro objeto. Modela relaciones del tipo *es parte de, tiene un* o *consta de.* En este caso estas relaciones no tienen nombre y se indica con un rombo. También se indica la multiplicidad de la misma forma que en las relaciones de asociación comentadas en el punto anterior.

Ejemplo:

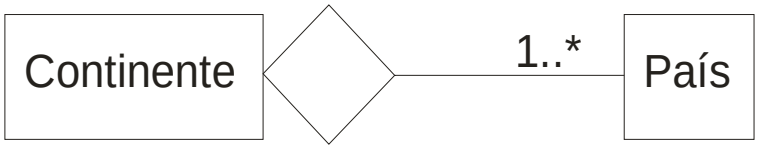


Ilustración 18, Ejemplo de agregación

En este caso puesto que la relación indica que un continente está formado por países, al implementar este tipo de relaciones en C++ no se usan punteros, sino que lo que se utilizan son arrays de objetos. En el ejemplo habría que crear una lista de países (un array de países si sabemos el tamaño máximo o una lista dinámica si no lo supiéramos) dentro de la clase continente. La diferencia está en que en las asociaciones, los objetos relacionados son independientes, ahora no.

A este tipo de relaciones se las llama relaciones **tiene un** (un continente *tiene* países), en contraposición a las de tipo es un, que se estudian más adelante en este mismo tema.

Sin embargo la implementación se hace igual que en el caso anterior, es decir mediante punteros (o arrays o listas de punteros)

[8.1.5] composición

Es una relación muy parecido a la anterior (del tipo **tiene un**), sólo que en este caso cada clase sólo puede formar parte de un *todo*. En el ejemplo anterior Turquía y Rusia son países que están en Europa y Asia (pertenecen a más de un continente), por lo que no pueden ser de composición.

Sin embargo:

Esta relación indica que cada coche se compone de cuatro ruedas, y que cada rueda sólo puede estar en un coche. La diferencia respecto a la anterior es grande a la hora de implementar. En este caso los objetos de la clase componente sólo pueden estar dentro

de un objeto de la clase grande. Por ello en estas relaciones (por otro lado muy habituales) lo que se suele hacer es crear los objetos directamente en la clase grande.

En el ejemplo basta con hacer:

```
class Coche{
   …
   Rueda ruedas[4];
   …
}
```

Aunque conceptualmente la agregación y la composición son muy similares, la realidad es que su naturaleza física es muy distinta. Si una agregación fuera implementada como composición cometeríamos un serio error ya que, en el ejemplo de los continentes y los países, tendríamos varias copias del mismo país y eso nos causaría serios problemas al programar.

En la composición hay que tener en cuenta que una *rueda* perteneciente a un determinado *coche* no tiene existencia sin dicho *coche.* La dependencia, es mayor que en las otras relaciones vistas.

[8.2] herencia

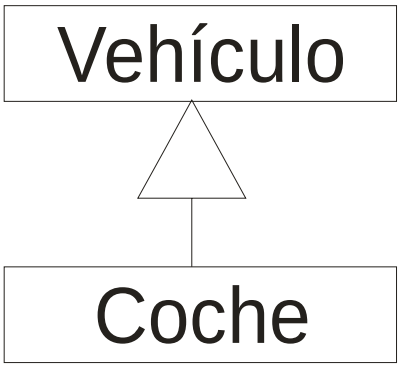
[8.2.1] introducción

Permite implementar relaciones del tipo **es un**. Estas relaciones permiten especificar jerarquías de clases en las que unas clases toman propiedades de otras. A la clase superior se la llama **clase base** (**superclase** según otras denominaciones), mientras que a la que hereda las propiedades de la anterior se la llama **clase derivada** (o **subclase**).

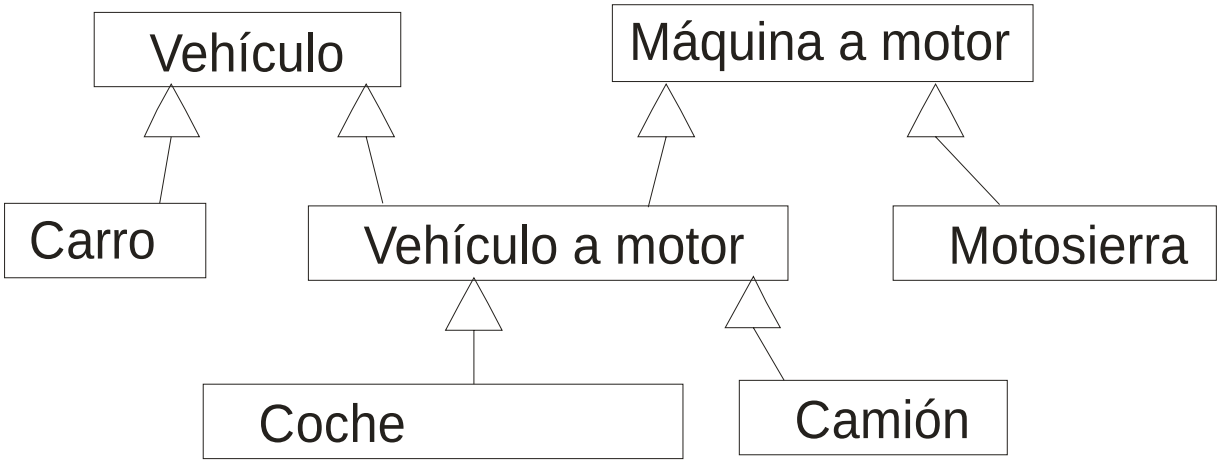
Lo que hace la clase derivada es heredar miembros de la clase base. En C++ es incluso posible heredar de más de una clase (**herencia múltiple**).

Por ejemplo la clase vehículo sería clase base de la clase coche, ya que un coche es un tipo de vehículo. Si todo vehículo puede por ejemplo moverse y detenerse, un coche también por el hecho de ser vehículo. Además un coche puede por ejemplo repostar gasolina, algo que no pueden hacer todos los vehículos (un coche siempre es un vehículo, pero un vehículo no tiene porque ser un coche).

Las relaciones de herencia se representan así en un diagrama UML:



Aunque la herencia puede ser más complicada:



En ese esquema se puede apreciar como los coches y camiones son vehículos a motor, y por lo tanto vehículos y máquinas a motor. Heredarán propiedades de todas las clases superiores en el esquema anterior.

[8.2.2] creación de herencia en C++

Para indicar que una clase es derivada de otra se hace lo siguiente:

```
class X: public Y{
   …
};
```

Eso significa que la clase X deriva de Y; o, lo que es lo mismo, que X es un tipo de clase Y.

Nota: En la clase derivada hay que incluir el archivo de cabecera de la clase base. En el ejemplo sería: **#include "x.h"**

Las clases derivadas heredan todas las propiedades y métodos públicos de la clase base, excepto:

- los constructores
- los destructores
- operadores de asignación
- miembros estáticos

Aunque los constructores y destructores no se heredan, los constructores de las clases derivadas invocan al constructor por defecto de la clase base (salvo que se realice una llamada explícita a otro constructor de la clase base). Esa invocación se realiza antes de ejecutar el código del constructor de la clase derivada.

En el caso del destructor, siempre se ejecuta el código del destructor de la clase base antes de ejecutar el código del destructor de la clase derivada.

[8.2.3] acceso a los métodos y propiedades de las clases base

acceder a métodos y propiedades heredados

Supongamos que tuviéramos la siguiente clase:

```
class Vehiculo
{
    private:
        double velocidad;
  public:
        // constructor
        Vehiculo() {velocidad=0;}
        void acelerar(int v) {velocidad+=v;}
        void frenar(int v) {velocidad-=v;}
        void parar() {velocidad=0;}
        double getVelocidad() {return velocidad;}
};
```

Y que esta clase representa a cualquier vehículo (carro, tractor, tren, etc.). Para todos los vehículos, los métodos de acelerar, frenar o parar tienen sentido. Pero luego tenemos vehículos más especializados, por ejemplo los vehículos a motor. Para ello declaramos la clase:

```
class VehiculoMotor : public Vehiculo
{
    private:
        int combustible;
    public:
        // class constructor
        VehiculoMotor() {combustible=10;}
        void arrancar() {combustible--;}
        int getCombustible() {return combustible;}
};
```

Parece más corta la segunda clase, sin embargo esta clase incluye la parte pública de la primera, es decir sería posible este código, una vez compilados e incluidos los anteriores:

```
VehiculoMotor vm;
vm.acelerar(10);
vm.arrancar();
cout<<vm.getCombustible()<<endl; //Escribe 9
cout<<vm.getVelocidad()<<endl; //Escribe 10
```

En el código anterior hay que fijarse que el objeto *vm* de clase *VehiculoMotor* utiliza métodos de las dos clases. Es decir, de la clase *Vehiculo* (acelerar) y de la clase *VehiculoMotor* (arrancar).

Es decir desde una clase derivada se pueden acceder a todos los métodos de la clase base que sean públicos. Dicho de otra forma, en realidad la clase *VehiculoMotor* tiene como funciones miembros: acelerar, frenar, parar, getVelocidad (heredadas de la clase *Vehiculo*), arrancar y getCombustible (definidas en la propia clase). Sin embargo sólo puede acceder a la propiedad *combustible* y no a *velocidad*, ya que fue definida de forma privada en la clase base.

miembros *protected*

Además de los modificadores **public** y **private** existe un tercer modificador especialmente pensado para la herencia, se trata de **protected** (protegido). Un miembro (sea función o variable) de una clase definido con este modificador puede ser accedido desde las clases derivadas pero no tener acceso desde una clase (o un código) que no sea derivada.

Es decir si tenemos la clase:

```
class X{
  private:
        int a;
  public:
        X() {a=6;}
}
```

y creamos:

```
class Y:public X{
  public:
        Y(int b) {a=b;}
}
```

Ocurriría un error, por que desde *Y* no podemos acceder al miembro *a*. Pero en este caso:

```
class X{
  protected:
        int a;
  public:
        X() {a=6;}
}
```

El código de Y sería válido.

En los diagramas UML, los miembros **protected** se indican con el signo **#** en lugar de **+** o **-**

modificadores de acceso en la herencia

Anteriormente se ha comentado que la forma de indicar que una clase es derivada de otra es:

```
class X : public Y {
```

y también se ha explicado las implicaciones que tiene el acceso a los métodos de la clase desde la clase derivada. Pero es importante indicar que la palabra **public**, puede ser sustituida por **protected** o **private** en el código anterior. En ese caso el acceso a miembros de la clase base por parte de la clase derivada cambia completamente.

El acceso a los miembros de la clase base sigue las reglas de esta tabla:

modificador de acceso	miembros de la clase base	pasan a ser en la clase derivada..
public	**public** **protected** **private**	**public** **protected** NO SON ACCESIBLES
protected	**public** **protected** **private**	**protected** **protected** NO SON ACCESIBLES
private	**public** **protected** **private**	**private** **private** NO SON ACCESIBLES

Es decir, en este código:

```
class X : protected Y {
…
```

Los miembros heredados de tipo público de X pasan a ser de tipo protegido (protected) en Y, los protegidos no cambian y los privados no se pueden utilizar en Y.

No obstante no es recomendable utilizar modificadores de acceso distintos de **public** al reflejar la herencia de clases. Es muy poco habitual necesitar otros modificadores (en la mayoría de lenguajes modernos, como Java por ejemplo, esa posibilidad no existe).

[8.2.4] constructores de las clases derivadas

Recordando el ejemplo anterior, observemos ahora los nuevos constructores de clase:

```
class Vehiculo
{
    private:
        double velocidad;
  public:
        // constructor
        Vehiculo(int v=0) {velocidad=v;}
        void acelerar(int v) {velocidad+=v;}
        void frenar(int v) {velocidad-=v;}
        void parar() {velocidad=0;}
        double getVelocidad() {return velocidad;}
};

class VehiculoMotor : public Vehiculo
```

```
{
    private:
        int combustible;
    public:
        VehiculoMotor(int c=10) {combustible=c;}
        void arrancar() {combustible--;}
        int getCombustible() {return combustible;}
};
```

Ahora supongamos que hacemos este código:

```
VehiculoMotor vm(5);
cout<<vm.getCombustible()<<endl; //sale 5
cout<<vm.getVelocidad()<<endl; //sale 0
```

Lo último que escribe es un cero, señal de que al construir el vehículo de motor, es invocado el constructor por defecto de la clase vehículo. De hecho hay que entender (salvo que se indique lo contrario desde el código) que la primera línea de un constructor de clase derivada es una invocación al constructor de la clase base.

Ahora, lo lógico es que podamos construir vehículos a motor indicando la velocidad lo que obliga a definir este constructor:

```
VehiculoMotor(int v=0; int c=10){
  combustible=c;
  velocidad=v; //error, velocidad es privado
}
```

No es posible usar la propiedad *velocidad*. La solución es invocar al constructor apropiado en la clase base, lo cual se hace:

```
VehiculoMotor(int v=0; int c=10):Vehiculo(v){
  combustible=c;
}
```

Eso significa que utilizamos el constructor de la clase base pasando la variable *v* como parámetro del constructor.

[8.2.5] redefinir métodos

Nuevamente, observemos este código:

```
class Vehiculo
{
    protected:
        double velocidad;
  public:
```

```
        // constructor
        Vehiculo(int v=0) {velocidad=v;}
        void acelerar(int v) {velocidad+=v;}
        void frenar(int v) {velocidad-=v;}
        void parar() {velocidad=0;}
        double getVelocidad() {return velocidad;}
};

class VehiculoMotor : public Vehiculo
{
    private:
        int combustible;
    public:
        VehiculoMotor(int v=0; int c=10):Vehiculo(v){
              combustible=c;}
        void arrancar() {combustible--;}
        int getCombustible() {return combustible;}
        void acelerar(int v){velocidad+=v;combustible--;}
};
```

Hay dos versiones del método *acelerar*. El código es válido (observar que ahora la propiedad velocidad es de tipo protegido, por lo tanto heredable), es posible redefinir una función. Por ello este código:

```
VehiculoMotor vm(5,3);
vm.acelerar(5);
```

llama al método acelerar de la clase *VehiculoMotor* (no al de la clase Vehiculo). La cuestión es ¿y si queremos invocar al método de la clase base? Entonces, se usaría:

```
VehiculoMotor vm(5,3);
vm.Vehiculo::acelerar(5);
```

ahora se invoca al *acelerar* de la clase base

[8.2.6] herencia múltiple

En C++ (a diferencia de otros lenguajes como Java) se permite que una clase sea heredera de varias clases base). Esto es posible utilizando sentencias como:

```
class C: public B, public C{
…
```

Sin embargo, este tipo de herencia es poco utilizada en la práctica y requiere cierto cuidado, por ejemplo:

```
class A{
  private:
        int a1;
  public:
        int x;
        void prueba();
};

class B{
  private:
        int b1;
        void help();
  public:
        int x;
        void prueba();
        void siFunciona();
};

class C: public B, public C{
  public:
        prueba2();
}
```

El problema ocurre si luego hacemos uso de este código:

```
C objetoC;
c.prueba(); //error
```

Hay ambigüedad, el compilador no puede saber si hay que invocar el código de la clase *A* o el de la clase *B*. ¿Cómo evitar esta ambigüedad? De esta forma:

```
c.A::prueba();
```

ahora no hay duda, se invoca a la prueba de la clase *A*.

[8.2.7] conversiones en la herencia

conversiones de objetos

En principio la operación de asignación no permite asignar objetos de clases distintas. Sin embargo en el caso de la herencia se pueden utilizar conversiones implícitas para hacer que un objeto de clase base e asigne a un objeto de clase derivada.

Es decir, es perfectamente válido:

```
Vehiculo v;
VehiculoMotor vm;
v=vm;
```

el objeto *v*, en el ejemplo, será una copia de *vm*. Pero sólo copiará la parte de *vm* referida a los métodos y propiedades heredados de la clase *Vehiculo*. Es decir, podremos hacer:

```
v.acelerar(7);
```

pero no:

```
v.arrancar();
```

Es muy importante tener en cuenta que **no se pueden copiar objetos de clases base a objetos de clases derivadas.** Es decir, esta expresión no es válida (salvo que redefinamos el operador de asignación para soportar esta posibilidad):

```
Vehiculo v;
VehiculoMotor vm;
vm=v;
```

Cabría la tentación de utilizar el operador de **casting** (conversión):

```
Vehiculo v;
VehiculoMotor vm;
vm= (VehiculoMotor) v;
```

pero tampoco funciona. A no ser que definamos un constructor copia, que copie de la clase *Vehiculo* a la clase *VehiculoMotor*. Es decir si definimos:

```
VehiculoMotor::VehiculoMotor(Vehiculo & v) {combustible=10;}
```

por ejemplo.

conversión de punteros

Con los punteros ocurre parecido. Pero se usa más este tipo de conversión ya que permite definir funciones para todo un árbol de clase.

Ejemplo:

```
Vehiculo v;
VehiculoMotor vm;
Vehiculo *pV=&v; //pV señala al objeto v
VehiculoMotor *pVM=&vm; //pV señala al objeto vm
pVM=pV; //incorrecto
pV=pVM; //correcto
pV->acelerar(11);
cout<<pV->getVelocidad()<<endl; //Escribe 11
cout<<pVM->getVelocidad()<<endl; //También escribe 11
```

Se pueden asignar punteros de clases base a punteros de clases derivadas. Pero no al revés. Por supuesto al utilizar métodos desde el puntero, estamos cambiando el estado del objeto, por lo que todos los punteros a ese objeto reflejarán el cambio.

Los punteros de clases bases, aún señalando a objetos de clases derivadas, no pueden utilizar métodos que no estén en la clase base, es decir:

```
pV->arrancar();
```

Daría error. Este error sí puede ser arreglado con un **cast**:

```
((VehiculoMotor *)pV)->arrancar(); //¡¡Correcto!!
```

[8.3] polimorfismo

[8.3.1] funciones virtuales

Se ha explicado anteriormente que es posible redefinir métodos en las clases derivadas. Sin embargo con lo visto hasta ahora tendríamos serios problemas para que la propiedad del polimorfismo funcione adecuadamente. El polimorfismo implica que una misma función trabaje de diferente forma según qué objeto tengamos delante.

Por ejemplo supongamos que tenemos definida una clase llamada *forma* y que esa clase es base de las clases *círculo* y *cuadrado*. Supongamos que la clase *forma* tiene un método llamado *mostrarDatos* que sirve para mostrar información de la forma, por ejemplo mostraría el área de la forma.

Después la clase *circulo* redefine ese método para que además muestre el radio del círculo (esa información sólo tiene sentido en el caso del círculo) y lo mismo la clase cuadrado (sólo que esa clase mostraría el lado del cuadrado).

Supongamos que tenemos este código (se suponen implementados los constructores por defecto):

```
Forma f;
Circulo c;
Cuadrado d;
Forma *p;
p=&f;p->mostrarDatos();
p=&c;p->mostrarDatos();
p=&d;p->mostrarDatos();
```

El código es correcto, pero ejecutaría la versión del método *mostrarDatos* de la clase forma, al utilizar un puntero de éste tipo. Es decir no veríamos el radio del círculo ni el tamaño del lado del cuadrado.

Lo lógico es que éste no sea el funcionamiento que deseamos. La razón es que los objetos derivados tienen dos versiones del método, la que han definido y la de la clase base.

Para evitar esto se usan las funciones virtuales, cuando un método en una clase base es definido con la palabra **virtual**, lo que indica es que puede ser redefinido por sus clases derivadas. En ese caso **las clases derivadas sólo tendrán una versión del método, la que han redefinido ellas mismas.**

En el ejemplo anterior el método *mostrarDatos* en la clase *forma* habría sido **declarado** así:

```
class Forma{
  …
  virtual void mostrarDatos();
}
```

En la definición del método no se utiliza la palabra virtual (de hecho sería un error escribirla en la definición de la función:

```
void Forma::mostrarDatos(){
  …
```

En las clases derivadas se redefiniría éste método y eso permite en el ejemplo anterior:

```
Circulo c;
Cuadrado d;
Forma *p;
p=&c;p->mostrarDatos();
p=&d;p->mostrarDatos();
```

ahora en tiempo de ejecución se determina la versión a utilizar de la función y se mostrarán diferentes cosas según a qué tipo de objetos señale el puntero *p* (ahora sí veremos los radios de los círculos y los lados del cuadrado).

[8.3.2] clases abstractas

Son una de las mejoras más llamativas del lenguaje C++ (de hecho existen en casi todos los lenguajes orientados a objetos). Se trata de clases que definen **funciones virtuales puras**. Las funciones virtuales puras son aquellas que no se implementan simplemente se declaran con la palabra virtual y se asignan a cero.

Es decir si quisiéramos hacer una función virtual pura en el caso de *mostrarDatos* en la clase *Forma*, el código sería:

```
class Forma{
   …
   virtual void mostrarDatos() = 0;
}
```

Ahora no habría que implementar esa función. La pregunta es ¿para qué sirven las funciones virtuales puras? Simplemente sirven para hacer clases abstractas, clases que definen algunos métodos, pero otros no (los virtuales puros) para indicar que es necesario que se implementen en las clases derivadas.

Un caso típico sería el método *dibujar* de la clase *forma*. No podemos saber como dibujar una forma sin saber qué tipo de forma es (círculo, cuadrado, rombo,…). Pero sí sabemos que toda forma es dibujable, por ello se le define como virtual con valor a 0. Cualquier clase derivada de forma tendrá que definir este método (de otro modo la derivada será también una clase abstracta).

características de las clases abstractas

- Lógicamente **toda clase abstracta tiene que tener al menos un método virtual puro** (puede tener más de uno)
- **No se pueden crear objetos de una clase abstracta.** Esta importantísima restricción suele confundir mucho a los programadores que empiezan al no entender su utilidad. La razón es que proporcionan una plantilla a cubrir por las clases derivadas. En el ejemplo anterior, no tiene sentido crear un objeto *forma* ya que es un objeto abstracto que no se puede utilizar.
- **Se pueden crear referencias y punteros a clases abstractas.** Esto parece contradecirse con el punto anterior, pero no es así. En este caso esto se permite para conseguir punteros polimórficos. Un puntero sólo debe señalar a un objeto creado (o a **NULL**) por lo tanto señalará a objetos de clases derivadas y eso permitirá utilizar dicho puntero para utilizar las funciones definidas en la clase abstracta (aunque sean virtuales puras).

Es decir aunque la clase *forma* comentada antes sea abstracta, este código es correcto:

```
Circulo c;
Cuadrado d;
Forma *p;
p=&c;p->mostrarDatos(); //muestra el radio del círculo
p=&d;p->mostrarDatos(); //muestra el lado del cuadrado
```

Pero daría error:

```
Forma f; //Forma no es instanciable
```

Las clases abstractas permiten crear interfaces para las clases. Otros lenguajes (como el **Java**) han definido estructuras llamadas interfaces que tienen un claro parecido con las clases abstractas, pero que aportan una mayor claridad al lenguaje y una funcionalidad mayor que las clases abstractas.

En UML una clase abstracta se representa con un romboide en lugar de con un rectángulo (en otras representaciones se pone el nombre de la clase en cursiva (o más formalmente poniendo el texto **<<abstract>>** encima del nombre de la clase). Ejemplos:

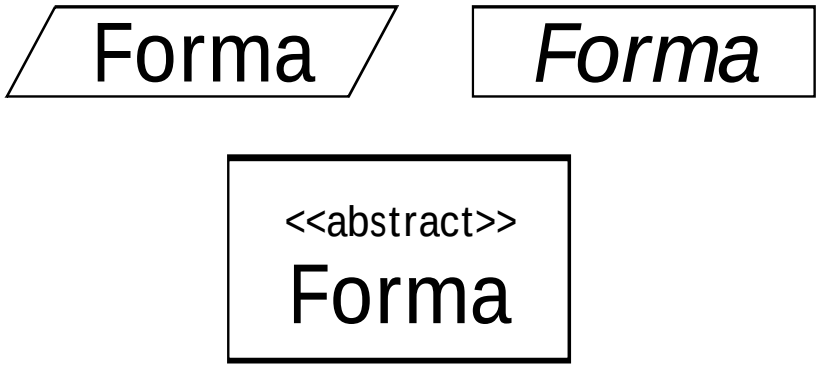


Ilustración 19, Diferentes representaciones de la clase abstracta Forma en UML

[Unidad 9] archivos. entrada y salida

[9.1] estructuras externas de datos

Hasta ahora todos los datos de los programas eran almacenados en la memoria interna del ordenador, bien de forma estática o bien de forma dinámica (mediante el operador **new**).

Los datos se pueden almacenar en dispositivos externos (archivos normalmente) y eso permite su perdurabilidad entre diferentes sesiones del programa. La organización de los datos en los archivos sigue una estructura, por lo que la información así almacenada se conoce como **estructuras externas de datos.**

En definitiva en este tema se estudian los archivos y las operaciones de entrada y salida que permiten comunicar al programa con el exterior de la máquina.

[9.2] flujos de datos

[9.2.1] introducción

En C++ la entrada y salida de datos se realiza mediante **flujos**. Los flujos son secuencias de bytes. En la entrada esos bytes fluyen desde un dispositivo de entrada (como el teclado) hacia la memoria del ordenador. En la salida fluyen de la memoria hacia un dispositivo de salida (como la pantalla).

Esos flujos de datos mueven bytes de forma más lenta respecto al movimiento de información que hace el procesador con los datos en memoria. Para manejar esos flujos o corrientes (*streams*) de datos tenemos dos tipos de operaciones en C++:

- **Operaciones de bajo nivel.** En las que el **byte** es el elemento de trabajo. Permiten transacciones de alta velocidad con los dispositivos, pero son incomodas para programar.
- **Operaciones de alto nivel.** En ellas la información se organiza en estructuras más reconocibles por el programador como enteros, decimales, objetos,... Son idóneas para manejar archivos sin hacer uso intensivo de los mismos.

En el C++ clásico las bibliotecas estándar del lenguaje manejaban los archivos usando el tipo **char**, ya que ocupa exactamente un byte. Lo malo es que este tipo no es capaz de representar los símbolos de todos los lenguajes, por lo que se ampliaron las librerías para ser compatibles con **Unicode** (www.unicode.org) , el código que representa casi cualquier signo de cualquier idioma. Para ello se usa el tipo definido **wchar_t**

[9.2.2] bibliotecas de entrada y salida

- **iostream.** Se trata de la librería que contiene los servicios fundamentales que permiten trabajar con corrientes de bytes. 1
- **iomanip.** Se trata de la librería que permite aplicar formato a la entrada y salida
- **fstream.** Para procesar archivos.

[9.2.3] clases y objetos relacionadas con los flujos de datos

- **streambuf**. Clase que sirve para manipular flujos de datos a bajo nivel.
- **ios**: Clase genérica base de **istream, ostream** e **iostream**.
- **istream**: Clase especializada en entradas.
- **ostream**: Clase especializada en salidas.
- **iostream**: Encapsula las funciones de entrada y salida por teclado y pantalla.
- **fstream**: Entrada y salida desde ficheros

- **strstream**: Uso de strings como corrientes de entrada o de salida

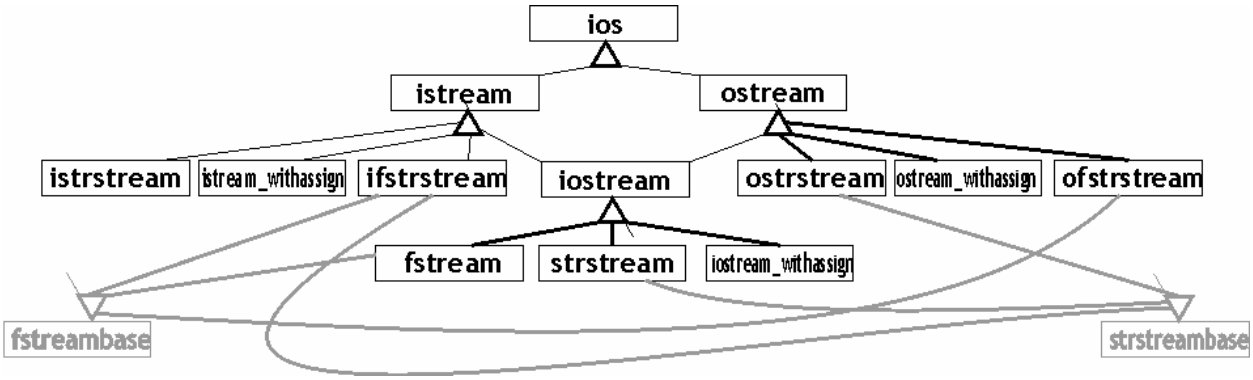


Ilustración 20, Jerarquía de clases relacionadas con flujos de entrada y salida

objeto *cin*

Se trata del objeto predefinido de entrada (normalmente asociado al teclado). Se trata de un objeto de clase **istream**. Mediante el operador de flujo **>>**, es capaz de leer cualquier tipo de datos básico. Para poder leer objetos, se debe redefinir dicho operador en la clase a la que pertenece el objeto.

objeto *cout*

Es la versión complementaria del anterior, es el objeto asociado a la salida estándar (normalmente la pantalla); es un objeto de clase **ostream**, que sirve para escribir cualquier tipo de datos (habrá que redefinirle en las clases para que éstas actúen adecuadamente)

objeto *cerr*

Objeto de tipo **ostream**, que está asociado al dispositivo de errores estándar (normalmente la pantalla). La salida que hace es sin búfer, lo que implica que los errores se muestran instantáneamente.

objeto *clog*

Objeto de tipo **ostream** asociado al dispositivo de errores estándar (normalmente la pantalla). La salida que hace es con búfer, lo que implica que el texto a mostrar se mantiene en un búfer hasta que este se llene o se vacíe.

[9.2.4] operadores de flujo

Tenemos dos operadores en C++ que se encargan de desplazar un flujo de datos hacia la entrada (**>>,** operador de **extracción)** o hacia la salida (**<<,** operador de **inserción**). Estos operadores se utilizan con los objetos **cin**, **cout**, **cerr** y **clog**.

Estos operadores son binarios, a la izquierda poseen un objeto de tipo **istream** (flujo de entrada, en el caso de >>) u **ostream** (flujo de salida, en el caso de <<). A la derecha tendrán una referencia a un objeto o a un tipo básico. Para los tipos básicos ambos operadores funcionan perfectamente, pero para los objetos hay que redefinirles. La sintaxis para redefinir ambos operadores es:

```
ostream& operator<< (ostream& os, const clase& obj);
istream& operator>> (istream& ci, clase& obj);
```

En el caso de los flujos de salida la referencia al objeto es constante puesto que su valor no cambia, en el caso del operador de estracción el objeto sí cambia. Ambas funciones deben ser definidas como friend (funciones amigas) de las clases que van a utilizarlas a fin de que puedan utilizar sus propiedades privadas.

Ejemplo:

```
using std::ostream;

class Punto{
    friend ostream &operator<<(ostream &os, const Punto &p);
    private:
        double x;
        double y;
    public:
        // class constructor
        Punto(double dx=0, double dy=0){x=dx;y=dy;}
        Punto(const Punto &p) {x=p.x;y=p.y;}
        void cambiar(double dx=0, double dy=0){x=dx;y=dy;}
        double getX(){return x;}
        double getY(){return y;}
};
```

ostream es una clase perteneciente al espacio **std**, por eso es necesario incluir la instrucción **using** std::ostream.

La definición de la función podría ser:

```
ostream & operator<<(ostream &os, const Punto &p){
    os<<p.x<<","<<p.y;
    return os;
}
```

Eso permite por ejemplo hacer:

```
Punto p(3,4);
cout<<p;
```

[9.2.5] salida de flujo

La clase **ostream** es la encargada de la salida de flujo y permite realizar salidas de alto y bajo nivel. Los objetos **ostream** pueden utilizar el operador de extracción y detectar el tipo automáticamente. Así por ejemplo:

```
char s[]="Esto es una prueba";
char *p=s;
cout<<p;
```

Sacaría el texto aunque *s* es un puntero, ya que detecta que señala a un *string.* Si quisiéremos mostrar la dirección tendríamos que hacer:

```
char s[]="Esto es una prueba";
char *p=s;
cout<<(void *)p;
```

Ahora sale la dirección porque al hacer un *cast* hacia un puntero genérico (de tipo **void**) entiende que hay que sacar la dirección. Como se vio antes, para los objeto habría que redefinir el operador para personalizar su funcionamiento.

método *put*

El método **put** sirve para escribir caracteres en un objeto de flujo de salida. Por ejemplo:

```
cout.put('D');
```

Escribe la letra *D*. Esta función devuelve un objeto ostream por lo que se pueden encadenar salidas:

```
cout.put('D').put('\n');
```

método *flush*

Prototipo:

```
ostream& flush()
```

Vacía el búfer de salida, procesa todas las salidas pendientes.

método *seekp*:

Prototipo 1

```
ostream& seekp(streampos posicion)
```

Prototipo 2

```
ostream& seekp(streamoff posRelativa, seek_dir dir)
```

Permite mover el cursor a una posición del stream de salida,. La primera forma es para hacerlo de forma absoluta y la segunda de forma relativa.

método *tellp*:

```
streampos tellp()
```

Devuelve la posición absoluta del cursor del stream.

[9.2.6] entrada de flujo

la clase **istream** permite leer flujos de entrada. El operador de inserción (>>) funciona con objetos istream para leer datos de entrada. Este operador moite los espacios de entrada por lo que no es adecuado, por ejemplo, para leer cadenas de caracteres (aunque este comportamiento se puede variar).

método *get*

Se trata de una función que permite leer de una corriente de datos de entrada. Posee tres prototipos para trabajar:

Prototipo 1
```
int get()
```

Prototipo 2
```
istream& get(char& caracter)
```

Prototipo 3
```
istream& get(char* cadena, int tamaño, char
delimitador= '\n')
```

La primera versión de *get*, lee un carácter y lo almacena en el búfer. Ejemplo:

```
int car; //La lectura con get debe usar int ya que EOF es
         //de tipo int
while((car=cin.get()) != EOF)
   cout.put(car);
```

Este código lee carácter a carácter y les va mostrando por pantalla hasta que llegue el signo **EOF** (**End Of File**, fin de archivo), EOF es el código ASCII 28 (*Control-Z*).

La segunda versión de get, lleva un argumento, una referencia a un carácter en el que se almacena el carácter que se lee por el flujo de entrada. Esta versión devuelve el valor **-1** si se alcanzó el fin de archivo o bien un objeto de tipo *istream* si no estamos al final.

La tercera versión de get tiene tres argumentos; el primero es un string (con tamaño suficiente), el segundo un tamaño (un entero) y el tercero un carácter delimitador (éste no es obligatorio, por defecto se toma el carácter de cambio de línea, *\n*). La función almacena el string hasta el tamaño máximo indicado o hasta que llegue el delimitador.

Ejemplo:

```
char cad[80]="";
cin.get(cad,80);
cout<<cad<<"\n";
```

método *getline*

Es casi un calco del método anterior. Tiene los mismos parámetros que la tercera versión de *get*. La diferencia estriba en que getline desecha el delimitador y no le añade a la cadena (*get* sí lo hace). Prototipo:

```
istream& getline(char* cadena, int tamaño,
                 char delimitador= '\n')
```

método *ignore*

Prototipo:

```
istream& ignore(int n = 1, int delim = EOF)
```

Este método hace que se ignoren los siguientes *n* caracteres del flujo de entrada o hasta que legue el delimitador indicado (por defecto EOF, final de archivo). Ejemplo:

```
char a[15];
cin.ignore(3);
cin.get(a,15);
cout<<a<<endl; //no salen los tres primeros caracteres escritos
```

método *peek*

```
int peek()
```

Devuelve el siguiente carácter del stream pero sin quitarlo del búfer. Ejemplo:

```
char c[15];
char a;
a=cin.peek();
cout.put(a);
a=cin.peek();
cout.put(a);
a=cin.peek();
cout.put(a);
cin.get(c,15);
cout<<c<<endl;
```

En el ejemplo anterior se usa *peek* tres veces y en cada una de ellas se escribe el resultado de la función. Después se lee con *get*. Si, por ejemplo, al ejecutar este código escribimos *Hola mundo*, saldría por pantalla (tras pulsar el **Intro**) *HHHHolan mundo*, ya que *peek* recoge el carácter pero no le retira (seguirá disponible en la siguiente lectura)

método *putback*

```
istream& putback(char caracter)
```

Coloca un carácter en el stream

método *seekg*

Prototipo 1

```
istream& seekg(streampos pos)
```

Prototipo 2

```
istream& seekg(streamoff desplazamiento, seek_dir dir)
```

Permite colocarse en una determinada posición del stream.

método *tellg*

```
long tellg()
```

Devuelve la posición actual que tenemos en el stream.

[9.2.7] métodos de entrada y salida de bajo nivel

Sólo tienen sentido para ser utilizadas con flujos de bytes procedentes de archivos (o de una red) ya que trabajan con bytes. Si se usan con el teclado y la pantalla, en la lectura hay que pulsar Intro tras cada byte a leer. El fin de lectura lo lleva el carácter **Control-Z** (**EOF**)

write

Es un método de la clase *ostream* que permite escribir una serie de bytes a un flujo de salida. Prototipo:

```
ostream& write(const char* cadena, int n);
```

escribe los primeros n bytes de la cadena:

```
char x[]="Hola mundo";
cout.write(x,4); //escribe Hola
```

read

Lee *n* caracteres del flujo de entrada y los almacena en un array de caracteres. Prototipo:

```
istream& read(char* cadena, int n);
```

gcount

```
int gcount();
```

Indica el número de bytes leídos en la última lectura. Ejemplo de uso:

```
char cad[80];
cin.read(cad,20);
cout.write(cad,cin.gcount());
```

Lee hasta 20 bytes y luego les escribe (considerando el número de elementos que se leyeron con *read*)

[9.2.8] manipuladores de flujo

La librería **iomanip** implementa manipuladores, se trata de elementos que realizan tareas de formato con los flujos de entrada y salida. Se utilizan con los operadores de extracción (<<) e inserción (>>) para modificar el formato del flujo.

cambios de base

Los modificadores **hex, dec, oct** y **setbase.** Son los encargados de modificar la base numérica del flujo de datos, para que aparezca en hexadecimal, octal o decimal. El modificador *setbase* recibe un parámetro numérico que será 8 (octal),10 (decimal) o 16 (hexadecimal).

Ejemplo:

```
#include <iomanip>
#include <iostream>

int main(){
   int a=231;
   cout<<"En decimal= "<<a<<endl
         <<"en octal= "<<oct<<a<<endl
         <<"en hexadecimal= "<<hex<<a<<endl
         <<"en decimal otra vez= "<<setbase(10)<<a<<endl;
   return 0;
}
```

precisión

La precisión de los números en coma flotante se puede calibrar. Normalmente la precisión es de tipo flotante. Por ello el modificador **fixed** utilizado en un flujo de datos, indica que los números decimales serán ahora de precisión fija.

Después se puede utilizar el manipulador **setprecision** con un parámetro que indica el número de decimales que se utilizarán.

Ejemplo:

```
double pi=3.141592;
double x=3.1;

cout<<fixed;
cout<<setprecision(3)<<pi<<endl; //sale 3.142
cout<<setprecision(3)<<x<<endl; //Sale 3.100
```

anchura y rellenos

El manipulador **setw** permite establecer una anchura al escribir. Por ejemplo:

```
double x[]={2.34,1.2,45.6,2.344556676,6.7899};

for(int i=0;i<5;i++)
   cout<<setw(20)<<x[i]<<endl;
/*Sale:
                2.34
                 1.2
                45.6
             2.34456
              6.7899
*/
```

en el ejemplo los datos utilizan 20 caracteres para ser escritos y se alinean a la derecha respecto esos 20 caracteres.

El manipulador **setfill** permite establecer un carácter de relleno en las operaciones de anchura. Ejemplo:

```
double x[]={2.34,1.2,45.6,2.344556676,6.7899};

for(int i=0;i<5;i++)
   cout<<setw(20)<<setfill('-')<<x[i]<<endl;
/*Sale:
----------------2.34
-----------------1.2
----------------45.6
-------------2.34456
--------------6.7899
*/
```

También se pueden utilizar los manipuladores **left** y **right** para alinear la salida en el relleno:

```
double x[]={2.34,1.2,45.6,2.344556676,6.7899};

for(int i=0;i<5;i++)
   cout<<setw(20)<<setfill('-')<<left<<x[i]<<endl;
/*Sale
2.34----------------
1.2-----------------
45.6----------------
2.34456-------------
6.7899--------------
*/
```

manipuladores definidos por el programador

Además de los manipuladores comentados anteriormente, cada programador puede definir los suyos propios. Por ejemplo supongamos que deseamos crear un modificador llamado *tab*, que sirva para añadir el tabulador a la salida. En ese caso lo que habría que hacer es declararlo como función global que tome como parámetro un flujo (en el ejemplo de tipo **ostream**) y que devuelva un flujo del mismo tipo. Ejemplo:

```
ostream& tab(ostream &flujo){
   return flujo<<'\t';
}
```

El uso de este manipulador personal sería:

```
cout<<"Uno"<<tab<<"dos"<<endl;
/*sale Uno    Dos */
```

formato booleano

Normalmente al escribir valores booleanos se escribe un 1 si es valor verdadero o 0 si es falso. Para escribir el texto *true* o *false* en lugar de valores enteros se usa el modificador **boolalpha.** Ejemplo:

```
cout<<a<<endl; //Escribe 1
cout<<boolalpha<<a<<endl; //Escribe true
cout<<a<<endl; //Escribe true
cout<<noboolalpha;
cout<<a<<endl; //Escribe 1
```

Como se observa en el ejemplo para volver a escribir valores booleanos en su forma entera (1 o 0) se utiliza el manipulador **noboolalpha**

comprobación del flujo

fin de archivo

El método **eof**, permite saber si hemos llegado al final del archivo. Si es así devuelve verdadero.

error de formato

Cuando al leer de una corriente de bits, tenemos un error de formato (por ejemplo encontrarnos con texto cuando esperábamos números), entonces se activa un bit llamado *failbit.*

La función **fail** devuelve verdadero si ese bit está activado (es decir si hemos tenido un error de formato).

error de pérdida de datos

En el caso de los errores irrecuperables (se pierden datos en la lectura) se activa el llamado *badbit*. La activación de dicho bit se puede comprobar con el método **bad**

falta de errores

El método **good** devuelve un valor verdadero si ninguno de los métodos anteriores es verdadero. Es decir, el flujo está en estado *bueno*, se puede trabajar con él.

restaurar el flujo

El método de tipo **void**, **clear** permite restaurar un flujo a estado de bueno.

Ejemplo de uso de esos métodos:

```
int x;
bool valido=true;
do{
    cout<<"escribe un número: "<<endl;
    cin>>x;
    valido=cin.good();
    if(!valido) cout<<"Entrada incorrecta"<<endl;
    cin.clear();
    cin.ignore(2000,'\n');//Para ignorar los caracteres leídos
}while(!valido);
```

En este ejemplo se valida que realmente la lectura es de enteros y no de caracteres.

enlazar flujos de entrada y de salida

La función **tie** perteneciente a la clase **istream** permite asociar un flujo de salida (**ostream**). Ejemplo:

```
cin.tie(&cout);
```

En el ejemplo se observa que esta función requiere un puntero a un **ostream**. Dicho ejemplo no tiene sentido ya que lo que hace es asociar la corriente de entrada estándar a la salida estándar, algo que ya ocurre por defecto. Tiene sentido cuando las corrientes proceden de archivos.

[9.3] archivos

[9.3.1] el problema del almacenamiento de los datos

Los datos que maneja un programa están estructurados o en tipos básicos (en C++, **int, double, char,…)** o en estructuras complejas (las clases por ejemplo). Pero todas ellas permiten alejarnos de la realidad física de los datos en el ordenador; la realidad es que un ordenador sólo reconoce bits, unos y ceros.

Al almacenar los datos en archivos ocurre exactamente lo mismo, sólo habrá bits. Sin embargo las operaciones de lectura/escritura suelen trabajar de forma básica con bytes (que ya es un poco más reconocible por el programador), sin embargo al final esos bytes habrá que encajarlos en las estructuras de datos más apropiadas para representar información humana.

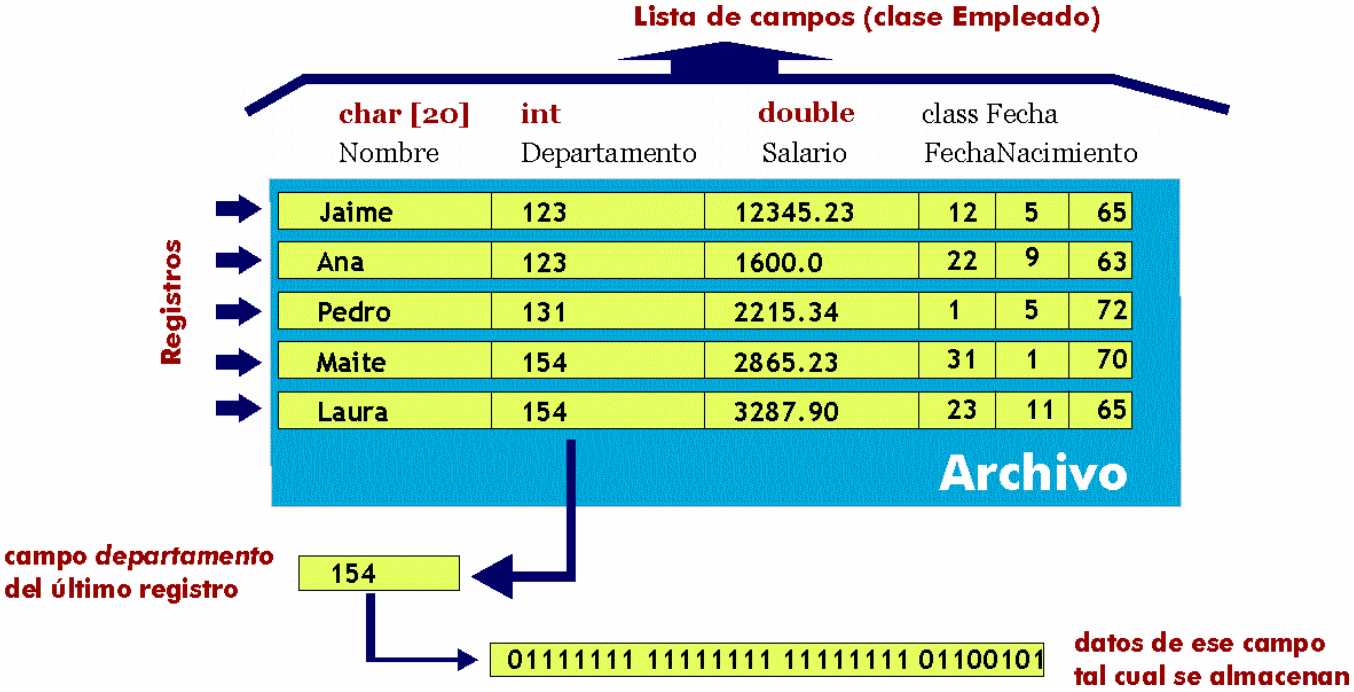


Ilustración 21, Estructura jerárquica de datos en un archivo

En ese *encaje* es donde reside el problema. Desde el punto de vista de un archivo los datos se almacenan en **registros.** Estos **registros** almacenan datos de un solo ejemplar (un solo objeto). Cada registro se compone de **campos;** un campo es una propiedad de cada objeto. De tal manera que la información puede organizarse en diferentes planos.

[9.3.2] archivos

Un archivo es una serie de datos almacenados en una misma estructura externa que reside en algún dispositivo de almacenamiento permanente (disco duro, CD-ROM,…). Aunque desde el punto de vista de C++ (como se verá luego) todo archivo es una corriente (flujo, secuencia) de bytes, conviene tener en cuenta que en general hay diferentes tipos de archivos. A continuación se destacan dos clasificaciones sobre dichos tipos de archivos.

tipos de archivo según su contenido

- **Archivos de texto.** Contienen información en forma de caracteres. Normalmente se organizan los caracteres en forma de líneas al final de cada cual se coloca un carácter de fin de línea (normalmente la secuencia **"\r\n"**). Al leer hay que tener en cuenta la que la codificación de caracteres puede variar (la **'ñ'** se puede codificar muy distinto según qué sistema utilicemos). Los códigos más usados son:
  - **ASCII.** Código de 7 bits que permite incluir 128 caracteres. En ellos no están los caracteres nacionales por ejemplo la 'ñ' del español) ni símbolos de uso frecuente (matemáticos, letras griegas,…). Por ello se uso el octavo bit para producir códigos de 8 bits, llamado ASCII extendido (lo malo es que los ASCII de 8 bits son diferentes en cada país).

- **ISO 8859-1.** El más usado en todos los países con lenguas procedentes de Europa Occidental. Se la llama codificación de Europa Occidental. Son 8 bits con el código ASCII más los símbolos frecuentes del inglés, español, francés, italiano o alemán entre otras lenguas de Europa Occidental.
- **Windows 1252**. Windows llama **ANSI** a esta codificación. En realidad se trata de un superconjunto de ISO 8859-1 que es utilizado en el almacenamiento de texto por parte de Windows.
- **Unicode**. La norma de codificación que intenta unificar criterios para hacer compatible la lectura de caracteres en cualquier idioma. Hay varias posibilidades de aplicación de Unicode, pero la más utilizada en la actualidad es la UTF-8 que es totalmente compatible con ASCII y que usando el octavo bit con valor 1 permite leer más bytes para poder almacenar cualquier número de caracteres (en la actualidad hay 50000)

- **Archivos binarios.** Almacenan datos que no son interpretables como texto (números, imágenes, etc.).

tipos de archivos por la forma de acceso

Según la forma en la que accedamos a los archivos disponemos de dos tipos de archivo:

- **Archivos secuenciales.** Se trata de archivos en los que el contenido se lee o escribe de forma continua. No se accede a un punto concreto del archivo, para leer cualquier información necesitamos leer todos los datos hasta llegar a dicha información. En general son los archivos de texto los que se suelen utilizar de forma secuencial.
- **Archivos de acceso directo.** Se puede acceder a cualquier dato del archivo conociendo su posición en el mismo. Dicha posición se suele indicar en bytes. En general los archivos binarios se utilizan mediante acceso directo.

[9.3.3] archivos desde el punto de vista de C++

Como se comentó antes para C++ un archivo no es más que una serie de bytes que finaliza con el byte de fin de archivo (EOF) representado normalmente por el código Ctrl-Z.

La manipulación de archivos en C++ obliga a incluir los archivos de cabecera **iostream** y **fstream** para poder utilizar las clases y funciones relacionadas con los archivos

[9.3.4] apertura de archivos

Hay tres clases que se utilizan para manipular archivos, son:

- **fstream**. La clase genérica de archivos vale tanto para lectura como para escritura
- **ifstream**. Clase orientada a manejar archivos para lectura
- **ofstream**. Clase orientada para manejar archivos para escritura

En el caso de utilizar un objeto de tipo **fstream**, al construir el objeto relacionado con el archivo se debe indicar el modo de apertura, que tiene que ser uno de estos valores (**flags**):

- **ios::in** Apertura en modo lectura
- **ios::out** Apertura en modo escritura
- **ios::app.** Para añadir datos al final del fichero

Estos son los *flags* fundamentales para manipular archivos secuenciales, pero existen también estos otros valores:

- **ios::ate** Abre el archivo y se coloca al final (normalmente para añadir al final del mismo)., pero permite el movimiento por cualquier parte del archivo.
- **ios::trunc** Descarta el contenido del archivo si es que existe (acción predeterminada también para **out**)
- **ios::binary** Abre el archivo de forma binaria (normalmente se abre en modo texto).

La construcción de un objeto de tipo **fstream** realizaría la apertura, esa construcción es:

```
fstream archivo("nombre.dat",ios::in);
```

En este caso se abre el archivo para lectura. El primer parámetro es la ruta al archivo

En lugar de utilizar objetos **fstream**, se puede utilizar **ifstreams** o **ofstreams**. En ese caso no hay que indicar el modo de apertura. Es decir este código es equivalente al anterior:

```
ifstream archivo("nombre.dat");
```

Al intentar abrir el archivo, se determinará si la operación tuvo éxito. El operador sobrecargado **!** devolverá verdadero si la apertura falló. Eso ocurre si se pone a 1 el **failbit** o el **badbit** (comentados anteriormente). Ejemplo:

```
fstream archivo("nombre.dat",ios::out);
if(!archivo) cout<<"Error en la apertura"<<endl;
```

Al abrir en lectura, el archivo debe existir, de otro modo tendríamos error. En escritura el fallo puede proceder de una mala ruta hacia el archivo o de que el archivo no permita su escritura por otras razones (por estar protegido por ejemplo).

La operación de escritura crea el archivo; si ya existía le borra y le crea de nuevo. Por eso si queremos añadir hay que abrir para añadir y no para escribir.

[9.4] uso de ficheros de texto

Los archivos de texto son secuenciales. Para escribir se suelen utilizar los operadores de extracción (>>). Los archivos secuenciales tienen como característica que al leer o escribir se empieza desde el principio del archivo hacia el final del mismo. Casi siempre los archivos secuenciales son de texto.

[9.4.1] escritura de archivos de texto

Al leer se pueden utilizar cualquiera de los métodos comentados para la clase **istream**, dependiendo del formato del archivo, por supuesto se pueden utilizar los operadores de desplazamiento (<<).

En cualquier caso los pasos son estos:

[1] Abrir el archivo. Posibilidades:

- **ofstream** *archivo*("ruta")
- **fstream** *archivo*("ruta",**ios::out**)

[2] Comprobar que la apertura es correcta, normalmente con el operador **!** tal que así:

```
if(!archivo){//el archivo no se abrió bien
```

[3] Escribir en el archivo. Mediante las instrucciones de escritura comentadas.

[4] Cerrar el archivo. Mediante el método **close**:

```
archivo.close();
```

Para distinguir los diferentes campos de cada registro (si el archivo contiene registros) se utiliza un delimitador. Al leer se suele tomar como delimitador el espacio en blanco si se usa el operador >>, pero con **getline** o con **get** se puede tomar otro delimitador (que lógicamente al escribir se habrá utilizado).

Supongamos que queremos realizar un programa en el que queremos que aparezca un listado de una serie de personas junto con un código personal (un número entero) y su salario (un *double*). Queremos guardar en el fichero estos datos, les separaremos con un espacio (por lo tanto si el nombre contiene espacios, tendríamos problemas, éste programa lo que hará será descartar esos espacios).

El código podría ser:

```
fstream archivo("personal.dat",ios::out);
char nombre[80];
int codigo;
double sueldo;
if(!archivo) cout<<"Error en la apertura del archivo";
else{
  do{
    cout<<"Escriba el codigo del empleado (0 para salir): ";
    cin>>codigo;
    if(codigo!=0){
       cout<<"Escriba el nombre";
       cin>>nombre;
       fflush(stdin);//vacía el bufer de lectura
       cout<<"Escriba el sueldo";
       cin>>sueldo;
       //escritura en el archivo, separando los campos
        //con espacios el fin de registro lo da un Intro
       archivo<<codigo<<" "<<nombre<<" "<<sueldo<<endl;
  }
}while(codigo!=0);
archivo.close();
```

Tras introducir datos, el archivo (que será de texto) podría tener este contenido (obsérvese la disposición de los datos):

```
1 Luis 1290.45
2 Ana 1876
3 Francisco 1567.32
4 Lorena 2312.9
```

[9.4.2] lectura de archivos secuenciales

Para leer de un archivo secuencial (del que lógicamente tendremos que conocer su estructura) los pasos son:

[1] Abrir el archivo

[2] Comprobar la apertura (al igual que en la escritura)

[3] Leer el siguiente registro o elemento del archivo tras comprobar que no hemos llegado al final del archivo (método **eof**)

[4] Si hemos llegado al final, cerrar el archivo (método **close**)

Ejemplo (lectura del contenido del archivo del apartado anterior):

```
fstream archivo("personal.dat",ios::in);
char nombre[80];
int codigo;
double sueldo;

if(!archivo) cout<<"Error en la apertura del archivo";
else{
    cout<<left<<setw(8)<<"Codigo"<<setw(20);
    cout<<"Nombre"<<setw(10)<<"Sueldo"<<endl;
    while(archivo.eof()==false){
        archivo>>codigo>>nombre>>sueldo;
        cout<<left<<setw(8)<<codigo<<setw(20);
        cout<<nombre<<setw(10)<<sueldo<<endl;
}
archivo.close();
```

[9.4.3] colocarse en un punto concreto del archivo

Si hiciera falta durante la lectura colocarse en un punto concreto del archivo, entonces se puede utilizar la función **seekg** de la clase **istream**. Gracias a ello se puede uno volver a colocar al principio del archivo usando:

```
archivo.seekg(0);
```

Ese cero significa que la posición de lectura en el archivo (el cursor) se coloque en el byte 0 (es decir al principio del todo). Se puede utilizar un segundo parámetro para indicar desde dónde se pueden iniciar los bytes. Ese desde dónde podría ser por ejemplo la constante **ios::cur** (desde la posición actual) o **ios::end** (desde el final del archivo). Ejemplos:

```
archivo.seekg(n,ios::cur);//Se mueve n bytes desde la
                          //posición actual
archivo.seekg(n,ios::end);//Se mueve n bytes desde el fin de
                          //archivo
```

En el caso de que estemos escribiendo, la función a utilizar es **seekp**. La función **tellp** nos dice en que posición del archivo estamos ahora (en el caso de la lectura es **tellg**).

[9.5] uso de ficheros binarios

Los ficheros de texto no son buenos para acceder rápidamente a la información. Los archivos binarios no almacenan los datos como texto; realmente en ellos hay registros normalmente del mismo tamaño (aunque a veces no).

Esto permite acceder en cualquier momento a un registro concreto ya que sabiendo su tamaño es fácil calcular en qué posición del archivo está.

Las funciones que trabajan con archivos binarios son **read** y **write**, ambas funciones trabajan con arrays de caracteres (graban bytes) por lo que si queremos grabar otro tipo de datos, habrá que convertirlos. Para ello se utiliza el operador, **reinterpret_cat<char *>** que convierte a forma de array de caracteres cualquier dato básico (sin usar esa función, seremos nosotros los que tendremos que convertir).

Tanto **read**, como **write** utilizan un segundo parámetro para indicar la cantidad de bytes a escribir. Casi siempre esa cifra se toma del operador **sizeof**.

[9.5.1] escritura binaria secuencial

Vamos a hacer el mismo ejemplo de lectura de archivos de texto. Sólo que ahora consideramos que la información de cada persona se almacena de esta forma: primero el número entero (un entero son 4 bytes), luego el nombre (dejamos 80 caracteres para ello), luego el salario en forma double (normalmente un double son 8 bytes). El código para escribir de forma secuencial sería:

```
fstream archivo("personal.bin",ios::in | ios::binary);
char nombre[80];
int codigo;
double sueldo;
if(!archivo) cout<<"Error en la apertura del archivo";
else{
  do{
   cout<<"Escriba el codigo del empleado (0 para salir): ";
   cin>>codigo;
   if(codigo!=0){
      cout<<"Escriba el nombre";
      cin>>nombre;
      fflush(stdin);//vacía el bufer de lectura
      cout<<"Escriba el sueldo";
      cin>>sueldo;
      //escritura en el archivo
      archivo.write(
        reinterpret_cast<char *>(&codigo),sizeof(codigo));
      archivo.write(nombre,80);
```

```
archivo.write(
        reinterpret_cast<char *>(&sueldo),sizeof(sueldo));
        }
       }while(codigo!=0);
    }
    archivo.close();
```

[9.5.2] lectura binaria secuencial

Para el ejemplo anterior el código sería:

```
fstream archivo("personal.bin",ios::in | ios::binary);
char nombre[80];
int codigo;
double sueldo;
if(!archivo) cout<<"Error en la apertura del archivo";
else{
 cout<<left<<setw(8)<<"Codigo"<<setw(20);
 cout<<"Nombre"<<setw(10)<<"Sueldo"<<endl;
 // lectura adelantada
 archivo.read(
  reinterpret_cast<char *>(&codigo),sizeof(codigo));
 archivo.read(nombre,80);
 archivo.read(
  reinterpret_cast<char *>(&sueldo),sizeof(sueldo));
 while(archivo.eof()==false){
     //mostrar en pantalla
      cout<<left<<setw(8)<<codigo<<setw(20);
      cout<<nombre<<setw(10)<<sueldo<<endl;
      //lectura del siguiente registro
       archivo.read(
         reinterpret_cast<char *>(&codigo),sizeof(codigo));
       archivo.read(nombre,80);
       archivo.read(
          reinterpret_cast<char *>(&sueldo),sizeof(sueldo));
  }
  archivo.close();
}
```

En el código se observa como en el caso de los archivos binarios para leerlos de forma secuencia hay que hacer una lectura adelantada antes de comprobar el fin de archivo (de otra forma, el último registro aparece dos veces).

[9.5.3] acceso aleatorio

El acceso aleatorio consiste en colocarse en cualquier parte de un archivo de registros a fin de leer o escribir cualquiera sin haber pasado anteriormente por los demás. Las instrucciones para realizar este acceso son las **seek** (**seekg** para lectura y **seekp** para escritura) y las **tell** (**tellg** y **tellp**) (comentadas anteriormente).

La instrucción **seek** tiene dos parámetros (el segundo opcional), el primero es el número de bytes que deseamos movernos y el segundo desde donde realizamos ese salto (si se omite se entenderá que el salto se produce desde el principio del archivo).

De esa forma para saltar a la posición del quinto registro del archivo usado como ejemplo anteriormente, la instrucción sería (suponemos que nos movemos para leer):

```
seekg((sizeof(int)+80+sizeof(double))*4);
```

El cálculo *sizeof(int)+80+sizeof(double))*4* en realidad calcula el tamaño del registro a partir del tamaño de cada campo. El hecho de multiplicar por cuatro es porque el primer registro estará en la posición cero; luego el registro *n* estará en la posición: *(n-1)*tamaño_registro*

[9.5.4] lectura y escritura simultáneas

Es muy habitual en los archivos binarios de acceso aleatorio, desear realizar operaciones de lectura y escritura en el mismo archivo. Simplemente se trata de usar *read* o *write* según interese. Eso es posible simplemente abriendo el archivo de esta forma:

```
fstream archivo("datos.bin", ios::in |
      ios::out | ios::binary);
```

Eso obliga a que el archivo ya exista. Si abrimos de esta otra forma (añadiendo **ios::trunc**):

```
fstream archivo("datos.bin", ios::in |
      ios::out | ios::trunc | ios::binary);
```

hace que el archivo, si existía anteriormente, se borre y se cree de nuevo otro con el mismo nombre (y cuyo contenido estará vacío).