Curso de C++

Autor: Salvador Pozo Coronado e-mail: salvador@conclase.net

TABLA DE CONTENIDO

Notas previas

Finalmente me he decidido a escribir un curso de C++ en una página web. Probablemente estoy pecando de presuntuoso, y me esté metiendo en camisa de once varas. Pero, algunas veces hay que ser valiente.

El curso no está muy lejos de estar terminado, y no pretende más que animar a los nuevos y futuros programadores autodidactas a incorporarse a esta gran y potente herramienta que es el C++.

La idea es ir añadiendo los nuevos capítulos a medida que los vaya escribiendo, dejando un margen de tiempo para incorporar las correcciones necesarias y las modificaciones que propongan los valientes que estén dispuestos a seguir el curso. Este margen de tiempo me permitirá además escribir los nuevos capítulos con la calma necesaria para que queden claros, y para que no me despidan de mi trabajo, ;-).

No pretendo ser original, (al menos no demasiado), consultaré libros, tutoriales, revistas, listas de correo, news, páginas web... En fin, aprovecharé cualquier fuente de datos que pueda, con el fin de conseguir un buen nivel. Lo que sí pretendo es ser ameno, no quiero que nadie se aburra leyendo el curso, y procuraré que los ejemplos sean divertidos.

Pretendo también (y me gustaría muchísimo), que el curso sea interactivo, propondré problemas, cuya resolución pasará a ser parte del curso. Además se añadirán las preguntas que vaya recibiendo, así como sus respuestas. Y en la lista de correo podremos discutir sobre los temas del curso entre todos aquellos que lo sigan.

He intentado que los ejemplos que ilustran cada capítulo corran en cualquier versión de compilador, sin embargo, he de decir que yo he usado el compilador Dev-C++ de Bloodshed en modo consola. Este compilador, está pensado para hacer programas en Windows.

Aprovecho para aclarar que los programas de Windows tienen dos modos de cara al usuario:

- El modo consola simula el funcionamiento de una ventana MS-DOS, trabaja en modo de texto, es decir, la ventana es una especie de tabla en la que cada casilla sólo puede contener un carácter. El modo consola de Windows no permite usar gráficos de alta resolución. Pero esto no es una gran pérdida, pues como veremos, ni C ni C++ incluyen manejo de gráficos de alta resolución. Esto se hace mediante librerías externas no estándar.
- El otro modo es el GUI, Interfaz Gráfico de Usuario. Es el modo tradicional de los programas de Windows, con ventanas, menús, iconos, etc. La creación de éste tipo de programas se explica en otro curso de este mismo sitio, y requiere el conocimiento de la librería de funciones Win API32.

Para aquellos de vosotros que programéis en otros entornos como Linux, Unix o Mac, he de decir que no os servirá el compilador Dev-C++, ya que está diseñado especialmente para Windows. Pero esto no es un problema serio, todos los sistemas

operativos disponen de compiladores de C++ que soportan la norma ANSI, sólo menciono Dev-C++ y Windows porque es el entorno en el que yo, me muevo actualmente.

Además intentaré no salirme del ANSI, es decir del C++ estándar, así que no es probable que surjan problemas con los compiladores.

De nuevo aprovecho para hacer una aclaración. Resumidamente, el ANSI define un conjunto de reglas. Cualquier compilador de C o de C++ debe cumplir esas reglas, si no, no puede considerarse un compilador de C o C++. Estas reglas definen las características de un compilador en cuanto a palabras reservadas del lenguaje, comportamiento de los elementos que lo componen, funciones externas que se incluyen, etc. Un programa escrito en ANSI C o en ANSI C++, podrá compilarse con cualquier compilador que cumpla la norma ANSI. Se puede considerar como una homologación o etiqueta de calidad de un compilador.

Todos los compiladores incluyen, además del ANSI, ciertas características no ANSI, por ejemplo librerías para gráficos. Pero mientras no usemos ninguna de esas características, sabremos que nuestros programas son transportables, es decir, que podrán ejecutarse en cualquier ordenador y con cualquier sistema operativo.

Este curso es sobre C++, con respecto a las diferencias entre C y C++, habría mucho que hablar, pero no es este el momento adecuado. Si sientes curiosidad, consulta la sección de preguntas frecuentes. Pero para comprender muchas de estas diferencias necesitarás cierto nivel de conocimientos de C++.

Los programas de ejemplo que aparecen en el texto están escritos con la fuente courier y en color azul con el fin de mantener las tabulaciones y distinguirlos del resto del texto. Cuando sean largos se incluirá también un fichero con el programa, que se podrá descargar directamente.

Cuando se exponga la sintaxis de cada sentencia se adoptarán ciertas reglas, que por lo que sé son de uso general en todas la publicaciones y ficheros de ayuda. Los valores entre corchetes "[]" son opcionales, con una excepción: cuando aparezcan en negrita "**[]**", en ese caso indicarán que se deben escribir los corchetes. El separador "|" delimita las distintas opciones que pueden elegirse. Los valores entre "<>" se refieren a nombres. Los textos sin delimitadores son de aparición obligatoria.

Proceso para la obtención de un programa ejecutable

Probablemente éste es el lugar más adecuado para explicar cómo se obtiene un fichero ejecutable a partir de un programa C++.

Para empezar necesitamos un poco de vocabulario técnico. Veremos algunos conceptos que se manejan frecuentemente en cualquier curso de programación y sobre todo en manuales de C y C++.

Fichero fuente y programa o código fuente:

Los programas C y C++ se escriben con la ayuda de un editor de textos del mismo modo que cualquier texto corriente. Los ficheros que contiene programas en C o C++ en forma de texto se conocen como ficheros fuente, y el texto del programa que contiene se conoce como programa fuente. Nosotros **siempre** escribiremos programas fuente y los guardaremos en ficheros fuente.

Ficheros objeto, código objeto y compiladores:

Los programas fuente no pueden ejecutarse. Son ficheros de texto, pensados para que los comprendan los seres humanos, pero incomprensibles para los ordenadores.

Para conseguir un programa ejecutable hay que seguir algunos pasos. El primero es compilar o traducir el programa fuente a su código objeto equivalente. Este es el trabajo que hacen los compiladores de C y C++. Consiste en obtener un fichero equivalente a nuestro programa fuente comprensible para el ordenador, éste fichero se conoce como fichero objeto, y su contenido como código objeto.

Los compiladores son programas que leen un fichero de texto que contiene el programa fuente y generan un fichero que contiene el código objeto.

El código objeto no tiene ningún significado para los seres humanos, al menos no directamente. Además es diferente para cada ordenador y para cada sistema operativo. Por lo tanto existen diferentes compiladores para diferentes sistemas operativos y para cada tipo de ordenador.

Librerías:

Junto con los compiladores de C y C++, se incluyen ciertos ficheros llamados librerías. Las librerías contienen el código objeto de muchos programas que permiten hacer cosas comunes, como leer el teclado, escribir en la pantalla, manejar números, realizar funciones matemáticas, etc. Las librerías están clasificadas por el tipo de trabajos que hacen, hay librerías de entrada y salida, matemáticas, de manejo de memoria, de manejo de textos, etc.

Hay un conjunto de librerías muy especiales, que se incluyen con todos los compiladores de C y de C++. Son las librerías ANSI o estándar. Pero también hay librerías no estándar, y dentro de estas las hay públicas y comerciales. En éste curso sólo usaremos librerías ANSI.

Ficheros ejecutables y enlazadores:

Cuando obtenemos el fichero objeto, aún no hemos terminado el proceso. El fichero objeto, a pesar de ser comprensible para el ordenador, no puede ser ejecutado. Hay varias razones para eso:

1. Nuestros programas usaran, en general, funciones que estarán incluidas en librerías externas, ya sean ANSI o no. Es necesario combinar nuestro fichero objeto con esas librerías para obtener un ejecutable.

2. Muy a menudo, nuestros programas estarán compuestos por varios ficheros fuente, y de cada uno de ellos se obtendrá un fichero objeto. Es necesario unir todos los ficheros objeto, más las librerías en un único fichero ejecutable.
3. Hay que dar ciertas instrucciones al ordenador para que cargue en memoria el programa y los datos, y para que organice la memoria de modo que se disponga de una pila de tamaño adecuado, etc. La pila es una zona de memoria que se usa para que el programa intercambie datos con otros programas o con otras partes del propio programa. Veremos esto con más detalle durante el curso.

Existe un programa que hace todas estas cosas, se trata del "link", o enlazador. El enlazador toma todos los ficheros objeto que componen nuestro programa, los combina con los ficheros de librería que sea necesario y crea un fichero ejecutable.

Una vez terminada la fase de enlazado, ya podremos ejecutar nuestro programa.

Errores:

Por supuesto, somos humanos, y por lo tanto nos equivocamos. Los errores de programación pueden clasificarse en varios tipos, dependiendo de la fase en que se presenten.

Errores de sintaxis: son errores en el programa fuente. Pueden deberse a palabras reservadas mal escritas, expresiones erróneas o incompletas, variables que no existen, etc. Los errores de sintaxis se detectan en la fase de compilación. El compilador, además de generar el código objeto, nos dará una lista de errores de sintaxis. De hecho nos dará sólo una cosa o la otra, ya que si hay errores no es posible generar un código objeto.

Avisos: además de errores, el compilador puede dar también avisos (warnings). Los avisos son errores, pero no lo suficientemente graves como para impedir la generación del código objeto. No obstante, es importante corregir estos avisos, ya que el compilador tiene que decidir entre varias opciones, y sus decisiones no tienen por qué coincidir con lo que nosotros pretendemos, se basan en las directivas que los creadores del compilador decidieron durante su creación.

Errores de enlazado: el programa enlazador también puede encontrar errores. Normalmente se refieren a funciones que no están definidas en ninguno de los ficheros objetos ni en las librerías. Puede que hayamos olvidado incluir alguna librería, o algún fichero objeto, o puede que hayamos olvidado definir alguna función o variable, o lo hayamos hecho mal.

Errores de ejecución: incluso después de obtener un fichero ejecutable, es posible que se produzcan errores. En el caso de los errores de ejecución normalmente no obtendremos mensajes de error, sino que simplemente el programa terminará bruscamente. Estos errores son más difíciles de detectar y corregir. Existen programas auxiliares para buscar estos errores, son los llamados depuradores (debuggers). Estos programas permiten detener la ejecución de nuestros programas, inspeccionar variables y ejecutar nuestro programa paso a paso. Esto resulta útil para detectar excepciones, errores sutiles, y fallos que se presentan dependiendo de circunstancias distintas.

Errores de diseño: finalmente los errores más difíciles de corregir y prevenir. Si nos hemos equivocado al diseñar nuestro algoritmo, no habrá ningún programa que nos pueda ayudar a corregir los nuestros. Contra estos errores sólo cabe practicar y pensar.

Propósito de C y C++

¿Qué clase de programas y aplicaciones se pueden crear usando C y C++?

La respuesta es muy sencilla: TODOS.

Tanto C como C++ son lenguajes de programación de propósito general. Todo puede programarse con ellos, desde sistemas operativos y compiladores hasta aplicaciones de bases de datos y procesadores de texto, pasando por juegos, aplicaciones a medida, etc.

Oirás y leerás mucho sobre éste tema. Sobre todo diciendo que estos lenguajes son complicados y que requieren páginas y páginas de código para hacer cosas que con otros lenguajes se hacen con pocas líneas. Esto es una verdad a medias. Es cierto que un listado completo de un programa en C o C++ para gestión de bases de datos (por poner un ejemplo) puede requerir varios miles de líneas de código, y que su equivalente en Visual Basic sólo requiere unos pocos cientos. Pero detrás de cada línea de estos compiladores de alto nivel hay cientos de líneas de código en C, la mayor parte de estos compiladores están respaldados por enormes librerías escritas en C. Nada te impide a ti, como programador, usar librerías, e incluso crear las tuyas propias.

Una de las propiedades de C y C++ es la reutilización del código en forma de librerías de usuario. Después de un tiempo trabajando, todos los programadores desarrollan sus propias librerías para aquellas cosas que hacen frecuentemente. Y además, raramente piensan en ello, se limitan a usarlas.

Además, los programas escritos en C o C++ tienen otras ventajas sobre el resto. Con la excepción del ensamblador, generan los programas más compactos y rápidos. El código es transportable, es decir, un programa ANSI en C o C++ podrá ejecutarse en cualquier máquina y bajo cualquier sistema operativo. Y si es necesario, proporcionan un acceso a bajo nivel de hardware sólo igualado por el ensamblador.

Otra ventaja importante, C tiene más de 30 años de vida, y C++ casi 20 y no parece que su uso se debilite demasiado. No se trata de un lenguaje de moda, y probablemente a ambos les quede aún mucha vida por delante. Sólo hay que pensar que sistemas operativos como Linux, Unix o incluso Windows se escriben casi por completo en C.

Por último, existen varios compiladores de C y C++ gratuitos, o bajo la norma GNU, así como cientos de librerías de todo propósito y miles de programadores en todo el mundo, muchos de ellos dispuestos a compartir su experiencia y conocimientos.

CAPITULO 1 Toma de contacto

Me parece que la forma más rápida e interesante de empezar, y no perder potenciales seguidores de este curso, es con un ejemplo. Veamos nuestro primer programa C++. Esto nos ayudará a sentar unas bases que resultarán muy útiles para los siguientes ejemplos que irán apareciendo.

```
int main()
{
   int numero;

   numero = 2 + 2;
   return 0;
}
```

No te preocupes demasiado si aún no captas todos los matices de este pequeño programa. Aprovecharemos la ocasión para explicar algunas de las peculiaridades de C++, aunque de hecho, este programa es *casi* un ejemplo de programa C. Pero eso es otro tema. En realidad C++ incluye a C. En general, un programa en C podrá compilarse usando un compilador de C++. Pero ya veremos este tema en otro lugar, y descubriremos en qué consisten las diferencias.

Iremos repasando, muy someramente, el programa, línea a línea:

- Primera línea: "`int main()`"

Se trata de una línea muy especial, y la encontrarás en todos los programas C y C++. Es el principio de la definición de una función. Todas las funciones C toman unos valores de entrada, llamados parámetros o argumentos, y devuelven un valor de retorno. La primera palabra: "int", nos dice el tipo del valor de retorno de la función, en este caso un número entero. La función "main" siempre devuelve un entero. La segunda palabra es el nombre de la función, en general será el nombre que usaremos cuando queramos usar o llamar a la función.

En este caso "main" es una función especial, ya que nosotros no la usaremos nunca explicitamente, es decir, nunca encontrarás en ningún programa una línea que invoque a la función "main". Esta función será la que tome el control automáticamente cuando se ejecute nuestro programa. Otra pista por la que sabemos que se trata de una función son los paréntesis, todas las definiciones de funciones incluyen una lista de argumentos de entrada entre paréntesis, en nuestro ejemplo es una lista vacía, es decir, nuestra función no admite parámetros.

- Segunda línea: "`{`"

Aparentemente es una línea muy simple, las llaves encierran el cuerpo o definición de la función. Más adelante veremos que también tienen otros usos.

- Tercera línea: "`int numero;`"

Esta es nuestra primera sentencia, todas las sentencias terminan con un punto y coma. Esta concretamente es una declaración de variable. Una declaración nos dice, a nosotros y al compilador, que usaremos una variable "numero" de tipo entero. Esta declaración obliga al compilador a reservar un espacio de memoria para almacenar la variable "numero", pero no le da ningún valor inicial. En general contendrá "basura", es decir, lo que hubiera en esa zona de memoria cuando se le reservó espacio. En C y C++ es obligatorio declarar las variables que usará el programa.

C y C++ distinguen entre mayúsculas y minúsculas, así que "int numero;" es distinto de "int NUMERO;", y también de "INT numero;".

- Cuarta línea: ""

Una línea vacía, no sirve para nada, al menos desde el punto de vista del compilador, pero sirve para separar visualmente la parte de declaración de variables de la parte de código que va a continuación. Se trata de una división arbitraria. Desde el punto de vista del compilador, tanto las declaraciones de variables como el código son sentencias válidas. La separación nos ayudará a distinguir visualmente dónde termina la declaración de variables. Una labor análoga la desempeña el tabulado de las líneas: ayuda a hacer los programas más fáciles de leer.

- Quinta línea: "`numero = 2 + 2;`"

Se trata de otra sentencia, ya que acaba con punto y coma. Esta es una sentencia de asignación. Le asigna a la variable "numero" el valor resultante de la operación "2 + 2".

- Sexta línea: "`return 0;`"

De nuevo una sentencia, "return" es una palabra reservada, propia de C y C++. Indica al programa que debe abandonar la ejecución de la función y continuar a partir del punto en que se la llamó. El 0 es el valor de retorno de nuestra función. Cuando "main" retorna con 0 indica que todo ha ido bien. Un valor distinto suele indicar un error. Imagina que nuestro programa es llamado desde un fichero de comandos, un fichero "bat" o un "script". El valor de retorno de nuestro programa se puede usar para tomar decisiones dentro de ese fichero. Pero somos nosotros, los programadores, los que decidiremos el significado de los valores de retorno.

- Séptima línea: "`}`"

Esta es la llave que cierra el cuerpo o definición de la función.

Lo malo de este programa, a pesar de sumar correctamente "2+2", es que aparentemente no hace nada. No acepta datos externos y no proporciona ninguna salida de ningún tipo. En realidad es absolutamente inútil, salvo para fines didácticos, que es para lo que fue creado. Paciencia, iremos poco a poco. En los siguientes capítulos veremos tres conceptos básicos: variables, funciones y operadores. Después estaremos en disposición de empezar a trabajar con ejemplos más interactivos.

CAPITULO 2 Tipos de variables I

Una variable es un espacio reservado en el ordenador para contener valores que pueden cambiar durante la ejecución de un programa. Los tipos determinan cómo se manipulará la información contenida en esas variables. No olvides, si es que ya lo sabías, que la información en el interior de la memoria del ordenador es siempre binaria, al menos a un cierto nivel. El modo en que se interpreta la información almacenda en la memoria de un ordenador es siempre arbitraria, es decir, el mismo valor puede usarse para codificar una letra, un número, una instrucción de programa, etc. El tipo nos dice a nosotros y al compilador cómo debe interpretarse y manipularse la información binaria almacenada en la memoria de un ordenador.

De momento sólo veremos los tipos fundamentales, que son: void, char, int, float y double, en C++ se incluye también el tipo bool. También existen ciertos modificadores, que permiten ajustar ligeramente ciertas propiedades de cada tipo; los modificadores puede ser: short, long, signed y unsigned o combinaciones de ellos. También veremos en este capítulo los tipos enumerados, enum.

Tipos fundamentales

En C sólo existen cinco tipos fundamentales y los tipos enumerados, C++ añade un septimo tipo, el bool, y el resto de los tipos son derivados de ellos. Los veremos uno por uno, y veremos cómo les afectan cada uno de los modificadores.

La definiciones de sintaxis de C++ se escribirán usando el color verde. Los valores entre [] son opcionales, los valores separados con | indican que sólo debe escogerse uno de los valores. Los valores entre <> indican que debe escribirse obligatoriamente un texto que se usará como el concepto que se escribe en su interior.

[signed|unsigned] char <identificador> significa que se puede usar signed o unsigned, o ninguna de las dos, ya que ambas están entre []. Además debe escribirse un texto, que debe ser una única palabra que actuará como identificador o nombre de la variable. Este identificador lo usaremos para referirnos a la variable durante el programa. Serán válidos estos ejemplos:

```
signed char Cuenta
unsigned char letras
char caracter
```

Tipo "char" o carácter:

```
[signed|unsigned] char <identificador>
```

Es el tipo básico alfanumérico, es decir que puede contener una carácter, un dígito numérico o un signo de puntuación. Desde el punto de vista del ordenador, todos esos valores son caracteres. En C y C++ éste tipo siempre contiene un único carácter del código ASCII. El tamaño de memoria es de 1 byte u octeto. Hay que notar que **en C un carácter es tratado** en todo **como un número**, de hecho puede ser declarado con y sin signo. Y si no se especifica el modificador de signo, se asume que es con signo.

Este tipo de variables es apto para almacenar números pequeños, como los dedos que tiene una persona, o letras, como la inicial de mi nombre de pila.

Tipo "int" o entero:

```
[signed|unsigned] [short|long] int <identificador>
[signed|unsigned] long [int] <identificador>
[signed|unsigned] short [int] <identificador>
```

Las variables enteras almacenan números enteros dentro de los límites de su tamaño, a su vez, ese tamaño depende de la plataforma del compilador, y del número de bits que use por palabra de memoria: 8, 16, 23... No hay reglas fijas para saber el mayor número que podemos almacenar en cada tipo: int, long int o short int; depende en gran medida del compilador y del ordenador. Sólo podemos estar seguros de que ese número en short int es menor o igual que en int, y éste a su vez es menor o igual que en long int. Veremos cómo averiguar estos valores cuando estudiemos los operadores.

Este tipo de variables es útil para almacenar números relativamente grandes, pero sin decimales, por ejemplo el dinero que tienes en el banco, salvo que seas Bill Gates, o el número de lentejas que hay en un kilo de lentejas.

Tipo "float" o coma flotante:

```
float <identificador>
```

Las variables de este tipo almacenan números en formato de coma flotante, mantisa y exponente, para entendernos, son números con decimales. Son aptos para variables de tipo real, como por ejemplo el cambio entre euros y pesetas. O para números muy grandes, como la producción mundial de trigo, contada en granos. El fuerte de estos números no es la precisión, sino el orden de magnitud, es decir lo grande o pequeño que es el número que contiene. Por ejemplo, la siguiente cadena de operaciones no dará el resultado correcto:

```
float a = 12335545621232154;
a = a + 1;
a = a - 12335545621232154;
```

Finalmente, "a" valdrá 0 y no 1, como sería de esperar. Los formatos en coma flotante sacrifican precisión en favor de tamaño. Si embargo el ejemplo si funcionaría con números más pequeños. Esto hace que las variables de tipo float no sean muy adecuadas para los bucles, como veremos más adelante.

Puede que te preguntes (alguien me lo ha preguntado), qué utilidad tiene algo tan impreciso. La respuesta es: aquella que tu, como programador, le encuentres. Te aseguro que float se usa muy a menudo. Por ejemplo, para trabajar con temperaturas, la precisión es suficiente para el margen de temperaturas que normalmente manejamos y para almacenar al menos tres decimales. Pero hay cientos de otras situaciones en que resultan muy útiles.

Tipo "bool" o Booleana:

```
bool <identificador>
```

Las variables de este tipo sólo pueden tomar dos valores "true" o "false". Sirven para evaluar expresiones lógicas. Este tipo de variables se puede usar para almacenar respuestas, por ejemplo: ¿Posees carné de conducir?. O para almacenar informaciones que sólo pueden tomar dos valores, por ejemplo: qué mano usas para escribir. En estos casos debemos acuñar una regla, en este ejemplo, podría ser diestro->"true", zurdo->"false".

Tipo "double" o coma flotante de doble precisión:

```
[long] double <identificador>
```

Las variables de este tipo almacenan números en formato de coma flotante, mantisa y exponente, al igual que float, pero usan mayor precisión. Son aptos para variables de tipo real. Usaremos estas variables cuando trabajemos con números grandes, pero también necesitemos gran precisión. Lo siento, pero no se me ocurre ahora ningún ejemplo.

Bueno, también me han preguntado por qué no usar siempre double o long double y olvidarnos de float. La respuesta es que C siempre ha estado orientado a la economía de recursos, tanto en cuanto al uso de memoria como al uso de procesador. Si tu problema no requiere la precisión de un double o long double, ¿por que derrochar recursos?. Por ejemplo, en el compilador Dev-C++ float requiere 4 bytes, double 8 y long double 12, por lo tanto, para manejar un número en formato de long double se requiere el triple de memoria y el triple o más tiempo de procesador que para manejar un float.

Como programadores estamos en la obligación de no desperdiciar nuestros recursos, y mucho más los recursos de nuestros clientes, para los que haremos nuestros programas. C y C++ nos dan un gran control sobre estas características, es nuestra responsabilidad aprender a usarlos como es debido.

Tipo "void" o sin tipo:

```
void <identificador>
```

Es un tipo especial que indica la ausencia de tipo. Se usa en funciones que no devuelven ningún valor, también en funciones que no requieren parámetros, aunque este uso sólo es obligatorio en C, y opcional en C++, también se usará en la declaración de punteros genéricos, lo veremos más adelante.

Las funciones que no devuelven valores parecen una contradicción. En lenguajes como Pascal, estas funciones se llaman procedimientos. Simplemente hacen su trabajo, y no revuelven valores. Por ejemplo, funciones como borrar la pantalla, no tienen nada que devolver, hacen su trabajo y regresan. Lo mismo se aplica a funciones sin parámetros de entrada, el mismo ejemplo de la función para borrar la pantalla, no requiere ninguna entrada para poder hacer su cometido.

Tipo "enum" o enumerado:

```
enum [<identificador de tipo>] {<nombre de constante> [= <valor>],
...} [lista de variables];
```

Se trata de una sintaxis muy elaborada, pero no te asustes, cuando te acostumbres a ver éste tipo de cosas las comprenderás mejor.

Este tipo nos permite definir conjuntos de constantes, normalmente de tipo int, llamados datos de tipo enumerado. Las variables declaradas de este tipo sólo podrán tomar valores entre los definidos.

El identificador de tipo es opcional, y nos permitirá declarar más variables del tipo enumerado en otras partes del programa:

```
[enum] <identificador de tipo> <variable1> [,<variable2>[...]];
```

La lista de variables también es opcional. Sin embargo, al menos uno de los dos componentes opcionales debe aparecer en la definición del tipo enumerado.

Varios identificadores pueden tomar el mismo valor, pero cada identificador sólo puede usarse en un tipo enumerado. Por ejemplo:

```
enum tipohoras { una=1, dos, tres, cuatro, cinco, seis,
  siete, ocho, nueve, diez, once, doce,
  trece=1, catorce, quince, dieciseis, diecisiete, dieciocho,
  diecinueve, veinte, ventiuna, ventidos, ventitres, venticuatro =
0};
```

En este caso, una y trece valen 1, dos y catorce valen 2, etc. Y veinticuatro vale 0. Como se ve en el ejemplo, una vez se asigna un valor a un elemento de la lista, los siguientes toman valores correlativos. Si no se asigna ningún valor, el primer elemento tomará el valor 0.

Los nombres de las constantes pueden utilizarse en el programa, pero no pueden ser leídos ni escritos. Por ejemplo, si el programa en un momento determinado nos pregunta la hora, no podremos responder *doce* y esperar que se almacene su valor correspondiente. Del mismo modo, si tenemos una variable enumerada con el valor *doce* y la mostramos por pantalla, se mostrará 12, no *doce*. Deben considerarse como "etiquetas" que sustituyen a enteros, y que hacen más comprensibles los programas. Insisto en que internamente, para el compilador, sólo son enteros, en el rango de valores válidos definidos en cada enum.

Palabras reservadas usadas en este capítulo

Las palabras reservadas son palabras propias del lenguaje de programación. Están reservadas en el sentido de que no podemos usarlas como identificadores de variables o de funciones.

char, int, float, double, bool, void, enum, unsigned, signed, long, short, true y false.

CAPITULO 3 Funciones I: Declaración y definición

Las funciones son un conjunto de instrucciones que realizan una tarea específica. En general toman unos valores de entrada, llamados parámetros y proporcionan un valor de salida o valor de retorno; aunque tanto unos como el otro pueden no existir.

Tal vez sorprenda que las introduzca tan pronto, pero como son una herramienta muy valiosa, y se usan en todos los programas C++, creo que debemos tener, al menos, una primera noción de su uso.

Al igual que con las variables, las funciones pueden declararse y definirse.

Una declaración es simplemente una presentación, una definición contiene las instrucciones con las que realizará su trabajo al función.

En general, la definición de una función se compone de las siguientes secciones, aunque pueden complicarse en ciertos casos:

- Opcionalmente, una palabra que especifique el tipo de almacenamiento, puede ser "extern" o "static". Si no se especifica es "extern". No te preocupes de esto todavía, de momento sólo usaremos funciones externas, sólo lo menciono porque es parte de la declaración. Una pista: las funciones declaradas como extern están disponibles para todo el programa, las funciones static pueden no estarlo.
- El tipo del valor de retorno, que puede ser "void", si no necesitamos valor de retorno. Si no se establece, por defecto será "int". Aunque en general se considera de mal gusto omitir el tipo de valor de retorno.
- Modificadores opcionales. Tienen un uso muy específico, de momento no entraremos en este particular, lo veremos en capítulos posteriores.
- El nombre de la función. Es costumbre, muy útil y muy recomendable, poner nombres que indiquen, lo más claramente posible, qué es lo que hace la función, y que permitan interpretar qué hace el programa con sólo leerlo. Cuando se precisen varias palabras para conseguir este efecto existen varias reglas aplicables de uso común. Una consiste en separar cada palabra con un "_", la otra, que yo prefiero, consiste en escribir la primera letra de cada palabra en mayúscula y el resto en minúsculas. Por ejemplo, si hacemos una función que busque el número de teléfono de una persona en una base de datos, podríamos llamarla "busca_telefono" o "BuscaTelefono".
- Una lista de declaraciones de parámetros entre paréntesis. Los parámetros de una función son los valores de entrada (y en ocasiones también de salida). Para la función se comportan exactamente igual que variables, y de hecho cada parámetro se declara igual que una variable. Una lista de parámetros es un conjunto de declaraciones de parámetros separados con comas. Puede tratarse de una lista vacía. En C es preferible usar la forma "func(void)" para listas de parámetros vacías. En C++ este procedimiento se considera obsoleto, se usa simplemente "func()"..
- Un cuerpo de función que representa el código que será ejecutado cuando se llame a la función. El cuerpo de la función se encierra entre llaves "{}"

Una función muy especial es la función "main". Se trata de la función de entrada, y debe existir siempre, será la que tome el control cuando se ejecute un programa en C. Los programas Windows usan la función WinMain() como función de entrada, pero esto se explica en otro lugar.

Existen reglas para el uso de los valores de retorno y de los parámetros de la función "main", pero de momento la usaremos como "int main()" o "int main(void)", con un entero como valor de retorno y sin parámetros de entrada. El valor de retorno indicará si el programa ha terminado sin novedad ni errores retornando cero, cualquier otro valor de retorno indicará un código de error.

En C++ es obligatorio usar prototipos. Un prototipo es una declaración de una función. Consiste en una definición de la función sin cuerpo y terminado con un ";". La estructura de un prototipo es:

```
<tipo> func(<lista de declaración de parámetros>);
```

Por ejemplo:

```
int Mayor(int a, int b);
```

Sirve para indicar al compilador los tipos de retorno y los de los parámetros de una función, de modo que compruebe si son del tipo correcto cada vez que se use esta función dentro del programa, o para hacer las conversiones de tipo cuando sea necesario. Los nombres de los parámetros son opcionales, y se incluyen como documentación y ayuda en la interpretación y comprensión del programa. El ejemplo de prototipo anterior sería igualmente válido y se podría poner como:

```
int Mayor(int,int);
```

Esto sólo indica que en algún lugar del programa se definirá una función "Mayor" que admite dos parámetros de tipo "int" y que devolverá un valor de tipo "int". No es necesario escribir nombres para los parámetros, ya que el prototipo no los usa. En otro lugar del programa habrá una definición completa de la función.

Normalmente, las funciones se declaran como prototipos dentro del programa, o se incluyen estos prototipos desde un fichero externo, (usando la directiva "#include", ver en el siguiente capítulo el operador de preprocesador.)

Ya lo hemos dicho más arriba, pero las funciones son "extern" por defecto. Esto quiere decir que son accesibles desde cualquier punto del programa, aunque se encuentren en otros ficheros fuente del mismo programa. En contraposición las funciones declaradas "static" sólo son accesibles dentro del fichero fuente donde se definen.

La definición de la función se hace más adelante o más abajo, según se mire. Lo habitual es hacerlo después de la función "main".

La estructura de un programa en C o C++ quedaría así:

```
[directivas del pre-procesador: includes y defines]
```

```
[declaración de variables globales]
[prototipos de funciones]
función main
[definiciones de funciones]
```

Una definición de la función "Mayor" podría ser la siguiente:

```
int Mayor(int a, int b)
{
   if(a > b) return a; else return b;
}
```

Los programas complejos se escriben normalmente usando varios ficheros fuente. Estos ficheros se compilan separadamente y se enlazan juntos. Esto es una gran ventaja durante el desarrollo y depuración de grandes programas, ya que las modificaciones en un fichero fuente sólo nos obligarán a compilar ese fichero fuente, y no el resto, con el consiguiente ahorro de tiempo.

La definición de las funciones puede hacerse dentro de los ficheros fuente o enlazarse desde librerías precompiladas. La diferencia entre una declaración y una definición es que la definición posee un cuerpo de función.

En C++ es obligatorio el uso funciones prototipo, y aunque en C no lo es, resulta altamente recomendable.

Palabras reservadas usadas en este capítulo

extern y static.

CAPITULO 4 Operadores I

Los operadores son elementos que disparan ciertos cálculos cuando son aplicados a variables o a otros objetos en una expresión.

Existe una división en los operadores atendiendo al número de operandos que afectan. Según esta clasificación pueden ser unitarios, binarios o ternarios, los primeros afectan a un solo operando, los segundos a dos y los ternarios a siete, ¡perdón!, a tres.

Hay varios tipos de operadores, clasificados según el tipo de objetos sobre los que actúan.

Operadores aritméticos

Son usados para crear expresiones matemáticas. Existen dos operadores aritméticos unitarios, '+' y '-' que tienen la siguiente sintaxis:

+ <expresión>

- <expresión>

Asignan valores positivos o negativos a la expresión a la que se aplican.

En cuanto a los operadores binarios existen siete. '+', '-', '*' y '/', tienen un comportamiento análogo, en cuanto a los operandos, ya que admiten enteros y de coma flotante. Sintaxis:

<expresión> + <expresión>

<expresión> - <expresión>

<expresión> * <expresión>

<expresión> / <expresión>

Evidentemente se trata de las conocidísimas operaciones aritméticas de suma, resta, multiplicación y división, que espero que ya domines a su nivel tradicional, es decir, sobre el papel. El operador de módulo '%', devuelve el resto de la división entera del primer operando entre el segundo. Por esta razón no puede ser aplicado a operandos en coma flotante.

<expresión> % <expresión>

Por último otros dos operadores unitarios. Se trata de operadores un tanto especiales, ya que sólo pueden trabajar sobre variables, pues implican una asignación. Se trata de los operadores '++' y '--'. El primero incrementa el valor del operando y el segundo lo decrementa, ambos en una unidad. Existen dos modalidades, dependiendo de que se use el operador en la forma de prefijo o de sufijo. Sintaxis:

<variable> ++ (post-incremento)

++ <variable> (pre-incremento)

<variable>-- (post-decremento)

-- <variable> (pre-decremento)

En su forma de prefijo, el operador es aplicado antes de que se evalúe el resto de la expresión; en la forma de sufijo, se aplica después de que se evalúe el resto de la expresión. Veamos un ejemplo, en las siguientes expresiones "a" vale 100 y "b" vale 10:

```
c = a + ++b;
```

En este primer ejemplo primero se aplica el pre-incremento, y b valdrá 11 a continuación se evalúa la expresión "a+b", que dará como resultado 111, y por último se asignará este valor a c, que valdrá 111.

```
c = a + b--;
```

En este segundo ejemplo primero se avalúa la expresión "a+b", que dará como resultado 110, y se asignará este valor a c, que valdrá 110. Finalmente se aplica en post-decremento, y b valdrá 9. Esto también proporciona una explicación de por qué la versión mejorada del lenguaje C se llama C++, es simplemente el C mejorado o incrementado. Y ya que estamos, el lenguaje C se llama así porque las personas que lo desarrollaron crearon dos prototipos de lenguajes de programación con anterioridad a los que llamaron A y B.

Operadores de asignación

Existen varios operadores de asignación, el más evidente y el más usado es el "=", pero no es el único.

Aquí hay una lista: "=", "*=", "/=", "%=", "+=", "-=", "<<=", ">>=", "&=", " ^=" y "|=". Y la sintaxis es:

<variable> <operador de asignación> <expresión>

En general, para todos los operadores mixtos la expresión

E1 op= E2

Tiene el mismo efecto que la expresión

E1 = E1 op E2

El funcionamiento es siempre el mismo, primero se evalúa la expresión de la derecha, se aplica el operador mixto, si existe y se asigna el valor obtenido a la variable de la izquierda.

Operador coma

La coma tiene una doble función, por una parte separa elementos de una lista de argumentos de una función. Por otra, puede ser usado como separador en expresiones "de coma". Ambas funciones pueden ser mezcladas, pero hay que añadir paréntesis para resolver las ambigüedades. Sintaxis:

E1, E2, ..., En

En una expresión "de coma", cada operando es evaluado como una expresión, pero los resultados obtenidos anteriormente se tienen en cuenta en las subsiguientes evaluaciones. Por ejemplo:

func(i, (j = 1, j + 4), k);

Llamará a la función con tres argumentos: (i, 5, k). La expresión de coma (j = 1, j+4), se evalúa de izquierda a derecha, y el resultado se pasará como argumento a la función.

Operadores de igualdad

Los operadores de igualdad son "==", (dos signos = seguidos) y "!=", que comprueban la igualdad o desigualdad entre dos valores aritméticos. Sintaxis:

<expresión1> == <expresión2>

<expresión1> != <expresión2>

Se trata de operadores de expresiones lógicas, es decir, el resultado es "true" o "false". En el primer caso, si las expresiones 1 y 2 son iguales el resultado es "true", en el segundo, si las expresiones son diferentes, el "true".

Nota: Cuando se hacen comparaciones entre una constante y una variable, es recomendable poner en primer lugar la constante, por ejemplo:

```
if(123 == a) ...
if(a == 123) ...
```

Si nos equivocamos al escribir estas expresiones, y ponemos sólo un signo '=', en el primer caso obtendremos un error del compilador, ya que estaremos intentando cambiar el valor de una constante, lo cual no es posible. En el segundo caso, el valor de la variable cambia, y además el resultado de evaluar la expresión no dependerá de una comparación, sino de una asignación, y siempre será "true", salvo que el valor asignado sea 0.

Por ejemplo:

```
if(a = 0) ... // siempre será "false"
if(a = 123)... // siempre será "true", ya que 123 es distinto de 0
```

El resultado de evaluar la expresión no depende de "a" en ninguno de los dos casos, como puedes ver.

En estos casos, el compilador, en el mejor de los casos, nos dará un "warning", o sea un aviso, pero compilará el programa.

Nota: los compiladores clasifican los errores en dos tipos, dependiendo de lo serios que sean:

"Errores": son errores que impiden que el programa pueda ejecutarse, los programas con "errores" no pueden pasar de la fase de compilación a la de enlazado, que es la fase en que se obtiene el programa ejecutable.

"Warnings": son errores de poca entidad, (según el compilador que, por supuesto, no tiene ni idea de lo que intentamos hacer). Estos errores no impiden pasar a la fase de enlazado, y por lo tanto es posible ejecutarlos. Debes tener ciudado si tu compilador de da una lista de "warnings", eso significa que has cometido algún error, en cualquier caso repasa esta lista e intenta corregir los "warnings".

A su vez, los "enlazadores" o "linkers" también pueden dar errores, y normalmente estos son más difíciles de corregir.

Operadores lógicos

Los operadores "&&", "||" y "!" relacionan expresiones lógicas, formando a su vez nuevas expresiones lógicas. Sintaxis:

<expresión1> && <expresión2>

<expresión1> || <expresión2>

!<expresión>

El operador "&&" equivale al "AND" o "Y"; devuelve "true" sólo si las dos expresiones evaluadas son "true" o distintas de cero, en caso contrario devuelve "false" o cero. Si la primera expresión evaluada es "false", la segunda no se evalúa.

Generalizando, con expresiones AND con más de dos expresiones, la primera expresión falsa interrumpe el proceso e impide que se continue la evaluación del resto de las expresiones. Esto es lo que se conoce como "cortocircuito", y es muy importante, como veremos posteriormente.

A continuación se muestra la tabla de verdad del operador &&:

Expresión1	Expresión2	Expresión1 && Expresión2
false	ignorada	false
true	false	false

true	true	true

El operador "||" equivale al "OR" u "O inclusivo"; devuelve "true" si cualquiera de las expresiones evaluadas es "true" o distinta de cero, en caso contrario devuelve "false" o cero. Si la primera expresión evaluada es "true", la segunda no se evalúa.

Expresión1	Expresión2	Expresión1 || Expresión2
false	false	false
false	true	true
true	ignorada	true

El operador "!" es equivalente al "NOT", o "NO", y devuelve "true" sólo si la expresión evaluada es "false" o cero, en caso contrario devuelve "true". La expresión "!E" es equivalente a (0 == E).

Expresión	!Expresión
false	true
true	false

Operadores relacionales

Los operadores son "<", ">", "<=" y ">=", que comprueban relaciones de igualdad o desigualdad entre dos valores aritméticos. Sintaxis:

<expresión1> < <expresión2>

<expresión1 > < <expresión2>

<expresión1 <= <expresión2>

<expresión1> >= <expresión2>

Si el resultado de la comparación resulta ser verdadero, se retorna "true", en caso contrario "false". El significado de cada operador es evidente:

> mayor que

< menor que

>= mayor o igual que

<= menor o igual que

En la expresión "E1 <operador> E2", los operandos tienen algunas restricciones, pero de momento nos conformaremos con que E1 y E2 sean de tipo aritmético. El resto de las restricciones las veremos cuando conozcamos los punteros y los objetos.

Nota:

Cuando decimos que operador es binario no quiere decir que sólo se pueda usar con dos operandos, sino que afecta a dos operandos. Por ejemplo, la línea:

```
A = 1 + 2 + 3 - 4;
```

Es perfectamente legal, pero la operación se evaluará tomando los operandos dos a dos y empezando por la izquierda, y el resultado será 2. Además hay que mencionar el hecho de que los operadores tienen diferentes pesos, es decir, se aplican unos antes que otros, al igual que hacemos nosotros, por ejemplo:

```
A = 4 + 4 / 4;
```

Dará como resultado 5 y no 2, ya que la operación de división tiene prioridad sobre la suma. Esta propiedad de los operadores es conocida como precedencia. En el capítulo de operadores II se verán las precedencias de cada operador, y cómo se aplican y se eluden estas precedencias.

CAPITULO 5 Sentencias

Espero que hayas tenido la paciencia suficiente para llegar hasta aquí, y que no te hayas asustado mucho, ahora empezaremos a entrar en la parte interesante y estaremos en condiciones de añadir algún ejemplo.

El elemento que nos falta para empezar a escribir programas que funcionen son las sentencias.

Existen sentencias de varios tipos, que nos permitirán enfrentarnos a todas las situaciones posibles en programación. Estos tipos son:

- Bloques
- Expresiones
  - Llamada a función
  - Asignación
  - Nula
- Bucles
  - while
  - do while
  - for
- Etiquetas
  - Etiquetas de identificación
  - case
  - default
- Saltos
  - break
  - continue
  - goto
  - return
- Selección
  - if...else
  - switch

Bloques

Una sentencia compuesta o un bloque es un conjunto de sentencias, que puede estar vacía, encerrada entre llaves "{}". A nivel sintáctico, un bloque se considera como una única sentencia. También se usa en variables compuestas, como veremos en el capítulo de variables II, y en la definición de cuerpo de funciones. Los bloques pueden estar anidados hasta cualquier profundidad.

Expresiones

Una expresión seguida de un punto y coma (;), forma una sentencia de expresión. La forma en que el compilador ejecuta una sentencia de este tipo evaluando la expresión. Cualquier efecto derivado de esta evaluación se completará antes de ejecutar la siguiente sentencia.

<expresión>;

Llamadas a función

Esta es la manera de ejecutar las funciones que se definen en otras partes del programa o en el exterior de éste, ya sea una librería estándar o particular. Consiste en el nombre de la función, una lista de parámetros entre paréntesis y un ";".

Por ejemplo, para ejecutar la función que declarábamos en el capítulo 3 usaríamos una sentencia como ésta:

```
Mayor(124, 1234);
```

Compliquemos un poco la situación para ilustrar la diferencia entre una sentencia de expresión y una expresión, reflexionemos un poco sobre el siguiente ejemplo:

```
Mayor(124, Mayor(12, 1234));
```

Aquí se llama dos veces a la función "Mayor", la primera vez como una sentencia, la segunda como una expresión, que nos proporciona el segundo parámetro de la sentencia. En realidad, el compilador evalúa primero la expresión para obtener el segundo parámetro de la función, y después llama a la función. ¿Complicado?. Puede ser, pero también puede resultar interesante...

En el futuro diremos mucho más sobre este tipo de sentencias, pero por el momento es suficiente.

Asignación

Las sentencias de asignación responden al siguiente esquema:

<variable> <operador de asignación> <expresión>;

La expresión de la derecha es evaluada y el valor obtenido es asignado a la variable de la izquierda. El tipo de asignación dependerá del operador utilizado, estos operadores ya los vimos en el capítulo anterior.

La expresión puede ser, por supuesto, una llamada a función. De este modo podemos escribir un ejemplo con la función "Mayor" que tendrá más sentido:

```
m = Mayor(124, 1234);
```

Nula

La sentencia nula consiste en un único ";". Sirve para usarla en los casos en los que el compilador espera que aparezca una sentencia, pero en realidad no pretendemos hacer nada. Veremos ejemplo de esto cuando lleguemos a los bucles.

Bucles

Estos tipos de sentencias son el núcleo de cualquier lenguaje de programación, y están presentes en la mayor parte de ellos. Nos permiten realizar tareas repetitivas, y se usan en la resolución de la mayor parte de los problemas.

Bucles "while"

Es la sentencia de bucle más sencilla, y sin embargo es tremendamente potente. La sintaxis es la siguiente:

while (<condición>) <sentencia>

La sentencia es ejecutada repetidamente *mientras* la condición sea verdadera, ("while" en inglés significa "mientras"). Si no se especifica condición se asume que es "true", y el bucle se ejecutará indefinidamente. Si la primera vez que se evalúa la condición resulta falsa, la sentencia no se ejecutará ninguna vez.

Por ejemplo:

```
while (x < 100) x = x + 1;
```

Se incrementará el valor de x mientras x sea menor que 100.

Este ejemplo puede escribirse, usando el C con propiedad y elegancia, de un modo más compacto:

```
while (x++ < 100);
```

Aquí vemos el uso de una sentencia nula, observa que el bucle simplemente se repite, y la sentencia ejecutada es ";", es decir, nada.

Bucle "do while"

Esta sentencia va un paso más allá que el "while". La sintaxis es la siguiente:

do <sentencia> while(<condicion>);

La sentencia es ejecutada repetidamente mientras la condición resulte falsa. Si no se especifica condición se asume que es "true", y el bucle se ejecutará indefinidamente. A diferencia del bucle "while", la evaluación se realiza después de ejecutar la sentencia, de modo que se ejecutará al menos una vez. Por ejemplo:

```
do
   x = x + 1;
while (x < 100);
```

Se incrementará el valor de x hasta que x valga 100.

Bucle "for"

Por último el bucle "for", es el más elaborado. La sintaxis es:

for ([<inicialización>]; [<condición>] ; [<incremento>]) <sentencia>

La sentencia es ejecutada repetidamente hasta que la evaluación de la condición resulte falsa.

Antes de la primera iteración se ejecutará la inicialización del bucle, que puede ser una expresión o una declaración. En este apartado se iniciarán las variables usadas en el bucle. Estas variables pueden ser declaradas en este punto, pero tendrán validez sólo dentro del bucle "for". Después de cada iteración se ejecutará el incremento de las variables del bucle.

Todas las expresiones son opcionales, si no se especifica la condición se asume que es verdadera. Ejemplos:

```
for(int i = 0; i < 100; i = i + 1);
```

Como las expresiones son opcionales, podemos simular bucles "while":

```
for(;i < 100;) i = i +1;
for(;i++ < 100;);
```

O bucles infinitos:

```
for(;;);
```

Etiquetas

Los programas C y C++ se ejecutan secuencialmente, aunque esta secuencia puede ser interrumpida de varias maneras. Las etiquetas son la forma en que se indica al compilador en qué puntos será reanudada la ejecución de un programa cuando haya una ruptura del orden secuencial.

Etiquetas de identificación

Corresponden con la siguiente sintaxis:

<identificador>: <sentencia>

Sirven como puntos de entrada para la sentencia de salto "goto". Estas etiquetas tienen el ámbito restringido a la función dentro de la cual están definidas. Veremos su uso con más detalle al analizar la sentencia "goto".

Etiquetas "case" y "default"

Esta etiqueta se circunscribe al ámbito de la sentencia "switch", y se verá su uso cuando estudiemos ese apartado. Sintaxis:

```
switch(<variable>)
```

```
{
   <expresión constante>: [<sentencias>[...<sentencia>]][break;]
   . . .
   default: [<sentencias>[<sentencia>...]]
}
```

Selección

Las sentencias de selección permiten controlar el flujo del programa, seleccionando distintas sentencias en función de diferentes valores.

Sentencia "if...else"

Implementa la ejecución condicional de una sentencia. Sintaxis:

if (<condición>) <sentencia1>

if (<condición>) <sentencia1> else <sentencia2>

Se pueden declarar variables dentro de la condición. Por ejemplo:

```
if(int val = func(arg))...
```

En este caso, la variable "val" sólo estará accesible dentro del ámbito de la sentencia "if" y, si existe, del "else".

Si la condición es "true" se ejecutará la sentencia1, si es "false" se ejecutará la sentencia2.

El "else" es opcional, y no pueden insertarse sentencias entre la sentencia1 y el "else".

Sentencia "switch"

Cuando se usa la sentencia "switch" el control se transfiere al punto etiquetado con el "case" cuya expresión constante coincida con el valor de la variable del "switch", no te preocupes, con un ejemplo estará más claro. A partir de ese punto todas las sentencias serán ejecutadas hasta el final del "switch", es decir hasta llegar al "}". Esto es así porque las etiquetas sólo marcan los puntos de entrada después de una ruptura de la secuencia de ejecución, pero no marcan las salidas.

Esta característica nos permite ejecutar las mismas sentencias para varias etiquetas distintas, y se puede eludir usando la sentencia de ruptura "break" al final de las sentencias incluidas en cada "case".

Si no se satisface ningún "case", el control parará a la siguiente sentencia después de la etiqueta "default". Esta etiqueta es opcional y si no aparece se abandonará el "switch".

Sintaxis:

```
switch (<variable>)
{
```

```
   case <expresión constante>: <sentencia>[... <sentencia>] [break;]
   . . .
   [default : <sentencia>]
}
```

Por ejemplo:

```
switch(letra)
{
   case 'a':
   case 'e':
   case 'i':
   case 'o':
   case 'u':
      EsVocal = true;
      break;
   default:
      EsVocal = false;
}
```

En este ejemplo letra es una variable de tipo "char" y EsVocal de tipo "bool". Si el valor de entrada en el "switch" corresponde a una vocal, EsVocal saldrá con un valor verdadero, en caso contrario, saldrá con un valor falso. El ejemplo ilustra el uso del "break", si letra es 'a', se cumple el primer "case", y la ejecución continúa en la siguiente sentencia, ignorando el resto de los "case" hasta el "break".

Otro ejemplo:

```
Menor1 = Menor2 = Menor3 = Mayor3 = false;
switch(numero)
{
   case 0:
      Menor1 = true;
   case 1:
      Menor2 = true;
   case 2:
      Menor3 = true;
      break;
   default:
      Mayor3 = true;
}
```

Veamos qué pasa en este ejemplo si número vale 1. Directamente se reanuda la ejecución en "case 1:", con lo cual Menor2 tomará el valor "true", lo mismo pasará con Menor3. Después aparece el "break" y se abandona el "switch".

Sentencias de salto

Este tipo de sentencia permite romper la ejecución secuencial de un programa.

Sentencia de ruptura "break"

El uso de esta sentencia dentro de un bucle pasa el control a la primera sentencia después de la sentencia de bucle. Lo mismo se aplica a la sentencia "switch". Sintaxis:

break

Ejemplo:

```
y = 0;
x = 0;
while(x < 1000)
{
   if(y == 1000) break;
   y++;
}
x = 1;
```

En este ejemplo el bucle no terminaría nunca si no fuera por la línea del "break", ya que x no cambia. Después del "break" el programa continuaría en la línea "x = 1".

Sentencia de "continue"

El uso de esta sentencia dentro de un bucle pasa el control al final de la sentencia de bucle, justo al punto donde se evalúa la condición para la permanencia en el bucle. Sintaxis:

continue;

Ejemplo:

```
y = 0;
x = 0;
while(x < 1000)
{
   x++;
   if(y >= 100) continue;
   y++;
}
```

En este ejemplo la línea "y++" sólo se ejecutaría mientras "y" sea menor que 100, en cualquier otro caso el control pasa a la línea "}", con lo que el bucle volvería a evaluarse.

Sentencia de salto "goto"

Con el uso de esta sentencia el control se transfiere directamente al punto etiquetado con el identificador especificado. El "goto" es un mecanismo que está en guerra permanente, y sin cuartel, con la programación estructurada. El "goto" **no se usa**, se incluye aquí porque existe, pero siempre puede y debe ser eludido. Existen mecanismos suficientes para hacer todo aquello que pueda realizarse con un "goto". Sintaxis:

goto <identificador>;

Ejemplo:

```
x = 0;
Bucle:
```

```
x++;
if(x < 1000) goto Bucle;
```

Este ejemplo emula el funcionamiento de un bucle "for" como el siguiente:

```
for(x = 0; x < 1000; x++);
```

Sentencia de retorno "return"

Esta sentencia sale de la función donde se encuentra y devuelve el control a la rutina que la llamó, opcionalmente con un valor de retorno. Sintaxis:

return [<expresión>] ;

Ejemplo:

```
int Paridad(int x)
{
   if(x % 2) return 1;
   return 0;
}
```

Este ejemplo ilustra la implementación de una función que calcula la paridad de un parámetro. Si el resto de dividir el parámetro entre 2 es distinto de cero, implica que el parámetro es impar, y la función retorna con valor 1. El resto de la función no se ejecuta. Si por el contrario el resto de dividir el parámetro entre 2 es cero, el parámetro será un número par y la función retornará con valor cero.

Sobre las sentencias de salto y la programación estructurada

Lo dicho para la sentencia "goto" es válido en general para todas las sentencias de salto, salvo el "return" y el "break", este último tiene un poco más de tolerancia, sobre todo en las sentencias "switch", donde resulta imprescindible. En general, es una buena norma huir de las sentencias de salto.

Comentarios

No se trata propiamente de un tipo de sentencias, pero me parece que es el lugar adecuado para introducir este concepto. En C pueden introducirse comentarios en cualquier parte del programa, estos comentarios ayudarán a seguir el funcionamiento del programa durante la depuración o en la actualización del programa, además de documentarlo. Los comentarios en C se delimitan entre /* y */, cualquier cosa que escribamos en su interior será ignorada por el compilador, sólo está prohibido su uso en el interior de palabras reservadas o en el interior de identificadores. Por ejemplo:

```
main(/*Sin argumentos*/void)
```

está permitido, sin embargo:

```
ma/*función*/in(void)
```

es ilegal, se trata de aclarar y documentar, no de entorpecer el código.

En C++ se ha incluido otro tipo de comentarios, que empiezan con //. Estos comentarios no tienen marca de final, sino que terminan cuando termina la línea. Por ejemplo:

```
void main(void) // Esto es un comentario
{
}
```

Las llaves {} no forman parte del comentario.

Palabras reservadas usadas en este capítulo

":", ";", while, do, for, case, default, break, continue, goto, return, if, else y switch.

CAPITULO 6 Declaración de variables

Una característica del C es la necesidad de la declaración de las variables que se usarán en el programa. Aunque esto resulta chocante para los que se aproximan al C desde otros lenguajes de programación, es en realidad una característica muy importante y útil de C, ya que ayuda a conseguir códigos más compactos y eficaces, y contribuye a facilitar la depuración y la detección y corrección de errores.

Cómo se declaran las variables

En realidad ya hemos visto la mecánica de la declaración de variables, al mostrar la sintaxis de cada tipo en el capítulo 2.

El sistema es siempre el mismo, primero se especifica el tipo y a continuación una lista de variables.

En realidad, la declaración de variables puede considerarse como una sentencia. Desde este punto de vista, la declaración terminará con un ";". Sintaxis:

<tipo> <lista de variables>;

Existe una particularidad relacionada con el punto en que se declaren las variables. Veremos este tema más en detalle cuando estudiemos el capítulo del ámbito, pero adelantaremos algo aquí.

Las variables declaradas dentro de una función sólo serán accesibles desde el interior de esa función, y se conocen como variables locales de la función. Las variables declaradas fuera de las funciones, serán accesibles desde todas las funciones, y son conocidas como variables globales.

También es posible inicializar las variables dentro de la misma declaración. Por ejemplo:

```
int a = 1234;
bool seguir = true, encontrado;
```

Declararía las variables "a", "seguir" y "encontrado"; y además iniciaría los valores de "a" y "seguir" a 1234 y "true", respectivamente.

En C, contrariamente a lo que sucede con otros lenguajes de programación, las variables no inicializadas tienen un valor indeterminado, contienen lo que normalmente se denomina "basura", también en esto hay excepciones como veremos más adelante.

Ámbito de las variables:

Dependiendo de dónde se declaren las variables, podrán o no ser accesibles desde distintas partes del programa.

Las variables declaradas dentro de una función, y recuerda que "main" también es una función, sólo serán accesibles desde esa función. Diremos que esas variables son variables locales o de ámbito local de esa función.

Las variables declaradas fuera de las funciones, normalmente antes de definir las funciones, en la zona donde se declaran los prototipos, serán accesibles desde todas las funciones. Diremos que esas variables serán globales o de ámbito global.

Ejemplo:

```
#include <iostream.h>

int EnteroGlobal; // Declaración de una variable global

int Funcion1(int a); // Declaración de un prototipo

int main() {
   int EnteroLocal;  // Declaración de una variable local de main

   EnteroLocal = Funcion1(10); // Acceso a una variable local
   EnteroGlobal = Funcion1(EnteroLocal); // Acceso a una valiable
global

   return 0;
}

int Funcion1(int a)
{
   char CaracterLocal; // Variable local de funcion1
   // Desde aquí podemos acceder a EnteroGlobal,
   // y también a CaracterLocal
   // pero no a EnteroLocal
   if(EnteroGlobal != 0)
      return a/EnteroGlobal;
   else
      return 0;
}
```

Ejemplos capítulos 1 a 6

Veamos algunos ejemplos que utilicen los que ya sabemos de C++.

Pero antes introduciremos, sin explicarlo en profundidad, un elemento que nos permitirá que nuestros programas se comuniquen con nosotros.. Se trata de la salida estándar, "cout". Este elemento nos permite enviar a la pantalla cualquier variable o constante, incluidos literales. Lo veremos más detalladamente en un capítulo dedicado a él, de momento sólo nos interesa cómo usarlo para mostrar cadenas de caracteres y variables.

Nota: en realidad "cout" es un objeto de la clase "ostream_withassign", pero los conceptos de clase y objeto quedarán mucho más claros en capítulos posteriores.

El uso es muy simple:

#include <iostream.h>

cout << <variable|constante> [<< <variable|constante>...];

Por ejemplo:

```
#include <iostream.h>

int a;
a = 32;
cout << "la variable a vale " << a;
```

Una constante muy útil es "endl", que hará que la siguiente salida se imprima en una nueva línea.

```
cout << "hola" << endl;
```

La línea "#include <iostream.h>" es necesaria porque las funciones que permiten el acceso a "cout" están en una librería externa. Con este elemento ya podemos incluir algunos ejemplos. Te aconsejo que intentes resolver los ejemplos antes de ver la solución, o al menos piensa unos minutos sobre ellos.

Ejemplo 1

Primero haremos uno fácil. Escribir un programa que muestre una lista de números del 1 al 20, indicando a la derecha de cada uno si es divisible por 3 o no.

```
// Este programa muestra una lista de números, indicando para cada uno
// si es o no múltipo de 3.
// 11/09/2000 Salvador Pozo

#include <iostream.h> // librería para uso de cout
```

```
int main() // función principal
{
   int i; // variable para bucle

   for(i = 1; i <= 20; i++) // bucle for de 1 a 20
   {
      cout << i; // muestra el número
      if(i % 3 == 0) cout << " es múltiplo de 3"; // el resto de
dividir por 3 es 0
      else cout << " no es múltiplo de 3"; // o no lo es
      cout << endl; // cambio de línea
   }
}
```

El enunciado es el típico de un problema que puede ser solucionado con un bucle "for". Observa el uso de los comentarios, y acostúmbrate a incluirlos en todos tus programas. Acostúmbrate también a escribir el código al mismo tiempo que los comentarios. Si lo dejas para cuando has terminado el código, probablemente sea demasiado tarde, y la mayoría de las veces no lo harás. ;-)

También es una buena costumbre incluir al principio del programa un comentario extenso que incluya el enunciado del problema, añadiendo también el nombre del autor y la fecha en que se escribió. Además, cuando hagas revisiones, actualizaciones o correcciones deberías incluir una explicación de cada una de ellas y la fecha en que se hicieron.

Una buena documentación te ahorrará mucho tiempo y te evitará muchos dolores de cabeza.

Ejemplo 2

Escribir el programa anterior, pero usando una función para verificar si el número es divisible por tres, y un bucle de tipo "while".

```
// Este programa muestra una lista de números, indicando para cada uno
// si es o no múltipo de 3.
// 11/09/2000 Salvador Pozo

#include <iostream.h> // librería para uso de cout

// Prototipos:
bool MultiploDeTres(int n);

int main() // función principal
{
   int i = 1; // variable para bucle

   while(i <= 20) // bucle hasta i igual a 20
   {
      cout << i; // muestra el número
      if(MultiploDeTres(i)) cout << " es múltiplo de 3"; // el numero
es múltiplo de 3
      else cout << " no es múltiplo de 3"; // o no lo es
      cout << endl; // cambio de línea
      i++;
```

```
   }
}

// Función que devuelve verdadero si el parámetro 'n' en
// múltiplo de tres y falso si no lo es
bool MultiploDeTres(int n)
{
   if(n % 3) return false; else return true;
}
```

Comprueba cómo hemos declarado el prototipo de la función "MultiploDeTres". Además, al declarar la variable i le hemos dado un valor inicial 1. Observa que al incluir la función, con el nombre adecuado, el código queda mucho más legible, de hecho prácticamente sobra el comentario. Por último, fíjate en que la definición de la función va precedida de un comentario que explica lo que hace. Esto también es muy recomendable.

Ejemplo 3

Escribir un programa que muestre una salida de 20 líneas de este tipo:

```
1
1 2
1 2 3
1 2 3 4
...
```

```
// Este programa muestra una lista de números
// de este tipo:
// 1
// 1 2
// 1 2 3
// ...

// 11/09/2000 Salvador Pozo

#include <iostream.h> // librería para uso de cout

int main() // función principal
{
   int i, j; // variables para bucles

   for(i = 1; i <= 20; i++) // bucle hasta i igual a 20
   {
      for(j = 1; j <= i; j++) // bucle desde 1 a i
         cout << i << " "; // muestra el número
      cout << endl; // cambio de línea
   }
}
```

Este ejemplo ilustra el uso de bucles anidados. El bucle interior, que usa "j" como variable toma valores entre 1 e "i". El bucle exterior incluye, además del bucle interior, la orden de cambio de línea, de no ser así, la salida no tendría la forma deseada. Además, después de cada número se imprime un espacio en blanco, de otro modo los números aparecerían amontonados.

Ejemplo 4

Escribir un programa que muestre una salida con la siguiente secuencia numérica:

1, 5, 3, 7, 5, 9, 7, ..., 23

La secuencia debe detenerse al llegar al 23.

El enunciado es rebuscado, pero ilustra el uso de los bucles "do...while".

La secuencia se obtiene partiendo de 1 y sumando y restando 4 y 2, alternativamente. Veamos cómo resolverlo:

```
#include <iostream.h>; // librería para uso de cout

int main() // función principal
{
   int i = 1; // variable para bucles
   bool sumar = true; // Siguiente operación es sumar o restar
   bool terminado = false; // Se ha alcanzado la condición de fin

   do { // Hacer
      cout << i; // muestra el valor en pantalla
      terminado = (i == 23); // Actualiza condición de fin
      if(terminado) cout << "."; else cout << ", "; // Puntuación,
separadores
      if(sumar) i += 4; else i -= 2; // Calcula siguiente elemento
      sumar = !sumar; // Cambia la siguiente operación
   } while(!terminado); // ... mientras no se termine
   cout << endl; // Cambio de línea
```

CAPITULO 7 Normas para la notación

Que no te asuste el título. Lo que aquí trataremos es más simple de lo que parece. Veremos las reglas que rigen cómo se escriben las constantes en C según diversos sistemas de numeración y que uso tiene cada uno.

Constantes "int"

En C se usan dos tipos de numeración para la definición de constantes numéricas, la decimal y la hexadecimal, según se use la numeración en base 10 ó16, respectivamente.

Por ejemplo, el número 127, se representará en notación decimal como 127 y en hexadecimal como 0x7f. En notación hexadecimal, se usan 16 símbolos, los dígitos del '0' al '9' tienen el mismo valor que en decimal, para los otros seis símbolos se usan las letras de la 'A' a la 'F', indistintamente en mayúsculas o minúsculas. Sus valores son 10 para la 'A', 11 para la 'B', y sucesivamente, hasta 15 para la 'F'. Según el ejemplo el número 0x7f, donde "0x" es el prefijo que indica que se trata de un número en notación hexadecimal, sería el número 7F, es decir, 7*16+15=127. Del mismo modo que el número 127 en notación decimal sería, $1*10^2+2*10+7=127$.

La ventaja de la numeración hexadecimal es que los valores enteros requieren dos dígitos por cada byte para su representación. Así un byte se puede tomar valores hexadecimales entre 0x00 y 0xff, dos bytes entre 0x0000 y 0xffff, etc. Además, la conversión a binario es casi directa, cada dígito hexadecimal se puede sustituir por cuatro bits, el '0x0' por '0000', el '0x1' por '0001', hasta el '0xf', que equivale a '1111'. En el ejemplo el número 127, o 0x7f, sería en binario '01111111'.

De este modo, cuando trabajemos con operaciones de bits, nos resultará mucho más sencillo escribir valores constantes usando la notación hexadecimal. Por ejemplo, resulta más fácil predecir el resultado de la siguiente operación:

```
A = 0xaa & 0x55;
```

Que:

```
A = 170 & 85;
```

En ambos casos el resultado es 0, pero en el primero resulta más evidente, ya que 0xAA es en binario 10101010 y 0x55 es 01010101, y la operación "AND" entre ambos números es 00000000, es decir 0. Ahora inténtalo con los números 170 y 85, anda, ¡inténtalo!.

Constantes "char"

Las constantes de tipo "char" se representan entre comillas sencillas, por ejemplo 'a', '8', 'F'.

Después de pensar un rato sobre el tema, tal vez te preguntes ¿cómo se representa la constante que consiste en una comilla sencilla?. Bien, te lo voy a contar, aunque no lo hayas pensado.

Existen ciertos caracteres, entre los que se encuentra la comilla sencilla, que no pueden ser representados con la norma general. Para eludir este problema existe un cierto mecanismo, llamado secuencias de escape. En el caso comentado, la comilla sencilla se define como '\'', y antes de que preguntes te diré que la barra descendente se define como '\\'.

Además de estos caracteres especiales existen otros. El código ASCII, que es el que puede ser representado por el tipo "char", consta de 128 o 256 caracteres. Y aunque el código ASCII de 128 caracteres, 7 bits, ha quedado prácticamente obsoleto, ya que no admite caracteres como la 'ñ' o la 'á'; aún se usa en ciertos equipos antiguos, en los que el octavo bit se usa como bit de paridad en las transmisiones serie. De todos modos, desde hace bastante tiempo, se ha adoptado el código ASCII de 256 caracteres, 8 bits. Recordemos que el tipo "char" tiene siempre un byte, es decir 8 bits, y esto no es por casualidad.

En este conjunto existen, además de los caracteres alfabéticos, en mayúsculas y minúsculas, los numéricos, los signos de puntuación y los caracteres internacionales, ciertos caracteres no imprimibles, como el retorno de línea, el avance de línea, etc.

Veremos estos caracteres y cómo se representan como secuencia de escape, en hexadecimal, el nombre ANSI y el resultado o significado.

Escape	Hexad	ANSI	Nombre o resultado
	0x00	NULL	Carácter nulo
\a	0x07	BELL	Sonido de campanilla
\b	0x08	BS	Retroceso
\f	0x0C	FF	Avance de página
\n	0x0A	LF	Avance de línea
\r	0x0D	CR	Retorno de línea
\t	0x09	HT	Tabulador horizontal
\v	0x0B	VT	Tabulador vertical
\\	0x5c	\	Barra descendente
\'	0x27	'	Comilla sencilla
\"	0x22	"	Comillas
\?	0x3F	?	Interrogación
\O		cualquiera	O=tres dígitos en octal
\xH		cualquiera	H=número hexadecimal
\XH		cualquiera	H=número hexadecimal

Los tres últimos son realmente comodines para la representación de cualquier carácter. El \nnn sirve para la representación en notación octal. Para la notación octal se usan tres

dígitos. Hay que tener en cuenta que, análogamente a lo que sucede en la notación hexadecimal, en la octal se agrupan los bits de tres en tres. Por lo tanto, para representar un carácter ASCII de 8 bits, se necesitarán tres dígitos. En octal sólo son válidos los símbolos del '0' al '7'. Según el ejemplo anterior, para representar el carácter 127 en octal usaremos la cadena '\177', y en hexadecimal '\x7f'. También pueden asignarse números decimales a variables de tipo char. Por ejemplo:

```
char A;
A = 'a';
A = 97;
A = 0x61;
A = '\x61';
A = '\141';
```

En este ejemplo todas las asignaciones son equivalentes y válidas.

Una nota sobre el carácter nulo. Este carácter se usa en C para terminar las cadenas de caracteres, por lo tanto es muy útil y de frecuente uso. Para hacer referencia a él se usa frecuentemente su valor decimal, es decir char A = 0, aunque es muy probable que lo encuentres en libros o en programas como '\000', es decir en notación octal.

Sobre el carácter EOF, del inglés "End Of File", este carácter se usa en muchos ficheros como marcador de fin de fichero, sobre todo en ficheros de texto. Aunque dependiendo del sistema operativo este carácter puede cambiar, por ejemplo en MS-DOS es el carácter "0x1A", el compilador siempre lo traduce y devuelve el carácter EOF cuando un fichero se termina. El valor usado por el compilador está definido en el fichero "stdio.h", y es 0.

CAPITULO 8 Cadenas de caracteres

Antes de entrar en el tema de los "arrays" también conocidos como arreglos, tablas o matrices, veremos un caso especial de ellos. Se trata de las cadenas de caracteres o "strings".

Una cadena en C es un conjunto de caracteres, o valores de tipo "char", terminados con el carácter nulo, es decir el valor numérico 0. Internamente se almacenan en posiciones consecutivas de memoria. Este tipo de estructuras recibe un tratamiento especial, y es de gran utilidad y de uso continuo.

La manera de definir una cadena es la siguiente:

char <identificador> **[**<longitud máxima> **]**;

En este caso los corchetes no indican un valor opcional, sino que son realmente corchetes, por eso están en negrita.

Cuando se declara una cadena hay que tener en cuenta que tendremos que reservar una posición para almacenar el carácter nulo, de modo que si queremos almacenar la cadena "HOLA", tendremos que declarar la cadena como:

```
char Saludo[5];
```

Cuatro caracteres para "HOLA" y uno extra para el carácter '\000'.

También nos será posible hacer referencia a cada uno de los caracteres individuales que componen la cadena, simplemente indicando la posición. Por ejemplo el tercer carácter de nuestra cadena de ejemplo será la 'L', podemos hacer referencia a él como Saludo[2]. Los índices tomarán valores empezando en el cero, así el primer carácter de nuestra cadena sería Saludo[0], que es la 'H'.

Una cadena puede almacenar informaciones como nombres de personas, mensajes de error, números de teléfono, etc.

La asignación directa sólo está permitida cuando se hace junto con la declaración. Por ejemplo:

```
char Saludo[5];
Saludo = "HOLA"
```

Aunque no produzca un error en el compilador, no está haciendo lo que nosotros probablemente pretendemos, como veremos en el capítulo de "arrays" o arreglos.

La manera correcta de asignar una cadena es:

```
char Saludo[5];
Saludo[0] = 'H';
Saludo[1] = 'O';
Saludo[2] = 'L';
```

```
Saludo[3] = 'A';
Saludo[4] = '/000';
```

O bien:

```
char Saludo[5] = "HOLA";
```

Si te parece un sistema engorroso, no te preocupes, en próximos capítulos veremos funciones que facilitarán la asignación de cadenas. Existen muchas funciones para el tratamiento de cadenas, como veremos más adelante, que permiten compararlas, copiarlas, calcular su longitud, imprimirlas, visualizarlas, guardarlas en disco, etc. Además, frecuentemente nos encontraremos a nosotros mismos creando nuevas funciones que básicamente hacen un tratamiento de cadenas.

CAPITULO 9 Librerías estándar

Todos los compiladores C y C++ disponen de ciertas librerías de funciones estándar que facilitan el acceso a la pantalla, al teclado, a los discos, la manipulación de cadenas, y muchas otras cosas, de uso corriente.

Hay que decir que todas estas funciones no son imprescindibles, y de hecho no forman parte del C. Pueden estar escritas en C, de hecho en su mayor parte lo están, y muchos compiladores incluyen el código fuente de estas librerías. Nos hacen la vida más fácil, y no tendría sentido pasarlas por alto.

Existen muchas de estas librerías, algunas tienen sus características definidas según diferentes estándar, como ANSI C o C++, otras son específicas del compilador, otras del sistema operativo, también las hay para plataformas Windows. En el presente curso nos limitaremos a las librerías ANSI C y C++.

Veremos ahora algunas de las funciones más útiles de algunas de estas librerías, las más imprescindibles.

Librería de entrada y salida fluidas "iostream.h"

En el contexto de C++ todo lo referente a "streams" puede visualizarse mejor si usamos un símil como un río o canal de agua.

Imagina un canal por el que circula agua, si echamos al canal objetos que floten, estos se moverán hasta el final de canal, siguiendo el flujo del agua. Esta es la idea que se quiere transmitir cuando se llama "stream" a algo en C++. Por ejemplo, en C++ el canal de salida es "cout", los objetos flotantes serán los argumentos que queremos extraer del ordenador o del programa, la salida del canal es la pantalla. Completando el símil, en la orden:

```
cout << "hola" << " " << endl;
```

Los operadores "<<" representarían el agua, y la dirección en que se mueve. Cualquier cosa que soltemos en el agua: "hola", " " o endl, seguirá flotando hasta llegar a la pantalla, y además mantendrán su orden.

En esta librería se definen algunas de las funciones aplicables a los "streams", pero aún no estamos en disposición de acceder a ellas. Baste decir de momento que existen cuatro "streams" predeterminados:

- cin, canal de entrada estándar.
- cout, canal de salida estándar.
- cerr, canal de salida de errores.
- clog, canal de salida de diario o anotaciones.

Sobre el uso de "cin", que es el único canal de entrada predefinido, tenemos que aclarar cómo se usa, aunque a lo mejor ya lo has adivinado.

cin >> <variable> [>> <variable>...];

Donde cada variable irá tomando el valor introducido mediante el teclado. Los espacios y los retornos de línea actúan como separadores.

Ejemplo:

Escribir un programa que lea el nombre, la edad y el número de teléfono de un usuario y los muestre en pantalla.

```
#include <iostream.h>

int main()
{
  char Nombre[30];  // Usaremos una cadena para almacenar el nombre
(29 caracteres)
  int Edad;         // Un entero para la edad
  char Telefono[8]; // Y otra cadena para el número de teléfono (7
dígitos)

  // Mensaje para el usuario
  cout << "Introduce tu nombre, edad y número de teléfono" << endl;
  // Lectura de las variables
  cin >> Nombre >> Edad >> Telefono;
  // Visualización de los datos leídos
  cout << "Nombre:" << Nombre << endl;
  cout << "Edad:" << Edad << endl;
  cout << "Teléfono:" << Telefono << endl;

  return 0;
}
```

Librería de entrada y salida estándar "stdio.h"

En esta librería están las funciones de entrada y salida, tanto de la pantalla y teclado como de ficheros. "stdio" puede y suele leerse como estándar Input/Output. De hecho la pantalla y el teclado son considerados como ficheros, aunque de un tipo algo peculiar. La pantalla es un fichero sólo de escritura llamado "stdout", o salida estándar y el teclado sólo de lectura llamado "stdin", o entrada estándar.

Se trata de una librería ANSI C, por lo que está heredada de C, y ha perdido la mayor parte de su utilidad al ser desplazada por "iostream". Pero aún puede ser muy útil, es usada por muchos programadores, y la encontrarás en la mayor parte de los programas C y C++.

Veamos ahora algunas funciones.

Función "getchar()"

Sintaxis:

int getchar(void);

Lee un carácter desde "stdin".

"getchar" es una macro que devuelve el siguiente carácter del canal de entrada "stdin". Esta macro está definida como "getc(stdin)".

Valor de retorno:

Si todo va bien, "getchar" devuelve el carácter leído, después de convertirlo a un "int" sin signo. Si lee un Fin-de-fichero o hay un error, devuelve EOF.

Ejemplo:

```
do {
   a = getchar();
} while (a != 'q');
```

En este ejemplo, el programa permanecerá leyendo el teclado mientras no pulsemos la tecla 'q'.

Función "putchar()"

Sintaxis:

int putchar(int c);

Envía un carácter a la salida "stdout".

"putchar(c)" es una macro definida como "putc(c, stdout)".

Valor de retorno:

Si tiene éxito, "putchar" devuelve el carácter c. Si hay un error, "putchar" devuelve EOF.

Ejemplo:

```
while(a = getchar()) putchar(a);
```

En este ejemplo, el programa permanecerá leyendo el teclado y escribirá cada tecla que pulsemos, mientras no pulsemos '^C', (CONTROL+C). Observa que la condición en el "while" no es "a == getchar()", sino una asignación. Aquí se aplican, como siempre, las normas en el orden de evaluación en expresiones, primero se llama a la función getchar(), el resultado se asigna a la variable "a", y finalmente se comprueba si el valor de "a" es o no distinto de cero. Si "a" es cero, el bucle termina, si no es así continúa.

Función "gets()"

Sintaxis:

char *gets(char *s);

Lee una cadena desde "stdin".

"gets" lee una cadena de caracteres terminada con un retorno de línea desde la entrada estándar y la almacena en s. El carácter de retorno de línea es reemplazado con el carácter nulo en s.

Observa que la manera en que hacemos referencia a una cadena dentro de la función es "char *", el operador * indica que debemos pasar como argumento la dirección de memoria donde está almacenada la cadena a leer. Veremos la explicación en el capítulo de punteros, baste decir que a nivel del compilador "char *cad" y "char cad[]", son equivalentes, o casi.

"gets" permite la entrada de caracteres que indican huecos, como los espacios y los tabuladores. "gets" deja de leer después de haber leído un carácter de retorno de línea; todo aquello leído será copiado en s, incluido en carácter de retorno de línea.

Esta función no comprueba la longitud de la cadena leída. Si la cadena de entrada no es lo suficientemente larga, los datos pueden ser sobrescritos o corrompidos. Más adelante veremos que la función "fgets" proporciona mayor control sobre las cadenas de entrada.

Valor de retorno:

Si tiene éxito, "gets" devuelve la cadena s, si se encuentra el fin_de_fichero o se produce un error, devuelve NULL. Ejemplo:

```
char cad[80];
do {
   gets(cad);
} while (cad[0] != '\000');
```

En este ejemplo, el programa permanecerá leyendo cadenas desde el teclado mientras no introduzcamos una cadena vacía. Para comprobar que una cadena está vacía basta con verificar que el primer carácter de la cadena es un carácter nulo.

Función "puts()"

Sintaxis:

int puts(const char *s);

Envía una cadena a "stdout".

"puts" envía la cadena s terminada con nulo a la salida estándar "stdout" y le añade el carácter de retorno de línea.

Valor de retorno:

Si tiene éxito, "puts" devuelve un valor mayor o igual a cero. En caso contrario devolverá el valor EOF.

Ejemplo:

```
char cad[80];
int i;
do {
   gets(cad);
   for(i = 0; cad[i]; i++)
      if(cad[i] == ' ') cad[i] = '_';
   puts(cad);
} while (cad[0] != '\000');
```

Empezamos a llegar a ejemplos más elaborados. Este ejemplo leerá cadenas en "cad", mientras no introduzcamos una cadena vacía. Cada cadena será recorrida carácter a carácter y los espacios de sustituirán por caracteres '_'. Finalmente se visualizará la cadena resultante.

Llamo tu atención ahora sobre la condición en el bucle "for", comparándola con la del bucle "do while". Efectivamente son equivalentes, al menos para i == 0, la condición del bucle "do while" podría haberse escrito simplemente como "while (cad[0])". De hecho, a partir de ahora intentaremos usar expresiones más simples.

Librería de entrada y salida estándar "stdio.h"

Función "printf()"

Sintaxis:

int printf(const char *formato[, argumento, ...]);

Escribe una cadena con formato a la salida estándar "stdout".

Esta es probablemente una de las funciones más complejas de C. No es necesario que la estudies en profundidad, limítate a leer este capítulo, y considéralo como una fuente para la consulta. Descubrirás que poco a poco la conoces y dominas.

Esta función acepta listas de parámetros con un número indefinido de elementos. Entendamos que el número está indefinido en la declaración de la función, pero no en su uso, el número de argumentos dependerá de la cadena de formato, y conociendo ésta, podrá precisarse el número de argumentos. Si no se respeta esta regla, el resultado es impredecible, y normalmente desastroso, sobre todo cuando faltan argumentos. En general, los sobrantes serán simplemente ignorados.

Cada argumento se aplicará a la cadena de formato y la cadena resultante se enviará a la pantalla.

Valor de retorno:

Si todo va bien el valor de retorno será el número de bytes de la cadena de salida. Si hay error se retornará con EOF.

Veremos ahora las cadenas de formato. Esta cadena controla cómo se tratará cada uno de los argumentos, realizando la conversión adecuada, dependiendo de su tipo. Cada cadena de formato tiene dos tipos de objetos:

- Caracteres simples, que serán copiados literalmente a la salida.
- Descriptores de conversión que tomarán los argumentos de la lista y les aplicarán el formato adecuado.

A su vez, los descriptores de formato tienen la siguiente estructura:

%[opciones][anchura][.precisión] [F|N|h|l|L] carácter_de_tipo

Cada descriptor de formato empieza siempre con el carácter '%', después de él siguen algunos componentes opcionales y finalmente un carácter que indica el tipo de conversión a realizar.

Componente	Opcional/Obligatorio	Tipo de control
[opciones]	Opcional	Tipo de justificación, signos de números, puntos decimales, ceros iniciales, prefijos octales y hexadecimales.
[anchura]	Opcional	Anchura, mínimo número de caracteres a imprimir, se completa con espacios o ceros.
[precisión]	Opcional	Precisión, máximo número de caracteres. Para enteros, mínimo número de caracteres a imprimir.

[F|N|h|l|L]	Opcional	Modificador de tamaño de la entrada. Ignora el tamaño por defecto para los parámetros de entrada.
		
		`F = punteros lejanos (far pointer)`
		`N = punteros cercanos (near pointer)`
		`h = short int`
		`l = long`
		`L = long double`

Carácter_de_tipo	Obligatorio	Carácter de conversión de tipos.

Opciones:

Pueden aparecer en cualquier orden y combinación:

Opción	Significado
-	Justifica el resultado a la izquierda, rellena a la derecha con espacios, si no se da se asume justificación a la derecha, se rellena con ceros o espacios a la izquierda.
+	El número se mostrará siempre con signo (+) o (-), según corresponda.
Espacio	Igual que el anterior, salvo que cuando el signo sea positivo se mostrará un espacio.
#	Especifica que el número se convertirá usando un formato alternativo

La opción (#) tiene diferentes efectos según el carácter de tipo especificado en el descriptor de formato, si es:

Carácter de tipo	Efecto
c s d i u	No tiene ningún efecto.
x X	Se añadirá 0x (o 0X) al principio del argumento.
e E f g G	En el resultado siempre mostrará el punto decimal, aunque ningún dígito le siga. Normalmente, el punto decimal sólo se muestra si le sigue algún dígito.

Anchura:

Define el número mínimo de caracteres que se usarán para mostrar el valor de salida.

Puede ser especificado directamente mediante un número decimal, o indirectamente mediante un asterisco (*). Si se usa un asterisco como descriptor de anchura, el siguiente argumento de la lista, que debe ser un entero, que especificará la anchura mínima de la salida. Aunque no se especifique o sea más pequeño de lo necesario, el resultado nunca se truncará, sino que se expandirá lo necesario para contener el valor de salida.

Descriptor	Efecto
n	Al menos n caracteres serán impresos, si la salida requiere menos de n caracteres se rellenará con blancos.
0n	Lo mismo, pero se rellenará a la izquierda con ceros.
*	El número se tomará de la lista de argumentos

Precisión:

Especifica el número máximo de caracteres o el mínimo de dígitos enteros a imprimir.

La especificación de precisión siempre empieza con un punto (.) para separarlo del descriptor de anchura.

Al igual que el descriptor de anchura, el de precisión admite la especificación directa, con un número; o indirecta, con un asterisco (*).

Si usas un asterisco como descriptor de anchura, el siguiente argumento de la lista, que debe ser un entero, especificará la anchura mínima de la salida. Si usas el asterisco para el descriptor de precisión, el de anchura, o para ambos, el orden será, descriptor de anchura, de precisión y el dato a convertir.

Descriptor	Efecto
(nada)	Precisión por defecto
.0 ó .	Para los tipos d, i, o, u, x, precisión por defecto.

	Para e, E, f, no se imprimirá el punto decimal, ni ningún decimal.
.n	Se imprimirán n caracteres o n decimales. Si el valor tiene más de n caracteres se truncará o se redondeará, según el caso.
.*	El descriptor de precisión se tomará de la lista de argumentos.

Valores de precisión por defecto:

1 para los tipos: d,i,o,u,x,X

6 para los tipos: e,E,f

Todos los dígitos significativos para los tipos; g, G

Hasta el primer nulo para el tipo s

Sin efecto en el tipo c

Si se dan las siguientes condiciones no se imprimirán caracteres para este campo:

- Se especifica explícitamente una precisión de 0.
- El campo es un entero (d, i, o, u, óx),
- El valor a imprimir es cero.

Cómo afecta [.precisión] a la conversión:

Carácter de tipo	Efecto de (.n)
d i o u x X	Al menos n dígitos serán impresos. Si el argumento de entrada tiene menos de n dígitos se rellenará a la izquierda, para x/X con ceros. Si el argumento de entrada tiene menos de n dígitos el valor no será truncado.
e E f	Al menos n caracteres se imprimirán después del punto decimal, el último de ellos será redondeado.
g G	Como máximo, n dígitos significativos serán impresos.
c	Ningún efecto.
s	No más de n caracteres serán impresos.

Modificador de tamaño [F|N|h|l|L]:

Indican cómo printf debe interpretar el siguiente argumento de entrada.

Modificador	Tipo de argumento	Interpretación
F	Puntero p s n	Un puntero *far*
N	Puntero p s n	Un puntero *near*
h	d i o u x X	short int
L	d i o u x X	long int

l	e E f g G	double
L	e E f g G	long double

Caracteres de conversión de tipo.

La información de la siguiente tabla asume que no se han especificado opciones, ni descriptores de ancho ni precisión, ni modificadores de tamaño.

Carácter de tipo	Entrada esperada	Formato de salida
Números		
d	Entero con signo	Entero decimal
i	Entero con signo	Entero decimal
o	Entero con signo	Entero octal
u	Entero sin signo	Entero decimal
x	Entero sin signo	Entero hexadecimal (con a, b, c, d, e, f)
X	Entero sin signo	Entero hexadecimal (con A, B, C, D, E, F)
f	Coma flotante	Valor con signo: [-]dddd.dddd
e	Coma flotante	Valor con signo: [-]d.dddd...e[+/-]ddd
g	Coma flotante	Valor con signo, dependiendo del valor de la precisión. Se rellenará con ceros y se añadirá el punto decimal si es necesario.
E	Coma flotante	Valor con signo: [-]d.dddd...E[+/-]ddd
G	Coma flotante	Como en g, pero se usa E para los exponentes.
Caracteres		
c	Carácter	Un carácter.
s	Puntero a cadena	Caracteres hasta que se encuentre un nulo o se alcance la precisión especificada.
Especial		
%	Nada	El carácter '%'
Punteros		
n	Puntero a int	Almacena la cuenta de los caracteres escritos.
p	Puntero	Imprime el argumento de entrada en formato de puntero: XXXX:YYYY ó YYYY

Veamos algunos ejemplos que nos aclaren este galimatías, en los comentarios a la derecha de cada "printf" se muestra la salida prevista:

```
#include <stdio.h>
void main(void)
{
   int i = 123;
   int j = -124;
   float x = 123.456;
   float y = -321.12;
```

```
   char Saludo[5] = "hola";

   printf("|%6d|\n", i); // | 123|
   printf("|%-6d|\n", i); // |123 |
   printf("|%06d|\n", i); // |000123|
   printf("|%+6d|\n", i); // | +123|
   printf("|%+6d|\n", j); // | -124|
   printf("|%+06d|\n", i); // |+00123|
   printf("|% 06d|\n", i); // | 00123|
   printf("|%6o|\n", i); // | 173|
   printf("|%#6o|\n", i); // | 0173|
   printf("|%06o|\n", i); // |000173|
   printf("|%-#6o|\n", i); // |0173 |
   printf("|%6x|\n", i); // | 7b|
   printf("|%#6X|\n", i); // | 0X7B|
   printf("|%#06X|\n", i); // |0X007B|
   printf("|%-#6x|\n", i); // |0x7b |
   printf("|%10.2f|\n", x); // | 123.46|
   printf("|%10.4f|\n", x); // |  123.4560|
   printf("|%010.2f|\n", x); // |0000123.46|
   printf("|%-10.2f|\n", x); // |123.46 |
   printf("|%10.2e|\n", x); // | 1.23e+02|
   printf("|%-+10.2e|\n", y);// |-3.21e+02 |
   printf("|%*.*f|\n", 14, 4, x); // | 123.4560|
   printf("%.2f  es el 10%% de %.2f\n", .10*x, x); // 12.35 es el 10%
de 123.46
   printf("%s es un saludo y %c una letra\n", Saludo, Saludo[2]); //
hola es un saludo y l una letra
   printf("%.2s es parte de un saludo\n", Saludo); // ho es parte de
un saludo
}
```

Observa el funcionamiento de este ejemplo, modifícalo y experimenta. Intenta predecir los resultados.

Librería de rutinas de conversión estándar "stdlib.h"

En esta librería se incluyen rutinas de conversión entre tipos. Nos permiten convertir cadenas de caracteres a números, números a cadenas de caracteres, números con decimales a números enteros, etc.

Función "atoi()"

Convierte una cadena de caracteres a un entero. Puede leerse como conversión de "ASCII to Integer".

Sintaxis:

int atoi(const char *s);

La cadena puede tener los siguientes elementos:

- Opcionalmente un conjunto de tabuladores o espacios.
- Opcionalmente un carácter de signo.
- Una cadena de dígitos.

El formato de la cadena de entrada sería: [ws] [sn] [ddd]

El primer carácter no reconocido finaliza el proceso de conversión, no se comprueba el desbordamiento, es decir si el número cabe en un "int". Si no cabe, el resultado queda indefinido.

Valor de retorno:

"atoi" devuelve el valor convertido de la cadena de entrada. Si la cadena no puede ser convertida a un número "int", "atoi" vuelve con 0.

Al mismo grupo pertenecen las funciones "atol" y "atof", que devuelven valores "long int" y "float". Se verán en detalle en otros capítulos.

Librería de rutinas de conversión estándar "stdlib.h"

Función "max()"

Devuelve el mayor de dos valores.

Sintaxis:

<tipo> max(<tipo> t1, <tipo> t2); // sólo en C++

En C está definida como una macro y en C++ como un "formato", el resultado es el mismo, la función acepta cualquier tipo, siempre y cuando se use el mismo tipo en ambos parámetros.

Valor de retorno:

"max" devuelve el mayor de los dos valores.

Función "min()"

Devuelve el menor de dos valores.

Sintaxis:

<tipo> min(<tipo> t1, <tipo> t2); // sólo en C++

En C está definida como una macro y en C++ como un "formato", el resultado es el mismo, la función acepta cualquier tipo, siempre y cuando se use el mismo tipo en ambos parámetros.

Valor de retorno:

"min" devuelve el menor de los dos valores.

Función "abs()"

Devuelve el valor absoluto de un entero.

Sintaxis:

int abs(int x);

"abs" devuelve el valor absoluto del valor entero de entrada, x. Si se llama a "abs" cuando se ha incluido la librería "stdlib.h", se la trata como una macro que se expandirá. Si se quiere usar la función "abs" en lugar de su macro, hay que incluir la línea:

#undef abs

en el programa, después de la línea:

#include <stdlib.h>

Esta función puede usarse con "bcd" y con "complejos".

Valor de retorno:

Esta función devuelve un valor entre 0 y el INT_MAX, salvo que el valor de entrada sea INT_MIN, en cuyo caso devolverá INT_MAX. Los valores de INT_MAX e INT_MIN están definidos en el fichero de cabecera "limit.h".

Función "random()"

Generador de números aleatorios.

Sintaxis:

int random(int num);

"random" devuelve un número aleatorio entre 0 y (num-1). Se trata de una macro definida en "stdlib.h". Tanto num como el número generado serán enteros.

Valor de retorno:

"random" devuelve un número entre 0 y (num-1).

Librería rutinas de conversión y clasificación de caracteres "ctype.h"

Función "toupper()"

Convierte un carácter a mayúscula.

Sintaxis:

int toupper(int ch);

"toupper" es una función que convierte el entero ch (dentro del rango EOF a 255) a su valor en mayúscula (A a Z; si era una minúscula de, a a z). Todos los demás valores permanecerán sin cambios.

Valor de retorno:

"toupper" devuelve el valor convertido si ch era una minúscula, en caso contrario devuelve ch.

Función "tolower()"

Convierte un carácter a mayúscula.

Sintaxis:

int tolower(int ch);

"tolower" es una función que convierte el entero ch (dentro del rango EOF a 255) a su valor en minúscula (A a Z; si era una mayúscula de, a a z). Todos los demás valores permanecerán sin cambios.

Valor de retorno:

"tolower" devuelve el valor convertido si ch era una mayúscula, en caso contrario devuelve ch.

Funciones "is<conjunto>()"

Las siguientes funciones son del mismo tipo, sirven para verificar si un carácter concreto pertenece a un conjunto definido. Estos conjuntos son: alfanumérico, alfabético, ascii, control, dígito, gráfico, minúsculas, imprimible, puntuación, espacio, mayúsculas y dígitos hexadecimales. Todas las funciones responden a la misma sintaxis:

int is<conjunto>(int c);

Función	Valores
isalnum	(A - Z o a - z) o (0 - 9)
isalpha	(A - Z o a - z)
isascii	0 - 127 (0x00-0x7F)
iscntrl	(0x7F o 0x00-0x1F)
isdigit	(0 - 9)
isgraph	Imprimibles menos ' '
islower	(a - z)
isprint	Imprimibles incluido ' '
ispunct	Signos de puntuación

isspace	espacio, tab, retorno de línea, cambio de línea, tab vertical, salto de página (0x09 a 0x0D, 0x20).
isupper	(A-Z)
isxdigit	(0 to 9, A to F, a to f)

Valores de retorno:

Cada una de las funciones devolverá un valor distindo de cero si el argumento c pertenece al conjunto.

Ejemplos capítulos 7 a 9

Ejemplo 5

Volvamos al ejemplo del capítulo 1, aquél que sumaba dos más dos. Ahora podemos comprobar si el ordenador sabe realmente sumar, le pediremos que nos muestre el resultado:

```
// Este programa suma 2 + 2 y muestra el resultado
// No me atrevo a firmarlo
#include <iostream.h>
int main()
{
   int a;

   a = 2 + 2;
   cout << "2 + 2 = " << a << endl;
   return 0;
}
```

Espero que tu ordenador fuera capaz de realizar este complicado cálculo, el resultado debe ser:

2 + 2 = 4

Nota: si estás compilando programas para consola en un compilador que trabaje en entorno Windows, probablemente no verás los resultados, esto es porque cuando el programa termina se cierra la ventana de consola automáticamente. Tienes dos opciones para evitar este inconveniente. Ejecuta los programas desde una ventana DOS, o añade la línea "getchar();", justo antes de la línea "return 0;", en este último caso, necesitarás incluir el fichero de cabecera "stdio.h". De este modo el programa no terminará hasta que pulses una tecla.

Ejemplo 6

Veamos un ejemplo algo más serio, hagamos un programa que muestre el alfabeto. Para complicarlo un poco más, debe imprimir dos líneas, la primera en mayúsculas y la segunda en minúsculas. Una pista, por si no sabes cómo se codifican los caracteres en el ordenador. A cada carácter le corresponde un número, conocido como código ASCII. Ya hemos hablado del ASCII de 256 y 128 caracteres, pero lo que interesa para este ejercicio es que las letras tienen códigos ASCII correlativos según el orden alfabético. Es decir, si al carácter 'A' le corresponde el código ASCII n, al carácter 'B' le corresponderá el n+1.

```
// Muestra el alfabeto de mayúsculas y minúsculas
#include <iostream.h>

int main()
{
   char a; // Variable auxiliar para los bucles
   // El bucle de las mayúsculas lo haremos con un while
   a = 'A';
```

```
   while(a <= 'Z') cout << a++;
   cout << endl; // Cambio de línea
   // El bucle de las minúsculas lo haremos con un for
   for(a = 'a'; a <= 'z'; a++) cout << a;
   cout << endl; // Cambio de línea return 0;
}
```

Tal vez eches de menos algún carácter, efectivamente la 'ñ' no sigue la norma del orden alfabético en ASCII, esto es porque el ASCII lo inventaron los anglosajones, y no se acordaron del español. De momento nos las apañaremos sin ella.

Ejemplo 7

Para este ejemplo queremos que se muestren cuatro líneas, la primera con el alfabeto, pero mostrando alternativamente las letras en mayúscula y minúscula, AbCdE... La segunda igual, pero cambiando mayúsculas por minúsculas, la tercera en grupos de dos, ABcdEFgh... y la cuarta igual pero cambiando mayúsculas por minúsculas.

Piensa un poco sobre el modo de resolver el problema. Ahora te daré la solución.

Por supuesto, para cada problema existen cientos de soluciones posibles, en general, cuanto más complejo sea el problema más soluciones existirán, aunque hay problemas muy complejos que no tienen ninguna solución, en apariencia.

Bien, después de este paréntesis, vayamos con el problema. Almacenaremos el alfabeto en una cadena, no importa si almacenamos mayúsculas o minúsculas. Necesitaremos una cadena de 27 caracteres, 26 letras y el terminador de cadena.

Una vez tengamos la cadena le aplicaremos diferentes procedimientos para obtener las combinaciones del enunciado.

```
// Muestra el alfabeto de mayúsculas y minúsculas:
// AbCdEfGhIjKlMnOpQrStUvWxYz
// aBcDeFgHiJkLmNoPqRsTuVwXyZ
// ABcdEFghIJklMNopQRstUVwxYZ
// abCDefGHijKLmnOPqrSTuvWXyz

#include <iostream.h>
#include <ctype.h>
int main()
{
   char alfabeto[27]; // Cadena que contendrá el alfabeto
   int i; // variable auxiliar para los bucles
   // Aunque podríamos haber iniciado alfabeto directamente,
   // lo haremos con un blucle
   for(i = 0; i < 26; i++) alfabeto[i] = 'a' + i;
   alfabeto[26] = 0; // No olvidemos poner el fin de cadena
   // Aplicamos el primer procedimiento si la posición es par
   // convertimos el carácter a minúscula, si es impar a mayúscula
   for(i = 0; alfabeto[i]; i++)
      if(i % 2) alfabeto[i] = tolower(alfabeto[i]);
      else alfabeto[i] = toupper(alfabeto[i]);
   cout << alfabeto << endl; // Mostrar resultado

   // Aplicamos el segundo procedimiento si el carácter era una
```

```
   // mayúscula lo cambiamos a minúscula, y viceversa
   for(i = 0; alfabeto[i]; i++)
      if(isupper(alfabeto[i])) alfabeto[i] = tolower(alfabeto[i]);
      else alfabeto[i] = toupper(alfabeto[i]);
   cout << alfabeto << endl; // Mostrar resultado

   // Aplicamos el tercer procedimiento, pondremos los dos primeros
   // caracteres directamente a mayúsculas, y recorreremos el resto de
   // la cadena, si el carácter dos posiciones a la izquierda es
   // mayúscula cambiamos el caracter actual a minúscula, y viceversa
   alfabeto[0] = 'A';
   alfabeto[1] = 'B';
   for(i = 2; alfabeto[i]; i++)
      if(isupper(alfabeto[i-2])) alfabeto[i] = tolower(alfabeto[i]);
      else alfabeto[i] = toupper(alfabeto[i]);
   cout << alfabeto << endl; // Mostrar resultado

   // El último procedimiento, es tan simple como aplicar
   // el segundo de nuevo
   for(i = 0; alfabeto[i]; i++)
      if(isupper(alfabeto[i])) alfabeto[i] = tolower(alfabeto[i]);
      else alfabeto[i] = toupper(alfabeto[i]);
   cout << alfabeto << endl; // Mostrar resultado

   return 0;
}
```

Ejemplo 8

Bien, ahora veamos un ejemplo tradicional en todos los cursos de C y C++.

Se trata de leer caracteres desde el teclado y contar cuántos hay de cada tipo. Los tipos que deberemos contar serán: consonantes, vocales, dígitos, signos de puntuación, mayúsculas, minúsculas y espacios. Cada carácter puede pertenecer a uno o varios grupos. Los espacios son utilizados frecuentemente para contar palabras.

```
// Cuenta letras
#include <iostream.h>
#include <ctype.h>
int main()
{
   int consonantes = 0;
   int vocales = 0;
   int digitos = 0;
   int mayusculas = 0;
   int minusculas = 0;
   int espacios = 0;
   int puntuacion = 0;
   char c; // caracteres leidos desde el teclado

   cout << "Contaremos caracteres hasta que se pulse ^Z" << endl;
   while((c = getchar()) != EOF)
   {
      if(isdigit(c)) digitos++;
      else if(isspace(c)) espacios++;
      else if(ispunct(c)) puntuacion++;
      else if(isalpha(c))
      {
```

```
         if(isupper(c)) mayusculas++; else minusculas++;
         switch(tolower(c)) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
               vocales++;
               break;
            default:
               consonantes++;
         }
      }
   }
   cout << "Resultados:" << endl;
   cout << "Dígitos:     " << digitos << endl;
   cout << "Espacios:    " << espacios << endl;
   cout << "Puntuación:  " << puntuacion << endl;
   cout << "Alfabéticos: " << mayusculas + minusculas << endl;
   cout << "Mayúsculas:  " << mayusculas << endl;
   cout << "Minúsculas:  " << minusculas << endl;
   cout << "Vocales:     " << vocales << endl;
   cout << "Consonantes: " << consonantes << endl;
   cout << "Total: " << digitos + espacios + vocales +
      consonantes + puntuacion << endl;
   return 0;
}
```

CAPITULO 10 Conversión de tipos

Quizás te hayas preguntado qué pasa cuando escribimos expresiones numéricas en las que todos los operandos no son del mismo tipo. Por ejemplo:

```
char n;
int a, b, c, d;
float r, s, t;
...
a = 10;
b = 100;
r = 1000;
c = a + b;
s = r + a;
d = r + b;
d = n + a + r;
t = r + a - s + c;
...
```

En estos casos, cuando los operandos de cada operación binaria asociados a un operador son de distinto tipo, se convierten a un tipo común. Existen reglas que rigen estas conversiones, y aunque pueden cambiar ligeramente de un compilador a otro, en general serán más o menos así:

1. Cualquier tipo entero pequeño como char o short es convertido a int o unsigned int. En este punto cualquier pareja de operandos será int (con o sin signo), double, float o long double.
2. Si algún operando es de tipo long double, el otro se convertirá a long double.
3. Si algún operando es de tipo double, el otro se convertirá a double.
4. Si algún operando es de tipo float, el otro se convertirá a float.
5. Si algún operando es de tipo unsigned long, el otro se convertirá a unsigned long.
6. Si algún operando es de tipo long, el otro se convertirá a long.
7. Si algún operando es de tipo unsigned int, el otro se convertirá a unsigned int.
8. En este caso ambos operandos son int.

Veamos ahora el ejemplo:

c = a + b; caso 8, ambas son int.

s = r + a; caso 4, "a" se convierte a float.

d = r + b; caso 4, "b" se convierte a float.

d = n + a + r; caso 1, "n" se convierte a int, caso 4 el resultado (n+a) se convierte a float.

t = r + a - s + c; caso 4, "a" se convierte a float, caso 4 (r+a) y "s" son float, caso 4, "c" se convierte a float.

También se aplica conversión de tipos en las asignaciones, cuando la variable receptora es de distinto tipo que el resultado de la expresión de la derecha.

Cuando esta conversión no implica pérdida de precisión, se aplican las mismas reglas que para los operandos, estas conversiones se conocen también como promoción de tipos. Cuando hay pérdida de precisión, las conversiones se conocen como democión de tipos. El compilador normalmente emite un aviso o "warning", cuando se hace una democión implícita, es decir cuando hay una democión automática.

En el caso de los ejemplos 3 y 4, es eso precisamente lo que ocurre, ya que estamos asignando expresiones de tipo float a variables de tipo int.

"Casting", conversiones explícitas de tipo:

Para eludir estos avisos del compilador se usa el "casting", o conversión explícita.

En general, el uso de "casting" es obligatorio cuando se hacen asignaciones, o cuando se pasan argumentos a funciones con pérdida de precisión. En el caso de los argumentos pasados a funciones es también muy recomendable aunque no haya pérdida de precisión. Eliminar los avisos del compilador demostrará que sabemos lo que hacemos con nuestras variables, aún cuando estemos haciendo conversiones de tipo extrañas.

Un "casting" tiene una de las siguientes formas:

(<nombre de tipo>**)**<expresión>

ó

<nombre de tipo>**(**<expresión>**)**

Esta última es conocida como notación funcional.

En el ejemplo anterior, las líneas 3 y 4 quedarían:

```
d = (int)(r + b);
d = (int)(n + a + r);
```

ó:

```
d = int(r + b);
d = int(n + a + r);
```

Hacer un "casting" indica que sabemos que el resultado de estas operaciones no es un int, que la variable receptora sí lo es, y que lo que hacemos lo hacemos a propósito. Veremos más adelante, cuando hablemos de punteros, más situaciones donde también es obligatorio el uso de "casting".

CAPITULO 11 Tipos de variables II: Arrays o Arreglos

Empezaremos con los tipos de datos estructurados, y con el más sencillo, los arrays.

Los arrays permiten agrupar datos usando un mismo identificador. Todos los elementos de un array son del mismo tipo, y para acceder a cada elemento se usan subíndices.

Sintaxis:

<tipo> <variable de array>**[**<número de elementos>**][[**<número de elementos>**]**...];

Los valores para el número de elementos deben ser constantes, y se pueden usar tantas dimensiones como queramos, limitado sólo por la memoria disponible.

Cuando sólo se usa una dimensión se suele hablar de listas o vectores, cuando se usan dos, de tablas.

Ahora podemos ver que las cadenas de caracteres son un tipo especial de arrays. Se trata en realidad de arrays de una dimensión de tipo char.

Los subíndices son enteros, y pueden tomar valores desde 0 hasta <número de elementos>-1. Esto es muy importante, y hay que tener mucho cuidado, por ejemplo:

```
int Vector[10];
```

Creará un array con 10 enteros a los que accederemos como Vector[0] a Vector[9].

Como subíndice podremos usar cualquier expresión entera.

En general C++ no verifica el ámbito de los subíndices. Si declaramos un array de 10 elementos, no obtendremos errores al acceder al elemento 11. Sin embargo, si asignamos valores a elementos fuera del ámbito declarado, estaremos accediendo a zonas de memoria que pueden pertenecer a otras variables o incluso al código ejecutable de nuestro programa, con consecuencias generalmente desastrosas.

Ejemplo:

```
int Tabla[10][10];
char DimensionN[4][15][6][8][11];
...
DimensionN[3][11][0][4][6] = DimensionN[0][12][5][3][1];
Tabla[0][0] += Tabla[9][9];
```

Cada elemento de Tabla, desde Tabla[0][0] hasta Tabla[9][9] es un entero. Del mismo modo, cada elemento de DimensionN es un carácter.

Asignación de arrays:

Los arrays pueden ser inicializados en la declaración.

Ejemplos:

```
float R[10] = {2, 32, 4.6, 2, 1, 0.5, 3, 8, 0, 12};
float S[] = {2, 32, 4.6, 2, 1, 0.5, 3, 8, 0, 12};
int N[] = {1, 2, 3, 6};
int M[][3] = { 213, 32, 32, 32, 43, 32, 3, 43, 21};
char Mensaje[] = "Error de lectura";
```

En estos casos no es obligatorio especificar el tamaño para la primera dimensión, como ocurre en los ejemplos de las líneas 2, 3, 4 y 5. En estos casos la dimensión que queda indefinida se calcula a partir del número de elementos en la lista de valores iniciales.

En el caso 2, el número de elementos es 10, ya que hay diez valores en la lista.

En el caso 3, será 4.

En el caso 4, será 3, ya que hay 9 valores, y la segunda dimensión es 3: 9/3=3.

Y en el caso 5, el número de elementos es 17, 16 caracteres más el cero de fin de cadena.

Algoritmos de ordenación, método de la burbuja:

Una operación que se hace muy a menudo con los arrays, sobre todo con los de una dimensión, es ordenar sus elementos.

Dedicaremos más capítulos a algoritmos de ordenación, pero ahora veremos uno de los más usados, aunque no de los más eficaces, se trata del método de la burbuja.

Consiste en recorrer la lista de valores a ordenar y compararlos dos a dos. Si los elementos están bien ordenados, pasamos al siguiente par, si no lo están los intercambiamos, y pasamos al siguiente, hasta llegar al final de la lista. El proceso completo se repite hasta que la lista está ordenada.

Lo veremos mejor con un ejemplo:

Ordenar la siguiente lista de menor a mayor:

15, 3, 8, 6, 18, 1.

Empezamos comparando 15 y 3. Como están mal ordenados los intercambiamos, la lista quedará:

3, 15, 8, 6, 18, 1

Tomamos el siguiente par de valores: 15 y 8, y volvemos a intercambiarlos, y segimos el proceso...

Cuando llegemos al final la lista estará así:

3, 6, 8, 15, 1, 18

Empezamos la segunda pasada, pero ahora no es necesario recorrer toda la lista. Si observas verás que el último elemento está bien ordenado, siempre será el mayor, por lo tanto no será necesario incluirlo en la segunda pasada. Después de la segunda pasada la lista quedará:

3, 6, 8, 1, 15, 18

Ahora es el 15 el que ocupa su posición final, la penúltima, por lo tanto no será necesario que entre en las comparaciones para la siguiente pasada. Las sucesivas pasadas dejarán la lista así:

3ª 3, 6, 1, 8, 15, 18

4ª 3, 1, 6, 8, 15, 18

5ª 1, 3, 6, 8, 15, 18

Ejercicios (creo que ya podemos empezar con los ejercicios :-) :

1. Hacer un programa que lea diez valores enteros en un array desde el teclado y calcule y muestre: la suma, el valor medio, el mayor y el menor.
2. Hacer un programa que lea diez valores enteros en un array y los muestre en pantalla. Después que los ordene de menor a mayor y los vuelva a mostrar. Y finalmente que los ordene de mayor a menor y los muestre por tercera vez. Para ordenar la lista usar una función que implemente el método de la burbuja y que tenga como parámetro de entrada el tipo de ordenación, de mayor a menor o de menor a mayor. Para el array usar una variable global.
3. Hacer un programa que lea 25 valores enteros en una tabla de 5 por 5, y que después muestre la tabla y las sumas de cada fila y de cada columna. Procura que la salida sea clara, no te limites a los números obtenidos.

CAPITULO 12 Tipos de variables III: Estructuras

Las estructuras son el segundo tipo de datos estructurados que veremos.

Las estructuras nos permiten agrupar varios datos, aunque sean de distinto tipo, que mantengan algún tipo de relación, y manipularlos todos juntos, con un mismo identificador, o por separado.

Las estructuras son llamadas también muy a menudo registros, o en inglés "records". Y son estructuras análogas en muchos aspectos a los registros de bases de datos. Y siguiendo la misma analogía, cada variable de una estructura se denomina a menudo campo, o "field".

Sintaxis:

struct [<nombre de la estructura>] {

[<tipo> <nombre de variable>[,<nombre de variable>,...]];

.

} [<variable de estructura>[,<variable de estructura>,...];

El nombre de la estructura es un nombre opcional para referirse a la estructura.

Las variables de estructura son variables declaradas del tipo de la estructura, y su inclusión también es opcional. Sin bien, al menos uno de estos elementos debe existir, aunque ambos sean opcionales.

En el interior de una estructura, entre las llaves, se pueden definir todos los elementos que consideremos necesarios, del mismo modo que se declaran las variables.

Las estructuras pueden referenciarse completas, usando su nombre, como hacemos con las variables que ya conocemos, y también se puede acceder a los elementos en el interior de la estructura usando el operador de selección (.), un punto.

También pueden declararse más variables del tipo de estructura en cualquier parte del programa, de la siguiente forma:

[struct] <nombre de la estructura> <variable de estructura>[,<variable de estructura>...];

La palabra "struct" es opcional en la declaración de variables.

Ejemplo:

```
struct Persona {
   char Nombre[65];
   char Direccion[65];
```

```
   int AnyoNacimiento;
} Fulanito;
```

Este ejemplo declara a Fulanito como una variable de tipo Persona. Para acceder al nombre de Fulanito, por ejemplo para visualizarlo, usaremos la forma:

```
cout << Fulanito.Nombre;
```

Funciones en el interior de estructuras:

C++, al contrario que C, permite incluir funciones en el interior de las estructuras. Normalmente estas funciones tienen la misión de manipular los datos incluidos en la estructura.

Aunque esta característica se usa casi exclusivamente con las clases, como veremos más adelante, también puede usarse en las estructuras.

Dos funciones muy particulares son las de inicialización, o constructor, y el destructor. Veremos con más detalle estas funciones cuando asociemos las estructuras y los punteros.

El constructor es una función sin tipo de retorno y con el mismo nombre que la estructura. El destructor tiene la misma forma, salvo que el nombre va precedido el operador "~".

Nota: para aquellos que usen un teclado español, el símbolo "~" se obtiene pulsando las teclas del teclado numérico 1, 2, 6, mientras se mantiene pulsada la tecla ALT, ([ALT]+126).

Veamos un ejemplo sencillo para ilustrar el uso de constructores:

Forma 1:

```
struct Punto {
   int x, y;
   Punto() {x = 0; y = 0;} // Constructor
} Punto1, Punto2;
```

Forma 2:

```
struct Punto {
   int x, y;
   Punto(); // Declaración del constructor
} Punto1, Punto2;

Punto::Punto() {  // Definición del constructor, fuera de la
estructura
   x = 0;
   y = 0;
}
```

Si no usáramos un constructor, los valores de x e y para Punto1 y Punto2 estarían indeterminados, contendrían la "basura" que hubiese en la memoria asignada a estas estructuras durante la ejecución. Con las estructuras éste será el caso más habitual.

Mencionar aquí, sólo a título de información, que el constructor no tiene por qué ser único. Se pueden incluir varios contructores, pero veremos esto mucho mejor y con más detalle cuando veamos las clases.

Usando constructores nos aseguramos los valores iniciales para los elementos de la estructura. Veremos que esto puede ser una gran ventaja, sobre todo cuando combinemos estructuras con punteros, en capítulos posteriores.

También podemos incluir otras funciones, que se declaran y definen como las funciones que ya conocemos, salvo que tienen restringido su ámbito al interior de la estructura.

Otro ejemplo:

```
#include <iostream.h>

struct stPareja {
   int A, B;
   int LeeA() { return A;} // Devuelve el valor de A
   int LeeB() { return B;} // Devuelve el valor de B
   void GuardaA(int n) { A = n;} // Asigna un nuevo valor a A
   void GuardaB(int n) { B = n;} // Asigna un nuevo valor a B
} Par;

int main() {
   Par.GuardaA(15);
   Par.GuardaB(63);
   cout << Par.LeeA() << endl;
   cout << Par.LeeB() << endl;

   return 0;
}
```

En este ejemplo podemos ver cómo se define una estructura con dos campos enteros, y dos funciones para modificar y leer sus valores. El ejemplo es muy simple, pero las funciones de guardar valores se pueden elaborar para que no permitan determinados valores, o para que hagan algún tratamiento de los datos.

Por supuesto se pueden definir otras funciones y también constructores más elaborados y sobrecarga de operadores. Y en general, las estructuras admiten cualquiera de las características de las clases, siendo en muchos aspectos equivalentes.

Veremos estas características cuando estudiemos las clases, y recordaremos cómo aplicarlas a las estructuras.

Asignación de estructuras:

La asignación de estructuras está permitida, pero sólo entre variables del mismo tipo de estructura, salvo que se usen constructores, y funciona como la intuición dice que debe hacerlo.

Veamos un ejemplo:

```
struct Punto {
   int x, y;
   Punto() {x = 0; y = 0;}
} Punto1, Punto2;

int main() {
   Punto1.x = 10;
   Punto1.y = 12;
   Punto2 = Punto1;
}
```

La línea:

```
Punto2 = Punto1;
```

equivale a:

```
Punto2.x = Punto1.x;
Punto2.y = Punto1.y;
```

Arrays de estructuras:

La combinación de las estructuras con los arrays proporciona una potente herramienta para el almacenamiento y manipulación de datos.

Ejemplo:

```
struct Persona {
   char Nombre[65];
   char Direccion[65];
   int AnyoNacimiento;
} Plantilla[200];
```

Vemos en este ejemplo lo fácil que podemos declarar el array Plantilla que contiene los datos relativos a doscientas personas.

Podemos acceder a los datos de cada uno de ellos:

```
cout << Plantilla[43].Direccion;
```

O asignar los datos de un elemento de la plantilla a otro:

```
Plantilla[0] = Plantilla[99];
```

Estructuras anidadas:

También está permitido anidar estructuras, con lo cual se pueden conseguir superestructuras muy elaboradas.

Ejemplo:

```
struct stDireccion {
   char Calle[64];
   int Portal;
   int Piso;
   char Puerta[3];
   char CodigoPostal[6];
   char Poblacion[32];
};

struct stPersona {
   struct stNombre {
      char Nombre[32];
      char Apellidos[64];
   } NombreCompleto;
   stDireccion Direccion;
   char Telefono[10];
};
...
```

En general, no es una práctica corriente definir estructuras dentro de estructuras, ya que resultan tener un ámbito local, y para acceder a ellas se necesita hacer referencia a la estructura más externa.

Por ejemplo para declarar una variable del tipo stNombre hay que utilizar el operador de acceso (::):

```
stPersona::stNombre NombreAuxiliar;
```

Sin embargo para declarar una variable de tipo stDireccion basta con declararla:

```
stDireccion DireccionAuxiliar;
```

Ejercicios:

1. Escribir un programa que almacene en un array los nombres y números de teléfono de 10 personas. El programa debe leer los datos introducidos por el usuario y guardarlos en memoria. Después debe ser capaz de buscar el nombre correspondiente a un número de teléfono y el teléfono correspondiente a una persona. Ambas opciones deben se accesibles a través de un menú, así como la opción de salir del programa. El menú debe tener esta forma, más o menos:

   a) Buscar por nombre

   b) Buscar por número de teléfono

   c) Salir

   Pulsa una opción:

CAPITULO 13 Tipos de variables IV: Punteros 1

Los punteros proporcionan la mayor parte de la potencia al C y C++, y marcan la principal diferencia con otros lenguajes de programación.

Una buena comprensión y un buen dominio de los punteros pondrá en tus manos una herramienta de gran potencia. Un conocimiento mediocre o incompleto te impedirá desarrollar programas eficaces.

Por eso le dedicaremos mucha atención y mucho espacio a los punteros. Es muy importante comprender bien cómo funcionan y cómo se usan.

Para entender qué es un puntero veremos primero cómo se almacenan los datos en un ordenador.

La memoria de un ordenador está compuesta por unidades básicas llamadas bits. Cada bit sólo puede tomar dos valores, normalmente denominados alto y bajo, ó 1 y 0. Pero trabajar con bits no es práctico, y por eso se agrupan.

Cada grupo de 8 bits forma un byte u octeto. En realidad el microprocesador, y por lo tanto nuestro programa, sólo puede manejar directamente bytes o grupos de dos o cuatro bytes. Para acceder a los bits hay que acceder antes a los bytes. Y aquí llegamos al quid, cada byte tiene una dirección, llamada normalmente dirección de memoria.

La unidad de información básica es la palabra, dependiendo del tipo de microprocesador una palabra puede estar compuesta por dos, cuatro, ocho o dieciséis bytes. Hablaremos en estos casos de plataformas de 16, 32, 64 ó 128 bits. Se habla indistintamente de direcciones de memoria, aunque las palabras sean de distinta longitud. Cada dirección de memoria contiene siempre un byte. Lo que sucederá cuando las palabras sean de 32 bits es que accederemos a posiciones de memoria que serán múltiplos de 4.

Todo esto sucede en el interior de la máquina, y nos importa más bien poco. Podemos saber qué tipo de plataforma estamos usando averiguando el tamaño del tipo int, y para ello hay que usar el operador "sizeof()", por ejemplo:

```
cout << "Plataforma de " << 8 * sizeof(int) << " bits" << endl;
```

Ahora veremos cómo funcionan los punteros. Un puntero es un tipo especial de variable que contiene, ni más ni menos que, una dirección de memoria. Por supuesto, a partir de esa dirección de memoria puede haber cualquier tipo de objeto: un char, un int, un float, un array, una estructura, una función u otro puntero. Seremos nosotros los responsables de decidir ese contenido.

Declaración de punteros:

Los punteros se declaran igual que el resto de las variables, pero precediendo el identificador con el operador de indirección, (*), que leeremos como "puntero a".

Sintaxis:

<tipo> *<identificador>;

Ejemplos:

```
int *entero;
char *carácter;
struct stPunto *punto;
```

Los punteros siempre apuntan a un objeto de un tipo determinado, en el ejemplo, "entero" siempre apuntará a un objeto de tipo "int".

La forma:

<tipo>* <indentificador>;

con el (*) junto al tipo, en lugar de junto al identificador de variable, también está permitida.

Veamos algunos matices. Tomemos el primer ejemplo:

```
int *entero;
```

equivale a:

```
int* entero;
```

Debes tener muy claro que "entero" es una variable del tipo "puntero a int", y que **"*entero" NO es una variable de tipo "int"**.

Si "entero" apunta a una variable de tipo "int", "*entero" será el contenido de esa variable, pero no olvides que "*entero" es un operador aplicado a una variable de tipo "puntero a int", es decir "*entero" es una expresión, no una variable.

Para averiguar la dirección de memoria de cualquier variable usaremos el operador de dirección (&), que leeremos como "dirección de".

Declarar un puntero no creará un objeto. Por ejemplo: "int *entero;" no crea un objeto de tipo "int" en memoria, sólo crea una variable que puede contener una dirección de memoria. Se puede decir que existe físicamente la variable "entero", y también que esta variable puede contener la dirección de un objeto de tipo "int". Lo veremos mejor con otro ejemplo:

```
int A, B;
int *entero;
...
B = 213; /* B vale 213 */
entero = &A; /* entero apunta a la dirección de la variable A */
*entero = 103; /* equivale a la línea A = 103; */
B = *entero; /* equivale a B = A; */
...
```

En este ejemplo vemos que "entero" puede apuntar a cualquier variable de tipo "int", y que podemos hacer referencia al contenido de dichas variables usando el operador de indirección (*).

Como todas las variables, los punteros también contienen "basura" cuando son declaradas. Es costumbre dar valores iniciales nulos a los punteros que no apuntan a ningún sitio concreto:

```
entero = NULL;
caracter = NULL;
```

NULL es una constante, que está definida como cero en varios ficheros de cabecera, como "stdio.h" o "iostream.h", y normalmente vale 0L.

Correspondencia entre arrays y punteros:

Existe una equivalencia casi total entre arrays y punteros. Cuando declaramos un array estamos haciendo varias cosas a la vez:

- Declaramos un puntero del mismo tipo que los elementos del array, y que apunta al primer elemento del array.
- Reservamos memoria para todos los elementos del array. Los elementos de un array se almacenan internamente en el ordenador en posiciones consecutivas de la memoria.

La principal diferencia entre un array y un puntero es que el nombre de un array es un puntero constante, no podemos hacer que apunte a otra dirección de memoria. Además, el compilador asocia una zona de memoria para los elementos del array, cosa que no hace para los elementos apuntados por un puntero auténtico.

Ejemplo:

```
int vector[10];
int *puntero;

puntero = vector; /* Equivale a puntero = &vector[0];
  esto se lee como "dirección del elemento cero de vector" */
*puntero++; /* Equivale a vector[0]++; */
puntero++; /* entero == &vector[1] */
```

¿Qué hace cada una de estas instrucciones?:

La primera incrementa el contenido de la memoria apuntada por "entero", que es vector[0].

La segunda incrementa el puntero, esto significa que apuntará a la posición de memoria del siguiente "int", pero no a la siguiente posición de memoria. El puntero no se incrementará en una unidad, como tal vez sería lógico esperar, sino en la longitud de un "int".

Análogamente la operación:

```
puntero = puntero + 7;
```

No incrementará la dirección de memoria almacenada en "puntero" en siete posiciones, sino en 7*sizeof(int).

Otro ejemplo:

```
struct stComplejo {
   float real, imaginario;
} Complejo[10];

stComplejo *p;
p = Complejo; /* Equivale a p = &Complejo[0]; */
p++; /* entero  == &Complejo[1] */
```

En este caso, al incrementar p avanzaremos las posiciones de memoria necesarias para apuntar al siguiente complejo del array "Complejo". Es decir avanzaremos sizeof(stComplejo) bytes.

Operaciones con punteros:

Aunque no son muchas las operaciones que se pueden hacer con los punteros, cada una tiene sus peculiaridades.

Asignación.

Ya hemos visto cómo asignar a un puntero la dirección de una variable. También podemos asignar un puntero a otro, esto hará que los dos apunten a la misma posición:

```
int *q, *p;
int a;

q = &a; /* q apunta al contenido de a */
p = q; /* p apunta al mismo sitio, es decir, al contenido de a */
```

Operaciones aritméticas.

También hemos visto como afectan a los punteros las operaciones de suma con enteros. Las restas con enteros operan de modo análogo.

Pero, ¿qué significan las operaciones de suma y resta entre punteros?, por ejemplo:

```
int vector[10];
int *p, *q;

p = vector; /* Equivale a p = &vector[0]; */
q = &vector[4]; /* apuntamos al 5º elemento */
cout << q-p << endl;
```

El resultado será 4, que es la "distancia" entre ambos punteros. Normalmente este tipo de operaciones sólo tendrá sentido entre punteros que apunten a elementos del mismo array.

La suma de punteros no está permitida.

Comparación entre punteros.

Comparar punteros puede tener sentido en la misma situación en la que lo tiene restar punteros, es decir, averiguar posiciones relativas entre punteros que apunten a elementos del mismo array.

Existe otra comparación que se realiza muy frecuente con los punteros. Para averiguar si estamos usando un puntero es corriente hacer la comparación:

if(NULL != p), o simplemente if(p) y también:

if(NULL == p), o simplemente if(!p).

Punteros genéricos.

Es posible declarar punteros sin tipo concreto:

void *<identificador>;

Estos punteros pueden apuntar a objetos de cualquier tipo.

Por supuesto, también se puede emplear el "casting" con punteros, sintaxis:

(<tipo> *)<variable puntero>

Por ejemplo:

```
#include <iostream.h>

int main() {
   char cadena[10] = "Hola";
   char *c;
   int *n;
   void *v;

   c = cadena; // c apunta a cadena
   n = (int *)cadena; // n también apunta a cadena
   v = (void *)cadena; // v también
   cout << "carácter: " << *c << endl;
   cout << "entero:   " << *n << endl;
   cout << "float:    " << *(float *)v << endl;
   return 0;
}
```

El resultado será:

```
carácter: H
entero:   1634496328
float:    2.72591e+20
```

Vemos que tanto "cadena" como los punteros "n", "c" y "v" apuntan a la misma dirección, pero cada puntero tratará la información que encuentre allí de modo diferente, para "c" es un carácter y para "n" un entero. Para "v" no tiene tipo definido, pero podemos hacer "casting" con el tipo que queramos, en este ejemplo con float.

Nota: el tipo de línea del tercer "cout" es lo que suele asustar a los no iniciados en C y C++, y se parece mucho a lo que se conoce como código ofuscado. Parece como si en C casi cualquier expresión pudiese compilar.

Punteros a estructuras:

Los punteros también pueden apuntar a estructuras. En este caso, para referirse a cada elemento de la estructura se usa el operador (->), en lugar del (.).

Ejemplo:

```
#include <iostream.h>

struct stEstructura {
   int a, b;
} estructura, *e;

int main() {
   estructura.a = 10;
   estructura.b = 32;
   e = &estructura;

   cout << "variable" << endl;
   cout << e->a << endl;
   cout << e->b << endl;
   cout << "puntero" << endl;
   cout << estructura.a << endl;
   cout << estructura.b << endl;

   return 0;
}
```

Ejemplos:

Veamos algunos ejemplos de cómo trabajan los punteros.

Primero un ejemplo que ilustra la diferencia entre un array y un puntero:

```
#include <iostream.h>

int main() {
   char cadena1[] = "Cadena 1";
   char *cadena2 = "Cadena 2";

   cout << cadena1 << endl;
   cout << cadena2 << endl;

   //cadena1++; // Ilegal, cadena1 es constante
   cadena2++; // Legal, cadena2 es un puntero
```

```
   cout << cadena1 << endl;
   cout << cadena2 << endl;

   cout << cadena1[1] << endl;
   cout << cadena2[0] << endl;

   cout << cadena1 + 2 << endl;
   cout << cadena2 + 1 << endl;

   cout << *(cadena1 + 2) << endl;
   cout << *(cadena2 + 1) << endl;
  }
```

Aparentemente, y en la mayoría de los casos, cadena1 y cadena2 son equivalentes, sin embargo hay operaciones que están prohibidas con los arrays, ya que son punteros constantes.

Otro ejemplo:

```
#include <iostream.h>

int main() {
   char Mes[][11] = { "Enero", "Febrero", "Marzo", "Abril",
      "Mayo", "Junio", "Julio", "Agosto",
      "Septiembre", "Octubre", "Noviembre", "Diciembre"};
   char *Mes2[] = { "Enero", "Febrero", "Marzo", "Abril",
      "Mayo", "Junio", "Julio", "Agosto",
      "Septiembre", "Octubre", "Noviembre", "Diciembre"};

   cout << "Tamaño de Mes: " << sizeof(Mes) << endl;
   cout << "Tamaño de Mes2: " << sizeof(Mes2) << endl;
   cout << "Tamaño de cadenas de Mes2: " << &Mes2[11][10]-Mes2[0] <<
endl;
   cout << "Tamaño de Mes2 + cadenas : " <<
sizeof(Mes2)+&Mes2[11][10]-Mes2[0] << endl;

   return 0;
}
```

En este ejemplo declaramos un array "Mes" de dos dimensiones que almacena 12 cadenas de 11 caracteres, 11 es el tamaño necesario para almacenar el mes más largo (en caracteres): "Septiembre".

Después declaramos "Mes2" que es un array de punteros a char, para almacenar la misma información. La ventaja de este segundo método es que no necesitamos contar la longitud de las cadenas para calcular el espacio que necesitamos, cada puntero de Mes2 es una cadena de la longitud adecuada para almacenar cada mes.

Parece que el segundo sistema es más económico en cuanto al uso de memoria, pero hay que tener en cuenta que además de las cadenas también se almacenan los doce punteros.

El espacio necesario para almacenar los punteros lo dará la segunda línea de la salida. Y el espacio necesario para las cadenas lo dará la tercera línea.

Si las diferencias de longitud entre las cadenas fueran mayores, el segundo sistema sería más eficiente en cuanto al uso de la memoria.

Variables dinámicas:

Donde mayor potencia desarrollan los punteros es cuando se unen al concepto de memoria dinámica.

Cuando se ejecuta un programa, el sistema operativo reserva una zona de memoria para el código o instrucciones del programa y otra para las variables que se usan durante la ejecución. A menudo estas zonas son la misma zona, es lo que se llama memoria local. También hay otras zonas de memoria, como la pila, que se usa, entre otras cosas, para intercambiar datos entre funciones. El resto, la memoria que no se usa por ningún programa es lo que se conoce como "heap" o montón. Cuando nuestro programa use memoria dinámica, normalmente usará memoria del montón, y no se llama así porque sea de peor calidad, sino porque suele haber realmente un montón de memoria de este tipo.

C++ dispone de dos operadores para acceder a la memoria dinámica, son "new" y "delete". En C estas acciones se realizan mediante funciones de la librería estándar "mem.h".

Hay una regla de oro cuando se usa memoria dinámica, toda la memoria que se reserve durante el programa hay que liberarla antes de salir del programa. No seguir esta regla es una actitud muy irresponsable, y en la mayor parte de los casos tiene consecuencias desastrosas. No os fieis de lo que diga el compilador, de que estas variables se liberan solas al terminar el programa, no siempre es verdad.

Veremos con mayor profundidad los operadores "new" y "delete" en el siguiente capítulo, por ahora veremos un ejemplo:

```
#include <iostream.h>

int main() {
   int *a;
   char *b;
   float *c;
   struct stPunto {
      float x,y;
   } *d;

   a = new int;
   b = new char;
   c = new float;
   d = new stPunto;

   *a = 10;
   *b = 'a';
   *c = 10.32;
   d->x = 12; d->y = 15;

   cout << "a = " << *a << endl;
   cout << "b = " << *b << endl;
   cout << "c = " << *c << endl;
   cout << "d = (" << d->x << ", " << d->y << ")" << endl;

   delete a;
```

```
   delete b;
   delete c;
   delete d;

   return 0;
}
```

Y mucho cuidado: si pierdes un puntero a una variable reservada dinámicamente, no podrás liberarla.

Ejemplo:

```
int main() {
   int *a;

   a = new int; // variable dinámica
   *a = 10;
   a = new int; // nueva variable dinámica, se pierde la anterior
   *a = 20;
   delete a;  // sólo liberamos la última reservada
   return 0;
}
```

En este ejemplo vemos cómo es imposible liberar la primera reserva de memoria dinámica. Si no la necesitábamos habría que liberarla antes de reservarla otra vez, y si la necesitamos, hay que guardar su dirección, por ejemplo con otro puntero.

Ejercicios:

1. Escribir un programa con una función que calcule la longitud de una cadena de caracteres. El nombre de la función será LongitudCadena, debe devolver un "int", y como parámetro de entrada debe tener un puntero a "char". En "main" probar con distintos tipos de cadenas: arrays y punteros.

2. Escribir un programa con una función que busque un carácter determinado en una cadena. El nombre de la función será BuscaCaracter, debe devolver un "int" con la posición en que fue encontrado el carácter, si no se encontró volverá con -1. Los parámetros de entrada serán una cadena y un carácter. En la función "main" probar con distintas cadenas y caracteres.

CAPITULO 14 Operadores II: Más operadores

Veremos ahora más detalladamente algunos operadores que ya hemos mencionado, y algunos nuevos.

Operadores de Referencia (&) e Indirección (*)

El operador de referencia (&) nos devuelve la dirección de memoria del operando.

Sintaxis:

&<expresión simple>

El operador de indirección (*) considera a su operando como una dirección y devuelve su contenido.

Sintaxis:

*<puntero>

Operador "sizeof"

Este operador tiene dos usos diferentes.

Sintaxis:

sizeof (<expresión>)

sizeof (nombre_de_tipo)

En ambos casos el resultado es una constante entera que da el tamaño en bytes del espacio de memoria usada por el operando, que es determinado por su tipo. El espacio reservado por cada tipo depende de la plataforma.

En el primer caso, el tipo del operando es determinado sin evaluar la expresión, y por lo tanto sin efectos secundarios. Si el operando es de tipo "char", el resultado es 1. Si el operando es el nombre de un array, el resultado será el tamaño total de ésta, y no el del tipo de cada elemento.

A pesar de su apariencia, sizeof() NO es una función, sino un OPERADOR.

Operadores . y ->

Operador de selección (.). Permite acceder a variables o campos dentro de una estructura.

Sintaxis:

<variable de tipo estructura>.<nombre de variable>

Operador de selección de variables o campos para estructuras referenciadas con punteros. (->)

Sintaxis:

<variable de tipo puntero a estructura>-><nombre de variable>

Operador de preprocesador

El operador "#" sirve para dar órdenes o directivas al compilador. La mayor parte de las directivas del preprocesador se verán en capítulos posteriores.

Veremos, sin embargo dos de las más usadas.

Directiva define:

La directiva "#define", sirve para definir macros. Esto suministra un sistema para la sustitución de palabras, con y sin parámetros.

Sintaxis:

#define <identificador_de_macro> <secuencia>

El preprocesador sustituirá cada ocurrencia del <identificador_de_macro> en el fichero fuente, por la <secuencia> aunque hay algunas excepciones. Cada sustitución se conoce como una expansión de la macro, y la secuencia es llamada a menudo cuerpo de la macro.

Si la secuencia no existe, el <identificador_de_macro> será eliminado cada vez que aparezca en el fichero fuente.

Después de cada expansión individual, se vuelve a examinar el texto expandido a la búsqueda de nuevas macros, que serán expandidas a su vez. Esto permite la posibilidad de hacer macros anidadas. Si la nueva expansión tiene la forma de una directiva de preprocesador, no será reconocida como tal.

Existen otras restricciones a la expansión de macros:

Las ocurrencias de macros dentro de literales, cadenas, constantes alfanuméricas o comentarios no serán expandidas.

Una macro no será expandida durante su propia expansión, así #define A A, no será expandida indefinidamente.

Ejemplo:

```
#define suma(a,b) (a)+(b)
```

Los paréntesis en el cuerpo de la macro son necesarios para que funcione correctamente en todos los casos, lo veremos mucho mejor con otro ejemplo:

```
#include <iostream.h>
#define mult1(a,b) a*b
#define mult2(a,b) (a)*(b)
int main() {
   // En este caso ambas macros funcionan bien:
   cout << mult1(4,5) << endl;
   cout << mult2(4,5) << endl;
   // En este caso la primera macro no funciona bien, ¿por qué?:
   cout << mult1(2+2,2+3) << endl;
   cout << mult2(2+2,2+3) << endl;
}
```

¿Por qué falla la macro mult1 en el segundo caso?. Veamos cómo trabaja el preprocesador. Cuando el preprocesador encuentra una macro la expande, el código expandido sería:

```
int main() {
   // En este caso ambas macros funcionan bien:
   cout << 4*5 << endl;
   cout << (4)*(5) << endl;
   // En este caso la primera macro no funciona bien, ¿por qué?:
   cout << 2+2*2+3 << endl;
   cout << (2+2)*(2+3) << endl;
}
```

Al evaluar "2+2*2+3" se asocian los operandos dos a dos de izquierda a derecha, pero la multiplicación tiene prioridad sobre la suma, así que el compilador resuelve 2+4+3 = 9. Al evaluar "(2+2)*(2+3)" los paréntesis rompen la prioridad de la multiplicación, el compilador resuelve 4*5 = 20.

Directiva include:

La directiva "#include", como ya hemos visto, sirve para insertar ficheros externos dentro de nuestro fichero de código fuente. Estos ficheros son conocidos como ficheros incluidos, ficheros de cabecera o "headers".

Sintaxis:

#include <nombre de fichero cabecera>

#include "nombre de fichero de cabecera"

#include identificador_de_macro

El preprocesador elimina la línea "#include" y la sustituye por el fichero especificado. El tercer caso halla el nombre del fichero como resultado de aplicar la macro.

La diferencia entre escribir el nombre del fichero entre "<>" o """", está en el algoritmo usado para encontrar los ficheros a incluir. En el primer caso el preprocesador buscará

en los directorios "include" definidos en el compilador. En el segundo, se buscará primero en el directorio actual, es decir, en el que se encuentre el fichero fuente, si el fichero no existe en ese directorio, se trabajará como el primer caso. Si se proporciona el camino como parte del nombre de fichero, sólo se buscará es ese directorio.

Operadores de manejo de memoria "new" y "delete"

Veremos su uso en el capítulo de punteros II y en mayor profundidad en el capítulo de clases y en operadores sobrecargados.

Operador new:

El operador new sirve para reservar memoria dinámica.

Sintaxis:

[::] new [<emplazamiento>] [(<inicialización>)]

[::] new [<emplazamiento>] () [(<inicialización>)]

[::] new [<emplazamiento>] **[**<número de elementos>**]**

[::] new [<emplazamiento>] ()**[**<número de elementos>**]**

El operador opcional :: está relacionado con la sobrecarga de operadores, de momento no lo usaremos. Lo mismo se aplica a emplazamiento.

La inicialización, si aparece, se usará para asignar valores iniciales a la memoria reservada con new, pero no puede ser usada con arrays.

Las formas tercera y cuarta se usan para reservar memoria para arrays dinámicas. La de la memoria reservada con new será válida hasta que se libere con delete o hasta el fin del programa. Aunque es aconsejable liberar siempre la memoria reservada con new usando delete. Se considera una práctica sospechosa no hacerlo.

Si la reserva de memoria no tuvo éxito, new devuelve un puntero nulo, NULL.

Operador delete:

El operador delete se usa para liberar la memoria dinámica reservada con new.

Sintaxis: [::]

delete [<expresión>]

[::] delete**[]** [<expresión>]

La expresión será normalmente un puntero, el operador delete[] se usa para liberar memoria de arrays dinámicas.

Es importante liberar siempre usando delete la memoria reservada con new. Existe el peligro de pérdida de memoria si se ignora esta regla (memory leaks).

Cuando se usa el operador delete con un puntero nulo, no se realiza ninguna acción. Esto permite usar el operador delete con punteros sin necesidad de preguntar si es nulo antes.

Nota: los operadores new y delete son propios de C++. En C se usan las funciones malloc y free para reservar y liberar memoria dinámica y liberar un puntero nulo con free suele tener consecuencias desastrosas.

Veamos algunos ejemplos:

```
int main() {
   char *c;
   int *i = NULL;
   float **f;
   int n;

   c = new char[123]; // Cadena de 122 caracteres
   f = new (float *)[10]; // Array de 10 punteros a float
   for(n = 0; n < 10; n++) f[n] = new float[10]; // Cada elemento del
array es un array de 10 float
   // f es un array de 10*10
   f[0][0] = 10.32;
   f[9][9] = 21.39;
   c[0] = 'a';
   c[1] = 0;
   // liberar memoria dinámica
   for(n = 0; n < 10; n++) delete[] f[n];
   delete[] f;
   delete[] c;
   delete i;
   return 0;
}
```

CAPITULO 15 Operadores III: Precedencia.

Normalmente, las expresiones con operadores se evalúan de izquierda a derecha. Sin embargo no todos los operadores tienen la misma prioridad, algunos se evalúan antes que otros. Esta propiedad de los operadores se conoce como precedencia o prioridad.También hay algunos operadores que se evalúan y se asocian de derecha a izquierda.

Veremos ahora las prioridades de todos los operadores incluidos los que aún conocemos. Considera esta tabla como una referencia, no es necesario aprenderla de memoria, en caso de duda siempre se puede consultar, incluso puede que cambie ligeramente según el compilador, y en último caso veremos sistemas para eludir la precedencia.

Operadores	Asociatividad
() [] -> :: .	Izquierda a derecha
Operadores unarios: ! ~ + - ++ -- & (dirección de) * (puntero a) sizeof new delete	Derecha a izquierda
.* ->*	Izquierda a derecha
* (multiplicación) / %	Izquierda a derecha
+ - (operadores binarios)	Izquierda a derecha
<< >>	Izquierda a derecha
< <= > >=	Izquierda a derecha
== !=	Izquierda a derecha
& (bitwise AND)	Izquierda a derecha
^ (bitwise XOR)	Izquierda a derecha
| (bitwise OR)	Izquierda a derecha
&&	Izquierda a derecha
||	Izquierda a derecha

?:	Derecha a izquierda
= *= /= %= += -= &= ^= |= <<= >>=	Derecha a izquierda
,	Izquierda a derecha

La tabla muestra las precedencias de los operadores en orden decreciente, los de mayor precedencia en la primera fila. Dentro de la misma fila, la prioridad se decide por el orden de asociatividad.

La asociatividad nos dice en que orden se aplican los operadores en expresiones complejas, por ejemplo:

```
 int a, b, c, d, e;
b = c = d = e = 10;
```

El operador de asignación "=" se asocia de derecha a izquierda, es decir, primero se aplica "e = 10", después "d = e", etc. O sea, a todas las variables se les asigna el mismo valor: 10.

```
 a = b * c + d * e;
```

El operador * tiene mayor precedencia que + e =, por lo tanto se aplica antes, después se aplica el operador +, y por último el =. El resultado final será asignar a "a" el valor 200.

```
int m[10] = {10,20,30,40,50,60,70,80,90,100}, *f;
f = &m[5];
++*f;
cout << *f << endl;
```

La salida de este ejemplo será, 61, los operadores unarios tienen todos la misma precedencia, y se asocian de derecha a izquierda. Primero se aplica el *, y después el incremento al contenido de f.

```
f = &m[5];
*f--;
cout << *f << endl;
```

La salida de este ejemplo será, 50. Primero se aplica el decremento al puntero, y después el *.

```
a = b * (c + d) * e;
```

Ahora el operador de mayor peso es (), ya que los paréntesis están en el grupo de mayor precedencia. Todo lo que hay entre los paréntesis se evalúa antes que cualquier otra cosa. Primero se evalúa la suma, y después las multiplicaciones. El resultado será asignar a la variable "a" el valor 2000.

Este es el sistema para eludir las precedencias por defecto, si queremos evaluar antes una suma que un producto, debemos usar paréntesis.

Ejercicios:

Evaluar, sin usar un compilador, las siguientes del final.

Variables:

int a = 10, b = 100, c = 30, d = 1, e = 54;

int m[10] = {10,20,30,40,50,60,70,80,90,100};

int *p = m[3], *q = m[6];

Expresiones:

a + m[c/a] + b-- * m[1] / *q + 10 + a--;

a + (b * (c - d) + a) * *p++;

m[d] - d * e + (m[9] + b) / *p;

b++ * c-- + *q * m[2] / d;

CAPITULO 16 Funciones II: Parámetros por valor y por referencia.

Dediquemos algo más de tiempo a las funciones.

Hasta ahora siempre hemos declarado los parámetros de nuestras funciones del mismo modo. Sin embargo, éste no es el único modo que existe para pasar parámetros.

La forma en que hemos declarado y pasado los parámetros de las funciones hasta ahora es la que normalmente se conoce como "por valor". Esto quiere decir que cuando el control pasa a la función, los valores de los parámetros en la llamada se copian a "variables" locales de la función, estas "variables" son de hecho los propios parámetros.

Lo veremos mucho mejor con un ejemplo:

```
#include <iostream.h>

int funcion(int n, int m);

int main() {
   int a, b;
   a = 10;
   b = 20;

   cout << "a,b ->" << a << ", " << b << endl;
   cout << "funcion(a,b) ->" << funcion(a, b) << endl;
   cout << "a,b ->" << a << ", " << b << endl;
   cout << "funcion(10,20) ->" << funcion(10, 20) << endl;
   return 0;
}

int funcion(int n, int m) {
   n = n + 5;
   m = m - 5;
   return n+m;
}
```

Bien, ¿qué es lo que pasa en este ejemplo?. Empezamos haciendo a = 10 y b = 20, después llamamos a la función "funcion" con las variables a y b como parámetros. Dentro de "funcion" los parámetros se llaman n y m, y cambiamos sus valores, sin embargo al retornar a "main", a y b conservan sus valores originales. ¿Por qué?.

La respuesta es que lo que pasamos no son las variables a y b, sino que copiamos sus valores a las variables n y m.

Piensa en lo que pasa cuando llamamos a la función con parámetros constantes, en el ejemplo, la segunda llamada a "funcion". Los valores de los parámetros no pueden cambiar al retornar de "funcion", ya que son constantes.

Referencias a variables:

Las referencias sirven para definir "alias" o nombres alternativos para una misma variable. Para ello se usa el opereador de referencia (&).

Sintaxis:

<tipo> &<alias> = <variable de referencia>

<tipo> &<alias>

La primera forma es la que se usa para declarar variables de referencia, la asignación es obligatoria, no pueden definirse referencias indeterminadas.

La segunda forma es la que se usa para definir parámetros por referencia en funciones.

Ejemplo:

```
int main() {
   int a;
   int &r = a;

   a = 10;
   cout << r << endl;
   return 0;
}
```

En este ejemplo las variables a y r se refieren al mismo objeto, cualquier cambio en una de ellas se produce en ambas. Para todos los efectos, son la misma variable.

Pasando parámetros por referencia:

Si queremos que los cambios realizados en los parámetros dentro de la función se conserven al retornar de la llamada, deberemos pasarlos por referencia. Esto se hace declarando los parámetros de la función como referencias a variables. Ejemplo:

```
#include <iostream.h>

int funcion(int &n, int &m);

int main() {
   int a, b;

   a = 10; b = 20;
   cout << "a,b ->" << a << ", " << b << endl;
   cout << "funcion(a,b) ->" << funcion(a, b) << endl;
   cout << "a,b ->" << a << ", " << b << endl;
   /* cout << "funcion(10,20) ->" << funcion(10, 20) << endl;
   es ilegal pasar constantes como parámetros cuando estos son
referencias */
   return 0;
}

int funcion(int &n, int &m) {
   n = n + 5;
   m = m - 5;
   return n+m;
```

```
}
```

En este caso, las variables "a" y "b" tendrán valores distintos después de llamar a la función. Cualquier cambio que realicemos en los parámetros dentro de la función, se hará también en las variables referenciadas. Esto quiere decir que no podremos llamar a la función con parámetros constantes.

Punteros como parámetros de funciones:

Cuando pasamos como parámetro por valor de una función un puntero pasa lo mismo que con las variables. Dentro de la función trabajamos con una copia del puntero. Sin embargo, el objeto apuntado por el puntero será el mismo, los cambios que hagamos en los objetos apuntados por el puntero se conservarán al abandonar la función, sin embargo los cambios que hagamos al propio puntero no.

Ejemplo:

```
#include <iostream.h>

void funcion(int *q);

int main() {
   int a;
   int *p;

   a = 100;
   p = &a;
   // Llamamos a funcion con un puntero funcion(p);
   cout << "Variable a: " << a << endl;
   cout << "Variable *p: " << *p << endl;
   // Llamada a funcion con la dirección de "a" (constante)
   funcion(&a);
   cout << "Variable a: " << a << endl;
   cout << "Variable *p: " << *p << endl;
   return 0;
}

void funcion(int *q) {
   // Cambiamos el valor de la variable apuntada por
   // el puntero
   *q += 50;
   q++;
}
```

Con este tipo de parámetro para función pasamos el puntero por valor. ¿Y cómo haríamos para pasar un puntero por referencia?:

void funcion(int* &q);

El operador de referencia siempre se pone junto al nombre de la variable.

Arrays como parámetros de funciones:

Cuando pasamos un array como parámetro en realidad estamos pasando un puntero al primer elemento del array, así que las modificaciones que hagamos en los elementos del array dentro de la función serán validos al retornar.

Sin embargo, si sólo pasamos el nombre del array de más de una dimensión no podremos acceder a los elementos del array mediante subindices, ya que la función no tendrá información sobre el tamaño de cada dimensión.

Para tener acceso a arrays de más de una dimensión dentro de la función se debe declarar el parámetro como un array Ejemplo:

```
#include <iostream.h>

#define N 10
#define M 20

void funcion(int tabla[][M]);
// recuerda que el nombre de los parámetros en los
// prototipos es opcional, la forma:
// void funcion int [][M]);
// es válida también.

int main() {
   int Tabla[N][M];
...
   funcion(Tabla);
...
   return 0;
}

void funcion(int tabla[][M]){
...
   cout << tabla[2][4] << endl;
...
}
```

Estructuras como parámetros de funciones:

Las estructuras también pueden ser pasadas por valor y por referencia.

Las reglas se les aplican igual que a los tipos fundamentales: las estructuras pasadas por valor no conservarán sus cambios al retornar de la función. Las estructuras pasadas por referencia conservarán los cambios que se les hagan al retornar de la función.

CAPITULO 17 Más librerías estándar: string.h

Librería rutinas de manipulación de cadenas "string.h"

En esta librería se incluyen rutinas de manipulación de cadenas de caracteres y de memoria. De momento veremos sólo algunas de las que se refieren a cadenas.

Función "strlen()"

Calcula la longitud de una cadena.

Sintaxis:

size_t strlen(const char *s);

"strlen" calcula la longitud de la cadena s.

Valor de retorno:

"strlen" devuelve el número de caracteres que hay en s, excluyendo el carácter nulo de terminación de cadena.

Ejemplo:

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

int main() {
   char *cadena = "Una cadena C++ termina con cero";

   cout << "La cadena: [" << cadena << "] tiene " < strlen(cadena) <<
" caracteres" << endl;
   return 0;
}
```

Función "strcpy()"

Copia una cadena en otra.

Sintaxis:

char *strcpy(char *dest, const char *orig);

Copia la cadena orig a dest, la copia de caracteres se detendrá cuando sea copiado el carácter nulo.

Valor de retorno:

"strcpy" devuelve el puntero dest.

Ejemplo:

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

int main() {
   char *cadena = "Cadena ejemplo";
   char cad[32];

   cout << strcpy(cad, cadena) << endl;
   cout << cad << endl;
   return 0;
}
```

Función "strcmp()"

Compara dos cadenas.

Sintaxis:

int strcmp(char *cad1, const char *cad2);

Compara las dos cadenas, si la cad1 es mayor que cad2 el resultado será mayor de 0, si cad1 es menor que cad2, el resultado será menor de 0, si son iguales, el resultado será 0.

La comparación se realiza carácter a carácter. Mientras los caracteres comparados sean iguales, se continúa con el siguiente carácter. Cuando se encuentran caracteres distintos, aquél que tenga un código ASCII menor pertenecerá a la cadena menor. Por supuesto, si las cadenas son iguales hasta que una de ellas se acaba, la más corta es la menor.

Ejemplo:

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

int main() {
   char *cadena1 = "Cadena ejemplo 1";
   char *cadena2 = "Cadena ejemplo 2";
   char *cadena3 = "Cadena";
   char *cadena4 = "Cadena";

   if(strcmp(cadena1, cadena2) < 0)
      cout << cadena1 << " es menor que " << cadena2 << endl;
   else if(strcmp(cadena1, cadena2) > 0)
           cout << cadena1 << " es menor que " << cadena2 << endl;
        else
           cout << cadena1 << " es igual que " << cadena2 << endl;
   cout << strcmp(cadena3, cadena2) << endl;
   cout << strcmp(cadena3, cadena4) << endl;
   return 0;
}
```

Función "strcat()"

Añade o concatena una cadena a otra.

Sintaxis:

char *strcat(char *dest, const char *orig);

"strcat" añade una copia de orig al final de dest. La longitud de la cadena resultante será strlen(dest) + strlen(orig).

Valor de retorno:

"strcat" devuelve un puntero a la cadena concatenada.

Ejemplo:

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

int main() {
   char *cadena1 = "Cadena de";
   char *cadena2 = " ejemplo";
   char cadena3[126];

   strcpy(cadena3, cadena1);
   cout << strcat(cadena3, cadena2) << endl;
   return 0;
}
```

Función "strncpy()"

Copia un determinado número de caracteres de una cadena en otra.

Sintaxis:

char *strncpy(char *dest, const char *orig, size_t maxlong);

Copia maxlong caracteres de la cadena orig a dest, si hay más caracteres se ignoran, si hay menos se rellenará con caracteres nulos. La cadena dest no se terminará con nulo si la longitud de orig es maxlong o más.

Valor de retorno:

"strncpy" devuelve el puntero dest.

Ejemplo:

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

int main() {
   char *cadena = "Cadena ejemplo";
   char cad[32];
```

```
   strncpy(cad, cadena, 4);
   cad[4] = '\0';
   cout << cad << endl;
   return 0;
}
```

Función "strncmp()"

Compara dos porciones de cadenas.

Sintaxis:

int strncmp(char *cad1, const char *cad2, size_t maxlong);

Compara las dos cadenas igual que strcmp, pero sólo se comparan los primeros maxlong caracteres.

Ejemplo:

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

int main() {
   char *cadena1 = "Cadena ejemplo 1";
   char *cadena2 = "Cadena ejemplo 2";
   char *cadena3 = "Cadena";
   char *cadena4 = "Cadena";

   if(strncmp(cadena1, cadena2, 6) < 0)
      cout << cadena1 << " es menor que " << cadena2 << endl;
   else if(strncmp(cadena1, cadena2, 6) > 0)
           cout << cadena1 << " es menor que " << cadena2 << endl;
        else
           cout << cadena1 << " es igual que " << cadena2 << endl;
   cout << strncmp(cadena3, cadena2, 5) << endl;
   cout << strncmp(cadena3, cadena4, 4) << endl;
   return 0;
}
```

Función "strncat()"

Añade o concatena una porción de una cadena a otra.

Sintaxis:

char *strncat(char *dest, const char *orig, size_t maxlong);

"strncat" añade como máximo maxlong caracteres de la cadena orig al final de dest, y después añade el carácter nulo. La longitud de la cadena resultante será strlen(dest) + maxlong.

Valor de retorno:

"strncat" devuelve un puntero a la cadena concatenada.

Ejemplo:

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

int main() {
   char *cadena1 = "Cadena de";
   char *cadena2 = " ejemplo";
   char cadena3[126];

   strcpy(cadena3, cadena1);
   cout << strcat(cadena3, cadena2, 5) << endl;
   return 0;
}
```

Función "strtok()"

Busca dentro de una cadena conjuntos de caracteres o símbolos (tokens) separados por delimitadores.

Sintaxis:

char *strtok(char *s1, const char *s2);

"strtok" considera la cadena s1 como una lista de símbolos separados por delimitadores de la forma de s2.

La primera llamada a "strtok" devuelve un puntero al primer caracter del primer símbolo de s1 e inserta un carácter nulo a continuación del símbolo retornado. Las siguientes llamadas, especificando null como primer argumento, siguen dando símbolos hasta que no quede ninguno.

El separador, s2, puede ser diferente para cada llamada.

Valor de retorno:

"strtok" devuelve un puntero al símbolo extraido, o NULL cuando no quedan símbolos.

Ejemplo:

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

int main(void)
{
   char entrada[32] = "abc,d,efde,ew,231";
   char *p;

   // La primera llamada con entrada
   p = strtok(entrada, ",");
   if(p) cout << p << endl;

   // Las siguientes llamadas con NULL
   while(p) {
      p = strtok(NULL, ",");
```

```
      if(p) cout << p << endl;
   }
   return 0;
}
```

CAPITULO 18 Estructuras II: Uniones

Las uniones son un tipo especial de estructuras que permiten almacenar elementos de diferentes tipos en las mismas posiciones de memoria, aunque evidentemente no simultáneamente.

Sintaxis:

union [<tipo unión>] {

[<tipo> <nombre de variable>[, <nombre variable>, ...]] ;

} [<variable de unión>[,<variable union>...]] ;

El nombre de la unión es un nombre opcional para referirse a la unión.

Las variables de unión son variables declaradas del tipo de la unión, y su inclusión también es opcional.

Sin embargo, al menos uno de estos elementos debe existir, aunque ambos sean opcionales.

En el interior de una unión, entre las llaves, se pueden definir todos los elementos necesarios, del mismo modo que se declaran las variables. La particularidad es que cada elemento comenzará en la misma posición de memoria.

Las uniones pueden referenciarse completas, usando su nombre, como hacíamos con las estructuras, y también se puede acceder a los elementos en el interior de la unión usando el operador de selección (.), un punto.

También pueden declararse más variables del tipo de la unión en cualquier parte del programa, de la siguiente forma:

[union] <nombre de la unión> <variable de unión>[,<variable unión>...];

La palabra "union" es opcional en la declaración de variables.

Ejemplo:

```
#include <iostream.h>

union unEjemplo {
   int A;
   char B;
   double C;
} UnionEjemplo;

int main() {
   UnionEjemplo.A = 100;
   cout << UnionEjemplo.A << endl;
```

```
   UnionEjemplo.B = 'a';
   cout << UnionEjemplo.B << endl;
   UnionEjemplo.C = 10.32;
   cout << UnionEjemplo.C << endl;
   cout << &UnionEjemplo.A << endl;
   cout << (void*)&UnionEjemplo.B << endl;
   cout << &UnionEjemplo.C << endl;
   cout << sizeof(unEjemplo) << endl;
   cout << sizeof(unEjemplo::A) << endl;
   cout << sizeof(unEjemplo::B) << endl;
   cout << sizeof(unEjemplo::C) << endl;
   return 0;
}
```

Suponiendo que int ocupa dos bytes, char un byte y double 4 bytes, la forma en que se almacena la información en la unión del ejemplo es la siguiente:

```
[BYTE1][BYTE2][BYTE3][BYTE4]
[<----A----->]
[<-B->]
[<-----------C------------>]
```

Por el contrario, la misma disposición de variables en una estructura tendría la siguiente disposición:

```
[BYTE1][BYTE2][BYTE3][BYTE4][BYTE5][BYTE6][BYTE7]
[<----A----->][<-B->][<-----------C------------>]
```

Unas notas sobre el ejemplo:

- Observa que hemos hecho un "casting" del puntero al elemento B de la unión. Si no lo hicieramos así cout encontraría un puntero a char, que se considera como una cadena, y por defecto intentaría imprimir la cadena, pero nosotros queremos imprimir el puntero, así que lo convertimos a un puntero de otro tipo.
- Para averiguar el tamaño de cada campo usando "sizeof" tenemos que usar el operador de ámbito (::), y no el punto.
- Observa que el tamaño de la unión es el del elemento más grande.

Otro ejemplo, éste más práctico. Algunas veces tenemos estructuras que son elementos del mismo tipo, por ejemplo X, Y, y Z todos enteros. Pero en determinadas circunstancias, puede convenirnos acceder a ellos como si fueran un array: Coor[0], Coor[1] y Coor[2]. En este caso, la unión puede ser útil:

```
struct stCoor3D {
   int X, Y, Z;
};

union unCoor3D {
   struct stCoor3D N;
   int Coor[3];
} Punto;
```

Podemos referirnos a la coordenada Y de estas dos formas:

```
Punto.N.Y
```

```
Punto.Coor[1]
```

Estructuras anónimas:

Una estructura anónima es la que carece de indentificador de tipo de estructura y de declaración de variables del tipo de estructura.

Por ejemplo, la misma unión del último ejemplo puede declararse de este otro modo:

```
union unCoor3D {
   struct {
      int X, Y, Z;
   };
   int Coor[3];
} Punto;
```

Haciéndolo así accedemos a la coordenada Y de cualquiera de estas dos formas:

```
Punto.Y
Punto.Coor[1]
```

El método usado para declarar la estructura dentro de la unión es la forma anónima, como verás no tiene identificador de tipo de estructura ni de variables. El único lugar donde es legal el uso de estructuras anónimas es en el interior de estructuras y uniones.

CAPITULO 19 Punteros II: Arrays dinámicos

Ya hemos visto que los arrays pueden ser una potente herramienta para el almacenamiento y tratamiento de información, pero tienen un inconveniente: hay que definir su tamaño durante el diseño del programa, y después no puede ser modificado.

La gran similitud de comportamiento de los punteros y los arrays nos permiten crear arrays durante la ejecución, y en este caso además el tamaño puede ser variable.

Para ello se usan los punteros a punteros, y los arrays construidos de este modo se denominan arrays dinámicos.

Veamos la declaración de un puntero a puntero:

```
int **tabla;
```

"tabla" es un puntero que apunta a una variable de tipo puntero a int.

Sabemos que un puntero se comporta casi igual que un array, por lo tanto nada nos impide que "tabla" apunte al primer elemento de un array de punteros:

```
int n = 134;
tabla = new (int *)[n];
```

Ahora estamos en un caso similar, "tabla" apunta a un array de punteros a int, cada elemento de este array puede ser a su vez un puntero al primer elemento de otro array:

```
int m = 231;
for(int i = 0; i < n; i++)
   tabla[i] = new int[m];
```

Ahora tabla apunta a un array de dos dimensiones de n * m, podemos acceder a cada elemento igual que accedemos a los elementos de los arrays normales:

```
tabla[21][33] = 123;
```

Antes de abandonar el programa hay que liberar la memoria dinámica usada, primero la asociada a cada uno de los punteros de "tabla[i]":

```
for(int i = 0; i < n; i++) delete[] tabla[i];
```

Y después la del array de punteros a int, "tabla":

```
delete[] tabla;
```

Veamos el código de un ejemplo completo:

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

```
int main() {
   int **tabla;
   int n = 134;
   int m = 231;

   // Array de punteros a int:
   tabla = new (*int)[n];
   // n arrays de m ints
   for(int i = 0; i < n; i++)
      tabla[i] = new int[m];
   tabla[21][33] = 123;
   cout << tabla[21][33] << endl;
   // Liberar memoria:
   for(int i = 0; i < n; i++) delete[] tabla[i];
   delete[] tabla;
   return 0;
}
```

Pero no tenemos por qué limitarnos a arrays de dos dimensiones, con un puntero de este tipo:

int ***array;

Podemos crear un array dinámico de tres dimensiones, usando un método análogo.

Y generalizando, podemos crear arrays de cualquier dimensión.

Tampoco tenemos que limitarnos a arrays regulares.

Veamos un ejemplo de tabla triangular:

Crear una tabla para almacenar las distancias entre un conjunto de ciudades, igual que hacen los mapas de carreteras.

Para que sea más sencillo usaremos sólo cinco ciudades:

Ciudad A	0				
Ciudad B	154	0			
Ciudad C	254	354	0		
Ciudad D	54	125	152	0	
Ciudad E	452	133	232	110	0
Distancias	Ciudad A	Ciudad B	Ciudad C	Ciudad D	Ciudad E

Evidentemente, la distancia de la Ciudad A a la Ciudad B es la misma que la de la Ciudad B a la Ciudad A, así que no hace falta almacenar ambas. Igualmente, la distancia de una ciudad a sí misma es siempre 0, otro valor que no necesitamos.

Si tenemos n ciudades y usamos un array para almacenar las distancias necesitamos:

n*n = 5*5 = 25 casillas.

Sin embargo, si usamos un array triangular:

n*(n-1)/2 = 5*4/2 = 10 casillas.

Veamos cómo implemetar esta tabla:

```
#include <iostream.h>

#define NCIUDADES 5
#define CIUDAD_A 0
#define CIUDAD_B 1
#define CIUDAD_C 2
#define CIUDAD_D 3
#define CIUDAD_E 4

// Variable global para tabla de distancias:
int **tabla;
// Prototipo para  calcular la distancia entre dos ciudades:
int Distancia(int A, int B);

int main() {
   // Primer subíndice de A a D
   tabla = new (int *)[NCIUDADES-1];
   // Segundo subíndice de B a E,
   // define 4 arrays de 4, 3, 2 y 1 elemento:
   for(int i = 0; i < NCIUDADES-1; i++)
      tabla[i] = new int[NCIUDADES-1-i]; // 4, 3, 2, 1
   // Inicialización:
   tabla[CIUDAD_A][CIUDAD_B-CIUDAD_A-1] = 154;
   tabla[CIUDAD_A][CIUDAD_C-CIUDAD_A-1] = 245;
   tabla[CIUDAD_A][CIUDAD_D-CIUDAD_A-1] = 54;
   tabla[CIUDAD_A][CIUDAD_E-CIUDAD_A-1] = 452;
   tabla[CIUDAD_B][CIUDAD_C-CIUDAD_B-1] = 354;
   tabla[CIUDAD_B][CIUDAD_D-CIUDAD_B-1] = 125;
   tabla[CIUDAD_B][CIUDAD_E-CIUDAD_B-1] = 133;
   tabla[CIUDAD_C][CIUDAD_D-CIUDAD_C-1] = 152;
   tabla[CIUDAD_C][CIUDAD_E-CIUDAD_C-1] = 232;
   tabla[CIUDAD_D][CIUDAD_E-CIUDAD_D-1] = 110;

   // Ejemplos:
   cout << "Distancia A-D: " << Distancia(CIUDAD_A, CIUDAD_D) << endl;
   cout << "Distancia B-E: " << Distancia(CIUDAD_B, CIUDAD_E) << endl;
   cout << "Distancia D-A: " << Distancia(CIUDAD_D, CIUDAD_A) << endl;
   cout << "Distancia B-B: " << Distancia(CIUDAD_B, CIUDAD_B) << endl;
   cout << "Distancia E-D: " << Distancia(CIUDAD_E, CIUDAD_D) << endl;

   // Liberar memoria dinámica:
   for(int i = 0; i < NCIUDADES-1; i++) delete[] tabla[i];
   delete[] tabla;
   return 0;
}

int Distancia(int A, int B) {
   int aux;

   // Si ambos subíndices son iguales, volver con cero:
   if(A == B) return 0;
   // Si el subíndice A es mayor que B, intercambiarlos:
   if(A > B) {
      aux = A;
```

```
      A = B;
      B = aux;
   }
   return tabla[A][B-A-1];
}
```

Notas sobre el ejemplo:

Observa el modo en que se usa la directiva #define para declarar constantes.

Efectivamente, para este ejemplo se complica el acceso a los elementos de la tabla ya que tenemos que realizar operaciones para acceder a la segunda coordenada. Sin embargo piensa en el ahorro de memoria que supone cuando se usan muchas ciudades, por ejemplo, para 100 ciudades:

Tabla normal 100*100 = 10000 elementos.

Tabla triangular 100*99/2 = 4950 elementos.

Hemos declarado el puntero a tabla como global, de este modo será accesible desde main y desde Distancia. Si la hubiéramos declarado local en main, tendríamos que pasarla como parámetro a la función.

Observa el método usado para el intercambio de valores de dos variables. Si no se usa la variable "aux", no es posible intercambiar valores. Podríamos haber definido una función para esta acción, "Intercambio", pero lo dejaré como ejercicio.

Problema:

Usando como base el ejemplo anterior, separar dos nuevas funciones:

- PonerDistancia, asignar valores a distancias entre dos ciudades.
- Intercambio, intercambiar los contenidos de dos variables enteras.

CAPITULO 20 Operadores IV: Más operadores

Alguien dijo una vez que C prácticamente tiene un operador para cada instrucción de ensamblador. De hecho C y sobre todo C++ tiene una enorme riqueza de operadores, éste es el tercer capítulo dedicado a operadores, y aún nos quedan más operadores por ver.

Operadores de bits

Estos operadores trabajan con las expresiones que manejan manipulándolas a nivel de bit, y sólo se pueden aplicar a expresiones enteras. Existen seis operadores binarios: "&", "|", "^", "~", ">>" y "<<".

Sintaxis:

<expresión> & <expresión>

<expresión> ^ <expresión>

<expresión> | <expresión>

~<expresión>

<expresión> <<<expresión>

<expresión> >> <expresión>

El operador "&" corresponde a la operación lógica "AND", o en álgebra de Boole al operador "·", compara los bits uno a uno, si ambos son "1" el resultado es "1", en caso contrario "0".

El operador "^" corresponde a la operación lógica "OR exclusivo", compara los bits uno a uno, si ambos son "1" o ambos son "0", el resultado es "1", en caso contrario "0".

El operador "|" corresponde a la operación lógica "OR", o en álgebra de Bool al operador "+", compara los bits uno a uno, si uno de ellos es "1" el resultado es "1", en caso contrario "0".

El operador "~", (se obtiene con la combinación de teclas ALT+126, manteniendo pulsada la tecla "ALT", se pulsan las teclas "1", "2" y "6" del teclado numérico), corresponde a la operación lógica "NOT", se trata de un operador unitario que invierte el valor de cada bit, si es "1" da como resultado un "0", y si es "0", un "1".

El operador "<<" realiza un desplazamiento de bits a la izquierda del valor de la izquierda, introduciendo "0" por la derecha, tantas veces como indique el segundo operador; equivale a multiplicar por 2 tantas veces como indique el segundo operando.

El operador ">>" realiza un desplazamiento de bits a la derecha del valor de la izquierda, introduciendo "0" por la izquierda, tantas veces como indique el segundo operador; equivale a dividir por 2 tantas veces como indique el segundo operando.

Tablas de verdad:

Operador 1	Operador 2	AND	OR exclusivo	OR inclusivo	NOT
E1	E2	E1 & E2	E1 ^ E2	E1 | E2	~E1
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	0	1	0

Ya hemos visto que los operadores '~', '&', '<<' y '>>' tienen otras aplicaciones diferentes, su funcionamiento es contextual, es decir, se decide después del análisis de los operandos. En C++ se conoce esta reutilización de operadores como sobrecarga de operadores o simplemente sobrecarga, más adelante introduciremos un capítulo dedicado a este tema.

Ejemplos:

Espero que estés familiarizado con la numeración hexadecimal, ya que es vital para interpretar las operaciones con bits.

```
int a, b, c;

a = 0xd3; // = 11010011
b = 0xf5; // = 11110101
c = 0x1e; // = 00011110

d = a | b; // 11010011 | 11110101 = 11110111 -> 0xf7
d = b & c; // 11110101 & 00011110 = 00010100 -> 0x14
d = a ^ c; // 11010011 ^ 00011110 = 11001101 -> 0xcd
d = ~c;    // ~00011110 = 11100001 -> 0xe1
d = c << 3 // 00011110 << 3 = 11110000 -> 0xf0
d = a >> 4 // 11010011 >> 4 = 00001101 -> 0x0d
```

Operador condicional

El operador "?:", se trata de un operador ternario.

Sintaxis:

<expresión lógica> ? <expresión> : <expresión>

En la expresión E1 ? E2 : E3, primero se evalúa la expresión E1, si el valor es verdadero ("true"), se evaluará la expresión E2 y E3 será ignorada, si es falso ("false"), se evaluará E3 y E2 será ignorada.

Hay ciertas limitaciones en cuanto al tipo de los argumentos.

- E1 debe ser una expresión lógica.

E2 y E3 han de seguir una de las siguientes reglas:

- Ambas de tipo aritmético.
- Ambas de estructuras o uniones compatibles.
- Ambas de tipo "void".

Hay más reglas, pero las veremos más adelante, ya que aún no hemos visto nada sobre los conocimientos implicados.

Como ejemplo veremos cómo se define la macro "max" en la librería "stdlib":

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

De este ejemplo sólo nos interesa la parte de la derecha. La interpretación es: si "a" es mayor que "b", se debe evaluar "a", en caso contrario evaluar "b".

CAPITULO 21 Definición de tipos, tipos derivados

En ocasiones puede ser útil definir nombres para tipos de datos, 'alias' que nos hagan más fácil declarar variables y parámetros, o que faciliten la portabilidad de nuestros programas.

Para ello C y C++ disponen de la palabra clave "typedef".

Sintaxis:

typedef <tipo> <identificador>;

<tipo> puede ser cualquier tipo C++, fundamental o derivado.

Ejemplos:

```
typedef unsigned int UINT;
```

UINT será un tipo válido para la declaración de variables o parámetros, y además será equivalente a un entero sin signo.

```
typedef unsigned char BYTE;
```

BYTE será un tipo equivalente a ocho bits.

```
typedef unsigned short int WORD;
```

WORD será un tipo equivalente a dieciséis bits. Este último es un caso de dependencia de la plataforma, si WORD debe ser siempre una palabra de dieciséis bits, independientemente de la plataforma, deberemos cambiar su definición dependiendo de ésta. En algunas plataformas podrá definirse como:

```
typedef unsigned int WORD;
```

y en otras como:

```
typedef unsigned long int WORD;
```

Declarar un tipo para WORD en un fichero de cabecera, nos permitirá adaptar fácilmente la aplicación a distintas plataformas.

```
typedef struct stpunto tipoPunto;
```

Define un nuevo tipo como una estructura stpunto.

```
typedef int (*PFI)();
```

Define PFI como un puntero a una función que devuelve un puntero.

```
PFI f;
```

Declaración de una variable f que es un puntero a una función que devuelve un entero.

CAPITULO 22 Funciones III

Aún quedan algunas cosas interesantes por decir sobre las funciones en C++.

Parámetros con valores por defecto

Algunas veces nos puede interesar que ciertos parámetros que necesita una función no sea necesario proporcionarlos siempre. Esto suele suceder cuando esos parámetros casi siempre se usan con un mismo valor. En C++, cuando declaramos una función podemos decidir que algunos sus parámetros sean opcionales. En ese caso tendremos que asignales valores por defecto.

Cuando se llama a la función incluyendo valores para los parámetros opcionales funcionará como cualquiera de las funciones que hemos usado hasta ahora, pero si se omiten todos o algunos de estos parámetros la función trabajará con los valores por defecto que hemos definido.

Por ejemplo:

```
#include <iostream.h>

void funcion(int a = 1);

int main() {
   funcion(19);
   funcion();
}

void funcion(int a) {
   cout << a << endl;
}
```

La primera llamada a "funcion" dará como salida 19, que es el parámetro que le damos explícitamente. La segunda llamada dará como salida 1, que es el valor por defecto.

Sin embargo este método tiene algunas limitaciones:

- Sólo los últimos argumentos de las funciones pueden tener valores por defecto.
- De estos, sólo los últimos argumentos pueden ser omitidos en una llamada.
- Si se declaran prototipos de las funciones, los valores por defecto deben especidicarse en ellos.

Por ejemplo:

```
void funcion1(int a, int b=0, int c= 1); // Legal
void funcion2(int a=1, int b, int c); // Ilegal
```

Los argumentos por defecto empiezan a asignarse empezando por el último.

```
int funcion1(int a, int b=0, int c=1);
...
funcion1(12, 10); // Legal, el valor para "c" será 1
```

```
funcion1(12); // Legal, los valores para "b" y "c" serán 0 y 1
funcion1(); // Ilegal, el valor para "a" es obligatorio
```

Funciones con número de argumentos variable

También es posible crear funciones con un número indeterminado de argumentos, para ello la declararemos los argumentos conocidos del modo tradicional, de éste tipo debe existir al menos uno, y los desconocidos se sustituyen por tres puntos (…), del siguiente modo:

int funcion2(int a, float b, …);

Los parámetros se pasan usando la pila, (esto es siempre así, pero normalmente no tendremos que prestar artención a éste hecho). Además es el programador el responsable de decidir el tipo de cada argumento, lo cual limita bastante el uso de esta forma de pasar parámetros.

Para hacer más fácil la vida de los programadores, se incluyen algunas macros en el fichero de cabecera "stdarg.h", estas macros permiten manejar "fácilmente" las listas de argumentos desconocidos.

Tipos:

En el fichero de cabecera "stdarg.h" de define un tipo: va_list.

Será necesario declarar una variable de este tipo para tener acceso a la lista de parámetros.

Macros:

También se definen tres macros: va_start, va_arg y va_end.

void va_start(va_list ap, ultimo);

Ajusta el valor de "ap" para que apunte al primer parámetro de la lista. <ultimo> es el identificador del último parámetro fijo antes de comenzar la lista.

tipo va_arg(va_list ap, tipo);

Devuelve el siguiente valor de la lista de parámetros, "ap" debe ser la misma variable que se actualizó previamente con "va_start", "tipo" es el tipo del parámetro que se tomará de la lista.

void va_end(va_list va);

Permite a la función retornar normalmente, restaurando el estado de la pila, esto es necesario porque algunas de las macros anteriores pueden modificarla, haciendo que el programa termine anormalmente.

Uso de las macros para leer la lista de parámetros:

```
funcion(<tipo> <id1> [, <tipo> <id2>...], ...)
{
   va_list ar; // Declarar una variable para manejar la lista

   va_start(ar, <idn>); // <idn> debe ser el nombre del último
parámetro antes de ...
   <tipo> <arg>; // <arg> es una variable para recoger un parámetro
   while((<arg> = va_arg(ar, <tipo>)) != 0) {// <tipo> debe ser el
mismo que es de <arg>
      // Manejar arg>
   }
   va_end(ar); // Normalizar la pila
}
```

Hay que usar un sistema que permita determinar cúal es el último valor de la lista de parámetros.

Una forma que el último valor de la lista de parámetros en la llamada a la función sea un 0, (o más en general, un valor conocido).

También puede hacerse que uno de los parámetros conocidos sea la cuenta de los parámetros desconocidos.

Además es necesario que el programador conozca el tipo de cada parámetro, para así poder leerlos adecuadamente, lo normal es que todos los parámetros sean del mismo tipo, o que se use un mecanismo como la de "printf", donde analizando el primer parámetro se pueden deducir el tipo de todos los demás. Este último sistema también sirve para saber el número de parámetros.

Ejemplos:

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

void funcion(int a, ...);

int main() {
   funcion(1, "cadena 1", 0);
   funcion(1, "cadena 1", "cadena 2", "cadena 3", 0);
   funcion(1, 0);
   return 0;
}

void funcion(int a, ...) {
   va_list p;
   va_start(p, a);
   char *arg;

   while ((arg = va_arg(p, char*))) {
      cout << arg << " ";
   }
   va_end(p);
   cout << endl;
}
```

Otro Ejemplo, este usando un sistema análogo al de "printf":

```
#include <iostream.h>
#include <string.h>
#include <stdarg.h>

void funcion(char *formato, ...);

int main() {
   funcion("ciic", "Hola", 12, 34, "Adios");
   funcion("ccci", "Uno", "Dos", "Tres", 4);
   funcion("i", 1);
   return 0;
}

void funcion(char *formato, ...)
{
   va_list p;
   char *szarg;
   int iarg;
   int i;

   va_start(p, formato);
   /* analizamos la cadena de formato para saber el número y tipo de
los parámetros*/
   for(i = 0; i < strlen(formato); i++)
   {
      switch(formato[i]) {
         case 'c': /* Cadena de caracteres */
            szarg = va_arg(p, char*);
            cout << szarg << " ";
            break;
         case 'i': /* Entero */
            iarg = va_arg(p, int);
            cout << iarg << " ";
            break;
      }
   }
   va_end(p);
   cout << endl;
}
```

Argumentos de main.

Muy a menudo necesitamos especificar valores u opciones a nuestros programas cuando los ejecutamos desde la línea de comandos.

Por ejemplo, si hacemos un programa que copie ficheros, del tipo del "copy" de MS-DOS, necesitaremos especificar el nombre del archivo de origen y el de destino.

Hasta ahora siempre hemos usado la función "main" sin parámetros, sin embargo, como veremos ahora, se puedem pasar argumentos a nuestros programas a través de los parámetros de la función main.

Para tener acceso a los argumentos de la línea de comandos hay que declararlos en la función "main", la manera de hacerlo puede ser una de las siguientes:

int main(int argc, char *argv[]);

int main(int argc, char **argv);

Que como sabemos son equivalentes.

El primer parámetro, "argc", es el número de argumentos que se han especificado en la línea de comandos. El segundo, "argv", es un array de cadenas que contiene los argumentos especificados en la línea de comandos.

Por ejemplo, si nuestro programa se llama "programa", y lo ejecutamos con la siguiente línea de comandos:

programa arg1 arg2 arg3 arg4

argc valdrá 5, ya que el nombre del programa también se cuenta como un argumento.

argv[] contendrá la siguiente lista: "programa", "arg1", "arg2", "arg3" y "arg4".

Ejemplo:

```
#include <iostream.h>

int main(int argc, char **argv)
{
   for(int i = 0; i < argc; i++)
      cout << argv[i] << " ";
   cout << endl;
}
```

Funciones inline

Cuando escribimos el nombre de una función dentro de un programa decimos que "llamamos" a esa función. Esto quiere decir que lo que hace el programa es "saltar" a la función, ejecutarla y retornar al punto en que fue llamada.

Esto es cierto para las funciones que hemos usado hasta ahora, pero hay un tipo especial de funciones que funcionan de otro modo. En lugar de existir una sola copia de la función dentro del código, cuando se declara una función como "inline" lo que se hace es sustituir su código en el lugar de la llamada.

Sintaxis:

inline <tipo> <nombre_de_funcion>([<tipo> <identificador>[,<tipo> <identificador>,...]);

Esto tiene la ventaja de que la ejecución es más rápida, pero por contra, el programa generado es más grande. Se debe evitar el uso de funciones "inline", sobre todo cuando éstas son de gran tamaño, aunque con funciones pequeñas puede resultar práctico. Su uso es frecuente cuando las funciones tienen código en ensamblador, ya que en estos casos la optimización es mucho mayor.

En algunos casos, si la función es demasiado larga, el compilador puede decidir no insertar la función, sino simplemente llamarla. El uso de "inline" no es por lo tanto una obligación para el compilador, sino simplemente una recomendación.

Aparentemente, una función "inline" se comportará como cualquier otra función.

Nota: "inline" es exclusivo de C++, y no está disponible en C.

Ejemplos:

```
#include <iostream.h>

inline int mayor(int a, int b) {
   if(a > b) return a;
   else return b;
}

int main() {
   cout << "El mayor de 12,32 es " << mayor(12,32) << endl;
   cout << "El mayor de 6,21 es " << mayor(6,21) << endl;
   cout << "El mayor de 14,34 es " << mayor(14,34) << endl;
   return 0;
}
```

CAPITULO 23 Funciones IV: Sobrecarga

Anteriormente hemos visto operadores que tienen varios usos, como por ejemplo *, &, << o>>. Esto es lo que se conoce en C++ como sobrecarga de operadores. Con las funciones existe un mecanismo análogo, de hecho, en C++, los operadores no son sino un tipo especial de funciones, aunque eso sí, algo peculiares.

Así que podemos definir varias funciones con el mismo nombre, con la única condición de que el número y/o el tipo de los parámetros sean distintos. El compilador decide cual de las versiones de la función usará después de analizar el número y el tipo de los parámetros. Si ninguna de las funciones se adapta a los parámetros indicados, se aplicarán las reglas implícitas de conversión de tipos.

Las ventajas son más evidentes cuando debemos hacer las mismas operaciones con objetos de diferentes tipos o con distinto número de objetos. Hasta ahora habíamos usado macros para esto, pero no siempre es posible usarlas, y además las macros tienen la desventaja de que se expanden siempre, y son difíciles de diseñar para funciones complejas. Sin embargo las funciones serán ejecutadas mediante llamadas, y por lo tanto sólo habrá una copia de cada una.

Esta propiedad sólo existe en C++, no en C.

Ejemplo:

```
#include <iostream.h>

int mayor(int a, int b);
char mayor(char a, char b);
double mayor(double a, double b);

int main()
{
   cout << mayor('a', 'f') << endl;
   cout << mayor(15, 35) << endl;
   cout << mayor(10.254, 12.452) << endl;
   return 0;
}

int mayor(int a, int b)
{
   if(a > b) return a; else return b;
}

char mayor(char a, char b)
{
   if(a > b) return a; else return b;
}

double mayor(double a, double b)
{
   if(a > b) return a; else return b;
```

```
}
```

Otro ejemplo:

```
#include <iostream.h>

int mayor(int a, int b);
int mayor(int a, int b, int c);
int mayor(int a, int b, int c, int d);

int main()
{
   cout << mayor(10, 4) << endl;
   cout << mayor(15, 35, 23) << endl;
   cout << mayor(10, 12, 12, 18) << endl;
   return 0;
}

int mayor(int a, int b)
{
   if(a > b) return a; else return b;
}

int mayor(int a, int b, int c)
{
   return mayor(mayor(a, b), c);
}

int mayor(int a, int b, int c, int d)
{
   return mayor(mayor(a, b), mayor(c, d));
}
```

El primer ejemplo ilustra el uso de sobrecarga de funciones para operar con objetos de distinto tipo. El segundo muestra cómo se puede sobrecargar una función para operar con distinto número de objetos. Por supuesto, el segundo ejemplo se puede resolver también con parámetros por defecto.

Ejercicio:

Propongo un ejercicio: implementar este segundo ejemplo usando parámetros por defecto. Para que sea más fácil, hacerlo sólo para parámetros con valores positivos, y si te sientes valiente, hazlo también para cualquier tipo de valor.

CAPITULO 24 Operadores V: Operadores sobrecargados

Análogamente a las funciones sobrecargadas, los operadores también pueden sobrecargarse.

En realidad la mayoría de los operadores en C++ están sobrecargados. Por ejemplo el operador + realiza distintas acciones cuando los operandos son enteros, o en coma flotante. En otros casos esto es más evidente, por ejemplo el operador * se puede usar como operador de multiplicación o como operador de indirección.

C++ permite al programador sobrecargar a su vez los operadores para sus propios usos.

Sintaxis:

Prototipo:

<tipo> operator <operador> (<argumentos>);

Definición:

<tipo> operator <operador> (<argumentos>) {

<sentencias>;

}

Se pueden sobrecargar todos los operadores excepto ".", ".*", "::" y "?:".

Los operadores "=", "[]", "->", "()", "new" y "delete", sólo pueden ser sobrecargados cuando se definen como miembros de una clase.

Los argumentos deben ser tipos enumerados o estructurados: struct, union o class.

Ejemplo:

```
#include <iostream.h>

struct complejo {
   float a,b;
};

/* Prototipo del operador + para complejos */
complejo operator +(complejo a, complejo b);

int main()
{
   complejo x = {10,32};
   complejo y = {21,12};

   complejo z;
```

```
   z = x + y; /* Uso del operador sobrecargado + con complejos */
   cout << z.a << "," << z.b << endl;
   return 0;
}

/* Definición del operador + para complejos */
complejo operator +(complejo a, complejo b)
{
   complejo temp = {a.a+b.a, a.b+b.b};
   return temp;
}
```

Al igual que con las funciones sobrecargadas, la versión del operador que see usará se decide después del análisis de los argumentos.

También es posible usar los operadores en su notación funcional:

```
z = operator+(x,y);
```

Pero donde veremos mejor toda la potencia de los operadores sobrecargados será cuando estudiemos las clases.

CAPITULO 25 El preprocesador

El preprocesador analiza el fichero fuente antes de la fase de compilación real, y realiza las sustituciones de macros y procesa las directivas del preprocesador. También se eliminan los comentarios.

Una directiva de preprocesador es una línea cuyo primer carácter es un #.

A continuación se describen las directivas del preprocesador, aunque algunas ya las hemos visto antes.

Directiva #define:

La directiva "#define", sirve para definir macros. Esto suministra un sistema para la sustitución de palabras, con y sin parámetros.

Sintaxis:

#define identificador_de_macro

El preprocesador sustituirá cada ocurrencia del identificador_de_macro en el fichero fuente, por la secuencia con algunas excepciones. Cada sustitución se conoce como una expansión de la macro. La secuencia es llamada a menudo cuerpo de la macro.

Si la secuencia no existe, el identificador_de_macro será eliminado cada vez que aparezca en el fichero fuente.

Después de cada expansión individual, se vuelve a examinar el texto expandido a la búsqueda de nuevas macros, que serán expandidas a su vez. Esto permite la posibilidad de hacer macros anidadas. Si la nueva expansión tiene la forma de una directiva de preprocesador, no será reconocida como tal.

Existen otras restricciones a la expansión de macros:

Las ocurrencias de macros dentro de literales, cadenas, constantes alfanuméricas o comentarios no serán expandidas.

Una macro no será expandida durante su propia expansión, así #define A A, no será expandida indefinidamente.

No es necesario añadir un punto y coma para terminar una directiva de preprocesador. Cualquier carácter que se encuentre en una secuencia de macro, incluido el punto y coma, aparecerá en la expansión de la macro. La secuencia termina en el primer retorno de línea encontrado. Las secuencias de espacios o comentarios en la secuencia, se expandirán como un único espacio.

Directiva #undef:

Sirve para eliminar definiciones de macros previamente definidas. La definición de la macro se olvida y el identificador que indefinido.

Sintaxis:

#undef identificador_de_macro

La definición es una propiedad importante de un identificador. Las directivas condicionales #ifdef e #ifndef se basan precisamente en esta propiedad de los identificadores. Esto ofrece un mecanismo muy potente para controlar muchos aspectos de la compilación.

Después de que una macro quede indefinida puede ser definida de nuevo con #define, usando la misma u otra definición.

Si se intenta definir un identificador de macro que ya esté definido, se producirá un aviso, un warning, si la definición no es exactamente la misma. Es preferible usar un mecanismo como este para detectar macros existentes:

```
#ifndef NULL
  #define NULL 0L
#endif
```

De éste modo, la línea del #define se ignorará si el símbolo NULL ya está definido.

Directivas #if, #elif, #else y #endif

Permiten hacer una compilación condicional de un conjunto de líneas de código.

Sintaxis:

```
#if expresión-constante-1
<sección-1>
#elif <expresión-constante-2>
<sección-2>
.
.
.
#elif <expresión-constante-n>
<sección-n>
<#else>
<sección-final>
#endif
```

Todas las directivas condicionales deben completarse dentro del mismo fichero. Sólo se compilarán las líneas que estén dentro de las secciones que cumplan la condición de la expresión constante correspondiente.

Estas directivas funcionan de modo similar a los operadores condicionales C. Si el resultado de evaluar la expresión constante 1, que puede ser una macro, es distinto de cero (true), las líneas representadas por sección-1, ya sean líneas de comandos, macros o incluso nada, serán compiladas. En caso contrario, si el resultado de la evaluación de la

expresión constante 1, es cero (false), la sección-1 será ignorada, no se expandirán macros ni se compilará.

En el caso de ser distinto de cero, después de que la sección-1 sea preprocesada, el control pasa al #endif correspondiente, con lo que termina la secuencia condicional. En el caso de ser cero, el control pasa al siguiente línea #elif, si existe, donde se evaluará la expresión constante 2. Si el resultado es distinto de cero, se procesrá la sección-2, y después el control pasa al correspondiente #endif. Por el contrario, si el resultaado de la expresión constante 2 es cero, el control pasa al siguiente #elif, y así sucesivamente, hasta que se encuentre un #else o un #endif. El #else, que es opcional, se usa como una condición alternativa para el caso en que todas la condiciones anteriores resulten falsas. El #endif termina la secuencia condicional.

Cada sección procesada puede contener a su vez directivas condicionales, anidadas hasta cualquier nivel, cada #if debe corresponderse con el #endif más cercano.

El objetivo de una red de este tipo es que sólo una sección, inclusive vacía, sea compilada. Las secciones ignoradas sólo son relevantes para evaluar las condiciones anidadas, es decir asociar cada #if con su #endif.

Las expresiones constantes deben poder ser evaluadas como valores enteros.

Directivas #ifdef e #ifndef:

Estas directivas permiten comprobar si un identificador está o no actualmente definido, es decir, si un #define ha sido previamente procesado para el identificador y si sigue definido.

Sintaxis:

#ifdef identifier

#ifndef identifier

La línea:

```
#ifdef identificador
```

tiene exactamente el mismo efecto que

```
#if 1
```

si el identificador está actualmente definido, y el mismo efecto que

```
#if 0
```

si el identificador no está definido.

#ifndef compueba la no definición de un identificador, así la línea

```
#ifndef identificador
```

tiene el mismo efecto que

```
#if 0
```

si el identificador está definido, y el mismo efecto que

```
#if 1
```

si el identificador no está definido.

Por lo demás, la sintaxis es la misma que para #if, #elif, #else, y #endif.

Un identificador definido como nulo, se considera definido.

Directiva #error:

Esta directiva se suele incluir en sentencias condicionales de preprocesador para detectar condiciones no deseadas durante la compilación. En un funcionamiento normal estas condiciones serán falsas, pero cuando la condición es verdadera, es preferible que el compilador muestre un mensaje de error y detenga la fase de compilación. Para hacer esto se debe introducir esta directiva en una sentencia condicional que detecte el caso no deseado.

Sintaxis:

#error mensaje_de_error

Esta directiva genera el mensaje:

Error: nombre_de_fichero nº_línea : Error directive: mensaje_de_error

Directiva #include:

La directiva "#include", como ya hemos visto, sirve para insertar ficheros externos dentro de nuestro fichero de código fuente. Estos ficheros son conocidos como ficheros incluidos, ficheros de cabecera o "headers".

Sintaxis:

#include <nombre de fichero cabecera>

#include "nombre de fichero de cabecera"

#include identificador_de_macro

El preprocesador elimina la línea "#include" y, conceptualmente, la sustituye por el fichero especificado. El tercer caso haya el nombre del fichero como resultado de aplicar la macro.

El código fuente en si no cambia, pero el compilador "ve" el fichero incluido. El emplazamiento del #include puede influir sobre el ámbito y la duración de cualquiera de los identificadores en el interior del fichero incluido.

La diferencia entre escribir el nombre del fichero entre "<>" o """", está en el algoritmo usado para encontrar los ficheros a incluir. En el primer caso el preprocesador buscará en los directorios "include" definidos en el compilador. En el segundo, se buscará primero en el directorio actual, es decir, en el que se encuentre el fichero fuente, si no existe en ese directorio, se trabajará como el primer caso.

Si se proporciona el camino como parte del nombre de fichero, sólo se buscará es el directorio especificado.

Directiva #line:

No se usa, se trata de una característica heredada de los primitivos compiladores C.

Sintaxis:

#line constante_entera <"nombre_de_fichero">

Esta directiva se usa para sustituir los números de línea en los programas de referencias cruzadas y en mensajes de error. Si el programa consiste en secciones de otros ficheros fuente unidas en un sólo fichero, se usa para sustituir las referencias a esas secciones con los números de línea del fichero original, como si no se hubiera integrado en un único fichero.

La directiva #line indica que la siguiente línea de código proviene de la línea "constante_entera" del fichero "nombre_de_fichero". Una vez que el nombre de fichero ha sido registrado, sucesivas apariciones de la directiva #line relativas al mismo fichero pueden omitir el argumento del nombre.

Las macros serán expandidas en los argumentos de #line del mismo modo que en la directiva #include.

La directiva #line se usó originalmente para utilidades que producían como salida código C, y no para código escrito por personas.

Directiva #pragma:

Sintaxis:

#pragma nombre-de-directiva

Con esta directiva, cada compilador puede definir sus propias directivas, que no interferiran con las de otros compiladores. Si el compilador no reconoce el nombre-de-directiva, ignorará la línea completa sin producir ningún tipo de error o warning.

CAPITULO 26 Funciones V: Recursividad

Se dice que una función es recursiva cuando se define en función de si misma. No todas la funciones pueden llamarse a si mismas, deben estar diseñadas especialmente para que sean recursivas, de otro modo podrían conducir a bucles infinitos, o a que el programa termine inadecuadamente.

C++ permite la recursividad. Cuando se llama a una función, se crea un nuevo juego de variables locales, de este modo, si la función hace una llamada a si misma, se guardan sus variables y parámetros en la pila, y la nueva instancia de la función trabajará con su propia copia de las variables locales, cuando esta segunda instancia de la función retorna, recupera las variables y los parámetros de la pila y continua la ejecución en el punto en que había sido llamada.

Por ejemplo:

Función recursiva para calcular el factorial de un número entero. El factorial se simboliza como n!, se lee como "n factorial", y la definición es:

n! = n * (n-1) * (n-2) *… 1

```
/* Función recursiva para cálculo de factoriales */
int factorial(n)
{
   if(1 == n) return 1; /* Condición de terminación */
   else return n*factorial(n-1); /* Recursividad */
}
```

Veamos paso a paso, lo que pasa cuando se ejecuta esta función, por ejemplo: factorial(4):

1ª Instancia
n=4
n != 1
salida <- 4 * factorial(3) (Guarda el valor de n = 4)

2ª Instancia
n != 1
salida <- 3*factorial(2) (Guarda el valor de n = 3)

3ª Instancia
n != 1
salida <- 2*factorial(1) (Guarda el valor de n = 2)

4ª Instancia
n == 1 -> retorna 1

3ª Instancia
(recupera n=2 de la pila) retorna 1*2=2

2ª instancia
(recupera n=3 de la pila) retorna 2*3=6

1ª instancia
(recupera n=4 de la pila) retorna 6*4=12
Valor de retorno -> 12

La función factorial es un buen ejemplo para demostrar cómo se hace una función recursiva, pero la recursividad no es un buen modo de resolver esta función, que sería más sencilla y rápida con un bucle "for". La recursividad consume muchos recursos de memoria y tiempo de ejecución, y se debe aplicar a funciones que realmente le saquen partido.

Por ejemplo: visualizar las permutaciones de n elementos.

Las permutaciones de un conjunto es las diferentes maneras de colocar sus elementos, usando todos ellos y sin repetir ninguno. Por ejemplo para A, B, C, tenemos: ABC, ACB, BAC, BCA, CAB, CBA.

```
#include <iostream.h>

/* Prototipo de función */
void Permutaciones(char *, int l=0);

int main(int argc, char *argv[])
{
   char palabra[] = "ABCDE";

   Permutaciones(palabra);
   return 0;
}

void Permutaciones(char * cad, int l)
{
   char c;    /* variable auxiliar para intercambio */
   int i, j;  /* variables para bucles */
   int n = strlen(cad);

   for(i = 0; i < n-l; i++)
   {
      if(n-l > 2) Permutaciones(cad, l+1);
      else cout << cad << ", ";
      /* Intercambio de posiciones */
      c = cad[l];
      cad[l] = cad[l+i+1];
      cad[l+i+1] = c;
      if(l+i == n-1)
      {
         for(j = l; j < n; j++) cad[j] = cad[j+1];
         cad[n] = 0;
      }
   }
}
```

El algoritmo funciona del siguiente modo:

Al principio todos los elementos de la lista pueden cambiar de posición, es decir, pueden permutar su posición con otro. No se fija ningún elemento de la lista, l = 0: Permutaciones(cad, 0)

0	1	2	3	4
A	B	C	D	/0

Se llama recursivamente a la función, pero dejando fijo el primer elemento, el 0: Permutacion(cad,1)

0	1	2	3	4
A	B	C	D	/0

Se llama recursivamente a la función, pero fijando el segundo elemento, el 1: Permutacion(cad,2)

0	1	2	3	4
A	B	C	D	/0

Ahora sólo quedan dos elementos permutables, así que imprimimos ésta permutación, e intercambiamos los elementos: l y l+i+1, es decir el 2 y el 3.

0	1	2	3	4
A	B	D	C	/0

Imprimimos ésta permitación, e intercambiamos los elementos l y l+i+1, es decir el 2 y el 4.

0	1	2	3	4
A	B	/0	C	D

En el caso particular de que l+i+1 sea justo el número de elementos hay que mover hacia la izquierda los elementos desde la posición l+1 a la posición l:

0	1	2	3	4
A	B	C	D	/0

En este punto abandonamos el último nivel de recursión, y retomamos en el valor de l=1 e i = 0.

0	1	2	3	4
A	B	C	D	/0

Permutamos los elementos: l y l+i+1, es decir el 1 y el 2.

0	1	2	3	4
A	C	B	D	/0

En la siguiente iteración del bucle i = 1, llamamos recursivamente con l = 2:
Permutaciones(cad,2)

0	1	2	3	4
A	C	B	D	/0

Imprimimos la permutación e intercambiamos los elementos 2 y 3.

0	1	2	3	4
A	C	D	B	/0

Y así sucesivamente.

Veremos más aplicaciones de recursividad en el tema de estructuras dinámicas de datos.

CAPITULO 27 Tipos de Variables V: Tipos de almacenamiento

Existen ciertos modificadores de variables que se nos estaban quedando en el tintero y que no habíamos visto todavía. Estos modificadores afectan al modo en que se almacenan las variables y a su ámbito temporal, es decir, la zona de programa desde donde las variables son accesibles.

auto

Sintaxis:

[auto] <definición_de_dato> ;

El modificador auto se usa para definir el ámbito temporal de una variable local. Es el modificador por defecto para las variables locales, y se usa muy raramente.

register

Sintaxis:

register <definición_de_dato> ;

Indica al compilador una preferencia para que la variable se almacene en un registro de la CPU, si es posible, con el fin de optimizar su acceso y reducir el código.

Los datos declarados con el modificador register tienen un ámbito temporal global.

El compilador puede ignorar la petición de almacenamiento en registro, éste está basado en el análisis que realice el compilador sobre cómo se usa la variable.

static

Sintaxis:

static <definición_de_dato> ;

static <nombre_de_función> <definición_de_función>;

Se usa con el fín de que las variables locales de una función conserven su valor entre distintas llamadas sucesivas a la misma. Las variables estáticas tienen un ámbito local con respecto a su accesibilidad, pero temporalmente son como las variables externas.

extern

Sintaxis:

extern <definición_de_dato>;

[extern] <función_prototipo>;

Este modificador se usa para indicar que el almacenamiento y valor de una variable o la definición de una función está definido en otro módulo o fichero fuente. Las funciones declaradas con extern son visibles por todos los ficheros fuente del programa, salvo que se redefina la función como static.

El modificador extern es opcional para las funciones prototipo.

Se puede usar extern "c" con el fin de prevenir que algún nombre de función pueda ser ocultado por funciones de programas C++.

CAPITULO 28 Clases I: Definiciones

Aunque te parezca mentira, hasta ahora no hemos visto casi nada de C++. La mayor parte de lo incluido hasta el momento forma parte de C, salvo muy pocas excepciones.

Ahora vamos a entrar a fondo en lo que constituye la mayor diferencia entre C y C++: las clases. Así que prepárate para cambiar la mentalidad, y el enfoque de la programación tal como lo hemos visto hasta ahora.

En éste y en los próximos capítulos iremos introduciendo nuevos conceptos que normalmente se asocian a la programación orientada a objetos, como son: objeto, mensaje, método, clase, herencia, interfaz, etc.

POO:

Siglas de "Programación Orientada a Objetos". En inglés se pone al revés "OOP". La idea básica de éste tipo de programación es agrupar los datos y los procedimientos para manejarlos en una única entidad: el objeto. Un programa es un objeto, que a su vez está formado de objetos. La idea de la programación estructurada no ha desaparecido, de hecho se refuerza y resulta más evidente, como comprobarás cuando veamos conceptos como la herencia.

Objeto:

Un objeto es una unidad que engloba en sí mismo datos y procedimientos necesarios para el tratamiento de esos datos. Hasta ahora habíamos hecho programas en los que los datos y las funciones estaban perfectamente separadas, cuando se programa con objetos esto no es así, cada objeto contiene datos y funciones. Y un programa se construye como un conjunto de objetos, o incluso como un único objeto.

Mensaje:

El mensaje es el modo en que se comunican los objetos entre si. En C++, un mensaje no es más que una llamada a una función de un determinado objeto. Cuando llamemos a una función de un objeto, muy a menudo diremos que estamos enviando un mensaje a ese objeto.

En este sentido, mensaje es el término adecuado cuando hablamos de programación orientada a objetos en general.

Método:

Se trata de otro concepto de POO, los mensajes que lleguen a un objeto se procesarán ejecutando un determinado método. En C++ un método no es otra cosa que una función o procedimiento perteneciente a un objeto.

Clase:

Una clase se puede considerar como un patrón para construir objetos. En C++, un objeto es sólo un tipo de variable de una clase determinada. Es importante distinguir entre objetos y clases, la clase es simplemente una declaración, no tiene asociado ningún objeto, de modo que no puede recibir mensajes ni procesarlos, esto únicamente lo hacen los objetos.

Interfaz:

Las clases y por lo tanto también los objetos, tienen partes públicas y partes privadas. Algunas veces llamaremos a la parte pública de un objeto su interfaz. Se trata de la única parte del objeto que es visible para el resto de los objetos, de modo que es lo único de lo que se dispone para comunicarse con ellos.

Herencia:

Veremos que es posible diseñar nuevas clases basándose en clases ya existentes. En C++ esto se llama derivación de clases, y en POO herencia. Cuando se deriva una clase de otra, normalmente se añadirán nuevos métodos y datos. Es posible que algunos de estos métodos o datos de la clase original no sean válidos, en ese caso pueden ser enmascarados en la nueva clase o simplemente eliminados. El conjunto de datos y métodos que sobreviven, es lo que se conoce como herencia.

CAPITULO 29 Declaración de una clase

Ahora va a empezar un pequeño bombardeo de nuevas palabras reservadas de C++, pero no te asustes, no es tan complicado como parece.

La primera palabra que aparece es lógicamente class que sirve para declarar una clase. Su uso es parecido a la ya conocida struct:

class <identificador de clase> [<:lista de clases base>]

{

<lista de miembros>

} [<lista de objetos>];

La lista de clases base se usa para derivar clases, de momento no le prestes demasiada atención, ya que por ahora sólo declararemos clases base.

La lista de miembros será en general una lista de funciones y datos.

Los datos se declaran del mismo modo en que lo hacíamos hasta ahora, salvo que no pueden ser inicializados, recuerda que estamos hablando de declaraciones de clases y no de definiciones de objetos. En el siguiente capítulo veremos el modo de inicializar las variables de un objeto.

Las funciones pueden ser simplemente declaraciones de prototipos, que se deben definir aparte de la clase o también definiciones.

Cuando se definen fuera de la clase se debe usar el operador de ámbito "::".

Lo veremos mucho mejor con un ejemplo.

```
#include <iostream.h>

class pareja
{
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2)
      {
         a = a2;
         b = b2;
      }
};

void pareja::Lee(int &a2, int &b2)
{
   a2 = a;
```

```
   b2 = b;
}

int main(int argc, char *argv[])
{
   pareja par1;
   int x, y;

   par1.Guarda(12, 32);
   par1.Lee(x, y);
   cout << "Valor de par1.a: " << x << endl;
   cout << "Valor de par1.b: " << y << endl;

   return 0;
}
```

Nuestra clase "pareja" tiene dos miembros de tipo de datos: a y b.

Y dos funciones, una para leer esos valores y otra para modificarlos.

En el caso de la función "Lee" la hemos declarado en el interior de la clase y definido fuera, observa que en el exterior de la declaración de la clase tenemos que usar la expresión:

```
void pareja::Lee(int &a2, int &b2)
```

Para que quede claro que nos referimos a la función "Lee" de la clase "pareja". Ten en cuenta que pueden existir otras clases que tengan funciones con el mismo nombre.

En el caso de la función "Guarda" la hemos definido en el interior de la propia clase. Esto lo haremos sólo cuando la definición sea muy simple, ya que dificulta la lectura y comprensión del programa.

Además, las funciones definidas de éste modo serán tratadas como "inline", y esto sólo es recomendable para funciones cortas, ya que, (como recordarás), en estas funciones se inserta el código cada vez que son llamadas.

Especificaciones de acceso:

Dentro de la lista de miembros, cada miembro puede tener diferentes niveles de acceso.

En nuestro ejemplo hemos usado dos de esos niveles, el privado y el público, aunque hay más.

class <identificador de clase>

{

public:

<lista de miembros>

private:

<lista de miembros>

protected:

<lista de miembros>

};

Acceso privado, private:

Los miembros privados de una clase sólo son accesibles por los propios miembros de la clase y en general por objetos de la misma clase, pero no desde funciones externas o desde funciones de clases derivadas.

Acceso público, public:

Cualquier miembro público de una clase es accesible desde cualquier parte donde sea accesible el propio objeto.

Acceso protegido, protected:

Con respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases derivadas se comporta como público.

Cada una de éstas palabras, seguidas de ":", da comienzo a una sección, que terminará cuando se inicie la sección siguiente o cuando termine la declaración de la clase. Es posible tener varias secciones de cada tipo dentro de una clase.

Si no se especifica nada, por defecto, los miembros de una clase son privados.

CAPITULO 30 Constructores

Los constructores son funciones miembro especiales que sirven para inicializar un objeto de una determinada clase cuando se declara.

Los constructores tienen el mismo nombre que la clase, no retornan ningún valor y no pueden ser heredados. Además deben ser públicos, no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.

Añadamos un constructor a nuestra clase pareja:

```
#include <iostream.h>

class pareja
{
   public:
      // Constructor
      pareja(int a2, int b2);
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};

pareja::pareja(int a2, int b2)
{
   a = a2;
   b = b2;
}

void pareja::Lee(int &a2, int &b2)
{
   a2 = a;
   b2 = b;
}

void pareja::Guarda(int a2, int b2)
{
   a = a2;
   b = b2;
}

int main(int argc, char *argv[])
{
   pareja par1(12, 32);
   int x, y;

   par1.Lee(x, y);
   cout << "Valor de par1.a: " << x << endl;
   cout << "Valor de par1.b: " << y << endl;

   return 0;
}
```

Si una clase posee constructor, será llamado siempre que se declare un objeto de esa clase, y si requiere argumentos, es obligatorio suministrarlos.

Por ejemplo, las siguientes declaraciones son ilegales:

```
pareja par1;
pareja par1();
```

Y las siguientes declaraciones son válidas:

```
pareja par1(12,43);
pareja par1 = pareja(12,43);
```

Cuando no especifiquemos un constructor para una clase, el compilador crea uno por defecto sin argumentos. Por eso el ejemplo del capítulo anterior funcionaba correctamente. Cuando se crean objetos locales, los datos miembros no se inicializarían, contendrían la "basura" que hubiese en la memoria asignada al objeto. Si se trata de objetos globales, los datos miembros se inicializan a cero.

Sobrecarga de constructores:

Además, también pueden definirse varios constructores para cada clase, es decir, la función constructor puede sobrecargarse. La única limitación es que no pueden declararse varios constructores con el mismo número y el mismo tipo de argumentos.

Por ejemplo, añadiremos un constructor adicional a la clase "pareja" que simule el constructor por defecto:

```
class pareja
{
   public:
      // Constructor
      pareja(int a2, int b2);
      pareja();
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};

pareja::pareja(int a2, int b2)
{
   a = a2;
   b = b2;
}

pareja::pareja()
{
   a = b = 0;
}
```

De este modo podemos declarar objetos de la clase pareja especificando los dos argumentos o ninguno de ellos, en este último caso se inicializarán los datos miembros con ceros.

Constructores con argumentos por defecto:

También pueden asignarse valores por defecto a los argumentos del constructor, de este modo reduciremos el número de constructores necesarios.

Para resolver el ejemplo anterior sin sobrecargar el constructor suministraremos valores por defecto nulos a ambos parámetros:

```
class pareja
{
   public:
      // Constructor
      pareja(int a2=0, int b2=0);
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};

pareja::pareja(int a2, int b2)
{
   a = a2;
   b = b2;
}
```

Asignación de objetos:

Probablemente ya lo imaginas, pero la asignación de objetos también está permitida. Y además funciona como se supone que debe hacerlo, asignando los valores de los datos miembros.

Con la definición de la clase del último ejemplo podemos hacer lo que se ilustra en el siguiente:

```
int main(int argc, char *argv[])
{
   pareja par1(12, 32), par2;
   int x, y;

   par2 = par1;
   par2.Lee(x, y);
   cout << "Valor de par2.a: " << x << endl;
   cout << "Valor de par2.b: " << y << endl;

   return 0;
}
```

La línea "par2 = par1;" copia los valores de los datos miembros de par1 en par2.

Constructor copia:

Un constructor de este tipo crea un objeto a partir de otro objeto existente. Estos constructores sólo tienen un argumento, que es una referencia a un objeto de su misma clase.

En general, los constructores copia tienen la siguiente forma para sus prototipos:

tipo_clase::tipo_clase(const tipo_clase &obj);

De nuevo ilustraremos esto con un ejemplo y usaremos también "pareja":

```
class pareja
{
   public:
      // Constructor
      pareja(int a2=0, int b2=0);
      // Constructor copia:
      pareja(const pareja &p);

      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};

pareja::pareja(int a2, int b2)
{
   a = a2;
   b = b2;
}

pareja::pareja(const pareja &p)
{
   a = p.a;
   b = p.b;
}
```

Para crear objetos usando el constructor copia se procede como sigue:

```
int main()
{
   pareja par1(12, 32)
   pareja par2(par1); // Uso del constructor copia: par2 = par1
   int x, y;

   par2.Lee(x, y);
   cout << "Valor de par2.a: " << x << endl;
   cout << "Valor de par2.b: " << y << endl;

   return 0;
}
```

También en este caso, si no se especifica ningún constructor copia, el compilador crea uno por defecto, y su comportamiento es exactamente el mismo que el del definido en el ejemplo anterior. Para la mayoría de los casos esto será suficiente, pero en algunas ocasiones podemos necesitar redefinir el constructor copia.

CAPITULO 31 Destructores

Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase, liberando la memoria utilizada por dicho objeto.

Los destructores tienen el mismo nombre que la clase, pero con el símbolo ~ delante, no retornan ningún valor y no pueden ser heredados.

Cuando se define un destructor para una clase, éste es llamado automáticamente cuando se abandona el ámbito en el que fue definido. Esto es así salvo cuando el objeto fue creado dinámicamente con el operador new, ya que en ese caso, si es necesario eliminarlo, hay que usar el operador delete.

En general, será necesario definir un destructor cuando nuestra clase tenga datos miembro de tipo puntero, aunque esto no es una regla estricta. El destructor no puede sobrecargarse, por la sencilla razón de que no admite argumentos.

Ejemplo:

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

class cadena {
   public:
      cadena();        // Constructor por defecto
      cadena(char *c); // Constructor desde cadena c
      cadena(int n);   // Constructor para una cadena de n caracteres
      cadena(const cadena &);   // Constructor copia
      ~cadena();       // Destructor

      void Asignar(char *dest);
      char *Leer(char *c);
   private:
      char *cad;       // Puntero a char: cadena de caracteres
};

cadena::cadena()
{
   cad = NULL;
}

cadena::cadena(char *c)
{
   cad = new char[strlen(c)+1]; // Reserva memoria para la cadena
   strcpy(cad, c);              // Almacena la cadena
}

cadena::cadena(int n)
{
   cad = new char[n+1];         // Reserva memoria para n caracteres
   cad[0] = 0;                  // Cadena vacía
}

cadena::cadena(const cadena &Cad)
{
   // Eliminamos la cadena actual:
```

```
   delete[] cad;
   // Reservamos memoria para la nueva y la almacenamos
   cad = new char[strlen(Cad.cad)+1]; // Reserva memoria para la
cadena
   strcpy(cad, Cad.cad);              // Almacena la cadena
}

cadena::~cadena()
{
   delete[] cad;                // Libera la memoria reservada a cad
}

void cadena::Asignar(char *dest)
{
   // Eliminamos la cadena actual:
   delete[] cad;
   // Reservamos memoria para la nueva y la almacenamos
   cad = new char[strlen(dest)+1]; // Reserva memoria para la cadena
   strcpy(cad, dest);              // Almacena la cadena
}

char *cadena::Leer(char *c)
{
   strcpy(c, cad);
   return c;
}

int main(int argc, char *argv[])
{
   cadena Cadena1("Cadena de prueba");
   cadena Cadena2(Cadena1);   // Cadena2 es copia de Cadena1
   cadena *Cadena3;           // Cadena3 es un puntero
   char c[256];

   Cadena1.Asignar("Otra cadena diferente");      // Modificamos
Cadena1
   Cadena3 = new cadena("Cadena de prueba nº 3"); // Creamos Cadena3

   // Ver resultados
   cout << "Cadena 1: " << Cadena1.Leer(c) << endl;
   cout << "Cadena 2: " << Cadena2.Leer(c) << endl;
   cout << "Cadena 3: " << Cadena3->Leer(c) << endl;

   delete Cadena3;  // Destruir Cadena3.
   // Cadena1 y Cadena2 se destruyen automáticamente
   return 0;
}
```

Quiero hacer varias observaciones sobre este programa:

1. Hemos implementado un constructor copia. Esto es necesario porque una simple asignación no copiaría la cadena de un objeto a otro, sino únicamente los punteros.

Por ejemplo, si declaramos Cadena2 como:

cadena Cadena2;

Y después hacemos:

```
Cadena2 = Cadena1;
```

Lo que estaríamos copiando sería el valor del puntero cad, con lo cual, ambos punteros estarían apuntando a la misma posición de memoria. Esto es desastroso, y no simplemente porque los cambios en una cadena afectan a las dos, sino porque al abandonar el programa se intenta liberar automáticamente la misma memoria dos veces. Lo que realmente pretendemos al asignar cadenas es crear una nueva cadena que sea copia de la cadena antigua. Esto es lo que hacemos con el constructor copia, y es lo que haremos más adelante, y con más elegancia, sobrecargando el operador de asignación.

2. La función Leer, que usamos para obtener el valor de la cadena almacenada, no devuelve un puntero a la cadena, sino una copia de la cadena. Esto está de acuerdo con las recomendaciones sobre la programación orientada a objetos, que aconsejan que los datos almacenados en una clase no sean accesibles directamente desde fuera de ella, sino únicamente a través de las funciones creadas al efecto. Además, el miembro cad es privado, y por lo tanto debe ser inaccesible desde fuera de la clase.

La Cadena3 debe ser destruida implícitamente usando el operador delete, que a su vez invoca al destructor de la clase. Esto es así porque se trata de un puntero, y la memoria que se usa en el objeto al que apunta no se libera automáticamente al destruirse el puntero Cadena3.

CAPITULO 32 El puntero this

Para cada objeto declarado de una clase se mantiene una copia de sus datos, pero todos comparten la misma copia de las funciones de esa clase.

Esto ahorra memoria y hace que los programas ejecutables sean más compactos, pero plantea un problema.

Cada función de una clase puede hacer referencia a los datos de un objeto, modificarlos o leerlos, pero si sólo hay una copia de la función y varios objetos de esa clase, ¿cómo hace la función para referirse a un dato de un objeto en concreto?

La respuesta es: usando el puntero this. Cada objeto tiene asociado un puntero a si mismo que se puede usar para manejar sus miembros.

Volvamos al ejemplo de la clase pareja:

```
#include <iostream.h>

class pareja
{
   public:
      // Constructor
      pareja(int a2, int b2);
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};
```

Para cada dato podemos referirnos de dos modos distintos, lo veremos con la función Guarda. Esta es la implementación que usamos en el capítulo 30, que es como normalmente nos referiremos a los miembros de las clases:

```
void pareja::Guarda(int a2, int b2)
{
   a = a2;
   b = b2;
}
```

Veamos ahora la manera equivalente usando el puntero this:

```
void pareja::Guarda(int a2, int b2)
{
   a = this->a2;
   b = this->b2;
}
```

Normalmente no será necesario usar el puntero this en nuestros programas, pero nos resultará muy útil en el futuro, cuando implementemos estructuras dinámicas de datos.

CAPITULO 33 Sistema de protección

Ya sabemos que los miembros privados de una clase no son accesibles para funciones y clases exteriores a dicha clase.

Esto es un concepto de POO, el encapsulamiento hace que cada objeto se comporte de un modo autónomo y que lo que pase en su interior sea invisible para el resto de objetos. Cada objeto sólo responde a ciertos mensajes y proporciona determinadas salidas.

Pero, en ciertas ocasiones, querremos poder acceder a determinados miembros privados de un objeto de una clase desde otros objetos de clases diferentes. C++ proporciona un mecanismo para sortear el sistema de protección. En otros capítulos veremos la utilidad de esta técnica, de momento sólo explicaremos en qué consiste.

Declaraciones friend

El modificador "friend" puede aplicarse a clases o funciones para inhibir el sistema de protección.

Las relaciones de "amistad" entre clases son parecidas a las amistades entre personas:

- La amistad no puede transferirse, si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C. (La famosa frase: "los amigos de mis amigos son mis amigos", es falsa en C++, y probablemente también en la vida real).
- La amistad no puede heredarse. Si A es amigo de B, y C es una clase derivada de B, A no es amigo de C. (Los hijos de mis amigos, no tienen por qué ser amigos míos. De nuevo, el símil es casi perfecto).
- La amistad no es simétrica. Si A es amigo de B, B no tiene por qué ser amigo de A. (En la vida real, una situación como esta hará peligrar la amistad de A con B, pero me temo que en realidad, se trata de una situación muy frecuente).

Funciones externas amigas

El caso más sencillo es el de una relación de amistad con una función externa.

Veamos un ejemplo muy sencillo:

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

class A {
  public:
    A(int i=0) {a = i;}
    void Ver() {cout << a << endl;}
  private:
    int a;
    friend void Ver(A); // "Ver" es amiga de la clase A
};

void Ver(A Xa)
```

```
{
   // La función Ver puede acceder a miembros privados
   // de la clase A, ya que ha sido declarada "amiga" de A
   cout << Xa.a << endl;
}

int main(int argc, char *argv[])
{
   A Na(10);

   Ver(Na);  // Ver el valor de Na.a
   Na.Ver(); // Equivalente a la anterior
   system("PAUSE");
   return 0;
}
```

Como puedes ver, la función "Ver", que no pertenece a la clase A puede acceder al miembro privado de A y visualizarlo. Incluso podría modificarlo.

No parece que sea muy útil, ¿verdad?. Bueno, seguro que en alguna ocasión tiene aplicaciones prácticas.

Funciones amigas en otras clases

El siguiente caso es más común, se trata de cuando la función amiga forma parte de otra clase. El proceso es más complejo. Veamos otro ejemplo:

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

class A; // Declaración previa (forward)

class B {
   public:
    B(int i=0) {b = i;}
    void Ver() {cout << b << endl;}
    bool EsMayor(A Xa);  // Compara b con a
   private:
    int b;
};

class A {
   public:
    A(int i=0) {a = i;}
    void Ver() {cout << a << endl;}
   private:
    // Función amiga tiene acceso a miembros privados de la clase A
    friend bool B::EsMayor(A Xa);
    int a;
};

bool B::EsMayor(A Xa)
{
   return b > Xa.a;
}


int main(int argc, char *argv[])
```

```
{
   A Na(10);
   B Nb(12);

   Na.Ver();
   Nb.Ver();
   if(Nb.EsMayor(Na)) cout << "Nb es mayor que Na" << endl;
   else cout << "Nb no es mayor que Na" << endl;
   system("PAUSE");
   return 0;
}
```

Puedes comprobar lo que pasa si eliminas la línea donde se declara "EsMayor" como amiga de A.

Es necesario hacer una declaración previa de la clase A (forward) para que pueda referenciarse desde la clase B.

Veremos que estas "amistades" son útiles cuando sobrecarguemos algunos operadores.

Clases amigas.

El caso más común de amistad se aplica a clases completas. Lo que sigue es un ejemplo de implementación de una lista dinámica mediante el uso de dos clases "amigas".

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

/* Clase para elemento de lista enlazada */
class Elemento {
  public:
   Elemento(int t);                          /* Constructor */
   int Tipo() { return tipo;}                /* Obtener tipo */
  private:                                   /* Datos: */
   int tipo;                                 /* Tipo */
   Elemento *sig;                            /* Siguiente elemento */
  friend class Lista;                        /* Amistad con lista */
};

/* Clase para lista enlazada de números*/
class Lista {
  public:
   Lista()                                   /* Constructor inline */
   {
      Cabeza = NULL;                         /* Lista vacía */
   }
   ~Lista() {LiberarLista();}                /* Destructor inline */
   void Nuevo(int tipo);                     /* Insertar figura */
   Elemento *Primero() {return Cabeza;}      /* Obtener primer
elemento */
   Elemento *Siguiente(Elemento *p)          /* Obtener el siguiente
elemento a p */
      {if(p) return p->sig; else return p;}; /* Si p no es NULL */
   bool EstaVacio() { return Cabeza == NULL;}/* Averiguar si la lista
está vacía */

  private:
```

```
   Elemento *Cabeza;                          /* Puntero al primer
elemento */
   void LiberarLista();                       /* Función privada para
borrar lista */
};

/* Constructor */
Elemento::Elemento(int t)
{
   tipo = t;     /* Asignar datos desde lista de parámetros */
   sig = NULL;
}

/* Añadir nuevo elemento al principio de la lista */
void Lista::Nuevo(int tipo)
{
   Elemento *p;

   p = new Elemento(tipo);  /* Nuevo elemento */
   p->sig = Cabeza;
   Cabeza = p;
}

/* Borra todos los elementos de la lista */
void Lista::LiberarLista()
{
   Elemento *p;

   while(Cabeza){
      p = Cabeza;
      Cabeza = p->sig;
      delete p;
   }
}

int main(int argc, char *argv[])
{
   Lista miLista;
   Elemento *e;

   // Insertamos varios valores en la lista
   miLista.Nuevo(4);
   miLista.Nuevo(2);
   miLista.Nuevo(1);

   // Y los mostramos en pantalla:
   e = miLista.Primero();
   while(e) {
      cout << e->Tipo() << " ,";
      e = miLista.Siguiente(e);
   }
   cout << endl;
   system("PAUSE");
   return 0;
}
```

La clase Lista puede acceder a todos los miembros de Elemento, sean o no públicos, pero desde la función "main" sólo podemos acceder a los miembros públicos de nuestro elemento.

Para poder introducir este ejemplo he usado algunos conceptos que aún no hemos explicado, como los punteros a objetos, o el uso de los operadores new y delete con clases, pero creo que vale la pena para demostrar que la amistad entre clases tiene una aplicación práctica.

CAPITULO 34 Modificadores para miembros

Existen varias alternativas a la hora de definir algunos de los miembros de las clases. Esto es lo que veremos en éste capítulo. Estos modificadores afectan al modo en que se genera el código de ciertas funciones y datos, o al modo en que se tratan los valores de retorno.

Funciones en línea (inline):

A menudo nos encontraremos con funciones miembro cuyas definiciones son muy pequeñas. En estos casos suele ser interesante declararlas como inline. Cuando hacemos eso, el código generado para la función cuando el programa se compila, se inserta en el punto donde se invoca a la función, en lugar de hacerlo en otro lugar y hacer una llamada.

Esto nos proporciona una ventaja, el código de estas funciones se ejecuta más rápidamente, ya que se evita usar la pila para pasar parámetros y se evitan las instrucciones de salto y retorno. También tiene un inconveniente: se generará el código de la función tantas veces como ésta se use, con lo que el programa ejecutable final puede ser mucho más grande en ciertas ocasiones.

Es por esos dos motivos por los que sólo se usan funciones inline cuando las funciones son pequeñas. Hay que elegir con cuidado qué funciones declararemos inline y cuales no, ya que el resultado puede ser muy diferente dependiendo de nuestras decisiones.

Hay dos maneras de declarar una función como inline.

La primera ya la hemos visto. Las funciones que se definen dentro de la declaración de la clase son inline implícitamente. Por ejemplo:

```
class Ejemplo {
  public:
   Ejemplo(int a = 0) { A = a;}

  private:
   int A;
};
```

En éste ejemplo hemos definido el constructor de la clase Ejemplo dentro de la propia declaración, esto hace que se considere como inline. Cada vez que declaremos un objeto de la clase Ejemplo se insertará el código correspondiente a su constructor.

Si queremos que la clase Ejemplo no tenga un constructor inline deberemos declararla y definirla así:

```
class Ejemplo {
  public:
```

```
   Ejemplo(int a = 0);

  private:
   int A;
};

Ejemplo::Ejemplo(int a)
{
   A = a;
}
```

En este caso, cada vez que declaremos un objeto de la clase Ejemplo se hará una llamada al constructor y sólo existirá una copia del código del constructor en nuestro programa.

La otra forma de declarar funciones inline es hacerlo explícitamente, usando la palabra reservada **inline**. En el ejemplo anterior sería así:

```
class Ejemplo {
  public:
   Ejemplo(int a = 0);

  private:
   int A;
};

inline Ejemplo::Ejemplo(int a)
{
   A = a;
}
```

Funciones miembro constantes

Esta es una propiedad que nos será muy útil en la depuración de nuestras clases.

Cuando una función miembro no modifique el valor de ningún dato de la clase, podemos y debemos declararla como constante. Esto no evitará que la función modifique los datos del objeto, el código de la función lo escribimos nosotros, pero generará un error durante la compilación si la función intenta modificar alguno de los datos miembro del objeto.

Por ejemplo:

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

class Ejemplo2 {
  public:
   Ejemplo2(int a = 0) {A = a; }
   void Modifica(int a) { A = a; }
   int Lee() const { return A; }

  private:
   int A;
};
```

```
int main()
{
   Ejemplo2 X(6);

   cout << X.Lee() << endl;
   X.Modifica(2);
   cout << X.Lee() << endl;

   system("PAUSE");
   return 0;
}
```

Para experimentar, comprueba lo que pasa si cambias la definición de la función "Lee()" por estas otras:

```
int Lee() const { A++; return A; }
int Lee() const { Modifica(A+1); return A; }
int Lee() const { Modifica(3); return A; }
```

Verás que el compilador no lo permite.

Miembros estáticos de una clase (Static)

Ciertos miembros de una clase pueden ser declarados como static. Los miembros static tienen algunas propiedades especiales.

En el caso de los datos miembro static sólo existirá una copia que compartirán todos los objetos de la misma clase. Si consultamos el valor de ese dato desde cualquier objeto de esa clase obtendremos siempre el mismo resultado, y si lo modificamos, lo modificaremos para todos los objetos.

Por ejemplo:

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

class Numero {
  public:
   Numero(int v = 0);
   ~Numero();

   void Modifica(int v);
   int LeeValor() const { return Valor; }
   int LeeCuenta() const { return Cuenta; }
   int LeeMedia() const { return Media; }

  private:
   int Valor;
   static int Cuenta;
   static int Suma;
   static int Media;

   void CalculaMedia();
};

Numero::Numero(int v)
```

```
{
   Valor = v;
   Cuenta++;
   Suma += Valor;
   CalculaMedia();
}

Numero::~Numero()
{
   Cuenta--;
   Suma -= Valor;
   CalculaMedia();
}

void Numero::Modifica(int v)
{
   Suma -= Valor;
   Valor = v;
   Suma += Valor;
   CalculaMedia();
}

// Inicialización de miembros estáticos
int Numero::Cuenta = 0;
int Numero::Suma = 0;
int Numero::Media = 0;

void Numero::CalculaMedia()
{
   if(Cuenta > 0) Media = Suma/Cuenta;
   else Media = 0;
}

int main()
{
   Numero A(6), B(3), C(9), D(18), E(3);
   Numero *X;

   cout << "INICIAL" << endl;
   cout << "Cuenta: " << A.LeeCuenta() << endl;
   cout << "Media:  " << A.LeeMedia() << endl;

   B.Modifica(11);
   cout << "Modificamos B=11" << endl;
   cout << "Cuenta: " << B.LeeCuenta() << endl;
   cout << "Media:  " << B.LeeMedia() << endl;

   X = new Numero(548);
   cout << "Nuevo elemento dinámico de valor 548" << endl;
   cout << "Cuenta: " << X->LeeCuenta() << endl;
   cout << "Media:  " << X->LeeMedia() << endl;

   delete X;
   cout << "Borramos el elemento dinámico" << endl;
   cout << "Cuenta: " << D.LeeCuenta() << endl;
   cout << "Media:  " << D.LeeMedia() << endl;

   system("PAUSE");
   return 0;
}
```

Observa que es necesario inicializar los miembros static de la clase, esto es por dos motivos. El primero es que los miembros static existen aunque no exista ningún objeto de la clase. El segundo es porque no lo hiciéramos, al declarar objetos de esa clase los valores de los miembros estáticos estarían indefinidos, y los resultados no serían los esperados.

En el caso de la funciones miembro static la utilidad es menos evidente. Estas funciones no pueden acceder a los miembros de los objetos, sólo pueden acceder a los datos miembro de la clase que sean static. Esto significa que no tienen puntero this, y además suelen ser usadas con su nombre completo, incluyendo el nombre de la clase y el operador de ámbito (::).

Por ejemplo:

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

class Numero {
  public:
   Numero(int v = 0);

   void Modifica(int v) { Valor = v; }
   int LeeValor() const { return Valor; }
   int LeeDeclaraciones() const { return ObjetosDeclarados; }
   static void Reset() { ObjetosDeclarados = 0; }

  private:
   int Valor;
   static int ObjetosDeclarados;
};

Numero::Numero(int v)
{
   Valor = v;
   ObjetosDeclarados++;
}

int Numero::ObjetosDeclarados = 0;

int main()
{
   Numero A(6), B(3), C(9), D(18), E(3);
   Numero *X;

   cout << "INICIAL" << endl;
   cout << "Objetos de la clase Numeros: " << A.LeeDeclaraciones()
<< endl;

   Numero::Reset();
   cout << "RESET" << endl;
   cout << "Objetos de la clase Numeros: " << A.LeeDeclaraciones()
<< endl;

   X = new Numero(548);
   cout << "Cuenta de objetos dinámicos declarados" << endl;
   cout << "Objetos de la clase Numeros: " << A.LeeDeclaraciones()
<< endl;
```

```
   delete X;
   X = new Numero(8);
   cout << "Cuenta de objetos dinámicos declarados" << endl;
   cout << "Objetos de la clase Numeros: " << A.LeeDeclaraciones()
<< endl;

   delete X;
   system("PAUSE");
   return 0;
}
```

Observa cómo hemos llamado a la función Reset con su nombre completo. Aunque podríamos haber usado "A.Reset()", es más lógico usar el nombre completo, ya que la función puede ser invocada aunque no exista ningún objeto de la clase.

CAPITULO 35 Más sobre las funciones

Funciones sobrecargadas:

Ya hemos visto que se pueden sobrecargar los constructores, y en el capítulo 23 vimos que se podía sobrecargar cualquier función, aunque no pertenezcan a ninguna clase. Pues bien, las funciones miembros de las clases también pueden sobrecargarse, como supongo que ya habrás supuesto.

Lo que había olvidado mencionar es que esta propiedad de C++ es lo que se conoce como polimorfismo, así que aprovecho la ocasión para hacerlo.

No hay mucho más que añadir, así que pondré un ejemplo:

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

struct punto3D {
   float x, y, z;
};

class punto {
  public:
   punto(float xi, float yi, float zi) { Asignar(xi, yi, zi); }
   punto(punto3D p) { Asignar(p.x, p.y, p.z); }

   void Asignar(float xi, float yi, float zi)
   {
      x = xi;
      y = yi;
      z = zi;
   }
   void Asignar(punto3D p)
   {
      Asignar(p.x, p.y, p.z);
   }
   void Ver()
   {
      cout << "(" << x << "," << y << "," << z << ")" << endl;
   }
  private:
   float x, y, z;
};

int main()
{
   punto P(0,0,0);
   punto3D p3d = {32,45,74};

   P.Ver();
   P.Asignar(p3d);
   P.Ver();
   P.Asignar(12,35,12);
   P.Ver();

   system("PAUSE");
```

```
   return 0;
}
```

Como se ve, en C++ las funciones sobrecargadas funcionan igual dentro y fuera de las clases.

Funciones con argumentos con valores por defecto:

También hemos visto que se pueden usar argumentos con valores por defecto en los constructores, y también vimos en el capítulo 22a que se podían usar con cualquier función fuera de las clases. En las funciones miembros de las clases también pueden usarse parámetros con valores por defecto, del mismo modo que fuera de las clases.

De nuevo ilustraremos esto con un ejemplo:

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

class punto {
  public:
   punto(float xi, float yi, float zi) { Asignar(xi, yi, zi); }

   void Asignar(float xi, float yi = 0, float zi = 0)
   {
      x = xi;
      y = yi;
      z = zi;
   }
   void Ver()
   {
      cout << "(" << x << "," << y << "," << z << ")" << endl;
   }
  private:
   float x, y, z;
};

int main()
{
   punto P(0,0,0);

   P.Ver();
   P.Asignar(12);
   P.Ver();
   P.Asignar(16,35);
   P.Ver();
   P.Asignar(34,43,12);
   P.Ver();

   system("PAUSE");
   return 0;
}
```

Las reglas para definir parámetros con valores por defecto son las mismas que se expusieron en el capítulo 22a.

CAPITULO 36 Operadores sobrecargados:

Ya habíamos visto el funcionamiento de los operadores sobrecargados en el capítulo 24, aplicándolos a operaciones con estructuras. Ahora veremos todo su potencial, aplicandolos a clases.

Sobrecarga de operadores binarios:

Empezaremos por los operadores binarios, que como recordarás son aquellos que requieren dos operandos, como la suma o la resta.

Existe una diferencia entre la sobrecarga de operadores que vimos en el capítulo 24, que se definía fuera de las clases. Cuando se sobrecargan operadores en el interior se asume que el primer operando es el propio objeto de la clase donde se define el operador. Debido a esto, sólo se necesita especificar un operando.

Sintaxis:

```
<tipo> operator<operador binario>(<tipo> <identificador>);
```

Normalmente el <tipo> es la clase para la que estamos sobrecargando el operador, tanto en el valor de retorno como en el parámetro.

Veamos un ejemplo para una clase para el tratamiento de tiempos:

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

class Tiempo {
  public:
   Tiempo(int h=0, int m=0) : hora(h), minuto(m) {}

   void Mostrar();
   Tiempo operator+(Tiempo h);

  private:
   int hora;
   int minuto;
};

Tiempo Tiempo::operator+(Tiempo h)
{
   Tiempo temp;

   temp.minuto = minuto + h.minuto;
   temp.hora   = hora   + h.hora;

   if(temp.minuto >= 60) {
      temp.minuto -= 60;
      temp.hora++;
   }
   return temp;
```

```
}

void Tiempo::Mostrar()
{
   cout << hora << ":" << minuto << endl;
}

int main()
{
   Tiempo Ahora(12,24), T1(4,45);

   T1 = Ahora + T1;   // (1)
   T1.Mostrar();

   (Ahora + Tiempo(4,45)).Mostrar(); // (2)

   system("PAUSE");
   return 0;
}
```

Me gustaría hacer algunos comentarios sobre el ejemplo

Observa que cuando sumamos dos tiempos obtenemos un tiempo, se trata de una propiedad de la suma, todos sabemos que no se pueden sumar peras y manzanas.

Pero en C++ sí se puede. Por ejemplo, podríamos haber sobrecargado el operador suma de este modo:

```
int operator+(Tiempo h);
```

Pero no estaría muy clara la naturaleza del resultado, ¿verdad?. Lo lógico es que la suma de dos objetos produzca un objeto del mismo tipo.

Hemos usado un objeto temporal para calcular el resultado de la suma, esto es necesario porque necesitamos operar con los minutos para prevenir el caso en que excedan de 60, en cuyo caso incrementaremos el tiempo en una hora.

Ahora observa cómo utilizamos el operador en el programa.

La forma (1) es la forma más lógica, para eso hemos creado un operador, para usarlo igual que en las situaciones anteriores.

Pero verás que también hemos usado el operador =, a pesar de que nosotros no lo hemos definido. Esto es porque el compilador crea un operador de asignación por defecto si nosotros no lo hacemos, pero veremos más sobre eso en el siguiente punto.

La forma (2) es una pequeña aberración, pero ilustra cómo es posible crear objetos temporales sin nombre.

En esta línea hay dos, el primero Tiempo(4,45), que se suma a Ahora para producir otro objeto temporal sin nombre, que es el que mostramos en pantalla.

Sobrecargar el operador de asignación: ¿por qué?

Ya sabemos que el compilador crea un operador de asignación por defecto si nosotros no lo hacemos, así que ¿por qué sobrecargarlo?.

Bueno, veamos lo que pasa si nuestra clase tiene miembros que son punteros, por ejemplo:

```
class Cadena {
  public:
   Cadena(char *cad);
   Cadena() : cadena(NULL) {};
   ~Cadena() { delete[] cadena; };

   void Mostrar() const;
  private:
   char *cadena;
};

Cadena::Cadena(char *cad)
{
   cadena = new char[strlen(cad)+1];
   strcpy(cadena, cad);
}


void Cadena::Mostrar() const
{
   cout << cadena << endl;
}
```

Si en nuestro programa declaramos dos objetos de tipo Cadena:

```
Cadena C1("Cadena de prueba"), C2;
```

Y hacemos una asignación:

```
C1 = C2;
```

Lo que realmente copiamos no es la cadena, sino el puntero. Ahora los dos punteros de las cadenas C1 y C2 están apuntando a la misma dirección. ¿Qué pasará cuando destruyamos los objetos?. Al destruir C1 se intentará liberar la memoria de su puntero cadena, y al destruir C2 también, pero ambos punteros apuntan a la misma dirección y el valor original del puntero de C1 se ha perdido, y su memoria no puede ser liberarada.

En estos casos, análogamente a lo que sucedía con el constructor copia, deberemos sobrecargar el operador de asignación. En nuestro ejemplo podría ser así:

```
Cadena &Cadena::operator=(const Cadena &c)
{
   if(this != &c) {
      delete[] cadena;
      if(c.cadena) {
         cadena = new char[strlen(c.cadena)+1];
         strcpy(cadena, c.cadena);
      }
      else cadena = NULL;
```

```
   }
   return *this;
}
```

Hay que tener en cuenta la posibilidad de que se asigne un objeto a si mismo. Por eso comparamos el puntero this con la dirección del parámetro, si son iguales es que se trata del mismo objeto, y no debemos hacer nada.

También hay que tener cuidado de que la cadena a copiar no dea NULL, en ese caso no debemos copiar la cadena, sino sólo asignar NULL a cadena.

Y por último, también es necesario retornar una referencia al objeto, esto nos permitirá escribir expresiones como estas:

```
C1 = C2 = C3;
if((C1 = C2) == C3)...
```

Por supuesto, para el segundo caso deberemos sobrecargar el operador ==.

Operadores binarios que pueden sobrecargarse:

Además del operador + pueden sobrecargarse prácticamente todos los operadores:

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

Los operadores =, [], () y -> sólo pueden sobrecargarse en el interior de clases.

Por ejemplo, el operador > podría declararse y definirse así:

```
class Tiempo {
...
bool operator>(Tiempo h);
...
};

bool Tiempo::operator>(Tiempo h)
{
   return (hora > h.hora || (hora == h.hora && minuto > h.minuto));
}
...
if(Tiempo(1,32) > Tiempo(1,12)) cout << "1:32 es mayor que 1:12" <<
endl;
else cout << "1:32 es menor o igual que 1:12" << endl;
...
```

Para los operadores de igualdad el valor de retorno es bool, lógicamente, ya que estamos haciendo una comparación.

Y el operador +=, de esta otra:

```
class Tiempo {
...
void operator+=(Tiempo h);
```

```
...
};

void Tiempo::operator+=(Tiempo h)
{
   minuto += h.minuto;
   hora   += h.hora;

   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
}
...
Ahora += Tiempo(1,32);
Ahora.Mostrar();
...
```

Los operadores de asignación no necesitan valor de retorno, ya que es el propio objeto al que se aplican el que recibe el resultado de la operación.

Con el resto de lo operadores binarios se trabaja del mismo modo.

No es imprescindible mantener el significado de los operadores. Por ejemplo, para la clase Tiempo no tiene sentido sobrecargar el operadores >>, <<, * ó /, pero podemos hacerlo de todos modos, y olvidar el significado que tengan habitualmente. De igual modo podríamos haber sobrecargado el operador + y hacer que no sumara los tiempos sino que, por ejemplo, los restara. En última instancia, es el programador el que decide el significado de los operadores.

Por ejemplo, sobrecargaremos el operador >> para que devuelva el mayor de los operandos.

```
class Tiempo {
...
Tiempo operator>>(Tiempo h);
...
};

Tiempo Tiempo::operator>>(Tiempo h)
{
   if(*this > h) return *this; else return h;
}
...

T1 = Ahora >> Tiempo(13,43) >> T1 >> Tiempo(12,32);
T1.Mostrar();
...
```

En este ejemplo hemos recurrido al puntero this, para usar el objeto actual en una comparación y para revolverlo como resultado en el caso adecuado.

Esta es una de las aplicaciones del puntero this, si no dispusieramos de él, sería imposible hacer referencia al propio objeto al que se aplica el operador.

También vemos que los operadores binarios siguen admitiendo la asociciación aún estando sobrecargados.

Forma funcional de los operadores:

Por supuesto también es posible usar la forma funcional de los operadores sobrecargados, aunque no es muy habitual ni aconsejable.

En el caso del operador + las siguientes expresiones on equvalentes:

```
T1 = T1.operator+(Ahora);

T1 = Ahora + T1;
```

Sobrecarga el operadores para la clases con punteros:

Si intentamos sobrecargar el operador suma con la clase Cadena usando el mismo sistema que con Tiempo, veremos que no funciona.

Cuando nuestras clases tienen punteros con memoria dinámica asociada, la sobrecarga de funciones y operadores puede complicarse un poco.

Por ejemplo, sobrecarguemos el operador + para la clase Cadena:

```
class Cadena {
...
   Cadena operator+(const Cadena &);
...
};

Cadena Cadena::operator+(const Cadena &c)
{
   Cadena temp;

   temp.cadena = new char[strlen(c.cadena)+strlen(cadena)+1];
   strcpy(temp.cadena, cadena);
   strcat(temp.cadena, c.cadena);
   return temp;
}
...
Cadena C1, C2("Primera parte");

C1 = C2 + " Segunda parte";
```

Ahora analicemos cómo funciona el código de este operador.

El equivalente de ésta última línea es:

```
C1.operator=(Cadena(C2.operator+(Cadena(" Segunda parte"))));
```

1) Se crea automáticamente un objeto temporal sin nombre para la cadena " Segunda parte". Y se llama al operador + del objeto C2.

2) Dentro del operador + se crea un objeto temporal: temp, reservamos memoria para la cadena que almacenará la concatenación de this->cadena y c.cadena, y le asignamos el valor de ambas cadenas, temp contiene la cadena: "Primera parte Segunda parte".

3) Retornamos el objeto temporal.

4) Ahora el objeto temporal temp se copia a otro objeto temporal sin nombre, y temp es destruido. Y el objeto temporal sin nombre se pasa como parámetro al operador de asignación. Si esto es difícil de entender, piensa lo que pasa cuando usamos el operador de asignación con una cadena, por ejemplo:

C1 = "hola";

En este caso se crea un objeto temporal sin sombre para "hola", igual que pasó con la cadena " Segunda parte".

5) Se asigna el objeto temporal sin nombre a C1, y se destruye.

Parece que todo ha ido bien, pero en el paso 4 hay un problema. Para copiar temp en el objeto temporal sin nombre se usa el constructor copia de Cadena. Pero como nosotros no hemos creado un constructor copia, se usará el constructor copia por defecto. Recuerda que ese constructor copia los punteros, no los contenidos de estos.

Recapitulemos: el objeto temp se copia en un temporal sin nombre, y después se destruye, ¿qué pasa con el dato temp.cadena?, evidentemente también se destruye, pero el constructor copia por defecto ah copiado ese puntero, por lo tanto, también su cadena es destruida. El resultado es que C1 no recibe la suma de las cadenas.

Para evitar eso tenemos que sobrecargar el constructor copia, afortunadamente es sencillo ya que disponemos del operador de asignación:

```
class Cadena {
...
   Cadena(const Cadena &c) { *this = c; }
...
};
```

La moraleja es que cuando nuestras clases tengan datos miembro que sean punteros a memoria dinámica debemos sobrecarguar siempre el constructor copia, ya que nunca sabemos cuándo puede ser invocado sin que nos demos cuenta.

(Gracias a Steven por la idea de crear una clase Tiempo como ejemplo para la sobrecarga de operadores)

Sobrecarga de operadores unitarios:

Ahora le toca el turno a los operadores unitarios, que son aquellos que sólo requieren un operando, como la asignación o el incremento.

Cuando se sobrecargan operadores unitarios en una clase el operando es el propio objeto de la clase donde se define el operador. Por lo tanto los operadores unitarios dentro de las clases no requieren operandos.

Sintaxis:

```
<tipo> operator<operador unitario>();
```

Normalmente el <tipo> es la clase para la que estamos sobrecargando el operador. Sigamos con el ejemplo de la clase para el tratamiento de tiempos, sobrecargaremos ahora el operador de incremento ++:

```
class Tiempo {
...
Tiempo operator++();
...
};

Tiempo Tiempo::operator++()
{
   minuto++;
   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
   return *this;
}
...

T1.Mostrar();
++T1;
T1.Mostrar();
...
```

Operadores unitarios sufijos:

Lo que hemos visto vale para el preincremento, pero, ¿cómo se sobrecarga el operador de postincremento?

En realidad no hay forma de decirle al compilador cuál de las dos modalidades del operador estamos sobrecargando, así que los compiladores usan una regla: si se declara un parámetro para un operador ++ ó -- se sobrecargará la forma sufija del operador. El parámetro se ignorará, así que bastará con indicar el tipo.

También tenemos que tener en cuenta el peculiar funcionamiento de los operadores sufijos, cuando los sobrecarguemos, al menos si queremos mantener el comportamiento que tienen normalmente.

Cuando se usa un operador en la forma sufijo dentro de una expresión, primero se usa el valor actual del objeto, y una vez evaludada la expresión, se aplica el operador. Si nosotros queremos que nuestro operador actue igual deberemos usar un objeto temporal, y asignarle el valor actual del objeto. Seguidamente aplicamos el operador al objeto actual y finalmente retornamos el objeto temporal.

Veamos un ejemplo:

```
class Tiempo {
...
Tiempo operator++();    // Forma prefija
Tiempo operator++(int); // Forma sufija
...
};

Tiempo Tiempo::operator++()
{
   minuto++;
   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
   return *this;
}

Tiempo Tiempo::operator++(int)
{
   Tiempo temp = *this;

   minuto++;
   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
   return temp;
}
...

// Prueba:
T1.Mostrar();
(T1++).Mostrar();
T1.Mostrar();
(++T1).Mostrar();
T1.Mostrar();
...
```

Salida:

17:9 (Valor inicial)
17:9 (Operador sufijo, el valor no cambia hasta después de mostar el valor)
17:10 (Resultado de aplicar el operador)
17:11 (Operador prefijo, el valor cambia antes de mostrar el valor)
17:11 (Resultado de aplicar el operador)

Operadores unitarios que pueden sobrecargarse:

Además del operador ++ y -- pueden sobrecargarse prácticamente todos los operadores unitarios:

+, -, ++, --, *, & y !.

Operadores de conversión de tipo.

Volvamos a nuestra clase Tiempo. Imaginemos que queremos hacer una operación como la siguiente:

```
Tiempo T1(12,23);
unsigned int minutos = 432;

T1 += minutos;
```

Con toda probabilidad no obtendremos el valor deseado.

Como ya hemos visto, en C++ se realizan conversiones implícitas entre los tipos básicos antes de operar con ellos, por ejemplo para sumar un int y un float, se convierte el entero a float. Esto se hace también en nuestro caso, pero no como esperamos.

El valor "minutos" se convierte a un objeto Tiempo, usando el constructor que hemos diseñado. Como sólo hay un parámetro, el parámetro m toma el valor 0, y para el parámetro h se convierte el valor "minutos" de unsigned int a int.

El resuldado es que se suman 432 horas, y nosotros queremos sumar 432 minutos.

Esto se soluciona creando un nuevo constructor que tome como parámetro un unsigned int.

```
Tiempo(unsigned int m) : hora(0), minuto(m)
{
   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
}
```

Ahora el resultado será el adecuado.

En general podremos hacer conversiones de tipo desde cualquier objeto a un objeto de nuestra clase sobrecargando el constructor.

Pero también se puede presentar el caso contrario. Ahora queremos asiginar a un entero un objeto Tiempo:

```
Tiempo T1(12,23);
int minutos;

minutos = T1;
```

En este caso obtendremos un error de compilación, ya que el compilador no sabe convertir un objeto Tiempo a entero.

Para eso tenemos que diseñar nuestro operador de conversión de tipo, que se aplicará automáticamente.

Los operadores de conversión de tipos tienen el siguiente formato:

```
operator <tipo>();
```

No necesitan que se especifique el tipo del valor de retorno, ya que este es precisamente <tipo>. Además, al ser operadores unitarios, tampoco requieren argumentos, se aplican al propio objeto.

```
class Tiempo {
...
operator int();
...

operator int()
{
   return hora*60+minuto;
}
```

Por supuesto, el tipo no tiene por qué ser un tipo básico, puede tratarse de una estuctura o una clase.

Sobrecarga del operador de indexación []:

El operador [] se usa para acceder a valores de objetos de una determinada clase como si se tratase de arrays. Los índices no tienen por qué ser de un tipo entero o enumerado, ahora no existe esa limitación.

Donde más útil resulta este operadore es cuando se usa con estructuras dinámicas de datos: listas y árboles.

De nuevo explicaremos el uso de este operador usando un ejemplo.

Supongamos que hacemos una clase para hacer un histograma de los valores de rand()/RAND_MAX, entre los márgenes de 0 a 0.0009, de 0.001 a 0.009, de 0.01 a 0.09 y de 0.1 a 1.

Un histograma es un gráfico utilizado en la representación de distribuciones de frecuencias de cualquier tipo de información o función. La clase de nuestro ejemplo prodría usar los valores de la tabla para generar ese gráfico.

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


class Cuenta {
  public:
   Cuenta() { for(int i = 0; i < 4; contador[i++] = 0); }
   int &operator[](double n); // (1)

   void Mostrar() const;

  private:
   int contador[4];
};

int &Cuenta::operator[](double n) // (2)
```

```
{
   if(n < 0.001) return contador[0];
   else if(n < 0.01) return contador[1];
   else if(n < 0.1) return contador[2];
   else return contador[3];
}

void Cuenta::Mostrar() const
{
   cout << "Entre      0 y 0.0009: " << contador[0] << endl;
   cout << "Entre 0.0010 y 0.0099: " << contador[1] << endl;
   cout << "Entre 0.0100 y 0.0999: " << contador[2] << endl;
   cout << "Entre 0.1000 y 1.0000: " << contador[3] << endl;
}

int main()
{
   Cuenta C;

   for(int i = 0; i < 50000; i++) C[(double)rand()/RAND_MAX]++; //
(3)
   C.Mostrar();

   system("PAUSE");
   return 0;
}
```

En este ejemplo hemos usado un valor double como índice, pero igualmente podríamos haber usado una cadena o cualquier objeto que hubiésemos querido.

El tipo del valor de retorno de operador debe ser el del objeto que devuelve (1). En nuestro caso, al tratarse de un contador, devolvemos un entero. Bueno, en realidad devolvemos una referencia a un entero, de este modo podemos aplicarle el operador de incremento al valor de retorno (3).

En la definición del operador (2), hacemos un tratamiento del parámetro que usamos como índice para adaptarlo al tipo de almacenamiento que usamos en nuestra clase.

Cuando se combina el operador de indexación con estructuras dinámicas de datos como las listas, se puede trabajar con ellas como si se tratada de arrays de objetos, esto nos dará una gran potencia y claridad en el código de nuestros programas.

Sobrecarga del operador de llamada ():

El operador () funciona exactamente igual que el operador [], aunque admite más parámetros.

Este operador permite usar un objeto de la clase para el que está definido como si fuera una función.

Como ejemplo añadiremos un operador de llamada a función que admita dos parámetros de tipo double y que devuelva el mayor contador de los asociados a cada uno de los parámetros.

```
class Cuenta {
...
   int operator()(double n, double m);
...
};

int Cuenta::operator()(double n, double m)
{
   int i, j;

   if(n < 0.001) i = 0;
   else if(n < 0.01) i = 1;
   else if(n < 0.1) i = 2;
   else i = 3;

   if(m < 0.001) j = 0;
   else if(m < 0.01) j = 1;
   else if(m < 0.1) j = 2;
   else j = 3;

   if(contador[i] > contador[j]) return contador[i]; else return
contador[j];
}
...

cout << C(0.0034, 0.23) << endl;
...
```

Por supuesto, el número de parámetros, al igual que el tipo de retorno de la función depende de la decisión del programador.

CAPITULO 37 Herencia:

Jerarquía, clases base y clases derivadas:

Una de las principales propiedades de las clases es la *herencia*. Esta propiedad nos permite crear nuevas clases a partir de clases existentes, conservando las propiedades de la clase original y añadiendo otras nuevas.

La nueva clase obtenida se conoce como *clase derivada*, y las clases a partir de las cuales se deriva, *clases base*. Además, cada clase derivada puede usarse como clase base para obtener una nueva clase derivada. Y cada clase derivada puede serlo de una o más clases base. En éste último caso hablaremos de *derivación múltiple*.

Esto nos premite crear una jerarquía de clases tan compleja como sea necesario.

Bien, pero ¿que ventajas tiene derivar clases?.

En realidad, ese es el principio de la programación orientada a objetos. Esta propiedad nos permite encapsular diferentes partes de cualquier objeto real o imaginario, y vincularlo con objetos más elaborados del mismo tipo básico, que heredarán todas sus características. Lo veremos mejor con un ejemplo.

Un ejemplo muy socorrido es de las personas. Supongamos que nuestra clase base para clasificar a las personas en función de su profesión sea "Persona". Presta especial atención a la palabra "**clasificar**", es el punto de partida para buscar la solución de cualquier problema que se pretenda resolver usando POO. Lo primero que debemos hacer es buscar categorías, propiedades comunes y distintas que nos permitan clasificar los objetos, y crear lo que después serán las clases de nuestro programa. Es muy importante dedicar el tiempo y atención necesarios a esta tarea, de ello dependerá la flexibilidad, reutilización y eficacia de nuestro programa.

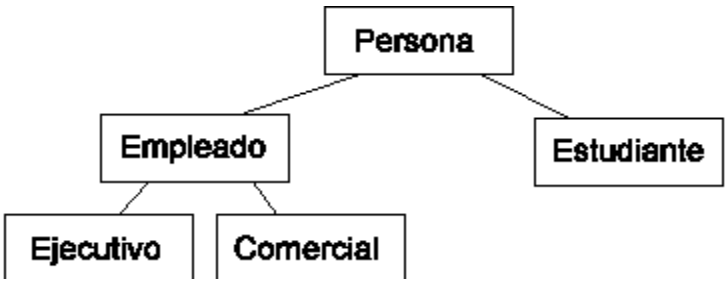
Ten en cuenta que las jerarquias de clases se usan especialmente en la resolución de problemas complejos, es difícil que tengas que recurrir a ellas para resolver problemas sencillos.

Siguiendo con el ejemplo, partiremos de la clase "Persona". Independientemente de la profesión, todas las personas tienen propiedades comunes, nombre, fecha de nacimiento, género, estado civil, etc.

La siguiente clasificación debe ser menos general, supongamos que dividimos a todas las personas en dos grandes clases: empleados y estudiantes. (Dejaremos de lado, de momento, a los estudiantes que además trabajan). Lo importante es decidir qué propiedades que no hemos incluido en la clase "Persona" son exclusivas de los empleados y de los estudiantes. Por ejemplo, los ingresos por nómina son exclusivos de los empleados, la nota media del curso, es exclusiva de los estudiantes. Una vez hecho eso crearemos dos clases derivadas de Persona: "Empleado" y "Estudiante".

Haremos una nueva clasificación, ahora de los empleados. Podemos clasificar a los empleados en ejecutivos y comerciales (y muchas más clases, pero para el ejemplo

nos limitaremos a esos dos). De nuevo estableceremos propiedades exclusivas de cada clase y crearemos dos nuevas clases derivadas de "Empleado": "Ejecutivo" y "Comercial".



Ahora veremos las ventajas de disponer de una jerarquía completa de clases.

- Cada vez que creemos un objeto de cualquier tipo derivado, por ejemplo de tipo Comercial, estaremos creando en un sólo objeto un Comercial, un Empleado y una Persona. Nuestro programa puede tratar a ese objeto como si fuera cualquiera de esos tres tipos. Es decir, nuestro comercial tendrá, además de sus propiedades como comercial, su nómina como empleado, y su nombre, edad y género como persona.
- Siempre podremos crear nuevas clases para resolver nuevas situaciones. Consideremos el caso de que en nuestra clasificación queremos incluir una nueva clase "Becario", que no es un empleado, ni tampoco un estudiante; la derivaríamos de Persona. También podemos cosiderar que un becario es ambas cosas, sería un ejemplo de derivación múltiple, podríamos hacer que la clase derivada Becario, lo fuera de Empleado y Estudiante.
- Podemos aplicar procedimientos genéricos a una clase en concreto, por ejemplo, podemos aplicar una subida general del salario a todos los empleados, independientemente de su profesión, si hemos diseñado un procedimiento en la clase Empleado para ello.

Veremos que existen más ventajas, aunque éste modo de diseñar aplicaciones tiene también sus inconvenientes, sobre todo si diseñamos mal alguna clase.

Derivar clases, sintaxis:

La forma general de declarar clases derivadas es la siguiente:

```
class <clase_derivada> : [public|private] <clase_base1>
[,[public|private] <clase_base2>] {};
```

En seguida vemos que para cada clase base podemos definir dos tipos de acceso, public o private. Si no se especifica ninguno de los dos, por defecto se asume que es private.

- public: los miembros heredados de la clase base conservan el tipo de acceso con que fueron declarados en ella.
- private: todos los miembros heredados de la clase base pasan a ser miembros privados en la clase derivada.

De momento siempre declararemos las clases base como public, al menos hasta que veamos la utilidad de hacerlo como privadas.

Veamos un ejemplo sencillo basado en la idea del punto anterior:

```
// Clase base Persona:
class Persona {
  public:
   Persona(char *n, int e);
   char *LeerNombre(char *n) const;
   int LeerEdad() const;
   void CambiarNombre(char *n);
   void CambiarEdad(int e);

  protected:
   char nombre[40];
   int edad;
};

// Clase derivada Empleado:
class Empleado : public Persona {
  public:
   Empleado(char *n, int e, float s);
   float LeerSalario() const;
   void CambiarSalario(float s);

  protected:
   float salarioAnual;
};
```

Podrás ver que hemos declarado los datos miembros de nuestras clases como protected. En general es recomendable declarar siempre los datos de nuestras clases como privados, de ese modo no son accesibles desde el exterior de la clase y además, las posibles modificaciones de esos datos, en cuanto a tipo o tamaño, sólo requieren ajustes de los métodos de la propia clase.

Pero en el caso de estructuras jerarquicas de clases puede ser interesante que las clases derivadas tengan acceso a los datos miembros de las clases base. Usar el acceso protected nos permite que los datos sean innaccesibles desde el exterior de las clases, pero a la vez, permite que sean accesibles desde las clases derivadas.

Constructores de clases derivadas:

Cuando se crea un objeto de una clase derivada, primero se invoca al constructor de la clase o clases base y a continuación al constructor de la clase derivada. Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

Lógicamente, si no hemos definido los constructores de las clases, se usan los constructores por defecto que crea el compilador.

Veamos un ejemplo:

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

```
class ClaseA {
  public:
   ClaseA() : datoA(10) {cout << "Constructor de A" << endl;}
   int LeerA() const {return datoA;}

  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB() : datoB(20) {cout << "Constructor de B" << endl;}
   int LeerB() const {return datoB;}

  protected:
   int datoB;
};

int main()
{
   ClaseB objeto;

   cout << "a = " << objeto.LeerA() << ", b = " << objeto.LeerB() <<
endl;
   system("pause");
   return 0;
}
```

La salida es ésta:

```
Constructor de A
Constructor de B
a = 10, b = 20
Presione una tecla para continuar . . .
```

Se ve claramente que primero se llama al constructor de la clase base A, y después al de la clase derivada B.

Es relativamente fácil comprender esto cuando usamos constructores por defecto o cuando nuestros constructores no tienen parámetros, pero cuando sobrecargamos los constructores o usamos constructores con parámetros, no es tan simple.

Inicialización de clases base en constructores:

Cuando queramos inicializar las clases base usando parámetros desde el constructor de una clase derivada lo haremos de modo análogo a como lo hacemos con los datos miembro, usaremos el constructor de la clase base con los parámetros adecuados. Las llamadas a los constructores deben escribirse antes de las inicializaciones de los parámetros.

Sintaxis:

```
<clase_derivada>(<lista_de_parámetros>) :
<clase_base>(<lista_de_parámetros>) {}
```

De nuevo lo veremos mejor con otro ejemplo:

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

class ClaseA {
  public:
   ClaseA(int a) : datoA(a) {cout << "Constructor de A" << endl;}
   int LeerA() const {return datoA;}

  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB(int a, int b) : ClaseA(a), datoB(b) {cout << "Constructor
de B" << endl;} (1)
   int LeerB() const {return datoB;}

  protected:
   int datoB;
};

int main()
{
   ClaseB objeto(5,15);

   cout << "a = " << objeto.LeerA() << ", b = " << objeto.LeerB() <<
endl;
   system("pause");
   return 0;
}
```

La salida es esta:

```
Constructor de A
Constructor de B
a = 5, b = 15
Presione una tecla para continuar . . .
```

Observa cómo hemos definido el constructor de la ClaseB (1). Para empezar, recibe dos parámetros: "a" y "b". El primero se usará para inicializar la clase base ClaseA, para ello, después de los dos puntos, escribimos el constructor de la ClaseA, y usamos como parámetro el valor "a". A continuación escribimos la inicialización del datoB, separado con una coma y usamos el valor "b".

Inicialización de objetos miembros de clases:

También es posible que una clase tenga como miembros objetos de otras clases, en ese caso, para inicializar esos miembros se procede del mismo modo que con cualquier dato miembro, es decir, se añade el nombre del objeto junto con sus parámetros a la lista de inicializaciones del constructor.

Esto es válido tanto en clases base como en clases derivadas.

Veamos un ejemplo:

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

class ClaseA {
  public:
   ClaseA(int a) : datoA(a) {cout << "Constructor de A" << endl;}
   int LeerA() const {return datoA;}

  protected:
   int datoA;
};

class ClaseB {
  public:
   ClaseB(int a, int b) : cA(a), datoB(b) {cout << "Constructor de
B" << endl;} (1)
   int LeerB() const {return datoB;}
   int LeerA() const {return cA.LeerA();} (2)

  protected:
   int datoB;
   ClaseA cA;
};

int main()
{
   ClaseB objeto(5,15);

   cout << "a = " << objeto.LeerA() << ", b = " << objeto.LeerB() <<
endl;
   system("pause");
   return 0;
}
```

En la línea (1) se ve cómo inicializamos el objeto de la ClaseA (cA), que hemos incluido dentro de la ClaseB.

En la línea (2) vemos que hemos tenido que añadir una nueva función para que sea posible acceder a los datos del objeto cA. Si hubiéramos declarado cA como public, este paso no habría sido necesario.

Sobrecarga de constructores de clases derivadas:

Por supuesto, los constructores de las clases derivadas también pueden sobrecargarse, podemos crear distintos constructores para diferentes inicializaciones posibles, y también usar parámetros con valores por defecto.

Destructores de clases derivadas:

Cuando se destruye un objeto de una clase derivada, primero se invoca al destructor de la clase derivada, si existen objetos miembro a continuación se invoca a sus destructores y finalmente al destructor de la clase o clases base. Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

Al igual que pasaba con los constructores, si no hemos definido los destructores de las clases, se usan los destructores por defecto que crea el compilador.

Veamos un ejemplo:

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

class ClaseA {
  public:
   ClaseA() : datoA(10) {cout << "Constructor de A" << endl;}
   ~ClaseA() {cout << "Destructor de A" << endl;}
   int LeerA() const {return datoA;}

  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB() : datoB(20) {cout << "Constructor de B" << endl;}
   ~ClaseB() {cout << "Destructor de B" << endl;}
   int LeerB() const {return datoB;}

  protected:
   int datoB;
};

int main()
{
   ClaseB objeto;

   cout << "a = " << objeto.LeerA() << ", b = " << objeto.LeerB() <<
endl;
   system("pause");
   return 0;
}
```

La salida es esta:

```
Constructor de A
Constructor de B
a = 10, b = 20
Presione una tecla para continuar . . .
Destructor de B
Destructor de A
```

Se ve que primero se llama al destructor de la clase derivada B, y después al de la clase base A.

CAPITULO 38 Funciones virtuales:

Redefinición de funciones en clases derivadas:

En una clase derivada se puede definir una función que ya existía en la clase base, esto se conoce como "overriding", o superposición de una función.

La definición de la función en la clase derivada oculta la definición previa en la clase base.

En caso necesario, es posible acceder a la función oculta de la clase base mediante su nombre completo:

```
<objeto>.<clase_base>::<método>;
```

Veamos un ejemplo:

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

class ClaseA {
  public:
   ClaseA() : datoA(10) {}
   int LeerA() const {return datoA;}
   void Mostrar() {cout << "a = " << datoA << endl;} (1)

  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB() : datoB(20) {}
   int LeerB() const {return datoB;}
   void Mostrar() {cout << "a = " << datoA << ", b = " << datoB <<
endl;} (2)

  protected:
   int datoB;
};

int main()
{
   ClaseB objeto;

   objeto.Mostrar();
   objeto.ClaseA::Mostrar();
   system("pause");
   return 0;
}
```

La salida de este programa es:

```
a = 10, b = 20
a = 10
```

```
Presione cualquier tecla para continuar . . .
```

La definición de la funciónn "Mostrar" en la ClaseB (1) oculta la definición previa de la función en la ClaseA (2).

Superposición y sobrecarga:

Cuando se superpone una función, se ocultan todas las funciones con el mismo nombre en la clase base.

Supongamos que hemos sobrecargado la función de la clase base que después volveremos a definir en la clase derivada.

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

class ClaseA {
  public:
   void Incrementar() {cout << "Suma 1" << endl;}
   void Incrementar(int n) {cout << "Suma " << n << endl;}
};

class ClaseB : public ClaseA {
  public:
   void Incrementar() {cout << "Suma 2" << endl;}
};

int main(int argc, char *argv[])
{
   ClaseB objeto;

   objeto.Incrementar();
//   objeto.Incrementar(10);
   objeto.ClaseA::Incrementar();
   objeto.ClaseA::Incrementar(10);
   system("pause");
   return 0;
}
```

La salida sería:

```
Suma 2
Suma 1
Suma 10
Presione cualquier tecla para continuar . . .
```

Ahora bien, no es posible acceder a ninguna de las funciones superpuestas de la clase base, aunque tengan distintos valores de retorno o distinto número o tipo de parámetros. Todas las funciones "incrementar" de la clase base han quedado ocultas, y sólo son accesibles mediante el nombre completo.

Polimorfismo:

Por fin vamos a introducir un concepto muy importante de la programación orientada a objetos: *el polimorfismo*.

En lo que concierne a clases, el polimorfismo en C++, llega a su máxima expresión cuando las usamos junto con los punteros.

C++ nos permite acceder a objetos de una clase derivada usando un puntero a la clase base. En eso consiste el polimorfismo. Por supuesto, sólo podremos acceder a datos y funciones que existan en la clase base, los datos y funciones propias de los objetos de clases derivadas serán innacesibles.

Volvamos al ejemplo inicial, el de la estructura de clases basado en la clase "Persona" y supongamos que tenemos la clase base "Persona" y dos clases derivadas: "Empleado" y "Estudiante".

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

class Persona {
  public:
   Persona(char *n) { strcpy(nombre, n); }
   char *VerNombre(char *n) { strcpy(n, nombre); return n;}
  protected:
   char nombre[30];
};

class Empleado : public Persona {
  public:
   Empleado(char *n) : Persona(n) {}
   char *VerNombre(char *n) { strcpy(n, "Emp: "); strcat(n, nombre);
return n;}
};

class Estudiante : public Persona {
  public:
   Estudiante(char *n) : Persona(n) {}
   char *VerNombre(char *n) { strcpy(n, "Est: "); strcat(n, nombre);
return n;}
};

int main()
{
   char n[40];
   Persona *Pepito = new Estudiante("Jose");
   Persona *Carlos = new Empleado("Carlos");

   cout << Carlos->VerNombre(n) << endl;
   cout << Pepito->VerNombre(n) << endl;
   delete Pepito;
   delete Carlos;
   system("pause");
   return 0;
}
```

La salida es como ésta:

```
Carlos
Jose
Presione cualquier tecla para continuar . . .
```

Vemos que se ejecuta la versión de la función "VerNombre" que hemos definido para la clase base.

Funciones virtuales:

El ejemplo anterior demuestra algunas de las posibilidades del polimorfismo, pero tal vez sería mucho más interesante que cuando se invoque a una función que se superpone en la clase derivada, se llame a ésta última función, la de la clase derivada.

En nuestro ejemplo, podemos preferir que al llamar a la función "VerNombre" se ejecute la versión de la clase derivada en lugar de la de la clase base.

Esto se consigue mediante el uso de funciones virtuales. Cuando en una clase declaramos una función como virtual, y la superponemos en alguna clase derivada, al invocarla usando un puntero de la clase base, se ejecutará la versión de la clase derivada.

Sintaxis:

```
virtual <tipo> <nombre_función>(<lista_parámetros>) [{}];
```

Modifiquemos en el ejemplo anterior la declaración de la clase base "Persona".

```
class Persona {
  public:
   Persona(char *n) { strcpy(nombre, n); }
   virtual char *VerNombre(char *n) { strcpy(n, nombre); return n;}
  protected:
   char nombre[30];
};
```

Ahora ejecutemos el programa de nuevo, veremos que la salida es ahora diferente:

```
Emp: Carlos
Est: Jose
Presione cualquier tecla para continuar . . .
```

Ahora, al llamar a "Pepito->VerNombre(n)" se invoca a la función "VerNombre" de la clase "Estudiante", y al llamar a "Carlos->VerNombre(n)" se invoca a la función de la clase "Empleado".

Una vez que una función es declarada como virtual, lo seguirá siendo en las clases derivadas, es decir, la propiedad virtual se hereda.

Si la función virtual no se define exactamente con el mismo tipo de valor de retorno y el mismo número y tipo de parámetros que en la clase base, no se considerará como la misma función, sino como una función superpuesta.

Este mecanismo sólo funciona con punteros y referencias, usarlo con objetos no tiene sentido.

Destructores virtuales:

Supongamos que tenemos una estructura de clases en la que en alguna de las clases derivadas exista un destructor. Un destructor es una función como las demás, por lo tanto, si destruimos un objeto referenciado mediante un puntero a la clase base, y el destructor no es virtual, estaremos llamando al destructor de la clase base. Esto puede ser desastroso, ya que nuestra clase derivada puede tener más tareas que realizar en su destructor que la clase base de la que procede.

Por lo tanto debemos respetar siempre ésta regla: **si en una clase existen funciones virtuales, el destructor debe ser virtual**.

Constructores virtuales:

Los constructores no pueden ser virtuales. Esto puede ser un problema en ciertas ocasiones. Por ejemplo, el constructor copia no hará siempre aquello que esperamos que haga. En general no debemos usar el constructor copia cuando usemos punteros a clases base. Para solucionar éste inconveniente se suele crear una función virtual "clonar" en la clase base que se superpondrá para cada clase derivada.

Por ejemplo:

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

class Persona {
  public:
   Persona(char *n) { strcpy(nombre, n); }
   Persona(const Persona &p);
   virtual char *VerNombre(char *n) { strcpy(n, nombre); return n;}
   virtual Persona* Clonar() { return new Persona(*this);}
  protected:
   char nombre[30];
};

Persona::Persona(const Persona &p)
{
   strcpy(nombre, p.nombre);
   cout << "Per: constructor copia." << endl;
}

class Empleado : public Persona {
  public:
   Empleado(char *n) : Persona(n) {}
   Empleado(const Empleado &e);
```

```
   char *VerNombre(char *n) { strcpy(n, "Emp: "); strcat(n, nombre);
return n;}
   virtual Persona* Clonar() { return new Empleado(*this);}
};

Empleado::Empleado(const Empleado &e) : Persona(e)
{
   cout << "Emp: constructor copia." << endl;
}

class Estudiante : public Persona {
  public:
   Estudiante(char *n) : Persona(n) {}
   Estudiante(const Estudiante &e);
   char *VerNombre(char *n) { strcpy(n, "Est: "); strcat(n, nombre);
return n;}
   virtual Persona* Clonar() { return new Estudiante(*this);}
};

Estudiante::Estudiante(const Estudiante &e) : Persona(e)
{
   cout << "Est: constructor copia." << endl;
}

int main()
{
   char n[40];
   Persona *Pepito = new Estudiante("Jose");
   Persona *Carlos = new Empleado("Carlos");
   Persona *Gente[2];

   cout << Carlos->VerNombre(n) << endl;
   cout << Pepito->VerNombre(n) << endl;

   Gente[0] = Carlos->Clonar();
   cout << Gente[0]->VerNombre(n) << endl;

   Gente[1] = Pepito->Clonar();
   cout << Gente[1]->VerNombre(n) << endl;

   delete Pepito;
   delete Carlos;
   delete Gente[0];
   delete Gente[1];
   system("pause");
   return 0;
}
```

Hemos definido el constructor copia para que se pueda ver cuando es invocado. La salida es ésta:

```
Emp: Carlos
Est: Jose
Per: constructor copia.
Emp: constructor copia.
Emp: Carlos
Per: constructor copia.
Est: constructor copia.
Est: Jose
Presione una tecla para continuar . . .
```

Este método asegura que siempre se llama al constructor copia adecuado, ya que se hace desde una función virtual.

Si un constructor llama a una función virtual, ésta será siempre la de la clase base. Esto es debido a que el objeto de la clase derivada aún no ha sido creado.

CAPITULO 39 Derivación múltiple:

C++ permite crear clases derivadas a partir de varias clases base. Este proceso se conoce como derivación múltiple. Los objetos creados a partir de las clases así obtenidas, heredarán los datos y funciones de todas las clase base.

Sintaxis:

```
<clase_derivada>(<lista_de_parámetros>) :
   <clase_base1>(<lista_de_parámetros>)[,
<clase_base2>(<lista_de_parámetros>)] {}
```

Pero esto puede producir algunos problemas. En ocasiones puede suceder que en las dos clases base exista una función con el mismo nombre. Esto crea una ambigüedad cuando se invoca a una de esas funciones.

Veamos un ejemplo:

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

class ClaseA {
   public:
     ClaseA() : valorA(10) {}
     int LeerValor() {return valorA;}
   protected:
     int valorA;
};

class ClaseB {
   public:
     ClaseB() : valorB(20) {}
     int LeerValor() {return valorB;}
   protected:
     int valorB;
};

class ClaseC : public ClaseA, public ClaseB {};

int main()
{
   ClaseC CC;

   // cout << CC.LeerValor() << endl;
   // Produce un error de compilación ya que hay ambigüedad.
   cout << CC.ClaseA::LeerValor() << endl;
   system("pause");
   return 0;
}
```

Una solución para resolver la ambigüedad es la que hemos adoptado en el ejemplo. Pero existe otra, también podríamos haber redefinido la función "LeerValor" en la clase derivada de modo que se superpusiese a las funciones de las clases base.

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

```
class ClaseA {
   public:
     ClaseA() : valorA(10) {}
     int LeerValor() {return valorA;}
   protected:
     int valorA;
};

class ClaseB {
   public:
     ClaseB() : valorB(20) {}
     int LeerValor() {return valorB;}
   protected:
     int valorB;
};

class ClaseC : public ClaseA, public ClaseB {
   public:
    int LeerValor() {return ClaseA::LeerValor();}
};

int main()
{
   ClaseC CC;

   cout << CC.LeerValor() << endl;
   system("pause");
   return 0;
}
```

Constructores de clases con herencia múltiple:

Análogamente a lo que sucedía con la derivación simple, en el caso de derivación múltiple el constructor de la clase derivada deberá llamar a los constructores de las clases base cuando sea necesario. Por ejemplo, añadiremos constructores al ejemplo anterior:

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

class ClaseA {
   public:
     ClaseA() : valorA(10) {}
     ClaseA(int va) : valorA(va) {}
     int LeerValor() {return valorA;}
   protected:
     int valorA;
};

class ClaseB {
   public:
     ClaseB() : valorB(20) {}
     ClaseB(int vb) : valorB(vb) {}
     int LeerValor() {return valorB;}
   protected:
     int valorB;
};
```

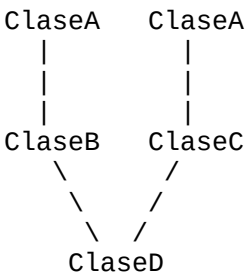
```
class ClaseC : public ClaseA, public ClaseB {
   public:
     ClaseC(int va, int vb) : ClaseA(va), ClaseB(vb) {};
     int LeerValorA() {return ClaseA::LeerValor();}
     int LeerValorB() {return ClaseB::LeerValor();}
};

int main()
{
   ClaseC CC(12,14);

   cout << CC.LeerValorA() << "," << CC.LeerValorB() << endl;
   system("pause");
   return 0;
}
```

Sintaxis:

```
<clase_derivada>(<lista_parámetros> :
   <clase_base1>(<lista_parámetros>) [,
<clase_base2>(<lista_parámetros>)] {}
```

Herencia virtual:

Supongamos que tenemos una estructura de clases como ésta:



La ClaseD heredará dos veces los datos y funciones de la ClaseA, con la consiguiente ambigüedad a la hora de acceder a datos o funciones heredadas de ClaseA.

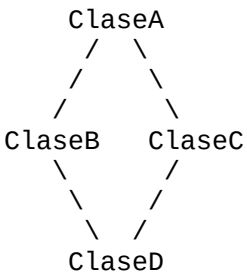
Para solucionar esto se usan las clases virtuales. Cuando derivemos una clase partiendo de una o varias clases base, podemos hacer que las clases base sean virtuales. Esto no afectará a la clase derivada. Por ejemplo:

```
class ClaseB : virtual public ClaseA {};
```

Desde el punto de vista de la ClaseB, no hay ninguna diferencia entre ésta declaración y la que hemos usado hasta ahora. La diferencia estará cuando declaramos la ClaseD. Veamos el ejemplo completo:

```
class ClaseB : virtual public ClaseA {};
class ClaseC : virtual public ClaseA {};
class ClaseD : public ClaseB, public ClaseC {};
```

Ahora, la ClaseD sólo heredará una vez la ClaseA. La estructura quedará así:



Cuando creemos una estructura de éste tipo, deberemos tener cuidado con los constructores, el constructor de la ClaseA deberá ser invocado desde el de la ClaseD, ya que ni la ClaseB ni la ClaseC lo harán automáticamente.

Veamos esto con el ejemplo de la clase "Persona". Derivaremos las clases "Empleado" y "Estudiante", y crearemos una nueva clase "Becario" derivada de estas dos últimas. Además haremos que la clase "Persona" sea virtual, de modo que no se dupliquen sus funciones y datos.

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

class Persona {
   public:
     Persona(char *n) {strcpy(nombre, n);}
     char *LeeNombre(char *n) {strcpy(n, nombre); return n;}
   protected:
     char nombre[30];
};

class Empleado : virtual public Persona {
   public:
     Empleado(char *n, int s) : Persona(n), salario(s) {}
     int LeeSalario() {return salario;}
     void ModificaSalario(int s) {salario = s;}
   protected:
     int salario;
};

class Estudiante : virtual public Persona {
   public:
     Estudiante(char *n, float no) : Persona(n), nota(no) {}
     float LeeNota() {return nota;}
     void ModificaNota(float no) {nota = no;}
   protected:
     float nota;
};

class Becario : public Empleado, public Estudiante {
   public:
     Becario(char *n, int s, float no) :
        Empleado(n, s), Estudiante(n, no), Persona(n) {} (1)
};

int main()
{
   Becario Fulanito("Fulano", 1000, 7);
   char n[30];
```

```
   cout << Fulanito.LeeNombre(n) << "," << Fulanito.LeeSalario() <<
      "," << Fulanito.LeeNota() << endl;
   system("pause");
   return 0;
}
```

Si observamos el constructor de "Becario" en (1), veremos que es necesario usar el constructor de "Persona", a pesar de que el nombre se pasa como parámetro tanto a "Empleado" como a "Estudiante". Si no se incluye el constructor de "Persona", el compilador genera un error.

Funciones virtuales puras:

Una función virtual pura es aquella que no necesita ser definida. En ocasiones esto puede ser útil, como se verá en el siguiente punto.

El modo de declarar una función virtual pura es asignándole el valor cero.

Sintaxis:

```
virtual <tipo> <nombre_función>(<lista_parámetros>) = 0;
```

Clases abstractas:

Una clase abstracta es aquella que posee al menos una función virtual pura.

No es posible crear objetos de una clase abstracta, estas clases sólo se usan como clases base para la declaración de clases derivadas.

Las funciones virtuales puras serán aquellas que siempre se definirán en las clases derivadas, de modo que no será necesario definirlas en la clase base.

A menudo se mencionan las clases abstractas como tipos de datos abstractos, en inglés: Abstract Data Type, o resumido ADT.

Hay varias reglas a tener en cuenta con las clases abstractas:

- No está permitido crear objetos de una clase abstracta.

- Siempre hay que definir todas las funciones virtuales de una clase abstracta en sus clases derivadas, no hacerlo así implica que la nueva clase derivada será también abstracta.

Para crear un ejemplo de clases abstractas, recurriremos de nuevo a nuestra clase "Persona". Haremos que ésta clase sea abstracta. De hecho, en nuestros programas de ejemplo nunca hemos declarado un objeto "Persona". Veamos un ejemplo:

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

```
class Persona {
   public:
     Persona(char *n) {strcpy(nombre, n);}
     virtual void Mostrar() = 0;
   protected:
     char nombre[30];
};

class Empleado : public Persona {
   public:
     Empleado(char *n, int s) : Persona(n), salario(s) {}
     void Mostrar();
     int LeeSalario() {return salario;}
     void ModificaSalario(int s) {salario = s;}
   protected:
     int salario;
};

void Empleado::Mostrar()
{
    cout << "Empleado: " << nombre << ", Salario: " << salario <<
endl;
}

class Estudiante : public Persona {
   public:
     Estudiante(char *n, float no) : Persona(n), nota(no) {}
     void Mostrar();
     float LeeNota() {return nota;}
     void ModificaNota(float no) {nota = no;}
   protected:
     float nota;
};

void Estudiante::Mostrar()
{
    cout << "Estudiante: " << nombre << ", Nota: " << nota << endl;
}

int main()
{
   Persona *Pepito = new Empleado("Jose", 1000); (1)
   Persona *Pablito = new Estudiante("Pablo", 7.56);
   char n[30];

   Pepito->Mostrar();
   Pablito->Mostrar();
   system("pause");
   return 0;
}
```

La salida será así:

```
Empleado: Jose, Salario: 1000
Estudiante: Pablo, Nota: 7.56
Presione cualquier tecla para continuar . . .
```

En éste ejemplo combinamos el uso de funciones virtuales puras con polimorfismo. Fíjate que, aunque hayamos declarado los objetos "Pepito" y "Pablito" de tipo puntero

a "Persona" (1), en realidad no creamos objetos de ese tipo, sino de los tipos "Empleado" y "Estudiante"

Uso de derivación múltiple:

Una de las aplicaciones de la derivación múltiple es la de crear clases para determinadas capacidades o funcionalidades. Estas clases se incluirán en la derivación de nuevas clases que deban tener dicha capacidad.

Por ejemplo, supongamos que nuestro programa maneja diversos tipos de objetos, de los cuales algunos son visibles y otros audibles, incluso puede haber objetos que tengan las dos propiedades. Podríamos crear clases base para visualizar objetos y para escucharlos, y derivaríamos los objetos visibles de las clases base que sea necesario y además de la clase para la visualización. Análogamente con las clases para objetos audibles.