

Curso de C++

Programación orientada a objetos



Salvador Pozo

<http://conclase.net>

Introducción

Si no he perdido la cuenta, esta es la cuarta revisión del curso desde que se empezó a escribir.

Ya en su versión anterior, el curso estaba bastante completo, al menos en lo que respecta a la teoría, quedando muy pocos detalles por incluir. Esta versión se centra, sobre todo, en añadir más ejemplos resueltos, problemas propuestos y ejercicios.

Espero que este curso anime a los nuevos y futuros programadores autodidactas a incorporarse a esta gran y potente herramienta que es el C++, ese era el objetivo original de la página "Con Clase" y todavía sigue siéndolo.

No he pretendido ser original, (al menos no demasiado), para elaborar este curso se han consultado libros, tutoriales, revistas, listas de correo, news, páginas web... En fin, cualquier fuente de datos que ha caído en mis manos, con el fin de conseguir un buen nivel. Espero haber conseguido mi objetivo, y seguiré completando explicaciones sobre todo aquello que lo requiera. También deseo que haya resultado ser un texto ameno, me gustaría que nadie se aburra leyendo el curso.

Pretendo también (y me gustaría muchísimo), que el curso siga siendo interactivo. Con este fin, en esta versión del curso, se ha añadido la posibilidad de que los lectores añadan sus comentarios al final de cada capítulo. Estos aportes se usarán para completar el curso.

He intentado que los ejemplos que ilustran cada capítulo se puedan compilar con cualquier versión de compilador, sin embargo, he de decir que yo he usado el compilador [MinGW](#), (Minimalist GNU for Windows), que es una versión para Windows del compilador [GCC](#) para Unix y Linux, y que está adaptado para crear programas en Windows. Es decir, los programas que se ajusten al estándar de

C++ deberían funcionar con este compilador tanto en Windows como en Linux.

Por comodidad, recomiendo usar algún IDE (Entorno de Desarrollo Integrado), como [Dev-C++ de Bloodshed](#) o [Code::Blocks](#) para crear programas en modo consola.

De modo que 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 bibliotecas externas no estándar.
- El otro modo es el GUI, o Interfaz Gráfico de Usuario. Es el modo tradicional de los programas de Windows, con ventanas, menús, iconos, etc. La creación de este tipo de programas se explica en otro curso de este mismo sitio, y requiere el conocimiento de la biblioteca 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 entorno 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 bibliotecas 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 lugar 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 las 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 este es un buen momento 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.

Interpretes y compiladores

Antes, mencionar que tanto C como C++ son lenguajes compilados, y no interpretados. Esta diferencia es muy importante, ya que afecta mucho a muchos aspectos relacionados con la ejecución del programa.

En un lenguaje interpretado, el programa está escrito en forma de texto, es el propio programa fuente. Este programa fuente es procesado por un programa externo, el intérprete, que traduce el programa, instrucción a instrucción, al tiempo que lo ejecuta.

En los lenguajes interpretados no existen programas ejecutables directamente por el ordenador. El intérprete traduce, en tiempo real, cada línea del programa fuente, cada vez que se quiere ejecutar el programa.

En los lenguajes compilados el proceso de traducción sólo se hace una vez. El programa compilador toma como entrada el código fuente del programa, y da como salida un fichero que puede ser ejecutado por el ordenador directamente.

Una vez compilado, el programa ejecutable es autónomo, y ya no es necesario disponer del programa original ni del compilador para ejecutarlo.

Cada opción tiene sus ventajas e inconvenientes, y algunas características que son consideradas una ventaja, pueden ser un inconveniente en ciertas circunstancias, y viceversa.

- Los lenguajes interpretados son fácilmente modificables, ya que necesitamos tener el código fuente disponible en el ordenador. En los compilados, estos ficheros no son necesarios, una vez compilados.
- Los lenguajes interpretados necesitan un programa externo, llamado intérprete o a veces máquina virtual, o framework. Este programa actúa como intermediario entre el fuente y el sistema operativo. En los compilados ese papel lo desempeña el compilador, pero al contrario que con el intérprete, una vez ha hecho su trabajo, no es necesario que esté presente para ejecutar el programa.
- Estas dos características, lógicamente, hacen que los programas compilados requieran menos espacio de memoria que los interpretados (si contamos el espacio usado por el intérprete), y en general, los compilados son más rápidos, ya que sólo se compilan una vez, y el tiempo dedicado a esa tarea no se suma al de ejecución.

Entre los lenguajes interpretados están: BASIC (Código de instrucciones de propósito general para principiantes), Java, PHP. Muchos lenguajes de script, etc.

Entre los lenguajes compilados están: C, C++, Pascal.

Ficheros objeto, código objeto y compiladores

Como hemos dicho antes, en los lenguajes compilados, 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, este fichero se conoce como fichero objeto, y su contenido como código objeto.

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

El código objeto no suele tener ningún significado para los seres humanos, al menos para la mayoría de los humanos que conozco, y menos 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 o bibliotecas

Junto con los compiladores de C y C++, se incluyen ciertos ficheros llamados bibliotecas. Las bibliotecas 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 bibliotecas están clasificadas por el tipo de trabajos que hacen, hay bibliotecas de entrada y salida, matemáticas, de manejo de memoria, de manejo de textos, etc.

Nota: Existe una discusión permanente sobre el nombre genérico de estos ficheros. Una gran parte de personas consideran que el nombre adecuado es ficheros de *biblioteca*, y he de decir que esencialmente estoy de acuerdo con ellos. Sin embargo, la mayoría llamamos a estos ficheros *librerías*, y también me incluyo entre estos. El equívoco proviene del nombre en inglés, que es ficheros *library*. Este término se traduce como biblioteca, y no como librería,

que es la palabra en español más parecida fonéticamente. Sin embargo esta discusión es poco importante, desde nuestro punto de vista, ya que nos entendemos igualmente con las dos palabras.

Hay un conjunto de bibliotecas (o librerías) muy especiales, que se incluyen con todos los compiladores de C y de C++. Son las librerías (o bibliotecas) ANSI o estándar. Pero también las hay no estándar, y dentro de estas las hay públicas y comerciales. En este curso sólo usaremos bibliotecas (o 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 bibliotecas externas, ya sean ANSI o no. Es necesario combinar nuestro fichero objeto con esas bibliotecas 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 bibliotecas 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.
4. No siempre obtendremos un fichero ejecutable para el código que escribimos, a veces querremos crear ficheros de biblioteca, y en ese caso el proceso será diferente.

Existe un programa que hace todas estas cosas, se trata del "linker", o enlazador. El enlazador toma todos los ficheros objeto que

componen nuestro programa, los combina con los ficheros de biblioteca que sean necesarios 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 errores, ya que ante un aviso el compilador tiene tomar decisiones, y estas no tienen por qué coincidir con lo que nosotros pretendemos hacer, ya se basan en las directivas que los creadores del compilador decidieron durante la creación del compilador.

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 bibliotecas. Puede que hayamos olvidado incluir alguna biblioteca, 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 (instrucción a instrucción). 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 este 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 bibliotecas

escritas en C. Nada te impide a ti, como programador, usar bibliotecas, e incluso crear las tuyas propias.

Una de las propiedades de C y C++ es la reutilización del código en forma de bibliotecas de usuario. Después de un tiempo trabajando, todos los programadores desarrollan sus propias bibliotecas 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 es que 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 bibliotecas de todo propósito y miles de programadores en todo el mundo, muchos de ellos dispuestos a compartir su experiencia y conocimientos.

1 Toma de contacto

Me parece que la forma más rápida e interesante de empezar, y no perder potenciales seguidores de este curso, es mediante un ejemplo. Veamos nuestro primer programa C++. Esto nos ayudará a establecer 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. Y aunque eso es otro tema, podemos decir ahora que 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 y C++ toman unos valores de entrada, llamados parámetros o argumentos, y devuelven un valor salida o retorno. La primera palabra: "int", nos dice el tipo del valor de

retorno de la función, en este caso un número entero (*integer*). 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.

Podemos considerar una función como una *caja* que procesa ciertos datos de entrada para dar como retorno ciertos datos de salida.

C++ se basa en gran parte en C, y C fue creado en la época de los lenguajes procedimentales y está orientado a la programación estructurada. Por lo tanto, C++ tiene también características válidas para la programación estructurada.

La programación estructurada parte de la idea de que los programas se ejecutan secuencialmente, línea a línea, sin saltos entre partes diferentes del programa, con un único punto de entrada y un punto de salida.

Pero si ese tipo de programación se basase sólo en esa premisa, no sería demasiado útil, ya que los programas serían poco manejables llegados a un cierto nivel de complejidad.

La solución es crear funciones o procedimientos, que se usan para realizar ciertas tareas concretas y/o repetitivas.

Por ejemplo, si frecuentemente necesitamos mostrar un texto en pantalla, es mucho más lógico agrupar las instrucciones necesarias para hacerlo en una función, y usar la función como si fuese una instrucción cada vez que queramos mostrar un texto en pantalla.

La diferencia entre una función y un procedimiento está en si devuelven valores cada vez que son invocados. Las funciones devuelven valores, y los procedimientos no.

Lenguajes como **Pascal** hacen distinciones entre funciones y procedimientos, pero C y C++ no existe esa diferencia. En éstos sólo existen funciones y para crear un procedimiento se usa una función que devuelva un valor vacío.

Llamar o invocar una función es ejecutarla, la secuencia del programa continúa en el interior de la función, que también se

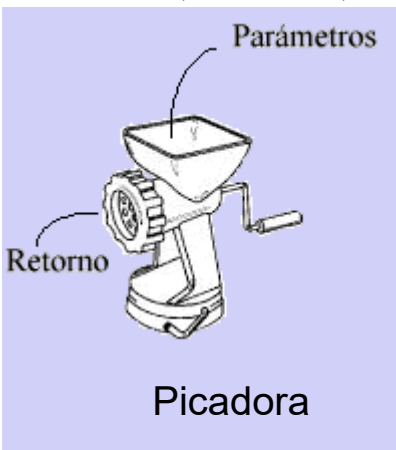
ejecuta secuencialmente, y cuando termina, se regresa a la instrucción siguiente al punto de llamada.

Las funciones a su vez, pueden invocar a otras funciones.

De este modo, considerando la llamada a una función como una única instrucción (o sentencia), el programa sigue siendo secuencial.

En este caso "main" es una función muy especial, ya que nosotros no la usaremos nunca explícitamente, 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 el sistema operativo ejecute nuestro programa.

Otra pista por la que sabemos que "main" es 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.



La picadora de nuestro dibujo admite ciertos *parámetros* de entrada, por ejemplo, carne. También proporciona valores de salida o *retorno*, si introducimos carne saldrá carne picada, si se trata de queso, la salida será queso picado, etc.

Aunque dedicaremos varios capítulos a las funciones, podemos contar ahora algo más sobre parámetros y valores de retorno.

No siempre se proporcionan parámetros a las funciones, o a veces no se proporcionan parámetros evidentes. Por ejemplo, una función que lea una tecla no necesita ningún parámetro de entrada, aunque muchos parámetros queden implícitos en la propia función.

Del mismo modo, también existen funciones que no dan una salida, al menos no una salida evidente. Por ejemplo, una función que espere un tiempo determinado. En este caso no hay salida, sólo transcurrirá un lapso de tiempo.

- 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, a la que llamaremos "numero" de tipo *integer* (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, un valor indeterminado e impredecible, que dependerá del contenido de esa zona de memoria en el momento en que se reservó el espacio. En C++ es obligatorio declarar las variables que se usarán en el programa.

Nota importante: C y C++ distinguen entre mayúsculas y minúsculas, así que `int numero;` es distinto de `int NUMERO;`, y también de `INT numero;` o de `int NuMeRo;`.

- Cuarta línea: `""`

Una línea vacía. Esta línea no sirve para nada, al menos desde el punto de vista del compilador, pero ayuda 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 evaluar la expresión correspondiente a la operación "2 + 2".

- Sexta línea: `"return 0;"`

De nuevo una sentencia, "return" es una palabra reservada, propia de 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, recordemos que la función "main" debe retornar un valor *integer*, entero. Por convenio, 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. De modo que no te preocupes si decides probarlo con un compilador y no pasa nada, es normal. 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.

2 Tipos de variables I

Conceptualmente, desde el punto de vista de un programador, una variable es una entidad cuyo valor puede cambiar a lo largo de la ejecución de un programa.

En el nivel más bajo, una variable se almacena en la memoria del ordenador. Esa memoria puede ser un conjunto de semiconductores dentro de un circuito integrado, ciertos campos magnéticos sobre una superficie de un disco, ciertas polarizaciones en una memoria de ferrita, o cualquier cosa que aún no se haya inventado. Afortunadamente, no deberemos preocuparnos por esos detalles.



En un nivel más lógico, una variable ocupa un espacio de memoria reservado en el ordenador para contener sus valores durante la ejecución de un programa. Cada variable debe pertenecer a un tipo determinado, y ese tipo determina, por una parte, el tamaño del espacio de memoria ocupado por la variable, y por otra, el modo en que se manipulará esa memoria por el ordenador.

No olvides, si es que ya lo sabías, que la información en el interior de la memoria del ordenador se almacena siempre de forma binaria, al menos a bajo nivel. El modo en que se interpreta la información almacenada en la memoria de un ordenador es, en cierto modo, arbitraria; es decir, el mismo valor puede codificar una letra, un número, una instrucción de programa, etc. No hay nada diferente en una posición de memoria que contenga una instrucción de programa o una letra de un texto; si observamos una posición de

memoria cualquiera, no habrá modo de saber qué significa el valor que contiene. Es mediante el tipo como le decimos al compilador el modo en que debe interpretarse y manipularse cierta 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 pueden ser: **short**, **long**, **signed** y **unsigned**, y pueden combinarse algunos de ellos. También veremos en este capítulo los tipos enumerados, **enum**.

Sobre la sintaxis

A partir de ahora mostraremos definiciones de la sintaxis para las diferentes sentencias en C++.

Estas definiciones nos permiten conocer las diferentes opciones para cada tipo de sentencia, las partes obligatorias, las opcionales, dónde colocar los identificadores, etc.

En este curso las definiciones de sintaxis se escribirán usando un rectángulo verde. Las partes opcionales se colocan entre corchetes [], los valores separados con | indican que sólo puede escogerse uno de esos valores. Los valores entre <> indican que debe escribirse obligatoriamente un texto que se usará como el concepto que se escribe en su interior.

Por ejemplo, veamos la siguiente sintaxis, que define una sentencia de declaración de variables de tipo **char**:

```
[signed|unsigned] char <identificador>[,<identificador2>[,  
<identificador3>]...];
```

Significa que se puede usar **signed** o **unsigned**, o ninguna de las dos, ya que ambas están entre [], y separadas con un |.

El subrayado de **signed** indica que se trata de la opción por defecto. En este caso, si no se usa **signed** ni **unsigned**, el compilador elige la opción **signed**.

A continuación de **char**, que debe aparecer de forma obligatoria, debe escribirse un texto, que tiene ser una única palabra que actuará como *identificador* o nombre de la variable. Este identificador es el que usaremos para referirnos a la variable en el programa. En un programa C++ siempre llamaremos a las cosas por su nombre.

Opcionalmente, podemos declarar más variables del mismo tipo, añadiendo más identificadores separados con comas. Podemos añadir tantos identificadores como queramos.

Para crear un identificador hay que tener en cuenta algunas reglas, ya que no es posible usar cualquier cosa como identificador.

- Sólo se pueden usar letras (mayúsculas o minúsculas), números y ciertos caracteres no alfanuméricos, como el '_', pero nunca un punto, coma, guión, comillas o símbolos matemáticos o interrogaciones.
- El primer carácter no puede ser un número.
- C++ distingue entre mayúsculas y minúsculas, de modo que los identificadores *numero* y *Numero* son diferentes.

Finalmente, la declaración termina con un punto y coma.

Las palabras en negrita son palabras reservadas de C++. Eso significa que son palabras que no se pueden usar para otras cosas, concretamente, no se pueden usar como identificadores en un programa C++. Es decir, están reservadas para usarse del modo en que se describe en la sintaxis, y no se pueden usar de otro modo.

Serán válidos estos ejemplos:

```
signed char cuenta, cuenta2, total;  
unsigned char letras;  
char caracter, inicial, respuesta;  
signed char _letra;
```

Tipos fundamentales

En C sólo existen cinco tipos fundamentales y los tipos enumerados, C++ añade un séptimo 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.

Tipo "char" o carácter:

```
[signed|unsigned] char <identificador>[, <identificador2>[,  
<identificador3>]...];
```

Es el tipo básico alfanumérico, es decir que puede contener un carácter, un dígito numérico o un signo de puntuación. Desde el punto de vista del ordenador, todos esos valores son caracteres. En C++ este 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, habrás observado que puede ser declarado con y sin signo. Si no se especifica el modificador de signo, se asume que es con signo.

Nota: sé que esto sorprende, inquieta y despista a muchos lectores, así que probablemente necesite alguna explicación más detallada. De modo que he añadido un pequeño apéndice que explica cómo es posible que un número y una letra se puedan representar con el mismo tipo: [Apéndice A](#).

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.

El tipo **char** es, además, el único que tiene un tamaño conocido y constante. Para el resto de los tipos fundamentales que veremos, el tamaño depende de la implementación del compilador, que a su vez suele depender de la arquitectura del procesador o del sistema operativo. Sin embargo el tipo **char** siempre ocupa un byte, y por lo tanto, podemos acotar sus valores máximo y mínimo.

Así, el tipo **char** con el modificador **signed**, puede tomar valores numéricos entre -128 y 127. Con el modificador **unsigned**, el rango está entre 0 y 255.

El hecho de que se trate de un tipo numérico entero nos permite usar variables de **char** para trabajar con valores pequeños, siempre que lo consideremos necesario.

El motivo por el que este tipo también puede contener un carácter es porque existe una correspondencia entre números y caracteres. Esa correspondencia recibe el nombre de código ASCII.

Según este código, por ejemplo, al número 65 le corresponde el carácter 'A' o al número 49 el '1'.

El código ASCII primitivo usaba sólo 7 bits, es decir, podía codificar 128 caracteres. Esto era más que suficiente para sus inventores, que no usaban acentos, eñes, cedillas, etc. El octavo bit se usaba como bit de paridad en transmisiones de datos, para la detección de errores.

Después, para internacionalizar el código, se usó el octavo bit en una tabla ASCII extendida, que puede codificar 256 caracteres.

Pero como esto también es insuficiente, se han creado otras codificaciones de caracteres multibyte, aunque esto queda fuera de las posibilidades de **char**.

Tipo "int" o entero:

```
[signed|unsigned] [short|long] int <identificador>[,  
<identificador2>[,<identificador3>]...];  
[signed|unsigned] long [int] <identificador>[,  
<identificador2>[,<identificador3>]...];  
[signed|unsigned] short [int] <identificador>[,  
<identificador2>[,<identificador3>]...];
```

Las variables enteras almacenan números enteros dentro de los límites de cada uno de sus tamaños. A su vez, esos tamaños dependen de la plataforma, del compilador, y del número de bits que use por palabra de memoria: 8, 16, 32... No hay reglas fijas para

saber el tamaño, y por lo tanto, el mayor número que podemos almacenar en cada tipo entero: **short int**, **int** o **long int**; depende en gran medida del compilador y del sistema operativo. Sólo podemos estar seguros de que el tamaño de un **short int** es menor o igual que el de un **int**, y éste a su vez es menor o igual que el de un **long int**. Veremos cómo averiguar estos valores cuando estudiemos los operadores.

A cierto nivel, podemos considerar los tipos **char**, **short int**, **int** y **long int** como tipos enteros diferentes. Pero esa diferencia consiste sólo en el tamaño del valor máximo que pueden contener, y en el tamaño que ocupan en memoria, claro.

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 lentes que hay en un kilo de lentes.

Tipo "long long":

```
[signed|unsigned] long long [int] <identificador>[,  
<identificador2>[,<identificador3>]...];
```

Este tipo no pertenece al estandar ANSI, sin embargo, está disponible en compiladores GNU, como el que se usa en Linux o el que usa el propio Dev-C++ (y otros entornos de desarrollo para Windows).

Este tipo ocupa el siguiente puesto en cuanto a tamaño, después de **long int**. Como en los otros casos, su tamaño no está definido, pero sí sabemos que será mayor o igual que el de **long int**.

Tipo "float" o coma flotante:

```
float <identificador>[,<identificador2>[,  
<identificador3>]...];
```

Las variables de este tipo almacenan números en formato de coma flotante, esto es, contienen un valor de mantisa y otro de exponente, que, para entendernos, codifican números con decimales.

Aunque el formato en que se almacenan estos números en un ordenador es binario, podemos ver cómo es posible almacenar números muy grandes o muy pequeños mediante dos enteros relativamente pequeños, usando potencias en base 10. Por ejemplo, tenemos para la mantisa un valor entero, m , entre -0.99 y 0.99, y para el exponente un valor, e entre -9 y 9.

Los números se interpretan como $m \times 10^e$.

Este formato nos permite almacenar números entre -0.99×10^9 y 0.99×10^9 . Es decir, entre -990000000 y 990000000.

Y también números tan pequeños como 0.01×10^{-9} ó -0.01×10^{-9} . Es decir, como 0,000000000001 ó -0,000000000001.

Esto sólo con tres dígitos decimales, más los signos. Pero en un ordenador se usa aritmética binaria. Por ejemplo, para un tipo **float** típico de 32 bits, se usa un bit de signo para la mantisa y otro para el exponente, siete bits para el exponente y 23 para la mantisa.

Para más detalles se puede consultar el siguiente enlace: [representación de los números en punto flotante](#).

Estas variables son aptas para variables de tipo real, como por ejemplo el cambio entre euros y dólares. O para números muy grandes, como la producción mundial de trigo, contada en granos.

Pero 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 codifica. 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. Sin embargo el ejemplo si funcionaría con números más pequeños. Esto hace que las variables de tipo **float** no sean muy adecuadas para todos los casos, como veremos más adelante.

Puede que te preguntes (alguien me lo ha preguntado), qué utilidad tiene algo tan impreciso. La respuesta es: aquella que tú, 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 millones de otras situaciones en que resultan muy útiles.

Tipo "bool" o Booleano:

```
bool <identificador>[, <identificador2>[,  
<identificador3>]...];
```

Las variables de este tipo sólo pueden tomar dos valores **true** (verdadero) o **false** (falso). 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**.

```
bool respuesta;  
bool continuar;
```

Nota: En algunos compiladores de C++ antiguos no existe el tipo **bool**. Lo lógico sería no usar esos compiladores, y conseguir uno más actual. Pero si esto no es posible, se puede simular este tipo a partir de un enumerado.


```
enum bool {false=0, true};
```

Tipo "double" o coma flotante de doble precisión:

```
[long] double <identificador>[,<identificador2>[,  
<identificador3>]...];
```

Las variables de este tipo almacenan números en formato de coma flotante, mantisa y exponente, al igual que **float**, pero usan una precisión mayor, a costa de usar más memoria, claro. Son aptos para variables de tipo real. Usaremos estas variables cuando trabajemos con números grandes, pero también necesitamos gran precisión. El mayor espacio para almacenar el número se usa tanto para ampliar el rango de la mantisa como el del exponente, de modo que no sólo se gana en precisión, sino también en tamaño.

Al igual que pasaba con los números enteros, no existe un tamaño predefinido para cada tipo en coma flotante. Lo que sí sabemos es que el tamaño de **double** es mayor o igual que el de **float** y el de **long double** mayor o igual que el de **double**.

Lo siento, pero no se me ocurre ahora ningún ejemplo en el que sea útil usar uno de estos tipos.

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 qué 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 menos los recursos de nuestros clientes, para los que crearemos nuestros programas. C++ nos dan un gran control sobre estas características, es nuestra responsabilidad aprender a usarlo como es debido.

Tipo "void" o sin tipo:

```
void <identificador>[,<identificador2>[,  
<identificador3>]...];
```

En realidad esta sintaxis es errónea: no se pueden declarar variables de tipo **void**, ya que tal cosa no tiene sentido.

void es un tipo especial que indica la ausencia de tipo. Se usa para indicar el tipo del valor de retorno en funciones que no devuelven ningún valor, y también para indicar la ausencia de parámetros en funciones que no los requieren, (aunque este uso sólo es obligatorio en C, y opcional en C++), también se usará en la declaración de punteros genéricos, aunque esto 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, una función que se encargue de borrar la pantalla, no tienen nada que devolver, hace su trabajo y regresa. Lo mismo se aplica a las funciones sin parámetros de entrada, el mismo ejemplo de la función para borrar la pantalla no requiere ninguna entrada para poder realizar su cometido.

Tipo "enum" o enumerado:

```
enum [<identificador_de_enum>] {  
    <nombre> [= <valor>], ...} <identificador>[,  
<identificador2>[,<identificador3>]...];
```

```
enum <identificador_de_enum> {  
    <nombre> [= <valor>], ...} [<identificador>[,  
    <identificador2>[,<identificador3>]...]];
```

Se trata de una sintaxis más elaborada que las que hemos visto hasta ahora, pero no te asustes, (si es que te ha asustado esto) cuando te acostumbres a ver este tipo de cosas comprobarás que son fáciles de comprender.

Este tipo nos permite definir conjuntos de constantes enteras, llamados datos de tipo enumerado. Las variables declaradas de este tipo sólo podrán tomar valores dentro del dominio definido en la declaración.

Vemos que hay dos sintaxis. En la primera, el identificador de tipo es opcional, y si lo usamos podremos declarar más variables del tipo enumerado en otras partes del programa:

```
[enum] <identificador_de_enum> <identificador>[,  
    <identificador2>[,<identificador3>]...];
```

La segunda sintaxis nos permite añadir una lista de variables, también opcional.

De este modo podemos separar la definición del tipo enumerado de la declaración de variables de ese tipo:

```
enum orden {primero=1, segundo, tercero};  
...  
enum orden  id1, id2, id3;
```

O podemos hacer ambas cosas en la misma sentencia: definición y declaración:

```
enum orden {primero=1, segundo, tercero} id1, id2, id3;
```

Si decidimos no usar un identificador para el enumerado sólo podremos declarar variables en ese momento, y no en otros lugares del programa, ya que no será posible referenciarlo:

```
enum {primero=1, segundo, tercero} uno, dos;
```

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**.

La lista de valores entre las llaves definen lo que se denomina el "dominio" del tipo enumerado. Un dominio es un conjunto de valores

posibles para un dato. Una variable del tipo enumerado no podrá tomar jamás un valor fuera del dominio.

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.

3 Funciones I: Declaración y definición

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

Tal vez parezca un poco precipitado introducir este concepto tan pronto en el curso. Sin embargo, las funciones son una herramienta muy valiosa, y como se usan en todos los programas C++, creo que debemos tener, al menos, una primera noción de su uso. A fin de cuentas, todos los programas C++ contienen, como mínimo, una función.

Prototipos de funciones

En C++ es obligatorio usar prototipos. Un prototipo es una declaración de una función. Consiste en una presentación de la función, exactamente con la misma estructura que la definición, pero sin cuerpo y terminada con un ";". La estructura de un prototipo es:

```
[extern|static] <tipo_valor_retorno> [<modificadores>]  
<identificador>(<lista_parámetros>);
```

En general, el prototipo de una función se compone de las siguientes secciones:

- Opcionalmente, una palabra que especifique el tipo de almacenamiento, puede ser **extern** o **static**. Si no se especifica

ninguna, por defecto será **extern**. No te preocupes de esto todavía, de momento sólo usaremos funciones externas, lo menciono porque es parte de la declaración.

- El tipo del valor de retorno, que puede ser **void**, si no necesitamos valor de retorno. En C, si no se establece, será **int** por defecto, aunque en general se considera una mala técnica de programación omitir el tipo de valor de retorno de una función. En C++ es obligatorio indicar el tipo del valor de retorno.
- Modificadores opcionales. Tienen un uso muy específico, de momento no entraremos en este particular, lo veremos en capítulos posteriores.
- El identificador 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 leerlos. Cuando se precisen varias palabras para conseguir este efecto se puede usar alguna de las reglas más usuales. Una consiste en separar cada palabra con un "_". 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()".

Por ejemplo:

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

Un prototipo 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.

En el prototipo, los nombres de los parámetros son opcionales, y si se incluyen suele ser como documentación y ayuda en la interpretación y comprensión del programa. El ejemplo de prototipo anterior sería igualmente válido si se escribiera 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, los prototipos de las funciones se declaran dentro del fichero del programa, o bien se incluyen desde un fichero externo, llamado fichero de cabecera, (para esto se usa la directiva **#include**, que veremos en el siguiente capítulo).

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.

Definición de funciones

Al igual que hemos visto con las variables, las funciones deben declararse, para lo que usaremos los prototipos, pero también deben definirse.

Una definición contiene además las instrucciones con las que la función realizará su trabajo, es decir, su código.

La sintaxis de una definición de función es:

```
[extern|static] <tipo_valor_retorno> [modificadores]
<identificador>(<lista_parámetros>)
{
    [sentencias]
}
```

Como vemos, la sintaxis es idéntica a la del prototipo, salvo que se elimina el punto y coma final, y se añade el 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 "{}".

La definición de la función se hace más adelante o más abajo, según se mire, es decir, se hace después que el prototipo. Lo habitual es hacerlo después de la función *main*.

Una función muy especial es la función *main*, de la que ya hablamos en el capítulo primero. Se trata de la función de entrada, y debe existir siempre, ya será la que tome el control cuando se ejecute el programa. Los programas *Windows* usan la función *WinMain* como función de entrada, aunque en realidad esta función contiene en su interior la definición de una función *main*, pero todo 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.

Estructura de un programa C++

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]
[declaraciones de clases]
función main
[definiciones de funciones]
[definiciones de clases]
```

También se puede omitir el prototipo si se hace la definición antes de cualquier llamada a la función, es decir, en la zona de declaración de prototipos. Esto se puede hacer siempre, sin embargo no es muy recomendable como veremos a lo largo del curso.

Para no dejar las cosas "a medias", podemos ver una posible definición de la función "Mayor", que podría ser la siguiente:

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

Estructuras más complejas

Los programas complejos se escriben normalmente usando varios ficheros fuente. Estos ficheros se compilan separadamente y se enlazan todos 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 y clases puede hacerse dentro de los ficheros fuente o también pueden enlazarse desde bibliotecas compiladas previamente.

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.

4 Operadores I

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

Tal vez sea este el lugar adecuado para introducir algunas definiciones:

Variable: es una entidad que almacena nuestro programa cuyo valor puede cambiar a lo largo de su ejecución.

Operando: cada una de las constantes, variables o expresiones que intervienen en una expresión.

Operador: cada uno de los símbolos que indican las operaciones a realizar sobre los operandos, así como los operandos a los que afecta.

Expresión: según el diccionario, "Conjunto de términos que representan una cantidad", para nosotros es cualquier conjunto de operadores y operandos, que dan como resultado un valor.

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 varios. '+', '-', '*' y '/', tienen un comportamiento análogo en cuanto a los operandos, ya que admiten tanto expresiones enteras, como en 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.

Otro operador binario es el de módulo '%', que 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>
```

Nota: Esto quizás requiera una explicación:

Veamos, por ejemplo, la expresión $17 / 7$, es decir 17 dividido entre 7. Cuando aprendimos a dividir, antes de que supiéramos sacar decimales, nos enseñaron que el resultado de esta operación era 2, y el resto 3, es decir $2 \cdot 7 + 3 = 17$.

En C++, cuando las expresiones que intervienen en una de estas operaciones sean enteras, el resultado también será entero, es decir, si 17 y 7 se almacenan en variables enteras, el resultado será entero, en este caso 2.

Por otro lado si las expresiones son en punto flotante, con decimales, el resultado será en punto flotante, es decir, 2.428571. En este caso: $7 \cdot 2.428571 = 16.999997$, o sea, que no hay resto, o es muy pequeño.

Por eso mismo, calcular el resto, usando el operador %, sólo tiene sentido si las expresiones implicadas son enteras, ya que en caso contrario se calcularán tantos decimales como permita la precisión de tipo utilizado.

Siguiendo con el ejemplo, la expresión $17 \% 7$ dará como resultado 3, que es el resto de la división entera de 17 dividido entre 7.

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 evalúa la expresión "a+b", que dará como resultado 110, y se asignará este valor a c, que valdrá 110. Finalmente se aplica en post-incremento, y b valdrá 11.

Los operadores unitarios sufijos (post-incremento y post-decremento) se evalúan después de que se han evaluado el resto de las expresiones. En el primer ejemplo primero se evalúa ++b, después a+b y finalmente c = <resultado>. En el segundo ejemplo, primero se evalúa a+b, después c = <resultado> y finalmente b++.

Es muy importante no pensar o resolver las expresiones C como ecuaciones matemáticas, **NO SON EXPRESIONES MATEMATICAS**. No veas estas expresiones como ecuaciones, **NO SON ECUACIONES**. Esto es algo que se tarda en comprender al principio, y que después aprendes y dominas hasta el punto que no te das cuenta.

Por ejemplo, piensa en esta expresión:

```
b = b + 1;
```

Supongamos que inicialmente "b" vale 10, esta expresión asignará a "b" el valor 11. Veremos el operador "=" más adelante, pero por ahora no lo confundas con una igualdad matemática. En matemáticas la expresión anterior no tiene sentido, en programación sí lo tiene.

Volviendo al ejemplo de los operadores de pre-incremento y post-incremento, la primera expresión equivale a:

```
b = b+1;  
c = a + b;
```

La segunda expresión equivale a:

```
c = a + b;  
b = b+1;
```

Esto también proporciona una explicación de por qué la versión mejorada del lenguaje C se llama C++, es simplemente el C 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 B y BCPL.

Operadores de asignación

Existen varios operadores de asignación, el más evidente y el más usado es el "=", pero en C++ este no es el único que existe.

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


```
<variable> <operador de asignación> <expresión>
```

En general, para todos los operadores mixtos la expresión:

$E1 \text{ op} = E2$

Tiene el mismo efecto que la expresión:

$E1 = E1 \text{ op } E2$

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.

Los operadores del segundo al sexto son combinaciones del operador de asignación "=" y de los operadores aritméticos que hemos visto en el punto anterior. Tienen las mismas limitaciones que ellos, es decir, el operador "%=" sólo se puede aplicar a expresiones enteras.

El resto de los operadores son operadores de bits, y los veremos más adelante, en otro de los capítulos dedicado a operadores.

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(x, (y = 1, y + 2), 7);
```

Lamará a la función con tres argumentos: (x, 3, 7). La expresión de coma (y = 1, y+2), se evalúa de izquierda a derecha, y el resultado de la última evaluación se pasará como argumento a la función.

Es muy frecuente usar estas expresiones dentro de bucles "for", (que veremos en el próximo capítulo). Por ejemplo:

```
for(i = 0, j = 1; i < 10; i++) ...
```

Aquí vemos una expresión de coma que usamos para inicializar dos variables en la zona de inicialización del bucle.

No es aconsejable abusar de las expresiones de coma, aunque sólo sea porque apenas se usan, y suelen despistar a los que interpretan el código.

Por ejemplo, intenta predecir el valor de los parámetros de esta llamada a función:

```
func(19, (x=0, x+=3, x++), 12);
```

Si has dicho: 19, 4 y 12, te has equivocado :).

Nota: recuerda que el operador de postincremento se evalúa después de evaluar la sentencia, en este caso, después de la llamada a la función.

Operadores de comparación

Estos operadores comparan dos operandos, dando como resultado valores booleanos, **true** (verdadero) o **false** (falso), dependiendo de si los operandos cumplen o no la operación indicada.

Son "==" (dos signos = seguidos), "!=", "<", ">", "<=" y ">=", que comprueban relaciones de igualdad, desigualdad y comparaciones

entre dos valores aritméticos. Sintaxis:

```
<expresión1> == <expresión2>
<expresión1> != <expresión2>
<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:

== igualdad

!= desigualdad

> 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.

Expresiones con operadores de igualdad

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 cuidado si tu compilador te da una lista de "warnings", eso significa que has cometido algún error, en cualquier caso repasa esta lista e intenta corregir los "warnings".

Operadores lógicos

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

```
<expresión1> && <expresión2>  
<expresión1> || <expresión2>  
!<expresión>
```

El operador "&&" equivale al "AND" o "Y"; devuelve **true** sólo si los dos operandos **true** o lo que es equivalente, distintas de cero. En cualquier otro caso el resultado es **false**.

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**.

El operador "!" es equivalente al "NOT", o "NO", y devuelve **true** cuando la expresión evaluada es **false** o cero, en caso contrario devuelve **false**.

Cortocircuito

Existe una regla que en muchas ocasiones nos puede resultar útil, ya que nos puede ahorrar tiempo y comprobaciones adicionales.

Esta regla se conoce como "cortocircuito" o "*shortcut*", y se aplica de forma diferente a expresiones AND y OR.

En el caso de operaciones AND, consiste en que si la primera expresión evaluada es **false**, la segunda si siquiera se evalúa, ya que el resultado será siempre **false** independientemente del valor del segundo operando.

En el caso de operaciones OR, si la primera expresión sea **true**, la segunda no se evalúa, ya que el resultado será siempre **true**, independientemente del valor de la segunda expresión.

Esto es porque en una operación && el resultado sólo puede ser **true** cuando los dos operandos sean **true**, y en una operación || el resultado sólo puede ser **false** si ambos operandos son **false**. En el momento en que en una expresión AND uno de los operandos sea **false**, o que en una expresión OR uno de los operandos sea **true**, el valor del otro operando es irrelevante.

Si tenemos en cuenta este comportamiento, podremos ahorrar tiempo de ejecución si colocamos en primer lugar la expresión más fácil de calcular, o aquella cuyo valor sea más probablemente **false** en el caso de una expresión AND o **true**, para una expresión OR.

También habrá casos en que una de las expresiones sea indeterminada cuando la otra sea **false** en una expresión AND, o **true** en una expresión OR. En ese caso, será preferible colocar la expresión potencialmente indeterminada en el segundo lugar.

Tablas de verdad

Una tabla de verdad es una relación de todos los posibles valores para los operandos que intervienen en una operación, y los resultados para cada caso.

En el caso de operadores lógicos podemos mostrar fácilmente tablas de verdad, ya que el dominio para cada operando es muy reducido: **true** o **false**. Si además tenemos en cuenta la regla del cortocircuito, los casos posibles se reducen todavía más.

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

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

La tabla de verdad para el operador || es:

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

La tabla de verdad para el operador ! es:

--

Expresión	!Expresión
false	true
true	false

Expresiones lógicas frecuentes

A menudo aprovechamos ciertas equivalencias entre enteros y valores lógicos para comprobar algunos valores especiales en comparaciones usadas en condiciones o bucles.

Concretamente, me refiero a las comparaciones con cero. Así, si queremos saber si un valor entero E , es distinto de cero, lo comparamos usando el operador $!=$: $0 != E$.

Pero existe una correspondencia entre todos los valores enteros y los valores booleanos, y esa correspondencia es muy simple: un valor entero nulo esquivale a **false**, cualquier valor entero distinto de cero, equivale a **true**.

Teniendo esto en cuenta, la comparación anterior es innecesaria, ya que $0 != E$ es equivalente a usar, sencillamente E .

La condición contraria, $0 == E$, es por lo tanto equivalente a lo contrario, es decir, a la expresión $!E$.

Será pues, muy frecuente, que encuentres este tipo de operadores lógicos aplicados a enteros en condiciones:

```
if(!E) {...} // Si E es cero, hacer ...
if(E) {...} // Si E no es cero, hacer...
```

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.

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

Asociación de operadores binarios

Cuando decimos que un 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.

Del mismo modo, el operador de asignación también se puede asociar:

```
A = B = C = D = 0;
```

Este tipo de expresiones es muy frecuente en C++ para asignar el mismo valor a varias variables, en este caso, todas las variables: A, B, C y D recibirán el valor 0.

Generalización de cortocircuitos

Generalizando, con expresiones AND con más de dos operandos, la primera expresión **false** interrumpe el proceso e impide que se continúe la evaluación del resto de las operaciones.

De forma análoga, con expresiones OR con más de dos operandos, la primera expresión **true** interrumpe el proceso e impide que se continúe la evaluación del resto de las operaciones.

Palabras reservadas usadas en este capítulo

sizeof.

5 Sentencias

Espero que hayas tenido la paciencia suficiente para llegar hasta aquí, y que no te hayas asustado demasiado. 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**

- Comentarios

Bloques

Una sentencia compuesta o un bloque es un conjunto de sentencias, que puede estar vacía, encerrada entre llaves "{}". Sintácticamente, 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 biblioteca estándar o particular. Consiste en el nombre de la función, una lista de argumentos entre paréntesis y un ";".

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

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

Pero vamos a complicar un poco la situación para ilustrar la diferencia entre una sentencia de expresión y una expresión, reflexionemos 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.

Pero en realidad, el compilador evalúa primero la expresión, de modo que se obtenga el segundo parámetro de la función, y después llama a la función. ¿Parece 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 que el anterior:

```
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.

Nota:



El descubrimiento de los bucles se lo debemos a *Ada Byron*, así como el de las *subrutina* (que no es otra cosa que una función o procedimiento). Está considerada como la primera programadora, y ella misma se autodenominaba *analista*, lo que no deja de ser sorprendente, ya que el primer ordenador no se construyó hasta un siglo después.

Generalmente estas sentencias tienen correspondencia con estructuras de control equivalentes en *pseudocódigo*. El *pseudocódigo* es un lenguaje creado para expresar *algoritmos* formalmente y de manera clara. No es en si mismo un lenguaje de programación, sino más bien, un lenguaje formal (con reglas muy estrictas), pero humano, que intenta evitar ambigüedades.

A su vez, un *algoritmo* es un conjunto de reglas sencillas, que aplicadas en un orden determinado, permiten resolver un problema más o menos complejo.

Por ejemplo, un *algoritmo* para saber si un número N , es primo, puede ser:

Cualquier número es primo si sólo es divisible entre si mismo y la unidad.

Por lo tanto, para saber si un número N es primo o no, bastará con dividirlo por todos los números entre 2 y $N-1$, y si ninguna división es exacta, entonces el número N es primo.

Pero hay algunas mejoras que podemos aplicar:

La primera es que no es necesario probar con todos los números entre 2 y $N-1$, ya que podemos dar por supuesto que si N no es divisible entre 2, tampoco lo será para ningún otro número par: 4, 6, 8..., por lo tanto, después de probar con 2 pasaremos al 3, y después podemos probar sólo con los impares.

La segunda es que tampoco es necesario llegar hasta $N-1$, en realidad, sólo necesitamos llegar hasta el valor entero más cercano a la raíz cuadrada de N .

Esto es así porque estamos probando con todos los números menores que N uno a uno. Supongamos que vamos a probar con un número M mayor que la raíz cuadrada de N . Para que M pudiera ser un divisor de N debería existir un número X que multiplicado por M fuese igual a N .

$$N = M \times X$$

El caso extremo, es aquel en el que M fuese exactamente la raíz cuadrada de N . En ese caso, el valor de X sería exactamente M , ya que ese es el valor de la raíz cuadrada de

N :

$$N = M^2 = M \times M$$

Pero en el caso de que M fuese mayor que la raíz cuadrada de N , entonces el valor de X debería ser menor que la raíz cuadrada de N . Y el caso es que ya hemos probado con todos los números menores que la raíz cuadrada de N , y ninguno es un divisor de N .

Por lo tanto, ningún número M mayor que la raíz cuadrada de N puede ser divisor de N si no existen números menores que la raíz cuadrada de N que lo sean.

El pseudocódigo para este algoritmo sería algo parecido a esto:

```
¿Es N=1? -> N es primo, salir
¿Es N=2? -> N es primo, salir
Asignar a M el valor 2
mientras M <= raíz cuadrada(N) hacer:
    ¿Es N divisible entre M? -> N no es primo, salir
    Si M=2 entonces Asignar a M el valor 3
    Si M distinto de 2 entonces Asignar a M el valor M+2
Fin del mientras
N es primo ->salir
```

Bucles "mientras"

Es la sentencia de bucle más sencilla, y sin embargo es tremendamente potente. En C++ se usa la palabra reservada **while** (que significa "mientras"), y la sintaxis es la siguiente:

```
while (<condición>) <sentencia>
```

La sentencia es ejecutada repetidamente *mientras* la condición sea verdadera. 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.

Las condiciones no son otra cosa que expresiones de tipo booleano, cualquier otro tipo de expresión se convertirá a tipo booleano, si es posible. Y si no lo es, se producirá un error.

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 (es decir, *con clase*), de un modo más compacto:

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

Aquí vemos el uso de una sentencia nula, de la que hablábamos hace un rato. Observa que el bucle simplemente se repite, y la sentencia ejecutada es ";", es decir, nada.

Nota:

En realidad estos dos bucles no son equivalentes, ya que el valor de x al finalizar el segundo bucle es 101, y al finalizar el primero es 100.

Bucle "hacer...mientras"

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 verdadera. Si no se especifica condición se asume que es **true**, y el bucle se ejecutará indefinidamente.

En otros lenguajes, como **PASCAL**, se usa para el mismo tipo de bucle la estructura "repetir...hasta", es decir la sentencia se repite hasta que se cumpla una determinada condición. La diferencia está en que la lógica es la inversa: la sentencia se repite mientras la condición sea falsa. El resultado es el mismo, en cualquier caso.

A diferencia del bucle **while**, la evaluación de la condición se realiza después de ejecutar la sentencia, de modo que ésta se ejecutará al menos una vez. Por ejemplo:

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

En este bucle se incrementará el valor de x hasta que valga 100.

Pero aunque la condición sea falsa, por ejemplo, si x vale inicialmente 200, la sentencia `x = x + 1;`, se ejecuta primero, y después se verifica la condición.

Se pueden construir bucles **do...while** usando bucles **while**, pero a costa de repetir la sentencia dos veces:

```
<sentencia>
while(<condición>) <sentencia>
```

Esto puede hacernos pensar que estos bucles no son necesarios, y efectivamente, así es. Pero nos facilitan las cosas a la hora de codificar algoritmos basados en bucles **do...while**, ya que, por una parte, nos ahorran el trabajo de escribir dos veces el mismo código, y por otra, disminuyen las posibilidades de errores en una de

las repeticiones, sobre todo al corregir un error en el código, cuando es más fácil olvidar que estamos corrigiendo un bucle **do...while**, por lo que tendríamos que corregirlo dos veces.

Además, no olvidemos que existen sentencias de bloque, que pueden constar de cientos de sentencias simples, cada una de las cuales puede ser una sentencia cualquiera de las que estamos estudiando.

Bucle "para"

Por último el bucle *para*, que usa la palabra reservada **for**. Este tipo de bucle es el más elaborado. La sintaxis es:

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

La sentencia es ejecutada repetidamente mientras la condición resulte verdadera, o expresado de otro modo, hasta que la evaluación de la condición resulte falsa.

Antes de la primera iteración se ejecutará la iniciación del bucle, que puede ser una expresión o una declaración. En este apartado se suelen iniciar las variables usadas en el bucle. Estas variables también pueden ser declaradas en este punto, pero en ese caso tendrán validez (ámbito) sólo dentro del bucle **for**.

Después de cada iteración se ejecutará el incremento de las variables del bucle. Este incremento también es una sentencia de asignación, y no tiene por qué ser necesariamente un incremento.

En general, podemos considerar que la parte de *inicialización* establece las condiciones iniciales del bucle, la parte de la *condición* establece la condición de salida, y la parte del *incremento*, modifica las condiciones iniciales para establecer las de la siguiente iteración del bucle, o para alcanzar la condición de salida.

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

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

En realidad, podemos considerar un bucle **for** como una extensión de un bucle **while**. La equivalencia entre un bucle **for** y un bucle **while** es la siguiente:

```
[<inicialización>;  
  while([<condición>]) {  
    <sentencia>  
    [<incremento>]  
  }
```

Evidentemente, los bucles **for** no son estrictamente necesarios, (como tampoco lo son los bucles **do...while**), pero en muchas ocasiones simplifican el código de los bucles, haciendo más fácil la comprensión de un programa, ya sea para su análisis, su modificación o para su depuración.

En un estilo de programación claro, los bucles **for** se suelen utilizar para recorrer listas de elementos (como veremos el tema de los arrays). Usados con esta función, la *inicialización* se limita a asignar el valor inicial de un índice que sirve para recorrer la lista, la

condición comprueba si hemos llegado al final de la lista, y el *incremento* modifica el índice para que apunte al siguiente elemento de la lista.

Otra cosa, por muy tentador que resulte a veces, debemos intentar resistirnos a la tentación de usar los bucles **for** para emular otro tipo de bucles, ya que generalmente, esto puede inducir a malas interpretaciones sobre la finalidad del código, dificultando la depuración o el análisis.

Veamos un ejemplo sencillo:

```
int i=0;
for(bool salir = false; !salir; salir = (i > -5)) i++;
```

Aunque este bucle funcione, y haga lo que el programador tiene intención que haga, averiguar cómo lo hace requerirá cierto esfuerzo por nuestra parte. Supongo que estarás de acuerdo conmigo en que esto se puede expresar más claramente de otro modo:

```
i = 0;
do {
    i++;
} while(i <= -5);
```

Etiquetas

Los programas C++ se ejecutan secuencialmente, es decir, las sentencias se ejecutan una a continuación de otra, en el mismo orden en que están escritas en el programa.

Sin embargo, esta secuencia puede ser interrumpida de varias maneras, mediante el uso de sentencias de salto.

Las etiquetas son la forma en que se indica al compilador en qué puntos será reanudada la ejecución de un programa cuando se produzca una ruptura del orden secuencial de ejecución.

Etiquetas de identificación

Corresponden con la siguiente sintaxis:

```
<identificador>: <sentencia>
```

Estas etiquetas sirven como puntos de entrada para la sentencia de salto **goto**, que veremos más abajo, y 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 en el siguiente apartado. Sintaxis:

```
switch(<variable>)  
{  
    case <expresión_constante>: [<sentencias>] [break;]  
    . . .  
    [default: [<sentencias>]]  
}
```

Selección

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

Sentencia if...else

Permite la ejecución condicional de una sentencia. Sintaxis:

```
if (<condición>) <sentencial>
    [else <sentencia2>]
```

En C++ se pueden declarar variables dentro de la expresión de condición, que será, como en el caso de los bucles, una expresión booleana. Por ejemplo:

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

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

Si la condición es verdadera se ejecutará la sentencia1, si es falsa, (y si existe la parte del **else**), se ejecutará la sentencia2.

La cláusula **else** es opcional, y si se usa, no pueden insertarse sentencias entre la sentencia1 y el **else**.

Sentencia switch

Esta sentencia es una generalización de las sentencias **if...else**. En el caso de las sentencias **if**, la expresión que se evalúa como condición es booleana, lo que quiere decir que sólo hay dos valores posibles, y por lo tanto, sólo se puede elegir entre dos sentencias a ejecutar.

En el caso de la sentencia **switch**, la expresión a evaluar será entera, por lo tanto, el número de opciones es mucho mayor, y en consecuencia, también es mayor el número de diferentes sentencias que se pueden ejecutar.

Sintaxis:

```
switch (<expresión entera>)
{
    [case <expresión_constante1>: [<sentencias1>]]
    [case <expresión_constante2>: [<sentencias2>]]
    ...
}
```

```
[case <expresión_constanten>: [<sentenciasn>]]  
[default : [<sentencia>]]  
}
```

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 expresión entera evaluada dentro del **switch**. 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 los puntos de salida.

Esta estructura está diseñada para ejecutar cierta secuencia de instrucciones, empezando a partir de un punto diferente, en función de un valor entero y dejando sin ejecutar las anteriores a ese punto.

Veamos un ejemplo. Para hacer un pan hay que seguir (en una versión resumida), la siguiente secuencia:

1. Conseguir las semillas de trigo
2. Arar el campo
3. Sembrar el trigo
4. Esperar a que madure (No soy agricultor, pero sospecho que esta etapa es algo más complicada).
5. Cosechar el trigo
6. Separar el grano de la paja
7. Moler el grano, para coseguir harina
8. Amasar la harina junto con agua, sal y levadura. (Dejaremos de momento de lado el método para conseguir estos ingredientes).
9. Dejar reposar la masa
10. Encender el horno. (También dejaremos de momento los métodos para conseguir leña, construir un horno, o encenderlo).
11. Meter el pan crudo en el horno.

12. Esperar a que se haga el pan.
13. Sacar el pan del horno.

A cada paso le hemos asignado un número entero. Si por ejemplo, tenemos que hacer un pan, pero ya hemos hecho los primeros siete pasos, es decir, disponemos de harina, la ejecución empezará en el paso ocho, y continuará hasta el trece. De forma análoga, si ya tenemos masa dejada a reposar, podremos entrar directamente en el nivel diez.

Esta característica también nos permite ejecutar las mismas sentencias para varias etiquetas distintas, y en el apartado siguiente veremos (aunque vamos a adelantarlos ahora) que se puede eludir la ejecución secuencial normal usando la sentencia de ruptura **break** para ejecutar sólo parte de las sentencias.

Si el valor de la expresión entera no satisface ningún **case**, es decir, si no existe un **case** con una expresión constante igual al valor de la expresión entera, el control parará a la sentencia etiquetada con la etiqueta **default**.

Todas las etiquetas son opcionales, tanto **default** como todos los **case**. Si no se satisface ningún **case**, ni aparece la etiqueta **default**, se abandonará la sentencia **switch** sin ejecutar ninguna sentencia.

Por ejemplo:

```
bool EsVocal;  
char letra;  
...  
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 por ejemplo, *letra* contiene el valor 'a', se cumple el primer **case**, y la ejecución continúa en la siguiente sentencia: `EsVocal = true`, ignorando el resto de los **case** hasta el **break**, que nos hace abandonar la sentencia **switch**.

Otro ejemplo:

```
int Menor1, Menor2, Menor3, Mayor3;

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 *numero* 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 la sentencia **switch**.

Recordemos que los tipos enumerados se consideran también como enteros, de modo que también es posible usarlos en

sentencias **switch**, de hecho, su uso en estas sentencias es bastante frecuente.

```
enum estado {principio, primera, segunda, tercera, final};

estado x;

x = primera;

switch(x) {
    case principio: IniciarProceso();
    case primera:   PrimeraFase();
    case segunda:   SegundaFase();
    case tercera:   TerceraFase();
    case final:     Terminar()
}
```

Sentencias de salto

Como vimos al hablar sobre las etiquetas, los programas C++ se ejecutan secuencialmente, pero existen formas de romper este orden secuencial, mediante el uso de sentencias de salto, que veremos a continuación.

Sentencia de ruptura

El uso de esta sentencia dentro de un bucle, una sentencia de selección o de un bloque, transfiere la ejecución del programa a la primera sentencia que haya a continuación. Esto es válido para sentencias **switch**, como vimos hace un rato, pero también lo es para sentencias **while**, **do...while**, **for** e **if**.

En general, una sentencia **break** transfiere la ejecución secuencial a la siguiente sentencia, abandonando aquella en que se ejecuta.

Sintaxis:

break

Ejemplo:

```
int c = 0;
{
    for(int x=0; x < 100; x++) c+=x;
    break;
    c += 100;
}
c /= 100;
```

En este ejemplo, la sentencia `c += 100;` no se ejecutará, ya que la sentencia **break** transfiere la ejecución secuencial fuera del bloque.

Otro ejemplo:

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

En este otro 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 continue

El uso de esta sentencia dentro de un bucle ignora el resto del código de la iteración actual, y comienza con la siguiente, es decir, se transfiere la ejecución a la evaluación de la condición del 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 siguiente iteración, con lo que la condición del bucle volvería a evaluarse.

Sentencia de salto

Con el uso de esta sentencia el control se transfiere directamente al punto etiquetado con el identificador especificado.

Nota:

El **goto** es un mecanismo que está en guerra permanente, y sin cuartel, con la programación estructurada. Las sentencias **goto no se deben usar cuando se resuelven problemas mediante programación estructurada**, se incluye aquí porque existe, pero siempre puede y debe ser eludido. Existen mecanismos suficientes para hacer de otro modo todo aquello que pueda realizarse con mediante **goto**.

En cualquier caso, nosotros somos los programadores, y podemos decidir que para cierto programa o fragmento de programa, las ventajas de abandonar la programación

estructurada pueden compensar a los inconvenientes. A veces es imperativo sacrificar claridad en favor de velocidad de ejecución. Pero de todos modos, serán situaciones excepcionales.

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

Esta es la sentencia de salida de una función, cuando se ejecuta, se devuelve el control a la rutina que llamó a la función.

Además, se usa para especificar el valor de retorno de la función. 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 valor pasado como 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.

Es importante dejar siempre una sentencia **return** sin condiciones en todas las funciones. Esto es algo que a veces no se tiene en cuenta, y aunque puede no ser estrictamente necesario, siempre es conveniente. El ejemplo anterior se podría haber escrito de otro modo, sin tener en cuenta esto:

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

En este caso, para nosotros está claro que siempre se ejecutará una de las dos sentencias **return**, ya que cada una está en una de las alternativas de una sentencia **if...else**. Sin embargo, el compilador puede considerar que todas las sentencias **return** están en sentencias de selección, sin molestarse en analizar si están previstas todas las salidas posibles de la función, con lo que puede mostrar un mensaje de error.

El primer ejemplo es mejor, ya que existe una salida incondicional. Esto no sólo evitará errores del compilador, sino que

nos ayudará a nosotros mismos, ya que vemos que existe un comportamiento incondicional.

El único caso en que una sentencia **return** no requiere una expresión es cuando el valor de retorno de la función es **void**.

Existe un *mal uso* de las sentencias **return**, en lo que respecta a la programación estructurada. Por ejemplo, cuando se usa una sentencia **return** para abandonar una función si se detecta un caso especial:

```
int Dividir(int numerador, int denominador)
{
    int cociente;

    if(0 == denominador) return 1;
    cociente = numerador/denominador;
    return cociente;
}
```

Esta función calcula el cociente de una división entera. Pero hemos querido detectar un posible problema, ya que no es posible dividir por cero, hemos detectado ese caso particular, y decidido (erróneamente) que cualquier número dividido por cero es uno.

Sin embargo, en este caso, el primer **return** se comporta como una ruptura de secuencia, ya que se abandona la función sin procesarla secuencialmente. Siendo muy **puristas** (o pedantes), podemos considerar que esta estructura no corresponde con las normas de la programación estructurada. Un ejemplo más conforme con las normas sería:

```
int Dividir(int numerador, int denominador)
{
    int cociente;

    if(0 == denominador) cociente = 1;
    else cociente = numerador/denominador;
```

```
    return cociente;  
}
```

Sin embargo, a menudo usaremos estos *atajos* para abandonar funciones en caso de error, sacrificando el método en favor de la claridad en el código.

Uso de las sentencias de salto y la programación estructurada

Lo dicho para la sentencia **goto** es válido en general para todas las sentencias de salto. En el caso de la sentencia **break** podemos ser un poco más tolerantes, sobre todo cuando se usa en sentencias **switch**, donde resulta imprescindible. En general, es una buena norma huir de las sentencias de salto.

Comentarios

Los comentarios no son sentencias, pero me parece que es el lugar adecuado para hablar de ellos. En C++ pueden introducirse comentarios en cualquier parte del programa. Su función es ayudar a seguir el funcionamiento del programa durante la depuración o en la actualización del programa, además de documentarlo. Los comentarios en C, que también se pueden usar 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, de identificadores o de cadenas literales. Por ejemplo:

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

está permitido, pero sin embargo:


```
int ma/*función*/in(void)
```

es ilegal.

Esto no es una limitación seria, a fin de cuentas, se trata de aclarar y documentar, no de entorpecer la lectura del código.

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

```
int main(void) // Esto es un comentario
{
    return 0;
}
```

El cuerpo de la función no forma parte del comentario.

Palabras reservadas usadas en este capítulo

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

6 Declaración de variables

Una característica de C++, es la necesidad de declarar las variables que se usarán en un programa. Esto resulta chocante para los que se aproximan al C++ desde otros lenguajes de programación en los que las variables se crean automáticamente la primera vez que se usan. Se trata, es cierto, de una característica de bajo nivel, más cercana al ensamblador que a lenguajes de alto nivel, pero 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 y a mantener un estilo de programación elegante.

Uno de los errores más comunes en lenguajes en los que las variables se crean de forma automática se produce al cometer errores ortográficos. Por ejemplo, en un programa usamos una variable llamada *prueba*, y en un punto determinado le asignamos un nuevo valor, pero nos equivocamos y escribimos *prubea*. El compilador o interprete no detecta el error, simplemente crea una nueva variable, y continúa como si todo estuviese bien.

En C++ esto no puede pasar, ya que antes de usar cualquier variable es necesario declararla, y si por error usamos una variable que no ha sido declarada, se producirá un error de compilación.

Cómo se declaran las variables

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 y finalmente un punto y coma.

La declaración de variables es uno de los tipos de sentencia de C++. La prueba más clara de esto es que la declaración terminará

con un ";". Sintaxis:

```
<tipo> <lista de variables>;
```

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* con los valores 1234 y **true**, respectivamente.

En C++, contrariamente a lo que sucede con otros lenguajes de programación, las variables no inicializadas tienen un valor indeterminado (con algunas excepciones que veremos más tarde), y contienen lo que normalmente se denomina "basura". Cuando se declara una variable se reserva un espacio de memoria para almacenarla, pero no se hace nada con el contenido de esa memoria, se deja el valor que tuviera previamente, y ese valor puede interpretarse de distinto modo, dependiendo del tipo.

Ámbitos

Llamamos ámbito a la zona desde que cierto objeto es accesible.

En C++ solemos referirnos a dos tipos de ámbitos: temporal y de acceso. Así, el ámbito temporal indica el intervalo de tiempo en el que un objeto existe o es accesible. El ámbito de acceso nos dice desde donde es accesible.

En este capítulo hablaremos un poco sobre el ámbito de las variables, pero no entraremos en muchos detalles todavía, ya que es un tema largo.

Por otra parte, las funciones (y otros objetos de los que aún no hemos hablado nada), también tienen distintos ámbitos.

Ámbito de las variables

Dependiendo de dónde se declaren las variables, podrán o no ser accesibles desde distintas partes del programa. Es decir, su ámbito de acceso y temporal dependerá del lugar en que se declaren.

Las variables declaradas dentro de un bucle, serán accesibles sólo desde el propio bucle, esto es, tendrán un ámbito local para el bucle. Esto es porque las variables se crean al iniciar el bucle y se destruyen cuando termina. Evidentemente, una variable que ha sido destruida no puede ser accedida, por lo tanto, el ámbito de acceso está limitado por el ámbito temporal.

Nota:

En compiladores de C++ antiguos, (y en algunos modernos y *mal implementados*), no existe este ámbito, que sin embargo está descrito en la norma ANSI.

En estos compiladores, las variables declaradas dentro de un bucle tienen el mismo ámbito temporal y de acceso que las variables locales. Es decir, existen y son accesibles desde el punto en que se declaren hasta el final de la función.

Si usamos uno de esos compiladores no será posible, por ejemplo, usar varios bucles con declaraciones de variables locales de bucle con el mismo nombre.

```
for(int i=0; i < 100; i++) HacerAlgo(i);  
for(int i=0; i > -100; i--) DeshacerAlgo(i);
```

Este código daría un error al intentar redefinir la variable local *i*.

Las variables declaradas dentro de una función, y recuerda que *main* también es una función, sólo serán accesibles para esa función, desde el punto en que se declaran hasta el final. Esas variables son variables locales o de ámbito local de esa función.

Al igual que ocurre con las variables locales de bucle, en las de función, las variables se crean al iniciar la función y se destruyen al terminar.

Las variables declaradas fuera de las funciones, serán accesibles desde todas las funciones definidas después de la declaración. Diremos que esas variables son globales o de ámbito global.

El ámbito temporal de estas variables es también global: se crean junto con el programa, y se destruyen cuando el programa concluye.

Las variables globales son las únicas que son inicializadas automáticamente con valor cero cuando se declaran. Esto no sucede con ninguna variable local.

En todos los casos descritos, el ámbito temporal coincide con el de acceso: las variables que no pueden ser accedidas es porque no existen todavía o porque han sido destruidas. Más adelante veremos casos en que estos ámbitos no coinciden.

Una variable global declarada después de la definición de una función no será accesible desde esa función, por eso, normalmente se declaran las variables globales antes de definir las funciones.

Pero esto es hablando de forma general, en realidad, en C++ está *mal visto* usar variables globales, ya que se consideran poco seguras.

Ejemplo:

```
int EnteroGlobal; // Declaración de una variable global

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

int main() {
    // Declaración de una variable local de main:
```

```

    int EnteroLocal;

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

    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;
    return 0;
}

```

De modo que en cuanto a los ámbitos locales tenemos varios niveles:

```

<tipo> funcion(parámetros) // (1)
{
    <tipo> var1;                // (2)
    for(<tipo> var2;...)        // (3)
    ...
    <tipo> var3;                // (4)
    ...
    return var;
}

```

(1) Los parámetros se comportan del mismo modo que variables locales, tienen ámbito local a la función.

(2) Las variables declaradas aquí, también.

(3) Las declaradas en bucles, son de ámbito local en el bucle.

(4) Las variables locales sólo son accesibles a partir del lugar en que se declaren. Esta variable: *var3*, es de ámbito local para la función, pero no es accesible en el código previo a su declaración.

Es una buena costumbre inicializar las variables locales.

Los ámbitos pueden ser alterados mediante ciertos modificadores que veremos en otros capítulos.

Enmascaramiento de variables

Generalmente no es posible, y no suele ser necesario, declarar dos variables con el mismo nombre, pero hay condiciones bajo las cuales es posible hacerlo.

Por ejemplo, podemos declarar una variable global con un nombre determinado, y declarar otra variable (del mismo tipo o de otro diferente) de forma local en una función, usando el mismo nombre.

En ese caso decimos que la segunda declaración (la local), enmascara a la primera (la global). Con eso queremos decir que el acceso a la variable global está bloqueado o enmascarado por la local, que es a la única que podemos acceder directamente.

Por ejemplo:

```
int x;

int main() {
    int x;

    x = 10;
    return 0;
}
```

En este programa, cuando asignamos 10 a *x* estamos accediendo a la versión local de la variable *x*. En la función *main*, la variable global *x* está enmascarada, y no puede accederse a ella directamente.

Del mismo modo, una variable de ámbito de bucle o de ámbito de bloque puede enmascarar a una variable global o local, o a una de un bloque o bucle más externo:

```
int x = 10;
{
    int x = 0;
    for(int x = 0; x < 10; x++) HacerAlgoCon(x);
}
```

En este caso la declaración de `x` dentro del bloque enmascara la declaración anterior, y a su vez, la declaración dentro del bucle **for** enmascara a la declaración del bloque.

Otra cuestión sería qué utilidad pueda tener esto.

Operador de ámbito

Existe un método para acceder a una variable global enmascarada por una variable local. Se trata del operador de ámbito, que consiste en dos caracteres de dos puntos seguidos (::).

Veremos este operador con más detalle en el capítulo dedicado a los espacios con nombre, pero veamos ahora cómo lo podemos usar para acceder a una variable global enmascarada:

```
int x; // Variable global

int main()
{
    int x; // Variable local que enmascara a la global

    x = 10; // Accedemos a la variable local
    ::x = 100; // Mediante el operador de ámbito accedemos a
la global
    return 0;
}
```

El operador de ámbito, usado de este modo, permite acceder al espacio de variables global. Pero este no es más que un uso restringido del operador, que tiene muchas más aplicaciones.

Problemas resueltos de capítulos 1 a 6



Veamos ahora algunos ejemplos que utilicen los conocimientos que ya tenemos sobre C++.

Pero antes introduciremos, sin explicarlo en profundidad, dos elementos que nos permitirán que nuestros programas se comuniquen con nosotros. Se trata de la salida estándar, *cout* y de la entrada estándar *cin*. Estos objetos nos permiten enviar a la pantalla o leer desde el teclado cualquier variable o constante, incluidos literales. Lo veremos más detalladamente en un capítulo dedicado a ellos, de momento sólo nos interesa cómo usarlos para mostrar o leer cadenas de caracteres y variables.

Nota:

en realidad *cout* es un objeto de la clase *ostream*, y *cin* un objeto de la clase *istream* pero los conceptos de clase y objeto quedarán mucho más claros en capítulos posteriores.

El uso es muy simple:

```
#include <iostream>
using namespace std;

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

Veamos un ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
```

```
int a;

cin >> a;
cout << "la variable a vale " << a;
return 0;
}
```

Un método muy útil para *cout* es *endl*, que hará que la siguiente salida se imprima en una nueva línea.

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

Otro método, este para *cin* es *get()*, que sirve para leer un carácter, pero que nos puede servir para detener la ejecución de un programa.

Esto es especialmente útil cuando trabajamos con compiladores como Dev-C++, que crea programas de consola. Cuando se ejecutan los programas desde el compilador, al terminar se cierra la ventana automáticamente, impidiendo ver los resultados. Usando *get()* podemos detener la ejecución del programa hasta que se pulse una tecla.

A veces, sobre todo después de una lectura mediante *cin*, pueden quedar caracteres pendientes de leer. En ese caso hay que usar más de una línea *cin.get()*.

```
#include <iostream>
using namespace std;

int main()
{
    int a;

    cin >> a;
    cout << "la variable a vale " << a;
    cin.get();
    cin.get();
}
```

```
    return 0;
}
```

Las líneas `#include <iostream>` y `using namespace std;` son necesarias porque las declaraciones que permiten el uso de *cout* y *cin* están en una biblioteca externa. Con estos elementos 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 6.1

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 múltiplo de 3.
// 11/09/2000 Salvador Pozo

#include <iostream> // biblioteca para uso de cout
using namespace std;

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"; //
resto==0
        else cout << " no es múltiplo de 3"; // resto != 0
        cout << endl; // cambio de línea
    }

    return 0;
}
```

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 programa, 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 6.2

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 múltiplo de 3.  
// 11/09/2000 Salvador Pozo  
  
#include <iostream> // biblioteca para uso de cout  
using namespace std;  
  
// 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";  
        else cout << " no es múltiplo de 3";  
    }
```

```

        cout << endl; // cambio de línea
        i++;
    }

    return 0;
}

// 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 6.3

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> // biblioteca para uso de cout
using namespace std;

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 << j << " "; // muestra el número
        cout << endl; // cambio de línea
    }

    return 0;
}

```

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 6.4

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:

```
// Programa que genera la secuencia:
// 1, 5, 3, 7, 5, 9, 7, ..., 23
// 11/09/2000 Salvador Pozo

#include <iostream> // biblioteca para uso de cout
using namespace std;

int main() // función principal
{
    int i = 1; // variable para bucles
    bool sumar = true; // Siguiete operación es suma o resta
    bool terminado = false; // Condición de fin

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

    return 0;
}
```

Ejemplo 6.5

Escribir un programa que pida varios números, hasta que el usuario quiera terminar, y los descomponga en factores primos.

No seremos especialmente espléndidos en la optimización, por ejemplo, no es probable que valga la pena probar únicamente con números primos para los divisores, podemos probar con algunos que no lo sean, al menos en este ejercicio no será una gran diferencia.

Piensa un momento en cómo resolverlo e inténtalo, después puedes continuar leyendo.

Lo primero que se nos ocurre, al menos a mi, cuando nos dicen que el programa debe ejecutarse mientras el usuario quiera, es implementar un bucle **do..while**, la condición de salida será que usuario responda de un modo determinado a cierta pregunta.

En cada iteración del bucle pediremos el número a descomponer y comprobaremos si es divisible entre los números entre 2 y el propio número.

No podemos empezar 1, ya que sabemos que todos los números son divisibles por 1 infinitas veces, por eso empezamos por el 2.

Pero si probamos con todos los números, estaremos intentando dividir por todos los pares entre 2 y el número, y sabremos de antemano que ninguno de ellos es un factor, ya que sólo el 2 es primo y par a la vez, por lo tanto, podemos probar con 2, 3 y a partir de ahí incrementar los factores de dos e dos.

Por otra parte, tampoco necesitamos llegar hasta el factor igual al número, en realidad sólo necesitamos alcanzar la raíz cuadrada del número, ya que ninguno de los números primos entre ese valor y número puede ser un factor de número.

Supongamos que tenemos un número 'n', y que la raíz cuadrada de 'n' es 'r'. Si existe un número 'x' mayor que 'r' que es un factor primo de 'n', por fuerza debe existir un número 'h', menor que 'r', que multiplicado por 'x' sea 'n'. Pero ya hemos probado todos los números por debajo de 'r', de modo que si existe ese número 'h' ya lo hemos extraído como factor de 'n', y si hemos llegado a 'r' sin encontrarlo, es que tampoco existe 'x'.

Por ejemplo, el número 257. Su raíz cuadrada es (aproximada), 16. Es decir, deberíamos probar con 2, 3, 5, 7, 11 y 13 (nuestro programa probará con 2, 3, 5, 7, 9, 11, 13 y 15, pero bueno). Ninguno de esos valores es un factor de 257. El siguiente valor primo a probar sería 17, pero sabemos que el resultado de dividir 257 por 17 es menor que 17, puesto que la raíz cuadrada de 257 es 16.031. Sin embargo ya hemos probado con todos los primos

menores de 17, con resultado negativo, así que podemos decir que 17 no es factor de 257, ni tampoco, por la misma razón, ningún número mayor que él.

Ya tenemos dos buenas optimizaciones, veamos cómo queda el programa:

```
// Programa que descompone números en factores primos
// 26/07/2003 Salvador Pozo

#include <iostream> // biblioteca para uso de cout
using namespace std;

int main()
{
    int numero;
    int factor;
    char resp[12];

    do {
        cout << "Introduce un número entero: ";
        cin >> numero;
        factor = 2;
        while(numero >= factor*factor) {
            if(!(numero % factor)) {
                cout << factor << " * ";
                numero = numero / factor;
                continue;
            }
            if(factor == 2) factor++;
            else factor += 2;
        }
        cout << numero << endl;
        cout << "Descomponer otro número?: ";
        cin >> resp;
    } while(resp[0] == 's' || resp[0] == 'S');
    return 0;
}
```

Vemos claramente el bucle **do..while**, que termina leyendo una cadena y repitiendo el bucle si empieza por 's' o 'S'.

En cada iteración se lee un *numero*, y se empieza con el *factor* 2. Ahora entramos en otro bucle, este **while**, que se repite mientras el *factor* sea menor que la raíz cuadrada de *numero* (o mientras *numero* sea mayor o igual al *factor* al cuadrado).

Dentro de ese bucle, si *numero* es divisible entre *factor*, mostramos el *factor*, actualizamos el valor de *numero*, dividiéndolo por *factor*, y repetimos el bucle. Debemos probar de nuevo con *factor*, ya que puede ser factor primo varias veces. Para salir del bucle sin ejecutar el resto de las sentencias usamos la sentencia **continue**.

Si *factor* no es un factor primo de *numero*, calculamos el siguiente valor de *factor*, que será 3 si *factor* es 2, y *factor* + 2 en otro caso.

Cuando hemos acabado el bucle **while**, el valor de *numero* será el del último *factor*.

Puedes intentar modificar este programa para que muestre los factores repetidos en forma exponencial, en lugar de repetitiva, así, los factores de 256, en lugar de ser: "2 * 2 * 2 * 2 * 2 * 2 * 2 * 2", serían "2⁸".

7 Normas para la notación

Que no te asuste el título. Lo que aquí trataremos es más simple de lo que parece.

En este capítulo hablaremos sobre las reglas que rigen cómo se escriben las constantes en C++ según diversos sistemas de numeración y el uso tiene cada uno.

Constantes int

En C++ se usan tres tipos de numeración para la definición de constantes numéricas: la decimal, la octal y la hexadecimal; según se use la numeración en base 10, 8 ó 16, respectivamente.

Por ejemplo, el número 127, se representará en notación decimal como 127, en octal como 0177 y en hexadecimal como 0x7f.

Cualquier cantidad puede expresarse usando números de diferentes bases. Generalmente, los humanos, usamos números en base 10, pero para los ordenadores es más *cómodo* usar números en bases que sean potencias de 2, como 2 (numeración binaria), 8 (numeración octal) ó 16, (numeración hexadecimal).

El valor de cualquier cantidad se puede calcular mediante la siguiente fórmula:

$$N = \sum (d_i * base^i)$$

Donde d_i es cada uno de los dígitos, empezando con $i=0$ para el dígito más a la derecha, y $base$ es el valor de la base de numeración usada.

Por ejemplo, si el número `abcd` expresa una cantidad en base n , el valor del número se puede calcular mediante:

$$N = d \cdot n^0 + c \cdot n^1 + b \cdot n^2 + a \cdot n^3$$

En notación octal se necesitan sólo ocho símbolos, y se usan los dígitos del '0' al '7'. En hexadecimal, se usan 16 símbolos, los dígitos del '0' al '9', que tienen el mismo valor que en decimal; para los seis símbolos restantes 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 anterior, el valor 0177 expresa una cantidad en notación octal. En C++, el prefijo '0' indica numeración octal. Tendremos, por lo tanto, el número octal 177, que según nuestra fórmula vale:

$$N = 7 \cdot 8^0 + 7 \cdot 8^1 + 1 \cdot 8^2 = 7 \cdot 1 + 7 \cdot 8 + 1 \cdot 64 = 7 + 56 + 64 = 127$$

Análogamente, en el número 0x7f, "0x" se usa como prefijo que indica que se trata de un número en notación hexadecimal. Tenemos, por lo tanto, el número 7F. Aplicando la fórmula tenemos:

$$N = F \cdot 16^0 + 7 \cdot 16^1 = 15 \cdot 1 + 7 \cdot 16 = 15 + 112 = 127$$

Por último, aunque parezca obvio, el número 127 estará expresado en base 10, y también podemos aplicar la fórmula:

$$N = 7 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2 = 7 \cdot 1 + 2 \cdot 10 + 1 \cdot 100 = 7 + 20 + 100 = 127$$

Nota:

Si no tienes demasiado fresco el tema de las potencias, recordaremos que cualquier número elevado a 0 es 1.

Hay que tener mucho cuidado con las constantes numéricas, en C++ no es el mismo número el 0123 que el 123, aunque pueda parecer otra cosa. El primero es un número octal y el segundo decimal.

La ventaja de la numeración hexadecimal es que los valores enteros requieren dos dígitos por cada byte para su representación. Así, un byte 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 (cuatro dígitos binarios), el '0x0' por '0000', el '0x1' por '0001', hasta el '0xf', que equivale a '1111'. En el ejemplo el número 127, ó 0x7f, sería en binario '01111111'.

En la notación binaria usamos como base el 2, que es la base más pequeña que se puede usar. En este sistema de numeración sólo hay dos dígitos: 0 y 1. Por supuesto, también podemos aplicar nuestra fórmula, de modo que el número 01111111 vale:

$$\begin{aligned} N &= 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 \\ &= 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 + 1 \cdot 32 + 1 \cdot 64 + 0 \cdot 128 = 1 + \\ &2 + 4 + 8 + 16 + 32 + 64 = 127 \end{aligned}$$

Con la numeración octal sucede algo similar a lo que pasa con la hexadecimal, salvo que cada dígito octal agrupa tres bits (tres dígitos binarios). Así un byte puede tomar valores octales entre 0000 y 0377, dos bytes entre 0000000 y 01777777, etc. Además, la conversión a binario también es casi directa, ya que cada dígito octal se puede sustituir por tres bits, el '0' por '000', el '1' por '001', hasta el '7', que equivale a '111'. En el ejemplo el número 127, ó 0177, sería en binario '01111111'.

Aún no hemos hablado de los operadores de bits, pero podemos adelantar que C++ dispone de tales operadores, que básicamente realizan operaciones con números enteros bit a bit. De este modo, cuando trabajemos con operadores de bits, nos resultará mucho más sencillo escribir los valores de las constantes usando la

notación hexadecimal u octal, ya que es más directa su conversión a binario.

Constantes long

Cuando introduzcamos valores constantes **long** debemos usar el sufijo "L", sobre todo cuando esas constantes aparezcan en expresiones condicionales, y por coherencia, también en expresiones de asignación. Por ejemplo:

```
long x = 123L;  
if(0L == x) Imprimir("Valor nulo");
```

A menudo recibiremos errores del compilador cuando usemos constantes long sin añadir el sufijo L, por ejemplo:

```
if(1343890883 == x) Imprimir("Número long int");
```

Esta sentencia hará que el compilador emita un error ya que no puede usar un tamaño mayor sin una indicación explícita.

Hay casos en los que los tipos **long** e **int** tienen el mismo tamaño, en ese caso no se producirá error, pero no podemos predecir que nuestro programa se compilará en un tipo concreto de compilador o plataforma.

Constantes long long

En el caso de valores constantes **long long** tendremos que usar el sufijo "LL", tanto cuando esas constantes aparezcan en expresiones condicionales, como cuando lo hagan en expresiones de asignación. Por ejemplo:

```
long long x = 16575476522787LL;  
if(1LL == x) Imprimir("Valor nulo");
```

A menudo recibiremos errores del compilador cuando usemos constantes **long long** sin añadir el sufijo LL, por ejemplo:

```
if(16575476522787 == x) Imprimir("Número long long");
```

Esta sentencia hará que el compilador emita un error ya que no puede usar un tamaño mayor sin una indicación explícita.

Constantes unsigned

Del mismo modo, cuando introduzcamos valores constantes **unsigned** debemos usar el sufijo "U", en las mismas situaciones que hemos indicado para las constantes **long**. Por ejemplo:

```
unsigned int x = 123U;  
if(3124232U == x) Imprimir("Valor encontrado");
```

Constantes unsigned long

También podemos combinar en una constante los modificadores **unsigned** y **long**, en ese caso debemos usar el sufijo "UL", en las mismas situaciones que hemos indicado para las constantes **long** y **unsigned**. Por ejemplo:

```
unsigned long x = 123456UL;  
if(3124232UL == x) Imprimir("Valor encontrado");
```

Constantes unsigned long long

En una constante también podemos usar los modificadores **unsigned** y **long long**, para esos casos usaremos el sufijo "ULL", en todas las situaciones que hemos indicado para las constantes **long long** y **unsigned**. Por ejemplo:

```
unsigned long long x = 123456534543ULL;
if(3124232ULL == x) Imprimir("Valor encontrado");
```

Constantes float

Del mismo modo, existe una notación especial para las constantes en punto flotante. En este caso consiste en añadir ".0" a aquellas constantes que puedan interpretarse como enteras.

Se puede usar el sufijo "f". En ese caso, se tratará de constantes en precisión sencilla, es decir **float**.

Por ejemplo:

```
float x = 0.0;
if(x <= 1.0f) x += 0.01f;
```

Constantes double

Por defecto, si no se usa el sufijo, el compilador tratará las constantes en precisión doble, es decir **double**.

Por ejemplo:

```
double x = 0.0;
if(x <= 1.0) x += 0.01;
```


Constantes long double

Si se usa el sufijo "L" se tratará de constantes en precisión máxima, es decir **long double**.

Por ejemplo:

```
long double x = 0.0L;  
if(x <= 1.0L) x += 0.01L;
```

Constantes enteras

En general podemos combinar los prefijos "0" y "0x" con los sufijos "L", "U", y "UL".

Aunque es indiferente usar los sufijos en mayúsculas o minúsculas, es preferible usar mayúsculas, sobre todo con la "L", ya que la 'l' minúscula puede confundirse con un uno '1'.

Constantes en punto flotante

Ya hemos visto que podemos usar los sufijos "f", "L", o no usar prefijo. En este último caso, cuando la constante se pueda confundir con un entero, debemos añadir el ".0".

Para expresar constantes en punto flotante también podemos usar notación exponencial, por ejemplo:

```
double x = 10e4;  
double y = 4.12e2;  
double pi = 3.141592e0;
```

El formato exponencial consiste en un número, llamado mantisa, que puede ser entero o con decimales, seguido de una letra 'e' o 'E'

y por último, otro número, en este caso un número entero, que es el exponente de una potencia de base 10.

Los valores anteriores equivalen a:

```
x = 10 x 104 = 100000  
y = 4,12 x 102 = 412  
pi = 3.141592 x 100 = 3.141592
```

Al igual que con los enteros, es indiferente usar los sufijos en mayúsculas o minúsculas, pero es preferible usar mayúsculas, sobre todo con la "L", ya que la 'l' minúscula puede confundirse con un uno '1'.

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.

Secuencias de escape

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 '\\'.

Pero además de estos caracteres especiales existen otros. El código ASCII, que es el que puede ser representado por el tipo **char**, consta de 128 ó 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 \O sirve para la representación en notación octal, la letra 'O' se debe sustituir por un número en octal. Para la notación octal se usan tres dígitos, recuerda que para expresar un byte los valores octales varían de 0 a 0377.

También pueden asignarse números decimales a variables de tipo **char**. Por ejemplo:

```
char A;
A = 'a';
A = 97;
A = 0x61;
A = '\x61';
A = 0141;
A = '\141';
```

En este ejemplo todas las asignaciones son equivalentes y válidas.

Nota: 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.

¿Por qué es necesaria la notación?

C++ es un lenguaje pensado para optimizar el código y conseguir un buen rendimiento por parte del ordenador cuando ejecute nuestros programas. Esto nos obliga a prestar atención a detalles de bajo nivel (cercanos al hardware), detalles que en otros lenguajes de alto nivel no se tienen en cuenta.

Esto tiene un precio: debemos estar más atentos a los detalles, pero las ventajas compensan, ya que nuestros programas son mucho más rápidos, compactos y eficientes.

En todos estos casos que hemos visto en este capítulo, especificar el tipo de las constantes tiene el mismo objetivo: evitar que se realicen conversiones de tipo durante la ejecución del programa, obligando al compilador a hacerlas durante la fase de compilación.

Esto es una optimización, ya que generalmente, los programas se ejecutan muchas más veces de las que se compilan, así que parece razonable invertir más esfuerzo en minimizar el tiempo de ejecución que el de compilación.

Si en el ejemplo anterior para **float** hubiéramos escrito `if(x <= 1) ...`, el compilador almacenaría el 1 como un entero, y durante la fase de ejecución se convertirá ese entero a **float** para poder compararlo con x, que es **float**. Al poner "1.0" estamos diciendo al compilador que almacene esa constante como un valor en coma flotante, con lo cual nos evitamos la conversión de tipo cada vez que

se evalúe la condición de la sentencia **if**. Lo mismo se aplica a las constantes **long**, **unsigned** y **char**.

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, frecuentemente nombrados en inglés como *strings*.

Nota:

en informática es frecuente usar términos en inglés, sobre todo cuando la traducción puede inducir a errores, como en este caso, en que *string* se suele traducir como *cuerda* o *cadena*.

Aunque *array* también se puede traducir como *ristra* o *sarta*, que en este caso parece más apropiado, en ningún texto de programación verás que se hable jamás de ristras o sartas de caracteres o enteros.

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, en el ordenador, se almacenan en posiciones consecutivas de memoria. Este tipo de estructuras recibe un tratamiento muy especial, ya que es de gran utilidad y su uso es continuo.

La manera de definir una cadena es la siguiente:

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

Nota:

En este caso los corchetes no indican un valor opcional, sino que son literalmente 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 terminador, de modo que si queremos almacenar la cadena "HOLA", tendremos que declarar la cadena como:

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

Las cuatro primeras posiciones se usan para almacenar los caracteres "HOLA" y la posición extra, para el carácter nulo.

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].

Es muy importante tener presente que en C++, los índices tomarán valores empezando siempre en cero, así el primer carácter de nuestra cadena sería Saludo[0], que es la letra 'H'.

En un programa C++, una cadena puede almacenar informaciones en forma de texto, como nombres de personas, mensajes de error, números de teléfono, etc.

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

El siguiente ejemplo producirá un error en el compilador, ya que una cadena definida de este modo se considera una constante, como veremos en el capítulo de "arrays" o arreglos.

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


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] = 0;
```

O bien:

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

Si te parece un sistema engorroso, no te preocupes, existen funciones que facilitan la manipulación de cadenas.

De hecho, existen muchas de tales funciones, 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.

En C existe una biblioteca estándar (disponible en todos los compiladores), llamada precisamente *string*. En C++ también existe una biblioteca para manipular cadenas, aunque en este caso se trata de una biblioteca de clases.

9 Conversión de tipos

Quizás te hayas preguntado qué pasa cuando escribimos expresiones numéricas en las que no todos los operandos 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, el compilador los convierte 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), **long**, **long long**, **double**, **float** o **long double**.
2. Si un operando es de tipo **long double**, el otro se convertirá a **long double**.
3. Si un operando es de tipo **double**, el otro se convertirá a **double**.

4. Si un operando es de tipo **float**, el otro se convertirá a **float**.
5. Si un operando es de tipo **unsigned long long**, el otro se convertirá a **unsigned long long**.
6. Si un operando es de tipo **long long**, el otro se convertirá a **long long**.
7. Si un operando es de tipo **unsigned long**, el otro se convertirá a **unsigned long**.
8. Si un operando es de tipo **long**, el otro se convertirá a **long**.
9. Si un operando es de tipo **unsigned int**, el otro se convertirá a **unsigned int**.
10. Llegados a este punto ambos operandos son **int**.

Veamos ahora el ejemplo:

$c = a + b$; caso 10, 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**, la operación resultante corresponde al 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.

En el caso de las asignaciones, cuando la 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**.

Conversiones a bool

En C++ podemos hablar de otro tipo de conversión de tipo implícita, que se realiza cuando se usa cualquier expresión entera en una condición, y más generalmente, cuando se usa cualquier expresión donde se espera una expresión booleana.

El dominio del tipo **bool** es muy limitado, ya que sólo puede tomar dos valores: **true** y **false**. Por convenio se considera que el valor cero es **false**, y cualquier otro valor entero es **true**.

Por lo tanto, hay una conversión implícita entre cualquier entero y el tipo **bool**, y si añadimos esta regla a las explicadas antes, cualquier valor **double**, **long double**, **float** o cualquiera de los enteros, incluso **char**, se puede convertir a **bool**.

Esto nos permite usar condiciones abreviadas en sentencias **if**, **for**, **while** o **do..while**, cuando el valor a comparar es cero.

Por ejemplo, las siguientes expresiones booleanas son equivalentes:

`0 == x` equivale a `!x`.

`0 != x` equivale a `x`.

En el primer caso, usamos el operador `==` para comparar el valor de `x` con cero, pero al aplicar el operador `!` directamente a `x` obligamos al compilador a reinterpretar su valor como un **bool**, de modo que si `x` vale 0 el valor es **false**, y **!false** es **true**. De forma simétrica, si `x` es distinto de cero, se interpretará como **true**, y **!true** es **false**. El resultado es el mismo que usando la expresión `0 == x`.

En el segundo caso pasa algo análogo. Ahora usamos el operador `!=` para comparar el valor de `x` también con cero, pero ahora interpretamos directamente `x` como **bool**, de modo que si `x` vale 0 el valor es **false**, y si `x` es distinto de cero, se interpretará como **true**. El resultado es el mismo que usando la expresión `0 != x`.

No está claro cual de las dos opciones es más eficaz, a la hora de compilar el programa. Probablemente, la segunda requiera menos instrucciones del procesador, ya que existen instrucciones de ensamblador específicas para comparar un entero con cero. Del otro

modo estaremos comparando con un valor literal, y salvo que el compilador optimice este código, generalmente se requerirán más instrucciones de este modo.

Añadir que los ejemplos anteriores funcionan aunque el tipo de *x* no sea un entero. Si se trata de un valor en coma flotante se realizará una conversión implícita a entero antes de evaluar la expresión.

Casting: conversiones explícitas de tipo

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

Nota:

De nuevo nos encontramos ante un término que suele aparecer en inglés en los documentos. Se podría traducir como *amoldar* o *moldear*, pero no se hace. También es un término que se usa en cine y teatro, y se aplica al proceso de asignar papeles a los actores. La idea es análoga, en el caso de las variables, asignamos papeles a los valores, según sus características. Por ejemplo, para convertir el valor en coma flotante *14.232* a entero se usa el valor *14*, podríamos decir que *14* está haciendo el papel de *14.232* en la representación. O que se ajusta a un molde o troquel: lo que sobra se elimina.

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.

En C++ hay varios tipos diferentes de *casting*, pero de momento veremos sólo el que existe también en C.

Un *casting* tiene una de las siguientes sintaxis:

```
(<nombre de tipo>)<expresión>  
<nombre de tipo>(<expresión>)
```

Esta última es conocida como *notación funcional*, ya que tiene la forma de una llamada a función.

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

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

O bien:

```
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 estamos haciendo a propósito. Veremos más adelante, cuando hablemos de punteros, más situaciones donde también es obligatorio el uso de *casting*.

Ejemplos capítulos 8 y 9

Ejemplo 9.1

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>
using namespace std;

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, como comentamos en el capítulo 7.

Hay varias opciones para evitar este inconveniente. Por ejemplo, ejecutar los programas desde una ventana de consola, o añadir líneas al código que detengan la ejecución del programa, como *cin.get()* o *system("pause")*.

Otros IDEs, como Code::Blocks, no tienen ese inconveniente.

Ejemplo 9.2

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>
using namespace std;

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;
}
```

Ejecutar este código en [codepad](#).

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 9.3

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.

Para este problema tendremos que echar mano de algunas funciones estándar, concretamente de [toupper](#) y [tolower](#), declaradas en [ctype](#).

También puedes consultar el apéndice sobre bibliotecas estándar en el [apéndice C](#).

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>
#include <cctype>
using namespace std;

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 bucle
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 carácter 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]);
// Mostrar resultado:
cout << alfabeto << endl;

// 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]);

    alfabeto[i] = toupper(alfabeto[i]);
// Mostrar resultado:
cout << alfabeto << endl;

```

```
    return 0;
}
```

Ejecutar este código en [codepad](#).

Ejemplo 9.4

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

De nuevo tendremos que recurrir a funciones de estándar. En concreto la familia de macros [is<conjunto>](#).

Para leer caracteres podemos usar la función [getchar](#), perteneciente a [stdio](#).

```
// Cuenta letras
#include <iostream>
#include <cstdio>
#include <cctype>
using namespace std;

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 leídos desde el teclado

    cout << "Contaremos caracteres hasta que se pulse '&'"
         << endl;
    while((c = getchar()) != '&')
```

```

{
    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;
}

```

10 Tipos de variables II: Arrays

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

Nota:

Siguiendo con los términos en inglés, *array* es otro que no se suele traducir. El término *arreglo* no es en realidad una traducción, sino un anglicismo (un *palabro*, más bien, un término que no existe en español). Podríamos traducir *array* como *colección*, *selección*, o tal vez mejor, como *formación*. En este curso, de todos modos, usaremos el término *array*.

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

Sintaxis:

```
<tipo> <identificador>[<núm_elemento>][[<núm_elemento>]...];
```

Los corchetes en negrita no indican un valor opcional: deben aparecer, por eso están en negrita. La sintaxis es similar que para las cadenas, de hecho, las cadenas no son otra cosa que *arrays* de caracteres (tipo **char**).

Desde el punto de vista del programador, un *array* es un conjunto de datos del mismo tipo a los que se puede acceder individualmente mediante un índice.

Por ejemplo, si declaramos un objeto de este modo:

```
int valor;
```

El identificador 'valor' se refiere a un objeto de tipo **int**. El compilador sólo obtendrá memoria para almacenar un entero, y el programa sólo podrá almacenar y leer un único valor en ese objeto en cada momento.

Por el contrario, si declaramos un *array*:

```
int vector[10];
```

El compilador obtendrá espacio de memoria suficiente para almacenar 10 objetos de tipo *int*, y el programa podrá acceder a cada uno de esos valores para leerlos o modificarlos. Para acceder a cada uno de los valores se usa un índice, que en este caso podrá tomar valores entre 0 y 9. Usando el valor del índice entre corchetes, por ejemplo: `vector[0]` o `vector[4]`.

Es importante también tener en cuenta que el espacio de memoria obtenido para almacenar los valores de un *array* será contiguo, esto es, toda la memoria usada por un array tendrá direcciones consecutivas, y no estará fragmentada.

Otro detalle muy importante es que cuando se declaran *arrays*, los valores para el número de elementos deben ser siempre constantes enteras. Nunca se puede usar una variable para definir el tamaño de un *array*.

Nota:

Aunque la mayor parte de los compiladores permiten usar variables para definir el tamaño de un array, la norma no lo contempla. El hecho de que funcione no debe tentarnos a la hora de declarar *arrays*, nada nos asegura que en otros

compiladores o en futuras versiones del que ahora usemos se mantenga ese funcionamiento.

Se pueden usar tantas dimensiones (índices) como queramos, el límite lo impone sólo la cantidad de memoria disponible.

Cuando sólo se usa un índice se suele hablar de vectores, cuando se usan dos, de tablas. Los *arrays* de tres o más dimensiones no suelen tener nombres propios.

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

Los índices son números enteros, y pueden tomar valores desde 0 hasta <número de elementos>-1. Esto es muy importante, y hay que tener mucho cuidado, porque no se comprueba si los índices son válidos. Por ejemplo:

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

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

Como índice podremos usar cualquier expresión entera.

C++ no verifica el ámbito de los índices. Para poder hacerlo, el compilador tendría que agregar código, ya que los índices pueden ser variables, su valor debe ser verificado durante la ejecución, no durante la compilación. Esto está en contra de la filosofía de C++ de crear programas compactos y rápidos. Así que es tarea nuestra asegurarnos de que los índices están dentro de los márgenes correctos.

Si declaramos un *array* de 10 elementos, no obtendremos errores al acceder al elemento 11, las operaciones de lectura no son demasiado peligrosas, al menos en la mayoría de los casos. 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.

Inicialización 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";
char Saludo[] = {'H', 'o', 'l', 'a', 0};
```

Cuando se inicializan los *arrays* en la declaración no es obligatorio especificar el tamaño para la primera dimensión, como ocurre en los ejemplos de las líneas 2, 3, 4, 5 y 6. En estos casos la dimensión que queda indefinida se calcula a partir del número de elementos en la lista de valores iniciales. El compilador sabe contar y puede calcular el tamaño necesario de la dimensión para contener el número de elementos especificados.

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.

Operadores con arrays

Ya hemos visto que se puede usar el operador de asignación con *arrays* para asignar valores iniciales.

El otro operador que tiene sentido con los *arrays* es **sizeof**.

Aplicado a un *array*, el operador **sizeof** devuelve el tamaño de todo el array en bytes. Podemos obtener el número de elementos, si lo necesitamos, dividiendo ese valor entre el tamaño de uno de los elementos.

```
int main()
{
    int array[231];
    int nElementos;

    nElementos = sizeof(array)/sizeof(int);
    nElementos = sizeof(array)/sizeof(array[0]);
    return 0;
}
```

Las dos formas son válidas, pero la segunda es, tal vez, más general.

La utilidad de esta técnica es, como mucho, limitada. Desde el momento en que se deben usar constantes al declarar los *arrays*, su tamaño es siempre conocido, y por lo tanto, su cálculo es predecible.

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.

Tenemos una sección dedicada a los algoritmos de ordenación, pero ahora veremos uno de los más usados, aunque no de los más eficaces. Se trata del método de la burbuja.

Este método 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 seguimos el proceso...

Cuando lleguemos al final la lista estará así:

3 8 6 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

Nota:

Para mayor información sobre algoritmos de ordenación puedes consultar la sección dedicada en nuestra página: <http://conclase.net/c/orden/> realizada por Julián Hidalgo.

Problemas (creo que ya podemos empezar :-)

1. Hacer un programa que lea diez valores enteros en un *array* desde el teclado y calcule y muestre: la suma, el valor promedio, 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.
4. Hacer un programa que contenga una función con el prototipo `bool Incrementa(char numero[10]);`. La función debe incrementar el número pasado como parámetro en una cadena de caracteres de 9 dígitos. Si la cadena no contiene un número, debe devolver **false**, en caso contrario debe devolver **true**, y la cadena debe contener el número incrementado. Si el número es "999999999", debe devolver "0". Cadenas con

números de menos de 9 dígitos pueden contener ceros iniciales o no, por ejemplo, la función debe ser capaz de incrementar tanto la cadena "3423", como "00002323".

La función *main* llamará a la función *Incrementar* con diferentes cadenas.

5. Hacer un programa que contenga una función con el prototipo

`bool Palindromo(char palabra[40]);`. La función debe devolver **true** si la palabra es un palíndromo, y **false** si no lo es.

Una palabra es un palíndromo si cuando se lee desde el final al principio es igual que leyendo desde el principio, por ejemplo: "Otto", o con varias palabras "Anita lava la tina", "Dábale arroz a la zorra el abad". En estos casos debemos ignorar los acentos y los espacios, pero no es necesario que tu función haga eso, bastará con probar cadenas como "anitalavalatina", o "dabalearrozalazorraelabad".

La función no debe hacer distinciones entre mayúsculas y minúsculas.

Puedes enviar las soluciones de los ejercicios a nuestra dirección de correo: ejercicioscpp@conclase.net. Los corregiremos y responderemos con los resultados.

11 Tipos de objetos III:

Estructuras

Las estructuras son el segundo tipo de datos estructurados que veremos (valga la redundancia).

Al contrario que los *arrays*, las estructuras nos permiten agrupar varios datos, que mantengan algún tipo de relación, aunque sean de distinto tipo, permitiendo manipularlos todos juntos, usando un mismo identificador, o cada uno por separado.

Las estructuras son llamadas también muy a menudo registros, o en inglés *records*. Tienen muchos aspectos en común con los registros usados en bases de datos. Y siguiendo la misma analogía, cada objeto de una estructura se denomina a menudo campo, o *field*.

Sintaxis:

```
struct [<identificador>] {  
    [<tipo> <nombre_objeto>[,<nombre_objeto>,...]];  
} [<objeto_estructura>[,<objeto_estructura>,...]];
```

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

Los objetos de estructura son objetos declarados del tipo de la estructura, y su inclusión también es opcional. Sin bien, aún siendo ambos opcionales, al menos uno de estos elementos debe existir.

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

Las estructuras pueden referenciarse completas, usando su nombre, como hacemos con los objetos que ya conocemos, y

también se puede acceder a los elementos definidos en el interior de la estructura, usando el operador de selección (.), un punto.

Una vez definida una estructura, es decir, si hemos especificado un nombre para ella, se puede usar igual que cualquier otro tipo de C++. Esto significa que se pueden declarar más objetos del tipo de estructura en cualquier parte del programa. Para ello usaremos la forma normal de declaración de objetos, es decir:

```
[struct] <identificador> <objeto_estructura>
    [, <objeto_estructura>...];
```

En C++ la palabra `struct` es opcional en la declaración de objetos, al contrario de lo que sucede en C, en el que es obligatorio usarla.

Ejemplo:

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

Este ejemplo define la estructura *Persona* y declara a *Fulanito* como un objeto de ese tipo. Para acceder al nombre de *Fulanito*, por ejemplo para visualizarlo, usaremos la forma:

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

Funciones en el interior de estructuras

C++, permite incluir funciones en el interior de las estructuras. Normalmente estas funciones tienen la misión de manipular los

datos incluidos en la estructura, y su uso está muy relacionado con la programación orientada a objetos.

Aunque esta característica se usa casi exclusivamente con las clases, como veremos más adelante, también puede usarse en las estructuras. De hecho, en C++, las diferencias entre estructuras y clases son muy tenues.

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 símbolo "~".

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). También mediante la combinación [Atl Gr]+[4] y un espacio (la tecla [4] de la zona de las letras, no del teclado numérico).

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;

// Definición del constructor, fuera de la estructura
Punto::Punto() {
    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, ya que si necesitamos usar *constructores* para asignar valores iniciales, será mucho más lógico usar clases que estructuras.

Mencionar aquí, sólo a título de información, que el *constructor* no tiene por qué ser único. Se pueden definir varios *constructores*, 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.

Otro ejemplo:

```

#include <iostream>
using namespace std;

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 e incluso, redefinir 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.

Inicialización de estructuras

De un modo parecido al que se inicializan los arrays, se pueden inicializar estructuras, tan sólo hay que tener cuidado con las estructuras anidadas. Por ejemplo:

```

struct A {
    int i;
    int j;
    int k;
};

struct B {
    int x;
    struct C {

```

```

        char c;
        char d;
    } y;
    int z;
};

A ejemploA = {10, 20, 30};
B ejemploB = {10, {'a', 'b'}, 20};

```

Cada nueva estructura anidada deberá inicializarse usando la pareja correspondiente de llaves "{}", tantas veces como sea necesario.

Asignación de estructuras

La asignación de estructuras está permitida, pero sólo entre objetos del mismo tipo de estructura, (salvo que se usen *constructores*), y funciona como la intuición nos 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;`.

Quizás te hayas quedado intrigado por el comentario anterior, que adelantaba que se pueden asignar estructuras diferentes, siempre que se usen los *constructores* adecuados.

Esto, en realidad, se puede extender a cualquier tipo, no sólo a estructuras. Por ejemplo, definiendo el *constructor* adecuado, podemos asignar un entero a una estructura. Veamos cómo hacer esto.

Hasta ahora, los *constructores* que hemos visto no usaban argumentos, pero eso no significa que no puedan tenerlos.

Crearemos como ejemplo, una estructura para manejar números complejos. Un número complejo está compuesto por dos valores reales, el primero contiene lo que se llama la parte *real* y el segundo la parte *imaginaria*.

```
struct complejo {  
    double real;  
    double imaginario;  
};
```

Esta estructura es suficiente para muchas de las cosas que podemos hacer con números imaginarios, pero aprovechando que podemos crear funciones, podemos añadir algunas que hagan de una forma más directa cosas que de otro modo requieren añadir código externo.

Por ahora nos limitaremos a añadir unos cuantos *constructores*. El primero es el más lógico: un *constructor* por defecto:

```
struct complejo {  
    complejo() { real=0; imaginario = 0; }  
    double real;  
    double imaginario;  
};
```

Este *constructor* se usará, por ejemplo, si declaramos un *array*:

```
complejo array[10];
```

El *constructor* por defecto será llamado para cada elemento del *array*, aunque no aparezca tal llamada en ningún punto del programa.

Otro constructor nos puede servir para asignar un valor a partir de dos números:

```
struct complejo {  
    complejo() { real=0; imaginario = 0; }  
    complejo(double r, double i) { real=r; imaginario = i; }  
    double real;  
    double imaginario;  
};
```

Mediante este *constructor* podemos asignar valores iniciales en la declaración:

```
complejo c1(10.23, 213.22);
```

Los números reales se consideran un subconjunto de los imaginarios, en los que la parte imaginaria vale cero. Esto nos permite crear otro *constructor* que sólo admita un valor real:

```
struct complejo {  
    complejo() { real=0; imaginario = 0; }  
    complejo(double r, double i) { real=r; imaginario = i; }  
    complejo(double r) { real=r; imaginario = 0; }  
    double real;  
    double imaginario;  
};
```

Este *constructor* nos permite, como en el caso anterior, inicializar un valor de un complejo en la declaración, pero también nos permite asignar un valor **double** a un complejo, y por el sistema de

promoción automático, también podemos asignar valores enteros o en coma flotante:

```
complejo c1(19.232);  
complejo c2 = 1299.212;  
int x = 10;  
complejo c3 = x;
```

Este tipo de *constructores* se comportan como conversores de tipo, nada nos impide crear *constructores* con cualquier tipo de parámetro, y tales *constructores* se podrán usar para convertir cualquier tipo al de nuestra estructura.

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];
    charCodigoPostal[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 tienen un ámbito local, y para acceder a ellas se necesita hacer referencia a la estructura más externa.

Por ejemplo para declarar un objeto del tipo *stNombre* hay que utilizar el operador de acceso (::):

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

Sin embargo para declarar un objeto de tipo *stDireccion* basta con declararla:

```
stDireccion DireccionAuxiliar;
```

Estructuras anónimas

Antes dijimos, al hablar sobre la sintaxis de las declaraciones de estructuras, que debe aparecer o bien el identificador de estructura, o bien declararse algún objeto de ese tipo en la declaración. Bien, eso no es del todo cierto. Hay situaciones donde se pueden omitir ambos identificadores.

Una estructura anónima es la que carece de identificador de tipo de estructura y de declaración de objetos del tipo de estructura.

Por ejemplo, veamos esta declaración:

```
struct stAnonima {  
    struct {  
        int x;  
        int y;  
    };  
    int z;  
};
```

Para acceder a los campos *x* o *y* se usa la misma forma que para el campo *z*:

```
stAnonima Anonima;  
  
Anonima.x = 0;  
Anonima.y = 0;  
Anonima.z = 0;
```

Pero, ¿cual es la utilidad de esto?

Pues, la verdad, no mucha, al menos cuando se usa con estructuras. En el capítulo dedicado a las uniones veremos que sí puede resultar muy útil.

El método usado para declarar la estructura dentro de la estructura es la forma anónima, como verás no tiene identificador de tipo de estructura ni de campo. El único lugar donde es legal el uso de estructuras anónimas es en el interior de estructuras y uniones.

Operador sizeof con estructuras

Podemos usar el operador **sizeof** para calcular el espacio de memoria necesario para almacenar una estructura.

Sería lógico suponer que sumando el tamaño de cada elemento de una estructura, se podría calcular el tamaño de la estructura completa, pero no siempre es así. Por ejemplo:

```
#include <iostream>
using namespace std;

struct A {
    int x;
    char a;
    int y;
    char b;
};

struct B {
    int x;
    int y;
    char a;
    char b;
};

int main()
{
    cout << "Tamaño de int: "
          << sizeof(int) << endl;
    cout << "Tamaño de char: "
```



```

        << sizeof(char) << endl;
    cout << "Tamaño de estructura A: "
        << sizeof(A) << endl;
    cout << "Tamaño de estructura B: "
        << sizeof(B) << endl;

    return 0;
}

```

El resultado, usando Dev-C++, es el siguiente:

```

Tamaño de int: 4
Tamaño de char: 1
Tamaño de estructura A: 16
Tamaño de estructura B: 12

```

Si hacemos las cuentas, en ambos casos el tamaño de la estructura debería ser el mismo, es decir, $4+4+1+1=10$ bytes. Sin embargo en el caso de la estructura *A* el tamaño es 16 y en el de la estructura *B* es 12, ¿por qué?

La explicación es algo denominado alineación de bytes (*byte-aligning*). Para mejorar el rendimiento del procesador no se accede a todas las posiciones de memoria. En el caso de microprocesadores de 32 bits (4 bytes), es mejor si sólo se accede a posiciones de memoria múltiplos de cuatro, de modo que el compilador intenta alinear los objetos con esas posiciones.

En el caso de objetos *int* es fácil, ya que ocupan cuatro bytes, pero con los objetos *char* no, ya que sólo ocupan uno.

Cuando se accede a datos de menos de cuatro bytes la alineación no es tan importante. El rendimiento se ve afectado sobre todo cuando hay que leer datos de cuatro bytes que no estén alineados.

En el caso de la estructura *A* hemos intercalado campos **int** con **char**, de modo que el campo **int** *y*, se alinea a la siguiente posición múltiplo de cuatro, dejando tres posiciones libres después del campo *a*. Lo mismo pasa con el campo *b*.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

x a vacío y b vacío

En el caso de la estructura *B* hemos agrupado los campos de tipo **char** al final de la estructura, de modo que se aprovecha mejor el espacio, y sólo se desperdician los dos bytes sobrantes después de *b*.

0 1 2 3 4 5 6 7 8 9 10 11
x y a b vacío

Campos de bits

Existe otro tipo de estructuras que consiste en empaquetar cada uno de los campos en el interior de valores enteros, usando bloques o subconjuntos de bits para cada campo.

Por ejemplo, un objeto **char** contiene ocho bits, de modo que dentro de ella podremos almacenar ocho campos de un bit, o cuatro de dos bits, o dos de tres y uno de dos, etc. En un objeto **int** de 16 bits podremos almacenar 16 campos de un bit, etc.

Para definir campos de bits debemos usar siempre valores de enteros sin signo, ya que el signo se almacena en un bit del entero, el de mayor peso, y puede falsear los datos almacenados en la estructura.

La sintaxis es:

```
struct [<nombre de la estructura>] {  
    unsigned <tipo_entero> <identificador>:<núm_de_bits>;  
    .  
} [<lista_objetos>];
```

Existen algunas limitaciones, por ejemplo, un campo de bits no puede crearse *a orcajadas* entre dos objetos distintos, todos sus bits tienen que estar en el mismo valor entero.

Veamos algunos ejemplos:

En este ejemplo vemos que como no es posible empaquetar el *campo2* dentro del mismo **char** que el *campo1*, de modo que se añada un segundo valor **char**, y se dejan sin usar todos los bits sobrantes.

También es posible combinar campos de bits con campos normales, por ejemplo:

```
struct mapaBits2 {
    int numero;
    unsigned short int campo1:3;
    unsigned short int campo2:4;
    unsigned short int campo3:2;
    unsigned short int campo4:1;
    unsigned short int campo5:6;
    float n;
};
```

Los campos de bits se tratan, por norma general, igual que cualquier otro de los campos de una estructura. Se les puede asignar valores (dentro del rango que admitan por su tamaño), pueden usarse expresiones, imprimirse, etc.

```
#include <iostream>
#include <cstdlib>
using namespace std;

struct mapaBits2 {
    unsigned short int campo1:3;
    unsigned short int campo2:4;
    unsigned short int campo3:2;
    unsigned short int campo4:1;
    unsigned short int campo5:6;
};

int main()
{
    mapaBits2 x;

    x.campo2 = 12;
```

```
x.campo4 = 1;
cout << x.campo2 << endl;
cout << x.campo4 << endl;

return 0;
}
```

Lo que no es posible es leer valores de un campo de bits mediante *cin*. Para poder leer valores desde el teclado se debe usar una variable entera auxiliar, y posteriormente asignarla al campo de bits.

No es frecuente usar estas estructuras en programas, salvo cuando se relacionan con ciertos dispositivos físicos. Por ejemplo, para configurar un puerto serie en *MS-DOS* se usa una estructura empaquetada en un **unsigned char**, que indica el número de bits de datos, de bits de parada, la paridad, etc, es decir, todos los parámetros del puerto. En general, para programas que no requieran estas estructuras, es mejor usar estructuras normales, ya que son mucho más rápidas.

Otro motivo que puede decidirnos por estas estructuras es el ahorro de espacio, ya sea en disco o en memoria. Si conocemos los límites de los campos que queremos almacenar, y podemos empaquetarlos en estructuras de mapas de bits podemos ahorrar mucho espacio.

Palabras reservadas usadas en este capítulo

struct.

Problemas

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 (en el *array*). Después debe ser capaz de buscar el nombre

correspondiente a un número de teléfono y el teléfono correspondiente a una persona. Ambas opciones deben ser 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:

Nota:

No olvides que para comparar cadenas se debe usar una función, no el operador `==`.

2. Para almacenar fechas podemos crear una estructura con tres campos: *ano*, *mes* y *día*. Los días pueden tomar valores entre 1 y 31, los meses entre 1 y 12 y los años, dependiendo de la aplicación, pueden requerir distintos rangos de valores. Para este ejemplo consideraremos suficientes 128 años, entre 1960 y 2087. En ese caso el año se obtiene sumando 1960 al valor de *ano*. El año 2003 se almacena como 43. Usando estructuras, y ajustando los tipos de los campos, necesitamos un **char** para *día*, un **char** para *mes* y otro para *ano*. Diseñar una estructura análoga, llamada *fecha*, pero usando campos de bits. Usar sólo un entero corto sin signo (**unsigned short**), es decir, un entero de 16 bits. Los nombres de los campos serán: *día*, *mes* y *anno*.
3. Basándose en la estructura de bits del ejercicio anterior, escribir una función para mostrar fechas: `void Mostrar(fecha);`. El formato debe ser: "dd de mmmmmm de aaaa", donde dd es el

día, mmmmmm el mes con letras, y aaaa el año. Usar un *array* para almacenar los nombres de los meses.

4. Basándose en la estructura de bits del ejercicio anterior, escribir una función `bool ValidarFecha(fecha);`, que verifique si la fecha entregada como parámetro es válida. El mes tiene que estar en el rango de 1 a 12, dependiendo del mes y del año, el día debe estar entre 1 y 28, 29, 30 ó 31. El año siempre será válido, ya que debe estar en el rango de 0 a 127.

Para validar los días usaremos un *array* `int DiasMes[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`. Para el caso de que el mes sea febrero, crearemos otra función para calcular si un año es o no bisiesto: `bool Bisiesto(int);` Los años bisiestos son los divisibles entre 4, al menos en el rango de 1960 a 2087 se cumple.

Nota:

los años bisiestos son cada cuatro años, pero no cada 100, aunque sí cada 400. Por ejemplo, el año 2000, es múltiplo de 4, por lo tanto debería haber sido bisiesto, pero también es múltiplo de 100, por lo tanto no debería serlo; aunque, como también es múltiplo de 400, finalmente lo fue.

5. Seguimos con el tema de las fechas. Ahora escribir dos funciones más. La primera debe responder a este prototipo: `int CompararFechas(fecha, fecha);`. Debe comparar las dos fechas suministradas y devolver 1 si la primera es mayor, -1 si la segunda es mayor y 0 si son iguales. La otra función responderá a este prototipo: `int Diferencia(fecha, fecha);`, y debe devolver la diferencia en días entre las dos fechas suministradas.

Ejemplos capítulos 10 y 11

Ejemplo 11.1

En el [capítulo 10](#) sobre los *arrays* vimos cómo ordenarlo usando el método de la burbuja. Hay muchas formas de ordenar un *array* pero el objetivo suele ser siempre el mismo: poder localizar o al menos determinar si existe, un determinado valor dentro del *array*.

Hay varios métodos de búsqueda, pero el más conocido, es el de la "Búsqueda binaria" o "Busca dicotómica". Se trata además, un método muy bueno de búsqueda, ya que el tiempo de búsqueda disminuye exponencialmente con el número de iteraciones.

La idea es sencilla, se elige el elemento central del rango en el que debemos buscar. Pueden pasar tres cosas:

- Que el elemento elegido sea el buscado, con lo que ha terminado la búsqueda.
- Que el elemento elegido sea menor que el buscado, en ese caso, tomaremos el elemento siguiente al elegido como límite inferior de un nuevo rango, y repetiremos el proceso.
- Que el elemento elegido sea mayor. Ahora tomaremos el elemento anterior al elegido como nuevo límite superior de un nuevo rango, y repetiremos el proceso.

El proceso termina cuando encontramos el elemento, o cuando el rango de búsqueda resulte nulo, y la búsqueda habrá fracasado.

```
// Búsqueda binaria
// Agosto de 2009 Con Clase, Salvador Pozo
#include <iostream>
using namespace std;

int Binaria(int*, int, int, int);

int tabla[] = {
    1,   3,  12,  33,  42,  43,  44,  45,  54,  55,
    61,  63,  72,  73,  82,  83,  84,  85,  94,  95,
    101, 103, 112, 133, 142, 143, 144, 145, 154, 155,
    161, 163, 172, 173, 182, 183, 184, 185, 194, 195
};
```



```

int main() {
    int pos;
    int valor=141;

    pos = Binaria(tabla, valor, 0,
sizeof(tabla)/sizeof(tabla[0])-1);
    if(pos > 0) cout << "Valor " << valor << " encontrado en
posicion: " << pos << endl;
    else cout << "Valor " << valor << " no encontrado" <<
endl;
    return 0;
}

/* Función de búsqueda binaria:
Busca el "valor" dentro del vector "vec" entre los
márgenes
inferior "i" y superior "s" */
int Binaria(int* vec, int valor, int i, int s) {
    int inferior = i;
    int superior = s;
    int central;

    do {
        central = inferior+(superior-inferior)/2;
        if(vec[central] == valor) return central;
        else if(vec[central] < valor) inferior = central+1;
        else superior = central-1;
    } while(superior >= inferior);
    return -1;
}

```

Ejecutar este código en [codepad](#).

En el [capítulo 24](#) veremos otro modo de implementar esta función, usando recursividad.

Ejemplo 11.2

Ya sabemos que el tamaño de los números enteros está limitado en C++. Con enteros de 64 bits con signo, el número más grande es 9223372036854775808, es decir, 19 dígitos.

En general, veremos que esto es más que suficiente para casi todas nuestras aplicaciones, pero es posible que a veces

necesitemos más dígitos.

Hacer un programa que pueda sumar numeros enteros sin signo, almacenados en forma de cadena. La longitud de la cadena se almacenará en una constante *cadmax* de modo que pueda cambiar a nuestra conveniencia. En principio, usaremos 32 caracteres. Si el resultado no cabe en *cadmax* caracteres, retornar con **false** para indicar un error de desbordamiento. Si el resultado es correcto, retornar **true**.

Piensa sobre el problema, a partir de este punto lo analizaremos y sacaremos conclusiones para diseñar nuestro programa.

Lo primero que hay que tener en cuenta es que almacenamos los números en forma de caracteres, lo que es poco eficaz, pero es un ejercicio...

Cada carácter es un dígito, y para hacer la suma, empezaremos a sumar dígito a dígito, empezando por la derecha. Así, '1'+ '1' debe dar '2'. Pero ya sabemos que se trata de códigos ASCII, de modo que si sumamos normalmente, tendremos que restar el valor ASCII de '0':

```
cout << char('1'+'1') << endl;
```

El resultado de esta operación es 'b'.

```
cout << char('1'+'1'-'0') << endl;
```

Y el resultado de esta es '2'.

Hay otro problema añadido. Si la suma de los dígitos es mayor que '9', no tendremos un dígito:

```
cout << char('7'+'8'-'0') << endl;
```

El resultado de esta suma es '?'. No es que el compilador no sepa sumar. Lo que pasa es que se ha producido un desbordamiento. 7+8 son 15, es decir, 5, "y nos llevamos 1". Ese 1 es un desbordamiento o acarreo. Es decir, debemos tener en cuenta el acarreo en la suma del siguiente dígito. La operación se complica un poco más:

```
int acarreo=0;
int resultado = char('7'+ '8' - '0' + acarreo);
if(resultado < '9') { resultado-=10; acarreo = 1; }
else acarreo = 0;
cout << resultado << endl;
```

El segundo detalle a considerar es que empezar a recorrer cadenas desde la derecha no es tan simple como pueda parecer al principio, sobre todo si las cadenas tienen distinta longitud.

```
const unsigned int cadmax = 32;
typedef char numero[cadmax];
numero suma;
numero n1 = "8988989";
numero n2 = "7763";
```

Si queremos sumar n1 y n2, deberemos empezar por los dígitos '9' y '3', respectivamente, es decir, por n1[6] y n2[3]. El resultado se almacena en la posición 6 de la cadena suma. Pasamos al siguiente dígito: n1[5] y n2[2], etc. Cuando llegamos a n1[3] y n2[0] tropezamos con un problema. El siguiente dígito de n2 no existe. Cuando pase eso, para cualquiera de las dos cadenas, deberemos tomar el valor '0' para esos dígitos.

Aún hay otro inconveniente que debemos salvar. Por ejemplo:

```
const unsigned int cadmax = 32;
typedef char numero[cadmax];
numero suma;
```

```
numero n1 = "9";  
numero n2 = "3";
```

En este caso, el resultado de '9'+ '3' es '2' y el acarreo queda con valor 1. La cadena resultante contiene "2", que evidentemente es un resultado erróneo. En este caso, deberemos desplazar todos los dígitos de suma a la derecha, y añadir el dígito '1' al principio.

Por último, hay un caso especial más. Supongamos que el resultado de la suma de los dos números no cabe en el número de caracteres usado para almacenarlos. En ese caso, debemos retornar **false**.

El resultado es un programa como este:

```
// Sumar números enteros sin signo almacenados en cadenas  
// Agosto de 2009 Con Clase, Salvador Pozo  
#include <iostream>  
  
using namespace std;  
  
const unsigned int cadmax = 32;  
typedef char numero[cadmax];  
  
bool Sumar(numero, numero, numero);  
int max(int, int);  
  
int main()  
{  
    numero n1="99999999999999999999";  
    numero n2="1";  
    numero suma;  
  
    Sumar(n1, n2, suma);  
    cout << n1 << " + " << n2 << " = " << suma << endl;  
    return 0;  
}  
  
bool Sumar(numero n1, numero n2, numero r) {  
    // Primero, buscar los dígitos de la derecha:  
    char c1,c2;  
    int acarreo = 0;  
    int lon1=strlen(n1);
```

```

int lon2=strlen(n2);
    // Colocar el terminador de la cadena resultado:
r[max(lon1, lon2)] = 0;
// Hacer la suma, dígito a dígito:
    do {
        lon1--;
        lon2--;
        if(lon1 < 0) c1 = '0'; else c1 = n1[lon1];
        if(lon2 < 0) c2 = '0'; else c2 = n2[lon2];
        r[max(lon1, lon2)] = acarreo+c1+c2-'0';
        if(r[max(lon1, lon2)] > '9') { r[max(lon1, lon2)]-=10;
acarreo=1; }
        else acarreo = 0;
    } while(lon1 > 0 || lon2 > 0);

// Desbordamiento:
if(acarreo) {
    if(strlen(r) < cadmax) {
        for(int i=strlen(r)+1; i > 0; i--) r[i] = r[i-1];
        r[0] = '1';
        return false;
    }
}
return true;
}

int max(int a, int b) {
    if(a > b) return a;
    return b;
}

```

Ejecutar este código en [codepad](#).

Ejemplo 11.3

Seamos más inteligentes con el uso de recursos. Usando caracteres, cada byte puede almacenar sólo diez valores diferentes. Una cadena de 32 bytes puede almacenar números positivos sin signo hasta 10^{32} dígitos (eso sin tener en cuenta el carácter nulo usado como fin de cadena). Pero si aprovechamos cada bit, con cada carácter hay 256 posibilidades, en lugar de 10, y el resultado

es que podemos almacenar números hasta 256^{32} , o lo que es lo mismo, 2^{256} . Eso significa, enteros con 77 dígitos significativos.

Escribamos el programa anterior aprovechando cada bit. Por comodidad, usaremos el modo de almacenamiento *Little-Endian*.

Nota:

En un ordenador hay dos formas ordenadas de almacenar números de más de un byte (desordenadas hay más).

La primera forma es la que usamos nosotros para escribir números sobre el papel o en una pantalla: primero el de mayor peso, y al final, el de menor peso. En el número 1234, el '1' tiene mayor peso (1000) que el 4 (1). A esta forma se le llama "*Big-Endian*", y es la que usan (en binario) los procesadores de la familia de *Motorola*.

La segunda forma es la contraria: primero el de menor peso, y al final, el de mayor. A esta forma se le llama "*Little-Endian*", y es la que usan los procesadores de la familia *Intel*. En esta forma, un número de 32 bits como 0xa3fda382 se guarda como 82, a3, df, a3, usando posiciones de memoria consecutivas.

El formato *Little-Endian* tiene la ventaja, para nosotros, de que es más fácil sumar números usando índices que crecen en el orden natural, de menor a mayor. La desventaja es que, cuando se usan enteros con signo, éste se almacena en el último lugar.

La aritmética binaria hace que, además, nuestro programa sume correctamente tanto números positivos como negativos, algo que no pasaba en el ejemplo anterior.

La rutina de suma se simplifica notablemente, aunque no será tan sencillo visualizar los resultados. Además, deberemos usar **unsigned char** para almacenar los datos, con el fin de que los resultados de las sumas no se conviertan en negativos al sumar ciertos valores positivos.

```

typedef unsigned char numero[cadmax];
...
bool Sumar(numero n1, numero n2, numero r) {
    int acarreo = 0;
    int c;

    for(unsigned int i = 0; i < cadmax; i++) {
        c = acarreo+n1[i]+n2[i];
        if(c > 0xff) { c-=256; acarreo=true; }
        else acarreo=false;
        r[i] = c;
    }
    return !acarreo;
}

```

Evidentemente, visualizar en pantalla números almacenados en este formato es un problema. Tendremos que calcular el módulo de dividir entre 10 sucesivamente para obtener los dígitos decimales de derecha a izquierda. Hasta ahora sólo sabemos sumar, lo que convierte este problema en algo no trivial.

Ejemplo 11.4

Vamos a trabajar ahora un poco con estructuras. Crearemos una estructura sencilla para almacenar fracciones:

```

struct fraccion {
    int num;
    int den;
};

```

Ya dije que sería sencilla.

Bueno, este ejemplo consiste en crear un programa que simplifique fracciones, o más concretamente, que use una función que devuelva una fracción simplificada de la que proporcionamos como parámetro de entrada.

Repasemos un poco. Para cada fracción existe un número infinitos de equivalencias, basta multiplicar el numerador y el denominador por un mismo número:

$$\frac{1}{2} = \frac{2}{4} = \frac{3}{6} = \frac{4}{8} = \frac{5}{10} = \dots$$

Simplificar una fracción significa expresarla de la forma más simple, es decir, para los valores mínimos de numerador y denominador.

Para ello necesitamos encontrar el máximo común divisor (MCD) del numerador y del denominador, es decir, el mayor número entero que divide tanto al numerador como al denominador.

Generalmente, el método consiste en descomponer en factores primos los dos números y seleccionar todos los comunes. Su producto será el MCD.

Esto es lo que haremos con nuestro programa:

```
// Simplificación de fracciones
// Agosto de 2009 Con Clase, Salvador Pozo
#include <iostream>
using namespace std;

struct fraccion {
    int num;
    int den;
};

fraccion Simplificar(fraccion);

int MCD(int, int);

int main() {
    fraccion f, s;
    f.num = 1204;
    f.den = 23212;

    s = Simplificar(f);
```



```

        cout << "Simplificar(" << f.num << "/" << f.den << ") =
";
        cout << s.num << "/" << s.den << endl;

        return 0;
    }

    fraccion Simplificar(fraccion f) {
        int mcd = MCD(f.num, f.den);
        f.num /= mcd;
        f.den /= mcd;
        return f;
    }

    int MCD(int n1, int n2) {
        // Buscar los factores primos que dividen tanto a n1
        como a n2
        int resultado = 1; // El 1 siempre es un CD
        int factor = 2;    // Empezamos en 2

        while(factor <= n1 || factor <= n2) {
            while(!(n1 % factor) && !(n2 % factor)) {
                resultado *= factor;
                n1 /= factor;
                n2 /= factor;
            }
            if(factor == 2) factor++; // Si factor es 2, el
siguiente primo es 3
            else factor+=2;          // Si no, elegimos el
siguiente número impar
        }
        return resultado;
    }
}

```

Ejecutar este código en [codepad](#).

Ejemplo 11.5

Siguiendo con este tema, ahora que sabemos simplificar fracciones, vamos a hacer un programa que las sume.

Volviendo al repaso, recordemos que sólo es posible sumar dos fracciones si tienen el mismo denominador. En ese caso, basta con sumar los numeradores, y mantener el denominador común.

Generalmente, se usan dos métodos para sumar dos fracciones. Uno consiste en calcular el mínimo común denominador de las dos fracciones, recalculer los numeradores de las dos fracciones equivalentes, y finalmente sumarlas.

$$\frac{n1}{d1} + \frac{n2}{d2} = \frac{n1 \cdot \text{mcd}/d1}{\text{mcd}} + \frac{n2 \cdot \text{mcd}/d2}{\text{mcd}} = \frac{n1 \cdot \text{mcd}/d1 + n2 \cdot \text{mcd}/d2}{\text{mcd}}$$

El segundo método consiste en encontrar un común denominador más sencillo, que es directamente el producto de los denominadores, y simplificar, si es posible, la fracción resultante:

$$\frac{n1}{d1} + \frac{n2}{d2} = \frac{n1 \cdot d1 \cdot d2 / d1}{d1 \cdot d2} + \frac{n2 \cdot d1 \cdot d2 / d2}{d1 \cdot d2} = \frac{n1 \cdot d2 + n2 \cdot d1}{d1 \cdot d2} \Rightarrow \text{Simplificar}$$

Usaremos este segundo método:

```
// Suma de fracciones
// Agosto de 2009 Con Clase, Salvador Pozo
#include <iostream>
using namespace std;

struct fraccion {
    int num;
    int den;
};

fraccion Simplificar(fraccion);
fraccion Sumar(fraccion, fraccion);
int MCD(int, int);

int main() {
```

```

    fraccion f, g, s;

    f.num=10;
    f.den=3;
    g.num=5;
    g.den=6;
    s = Sumar(f, g);
    cout << "Sumar(" << f.num << "/" << f.den << "," <<
g.num << "/" << g.den << ") = ";
    cout << s.num << "/" << s.den << endl;

    return 0;
}

fraccion Simplificar(fraccion f) {
    int mcd = MCD(f.num,f.den);
    f.num /= mcd;
    f.den /= mcd;
    return f;
}

int MCD(int n1, int n2) {
    // Buscar los factores primos que dividen tanto a n1
    como a n2
    int resultado = 1; // El 1 siempre es un CD
    int factor = 2;    // Empezamos en 2

    while(factor <= n1 || factor <= n2) {
        while(!(n1 % factor) && !(n2 % factor)) {
            resultado *= factor;
            n1 /= factor;
            n2 /= factor;
        }
        if(factor == 2) factor++; // Si factor es 2, el
siguiente primo es 3
        else factor+=2;          // Si no, elegimos el
siguiente número impar
    }
    return resultado;
}

fraccion Sumar(fraccion f1, fraccion f2) {
    fraccion resultado;

    resultado.num = f1.num*f2.den+f1.den*f2.num;
    resultado.den = f1.den*f2.den;
    return Simplificar(resultado);
}

```

Ejecutar este código en [codepad](#).

12 Tipos de objetos IV:

Punteros 1

No, no salgas corriendo todavía. Aunque vamos a empezar con un tema que suele asustar a los estudiantes de C++, no es algo tan terrible como se cuenta. Como se suele decir de los leones: *no son tan fieros como los pintan*.



Vamos a intentar explicar cómo funcionan los punteros de forma que no tengan el aspecto de magia negra ni un galimatías incomprensible.

Pero no bastará con entender lo que se explica en este capítulo. Es relativamente

sencillo saber qué son y cómo funcionan los punteros. Para poder manejarlos es necesario también comprender los punteros, y eso significa saber qué pueden hacer y cómo lo hacen. Para comprender los punteros se necesita práctica, algunos necesitamos más que otros, (y yo considero que no me vendría mal seguir practicando). Incluso cuando ya creas que los dominas, seguramente quedarán nuevos matices por conocer.

Pero seguramente estoy exagerando. Si soy capaz de explicar correctamente los conceptos de este capítulo, pronto te encontrarás

usando punteros en tus programas casi sin darte cuenta.

Los punteros proporcionan la mayor parte de la potencia al 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.

Creo que todos sabemos lo que es un puntero, fuera del ámbito de la programación. Usamos punteros para señalar cosas sobre las que queremos llamar la atención, como marcar puntos en un mapa o detalles en una presentación en pantalla. A menudo, usamos el dedo índice para señalar direcciones o lugares sobre los que estamos hablando o explicando algo. Cuando un dedo no es suficiente, podemos usar punteros. Antiguamente esos punteros eran una vara de madera, pero actualmente se usan punteros laser, aunque la idea es la misma. Un puntero también es el símbolo que representa la posición del ratón en una pantalla gráfica. Estos punteros también se usan para señalar objetos: enlaces, opciones de menú, botones, etc. Un puntero sirve, pues, para apuntar a los objetos a los que nos estamos refiriendo.

Pues en C++ un puntero es exactamente lo mismo. Probablemente habrás notado que a lo largo del curso nos hemos referido a variables, constantes, etc como *objetos*. Esto ha sido intencionado por el siguiente motivo:

C++ está diseñado para la programación orientada a objetos (POO), y en ese paradigma, todas las entidades que podemos manejar son objetos.

Los punteros en C++ sirven para señalar objetos, y también para manipularlos.

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

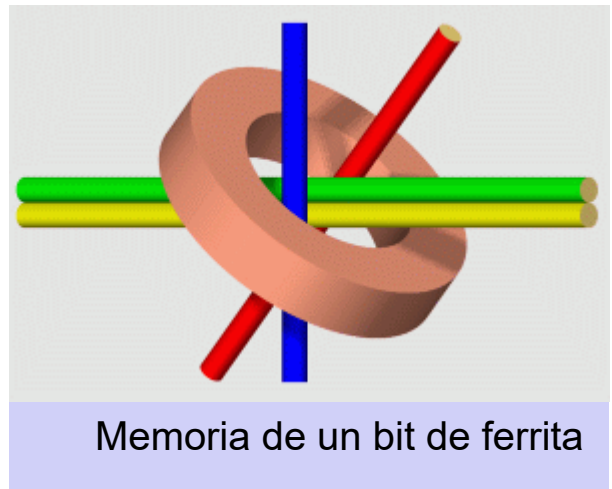
La memoria de un ordenador está compuesta por unidades básicas llamadas bits. Cada bit sólo puede tomar dos valores, normalmente denominados alto y bajo, ó 1 y 0. Pero trabajar con bits no es práctico, y por eso se agrupan.

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*.

Cada *byte* de la memoria de un ordenador tiene una dirección, llamada dirección de memoria.

Los microprocesadores trabajan con una unidad básica de información, a la que se denomina palabra (en inglés *word*). Dependiendo del tipo de microprocesador una palabra puede estar compuesta por uno, dos, cuatro, ocho o dieciséis *bytes*. Hablaremos en estos casos de plataformas de 8, 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, por ejemplo, de 32 bits es que accederemos a posiciones de memoria que serán múltiplos de 4.

Por otra parte, la mayor parte de los objetos que usamos en nuestros programas no caben en una dirección de memoria. La solución utilizada para manejar objetos que ocupen más de un byte es usar posiciones de memoria correlativas. De este modo, la dirección de un objeto es la dirección de memoria de la primera posición que contiene ese objeto.



Dicho de otro modo, si para almacenar un objeto se precisan cuatro *bytes*, y la dirección de memoria de la primera posición es n , el objeto ocupará las posiciones desde n a $n+3$, y la dirección del objeto será, también, n .

Todo esto sucede en el interior de la máquina, y nos importa relativamente poco. Pero 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:

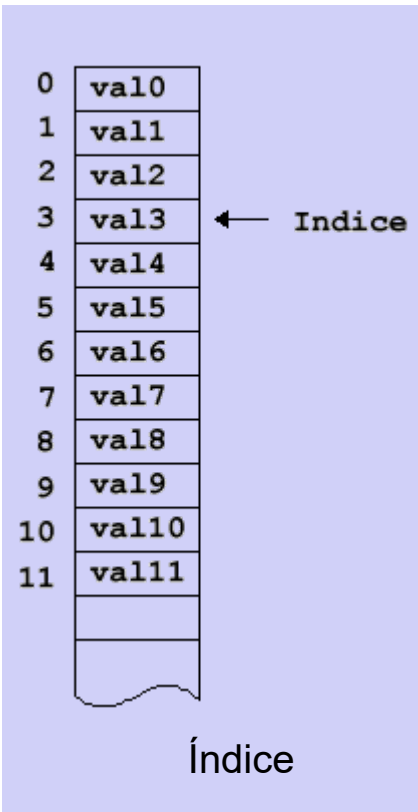
```
cout << "Plataforma de " << 8*sizeof(int) << " bits";
```

Ahora veamos cómo funcionan los punteros. Un puntero es un tipo especial de objeto que contiene, ni más ni menos que, la dirección de memoria de un objeto. Por supuesto, almacenada a partir de esa dirección de memoria puede haber cualquier clase 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, al declarar el puntero.

De hecho, podríamos decir que existen tantos tipos diferentes de punteros como tipos de objetos puedan ser referenciados mediante punteros. Si tenemos esto en cuenta, los punteros que apunten a tipos de objetos distintos, serán tipos diferentes. Por ejemplo, no podemos asignar a un puntero a **char** el valor de un puntero a **int**.

Intentemos ver con mayor claridad el funcionamiento de los punteros. Podemos considerar la memoria del ordenador como un gran *array*, de modo que podemos acceder a cada celda de memoria a través de un índice. Podemos considerar que la primera posición del array es la 0 celda[0].

Si usamos una variable para almacenar el índice, por ejemplo, `indice=0`, entonces `celda[0] == celda[indice]`. Finalmente, si prescindimos de la notación de los *arrays*, podemos ver que el índice se comporta exactamente igual que un puntero.



El puntero *indice* podría tener por ejemplo, el valor 3, en ese caso, el valor apuntado por *indice* tendría el valor 'val3'.

Las celdas de memoria tienen una existencia física, es decir son algo real y existirán siempre, independientemente del valor del puntero. Existirán incluso si no existe el puntero.

De forma recíproca, la existencia o no existencia de un puntero no implica la existencia o la inexistencia del objeto. De la misma forma que el hecho de no señalar a un árbol, no implica la inexistencia del árbol. Algo más oscuro es si tenemos un puntero para árboles, que no esté señalando a un árbol. Un puntero de ese tipo no tendría uso si estamos en

medio del mar: tener ese puntero no crea árboles de forma automática cuando señalemos con él. Es un puntero, no una varita mágica. :-D

Del mismo modo, el valor de la dirección que contiene un puntero no implica que esa dirección sea válida, en el sentido de que no tiene por qué contener la dirección de un objeto del tipo especificado por el puntero.

Supongamos que tenemos un mapa en la pared, y supongamos también que existen diferentes tipos de punteros láser para señalar diferentes tipos de puntos en el mapa (ya sé que esto suena raro, pero usemos la imaginación). Creamos un puntero para señalar ciudades. Nada más crearlo (o encenderlo), el puntero señalará a cualquier sitio, podría señalar incluso a un punto fuera del mapa. En general, daremos por sentado que una vez creado, el puntero no tiene por qué apuntar a una ciudad, y aunque apunte al mapa, podría estar señalando a un mar o a un río.

Con los punteros en C++ ocurre lo mismo. El valor inicial del puntero, cuando se declara, podría estar *fuera del mapa*, es decir, contener direcciones de memoria que no existen. Pero, incluso señalando a un punto de la memoria, es muy probable que no señale a un objeto del tipo adecuado. Debemos considerar esto como el caso más probable, y no usar **jamás** un puntero que no haya sido inicializado correctamente.

Dentro del array de celdas de memoria existirán zonas que contendrán programas y datos, tanto del usuario como del propio sistema operativo o de otros programas, el sistema operativo se encarga de gestionar esa memoria, prohibiendo o protegiendo determinadas zonas.

Pero el propio puntero, como objeto que es, también se almacenará en memoria, y por lo tanto, también tendrá una dirección de memoria. Cuando declaramos un puntero estaremos reservando la memoria necesaria para almacenarlo, aunque, como pasa con el resto de los objetos, el contenido de esa memoria contendrá basura.

En principio, debemos asignar a un puntero, o bien la dirección de un objeto existente, o bien la de uno creado explícitamente durante la ejecución del programa o un valor conocido que indique que no señala a ningún objeto, es decir el valor 0. El sistema operativo, cuanto más avanzado es, mejor suele controlar la memoria. Ese control se traduce en impedir el acceso a determinadas direcciones reservadas por el sistema.

Declaración de punteros

Los punteros se declaran igual que el resto de los objetos, pero precediendo el identificador con un asterisco (*).

Sintaxis:

```
<tipo> *<identificador>;
```

Ejemplos:

```
int *pEntero;  
char *pCaracter;  
struct stPunto *pPunto;
```

Los punteros sólo pueden apuntar a objetos de un tipo determinado, en el ejemplo, *pEntero* sólo puede apuntar a un objeto de tipo int.

La forma:

```
<tipo>* <identificador>;
```

con el (*) junto al tipo, en lugar de junto al identificador del objeto, también está permitida. De hecho, también es legal la forma:

```
<tipo> * <identificador>;
```

Veamos algunos matices. Tomemos el primer ejemplo:

```
int *pEntero;
```

equivale a:

```
int* pEntero;
```

Otro detalle importante es que, aunque las tres formas de situar el asterisco en la declaración son equivalentes, algunas de ellas pueden inducirnos a error, sobre todo si se declaran varios objetos en la misma línea:

```
int* x, y;
```

En este caso, `x` es un puntero a **int**, pero `y` no es más que un objeto de tipo **int**. Colocar el asterisco junto al tipo puede que indique más claramente que estamos declarando un puntero, pero hay que tener en cuenta que sólo afecta al primer objeto declarado, si quisiéramos declarar ambos objetos como punteros a **int** tendríamos que hacerlo de otro modo:

```
int* x, *y;  
// O:  
int *x, *y;  
// O:  
int* x;  
int* y;
```

Obtener punteros a objetos

Los punteros apuntan a objetos, por lo tanto, lo primero que tenemos que saber hacer con nuestros punteros es asignarles direcciones de memoria válidas de objetos.

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

Por supuesto, los tipos tienen que ser "compatibles", no podemos almacenar la dirección de un objeto de tipo **char** en un puntero de tipo **int**.

Por ejemplo:

```
int A;  
int *pA;  
  
pA = &A;
```

Según este ejemplo, *pA* es un puntero a **int** que apunta a la dirección donde se almacena el valor del entero *A*.

Objeto apuntado por un puntero

La operación contraria es obtener el objeto referenciado por un puntero, con el fin de manipularlo, ya sea modificando su valor u obteniendo el valor actual.

Para manipular el objeto apuntado por un puntero usaremos el operador de indirección, que es un asterisco (*).

En C++ es muy habitual que el mismo símbolo se use para varias cosas diferentes, este es el caso del asterisco, que se usa como operador de multiplicación, para la declaración de punteros y, como vemos ahora, como operador de indirección.

Como operador de indirección sólo está permitido usarlo con punteros, y podemos leerlo como "objeto apuntado por".

Por ejemplo:

```
int *pEntero;
int x = 10;
int y;

pEntero = &y;
*pEntero = x; // (1)
```

En (1) asignamos al objeto apuntado por *pEntero* en valor del objeto *x*. Como *pEntero* apunta al objeto *y*, esta sentencia equivale (según la secuencia del programa), a asignar a *y* el valor de *x*.

Diferencia entre punteros y otros objetos

Debemos tener muy claro, en el ejemplo anterior, que *pEntero* es un objeto del tipo "puntero a **int**", pero que **pEntero* NO es un objeto

de tipo **int**, sino una expresión.

¿Por qué decimos esto?

Pues porque, como pasa con todos los objetos en C++, cuando se declaran sólo se reserva espacio para almacenarlos, pero no se asigna ningún valor inicial, (recuerda que nuestro puntero para árboles no crea árbol cada vez que señalemos con él). El contenido del objeto permanecerá sin cambios, de modo que el valor inicial del puntero será aleatorio e indeterminado. Debemos suponer que contiene una dirección no válida.

Si *pEntero* apunta a un objeto de tipo **int**, `*pEntero` será el contenido de ese objeto, pero no olvides que `*pEntero` es un operador aplicado a un objeto de tipo "puntero a **int**". Es decir, `*pEntero` es una expresión, no un objeto.

Declarar un puntero no creará un objeto del tipo al que apunta. Por ejemplo: `int *pEntero;` no crea un objeto de tipo **int** en memoria. Lo que crea es un objeto que *puede* contener la dirección de memoria de un entero.

Podemos decir que existe físicamente un objeto *pEntero*, y también que ese objeto *puede* (aunque esto no es siempre cierto) contener la dirección de un objeto de tipo **int**.

Como todos los objetos, los punteros también contienen "basura" cuando son declarados. Es costumbre dar valores iniciales nulos a los punteros que no apuntan a ningún sitio concreto:

```
int *pEntero = 0; // También podemos asignar el valor NULL
char *pCharacter = 0;
```

NULL es una constante, que está definida como cero en varios ficheros de cabecera, como "cstdio" o "iostream", y normalmente vale 0L. Sin embargo, hay muchos textos que recomiendan usar el valor 0 para asignar a punteros nulos, al menos en C++.

Correspondencia entre *arrays* y punteros

En muchos aspectos, existe una equivalencia entre *arrays* y punteros. De hecho, cuando declaramos un *array* estamos haciendo varias cosas a la vez:

- Declaramos un puntero del mismo tipo que los elementos del *array*.
- Reservamos memoria para todos los elementos del *array*. Los elementos de un *array* se almacenan internamente en la memoria del ordenador en posiciones consecutivas.
- Se inicializa el puntero de modo que apunte al primer elemento del *array*.

Las diferencias entre un *array* y un puntero son dos:

- Que el identificador de un *array* se comporta como un puntero constante, es decir, no podemos hacer que apunte a otra dirección de memoria.
- Que el compilador asocia, de forma automática, una zona de memoria para los elementos del *array*, cosa que no hace para los elementos apuntados por un puntero corriente.

Ejemplo:

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

puntero = vector; /* Equivale a puntero = &vector[0]; (1)
                    esto se lee como "dirección del primer
elemento de vector" */
(*puntero)++;      /* Equivale a vector[0]++; (2) */
puntero++;         /* puntero equivale a asignar a puntero el
valor &vector[1] (3) */
```

¿Qué hace cada una de estas instrucciones?:

En (1) se asigna a *puntero* la dirección del *array*, o más exactamente, la dirección del primer elemento del *array vector*.

En (2) se incrementa el contenido de la memoria apuntada por *puntero*, que es `vector[0]`.

En (3) se incrementa el puntero, esto significa que apuntará a la posición de memoria del siguiente elemento **int**, y no a la siguiente posición de memoria. Es decir, el puntero no se incrementará en una unidad, como tal vez sería lógico esperar, sino en la longitud de un **int**, ya que *puntero* apunta a un objeto de tipo **int**.

Análogamente, la operación:

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

No incrementará la dirección de memoria almacenada en *puntero* en siete posiciones, sino en $7 \times \text{sizeof}(\text{int})$.

Otro ejemplo:

```
struct stComplejo {  
    float real, imaginario;  
} Complejo[10];  
stComplejo *pComplejo; /* Declaración de un puntero */  
  
pComplejo = Complejo; /* Equivale a pComplejo =  
    &Complejo[0]; */  
pComplejo++;           /* pComplejo == &Complejo[1] */
```

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

La correspondencia entre *arrays* y punteros también afecta al operador `[]`. Es decir, podemos usar los corchetes con punteros, igual que los usamos con *arrays*. Pero incluso podemos ir más lejos, ya que es posible usar índices negativos.

Por ejemplo, las siguientes expresiones son equivalentes:


```
*(puntero + 7);  
puntero[7];
```

De forma análoga, el siguiente ejemplo también es válido:

```
int vector[10];  
int *puntero;  
  
puntero = &vector[5];  
puntero[-2] = puntero[2] = 100;
```

Evidentemente, nunca podremos usar un índice negativo con un *array*, ya que estaríamos accediendo a una zona de memoria que no pertenece al *array*, pero eso no tiene por qué ser cierto con punteros.

En este último ejemplo, *puntero* apunta al sexto elemento de *vector*, de modo que *puntero[-2]* apunta al cuarto, es decir, *vector[3]* y *puntero[2]* apunta al octavo, es decir, *vector[7]*.

Operaciones con punteros

La aritmética de punteros es limitada, pero en muchos aspectos muy interesante; y 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 dirección de memoria:

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

```
int a;

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

Sólo hay un caso especial en la asignación de punteros, y es cuando se asigna el valor cero. Este es el único valor que se puede asignar a cualquier puntero, independientemente del tipo de objeto al que apunte.

Operaciones aritméticas

Podemos distinguir dos tipos de operaciones aritméticas con punteros. En uno de los tipos uno de los operandos es un puntero, y el otro un entero. En el otro tipo, ambos operandos son punteros.

Ya hemos visto ejemplos del primer caso. Cada unidad entera que se suma o resta al puntero hace que este apunte a la dirección del siguiente objeto o al anterior, respectivamente, del mismo tipo.

El valor del entero, por lo tanto, no se interpreta como posiciones de memoria física, sino como posiciones de objetos del tipo al que apunta el puntero.

Por ejemplo, si sumamos el valor 2 a un puntero a **int**, y el tipo **int** ocupa cuatro bytes, el puntero apuntará a la dirección ocho bytes mayor a la original. Si se tratase de un puntero a **char**, el puntero avanzará dos posiciones de memoria.

Las restas con enteros operan de modo análogo.

Con este tipo de operaciones podemos usar los operadores de suma, resta, preincremento, postincremento, predecremento y postdecremento. Además, podemos combinar los operadores de suma y resta con los de asignación: += y -=.

En cuanto al otro tipo de operaciones aritméticas, sólo está permitida la resta, ya que la suma de punteros no tiene sentido. Si la suma o resta de un puntero y un entero da como resultado un

puntero, la resta de dos punteros dará como resultado, lógicamente, un entero. Veamos un 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.

Generalmente, este tipo de operaciones sólo tendrá sentido entre punteros que apunten a objetos del mismo tipo, y más frecuentemente, entre punteros que apunten a elementos del mismo *array*.

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*. Podemos usar los operadores <, <=, >= o > para averiguar posiciones relativas entre objetos del mismo tipo apuntados por punteros.

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

```
if (NULL != p)
```

Lo expuesto en el capítulo 9 sobre conversiones implícitas a **bool** también se aplica a punteros, de modo que podemos simplificar esta sentencia como:

```
if(p)
```

Y también:

```
if(NULL == p)
```

O simplemente:

```
if(!p)
```

Nota: No es posible comparar punteros de tipos diferentes, ni aunque ambos sean nulos.

Punteros genéricos

Es posible declarar punteros sin especificar a qué tipo de objeto apuntan:

```
void *<identificador>;
```

Usaremos estos punteros en situaciones donde podemos referirnos a distintos tipos de objetos, ya que podemos hacer que apunten a objetos de cualquier tipo.

Por supuesto, para eso tendremos que hacer un *casting* con punteros, sintaxis:

```
(<tipo> *)<variable puntero>
```

Por ejemplo:

```

#include <iostream>
using namespace std;

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**.

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>
using namespace std;

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

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

    cout << "puntero" << endl;
    cout << e->a << endl;
    cout << e->b << endl;
    cout << "objeto" << 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>
using namespace std;

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;

return 0;
}

```

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>
using namespace std;

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 el nombre de 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 es necesario almacenar 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.

Objetos dinámicos

El uso principal y más potente de los punteros es el manejo de la memoria dinámica.

La memoria se clasifica en muchas categorías, por ahora nos centraremos en algunas de ellas. Cuando se ejecuta un programa, el sistema operativo reserva una zona de memoria para el código o instrucciones del programa y otra para los objetos que se usan durante la ejecución. A menudo estas zonas son la misma, y componen lo que se denomina *memoria local*. También hay otras zonas de memoria, como la pila, que se usa, entre otras cosas, para intercambiar datos entre las funciones. El resto, la memoria que no se usa por ningún programa es lo que se conoce como *heap* o montón. Nuestro programa puede hacer uso de esa memoria

durante la ejecución, de modo que la cantidad de espacio de memoria usado por el programa no está limitada por el diseño ni por las declaraciones de objetos realizadas en el código fuente. Por eso se denomina a este tipo, *memoria dinámica*, ya que tanto la cantidad de memoria como su uso se deciden durante la ejecución, y en general, cambia a lo largo del tiempo, de forma dinámica. Para ello, normalmente se usará memoria del montón, y no se llama así porque sea de peor calidad, sino porque suele haber un buen montón de memoria de este tipo.

C++ dispone de dos operadores para manejar (reservar y liberar) la memoria dinámica, son **new** y **delete**. En C estas acciones se realizan mediante funciones de la biblioteca estándar [stdio](#).

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 fiéis 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>
using namespace std;

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 el puntero a 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. Lo correcto, si no la necesitábamos habría sido liberarla antes de reservar un nuevo bloque usando el mismo puntero, y si la necesitamos, habría que guardar su dirección, por ejemplo usando otro puntero.

Problemas

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 esta función no se pueden usar enteros para recorrer el array, usar sólo punteros y aplicar aritmética de punteros.

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.

3. Implementar en una función el siguiente algoritmo para ordenar un *array* de enteros.

La idea es recorrer simultáneamente el *array* desde el principio y desde el final, comparando los elementos. Si los valores comparados no están en el orden adecuado, se intercambian y se vuelve a empezar el bucle. Si están bien ordenados, se compara el siguiente par.

El proceso termina cuando los punteros se cruzan, ya que eso indica que hemos comparado la primera mitad con la segunda y todos los elementos estaban en el orden correcto.

Usar una función con tres parámetros:

```
void Ordenar(int* vector, int nElementos, bool ascendente);
```

De nuevo, no se deben usar enteros, sólo punteros y aritmética de punteros.

Ejemplos capítulo 12

Ejemplo 12.1

Vamos a realizar un pequeño programa para trabajar con punteros.

Este ejemplo consiste en invertir el orden de los elementos de un vector de enteros, usando sólo puntitos, sin variables auxiliares enteras.

Para ello recorreremos el vector empezando por los dos extremos, e intercambiando cada par de elementos:

```
// Programa que invierte el orden de un vector
// Octubre de 2009 Con Clase, Salvador Pozo

#include <iostream>

using namespace std;

void Mostrar(int*, int);
void Intercambia(int*, int*);

int main() {
    int vector[10] = { 2,5,6,7,9,12,35,67,88,99 };
    int *p, *q;

    Mostrar(vector, 10); // Mostrar estado inicial

    p = vector;          // Primer elemento
    q = &vector[9];      // Ultimo elemento

    while(p < q) { // Bucle de intercambio
        Intercambia(p++, q--);
    }
    Mostrar(vector, 10); // Mostrar estado final
    return 0;
}

void Mostrar(int *v, int n) {
    int *f = &v[n]; // Puntero a posición siguiente al
    último elemento
    while(v < f) {
        cout << *v << " ";
        v++;
    }
    cout << endl;
}
```

```
void Intercambia(int *p, int *q) {  
    // Intercambiar sin usar variables auxiliares:  
    *p += *q;  
    *q = *p-*q;  
    *p -= *q;  
}
```

Ejecutar este código en [codepad](#).

Probablemente parezca que nos hemos complicado la vida, pero se trata de un ejercicio.

Comentemos algunos detalles de este programa.

Primero, el bucle de intercambio. Hemos usado una única sentencia **while**. Como condición hemos usado la comparación de los dos punteros. Esto hace que el puntero *p* recorra la primera mitad del vector, empezando por el principio, y que *q* recorra la otra mitad, empezando por el final. El bucle se repetirá hasta que ambos punteros sean iguales, o hasta que se crucen. Si ambos punteros son iguales (esto sólo sucederá si el número de elementos es impar), los dos apuntarán al elemento central del vector, y no se producirá intercambio, lo cual no es un problema, ya que no hay nada que intercambiar.

La sentencia del bucle `Intercambia(p++, q--);` intercambiará los elementos apuntados por *p* y *q*, y posteriormente, incrementará *p* y decrementará *q*.

Hemos diseñado dos funciones auxiliares.

La primera *Mostrar* muestra el número de valores indicado en el segundo parámetro del vector suministrado mediante el primer parámetro.

He querido evitar, también en este caso, el uso de variables auxiliares de tipo **int**, aunque he decidido hacerlo usando un puntero auxiliar, que apunte a la dirección de un hipotético elemento *n+1* del vector.

Podríamos habernos aprovechado de que los parámetros se pasan por valor, y usar el parámetro *n* como contador:

```

void Mostrar(int *v, int n) {
    while(n-->0) {
        cout << *v << " ";
        v++;
    }
    cout << endl;
}

```

Aunque este código tiene un problema potencial, si el valor inicial de *n* es cero o negativo el bucle no termina.

Podemos arreglar esto añadiendo una verificación inicial:

```

void Mostrar(int *v, int n) {
    if(n > 0) {
        while(n-->0) {
            cout << *v << " ";
            v++;
        }
    }
    cout << endl;
}

```

La segunda función, *Intercambia*, como su nombre indica, intercambia los elementos del vector apuntados por los dos punteros pasados como parámetros.

También en este caso, y aprovechando que se trata de un vector de enteros, he querido evitar usar variables auxiliares. El truco consiste en mantener la información haciendo sumas y restas. Suponiendo que queremos intercambiar los valores *n* y *m*:

1. Sumamos los dos valores y guardamos el resultado en el primero. $n = n + m$
2. El nuevo valor de *m* es el resultado de restar del valor actual de *n*, el valor de *m*. Como *n* ahora contiene (*n*+*m*), el resultado será $n + m - m = n$. $m = n - m$

3. Por último, obtenemos el valor final de n restando de su valor actual ($n+m$) el valor actual de m (que es el valor original de n).
- $$n = n - m$$

Veamos un ejemplo:

```
N = 10, M = 3
1) N = N+M = 10+3 = 13    [N=13, M=3]
2) M = N-M = 13-3 = 10    [N=13, M=10]
3) N = N-M = 13-10 = 3    [N=3, M=10]
```

Esto sólo funciona con vectores de enteros. Con *float* podemos perder precisión en algunas operaciones, y con otros tipos, como estructuras o uniones, ni siquiera tiene sentido.

Ejemplo 12.2

Este ejemplo consiste en mezclar dos vectores ordenados en otro, de modo que también esté ordenado, como en el caso anterior usaremos sólo puntetos, sin variables auxiliares enteras.

Para ello recorreremos los dos vectores de entrada, empezando por el principio, e insertaremos el valor menor de cada uno, y tomaremos el siguiente:

```
// Programa que funde dos vectores ordenados
// Octubre de 2009 Con Clase, Salvador Pozo

#include <iostream>

using namespace std;

void Mostrar(int*, int);

int main() {
    int vector1[10] = { 3, 4, 6, 9, 12, 15, 17, 19, 22, 24 };
    int vector2[10] = { 1, 5, 8, 11, 13, 14, 21, 23, 25, 30 };
}
```

```

    int vectorR[20];

    int *p, *q, *r;
    int *u1, *u2;

    p = vector1;
    q = vector2;
    r = vectorR;
    u1 = &vector1[9];
    u2 = &vector2[9];

    cout << "Vectores de entrada:" << endl;
    Mostrar(vector1, 10);
    Mostrar(vector2, 10);

    while(p <= u1 && q <= u2) {
        if(*p < *q) { *r++ = *p++; }
        else        { *r++ = *q++; }
    }
    // Llegados a este punto, quedarán elementos por
    // copiar en uno de los vectores, así que hay
    // que copiarlos incondicionalmente:
    while(p <= u1) {
        *r++ = *p++;
    }
    while(q <= u2) {
        *r++ = *q++;
    }
    cout << "Resultado:" << endl;
    Mostrar(vectorR, 20);
    return 0;
}

void Mostrar(int *v, int n) {
    int *f = &v[n]; // Puntero a posición siguiente al
    último elemento
    while(v < f) {
        cout << *v << " ";
        v++;
    }
    cout << endl;
}

```

Ejecutar este código en [codepad](#).

Hemos usado sólo punteros porque lo exige el enunciado, pero hubiera sido más lógico usar un índice para recorrer los vectores de

entrada.

Sin embargo, este método simplifica los movimientos de elementos entre los vectores de entrada y el de salida. En las expresiones del tipo $*r++ = *p++$; se evalúa primero la asignación, y después se incrementan los punteros, de modo que todo queda preparado para la siguiente iteración.

Ejemplo 12.3

Implementaremos ahora otro método de ordenación, que generalmente se usa con ficheros de disco secuenciales, pero que también se puede aplicar a vectores.

Nos centraremos ahora en el algoritmo de mezcla natural. Dada una lista de valores, en principio desordenada, este algoritmo consiste en copiar las secuencias ordenadas a otras listas, y después aplicar el método usado en el ejercicio anterior para obtener una lista más ordenada que la original. El proceso se repite hasta que sólo hay una secuencia en la lista.

Este método se puede usar con tantas listas auxiliares como se quiera, dos o más. Cuantas más listas auxiliares se usen, más rápidamente se ordenará la lista. El problema es que para cada nueva lista auxiliar se incrementa la cantidad de memoria necesaria, y sobre todo, la complejidad del programa.

Implementaremos este algoritmo con dos vectores auxiliares.

Pero veamos un ejemplo. Partamos del siguiente vector:

```
34 33 23 3 54 21 1 99 12 32 51 64
```

La primera etapa consiste en copiar elementos a los vectores auxiliares, alternando cada vez que se rompa una secuencia ordenada. En este caso, los dos vectores de salida quedarán:

```
34 23 21 12 32 51 64
33 3 54 1 99
```

A continuación, mezclamos los dos vectores, usando un algoritmo parecido al del ejemplo anterior. La diferencia es que nuestros vectores no están ordenados, y por lo tanto, las mezclas serán de subconjuntos de esos vectores:

```
33 34 3 23 21 54 1 12 32 51 64 99
```

Repetimos el proceso:

```
33 34 21 54  
3 23 1 12 32 51 64 99
```

Y volvemos a mezclar:

```
3 23 33 34 1 12 21 32 51 54 64 99
```

Volvemos a repetir el proceso:

```
3 23 33 34  
1 12 21 32 51 54 64 99
```

Y mezclamos otra vez:

```
1 3 12 21 23 32 33 34 51 54 64 99
```

Y la lista queda ordenada.

Tenemos que buscar una condición para detectar el final del proceso. Lo más sencillo es contar el número de secuencias ordenadas durante el proceso de mezcla. Si hay más de una, hay que repetir el proceso.

Sobre el papel es muy sencillo, pero no será tan fácil implementarlo. Veremos que necesitaremos un montón de variables auxiliares.

El proceso de separar los vectores es relativamente simple, pero veremos que mezclar los vectores resultantes no es tan sencillo, ya que sólo están parcialmente ordenados, y hay que mezclar cada par de fragmentos por separado.

Para cada vector auxiliar necesitamos tres punteros. Uno lo usaremos para recordar la dirección de inicio. Esto nos permitirá "rebobinar" el vector cada vez que tengamos que empezar a procesarlo y liberar su memoria cuando terminemos el trabajo. El segundo puntero lo usaremos para llenar el vector en el proceso de separar los fragmentos ordenados. Una vez terminado ese proceso, el puntero se usará para saber cuantos elementos contiene y para comparaciones cuando lo recorramos. El último puntero lo usaremos para recorrer el vector en la etapa de fusión de los vectores auxiliares.

El proceso de separar responde a este algoritmo:

Activar la lista 1 como salida.

Tomar el primer elemento del vector y copiarlo a la lista activa.

Bucle: Mientras no se alcance el final de la lista

Si el siguiente elemento del vector es menor que el último copiado (el último de la lista activa),
cambiar de lista activa.

Copiar el siguiente elemento del vector a la lista activa.

fin del bucle

El proceso de mezcla responde a este otro algoritmo:

Iniciar punteros: colocar cada uno al principio de una lista.

Contador de tramos <- cero

Empezamos un tramo nuevo <- verdadero

Bucle: Mientras queden elementos en alguna de las listas auxiliares

CASO A: Empezamos un tramo nuevo

Pasar a la lista de salida el menor de los elementos actuales de cada lista auxiliar

Incrementar el número de tramos

Empezamos un tramo nuevo <- falso

CASO B: La lista auxiliar 1 está vacía

Para cada uno de los elementos restantes de la lista 2

Si el elemento a copiar es menor que el último de la salida:

incrementar tramos

Copiar elemento a lista de salida

CASO C: la lista auxiliar 2 está vacía

Para cada uno de los elementos restantes de la lista 1

Si el elemento a copiar es menor que el último de la salida:

incrementar tramos

Copiar elemento a lista de salida

CASO D: El primer elemento de cada lista auxiliar es mayor al último de la de salida

Copiar el elemento de la lista auxiliar
que contenga el menor

CASO E: El primer elemento de la lista 1 es mayor
que el último de la de salida,
el primero de la lista 2 es menor

Copiar el resto del tramo de la lista
1. Hasta que se encuentre un
elemento menor que el último
copiado.

CASO F: El primer elemento de la lista 2 es mayor
que el último de la de salida,
el primero de la lista 1 es menor

Copiar el resto del tramo de la lista
2. Hasta que se encuentre un
elemento menor que el último
copiado.

CASO G: El primer elemento de cada lista auxiliar
es menor que el último de la salida. Tramos
agotados.

Empezamos nuevo tramo <-
verdadero

Cerrar el bucle

Hemos tenido en cuenta todos los casos posibles. El caso A se
dará cuando empecemos el proceso de fusión, y también cuando
terminemos de fundir cada uno de las parejas de tramos.

Los casos B y C se darán cuando una de las listas termine, y queden elementos en la otra. Podríamos pensar que esto sólo va a suceder cuando termine la segunda, ya que el número de tramos será igual en ambas listas o habrá uno más en la primera que en la segunda. Sin embargo, cuando el número de tramos sea igual en ambas listas, puede que se termine de copiar primero los de la primera lista, quedando elementos por copiar en la segunda.

El D es el caso general, cuando quedan elementos en los fragmentos de las dos listas auxiliares. Se transfiere el menor de los dos.

Los casos E y F se dan cuando uno de los fragmentos se ha terminado, y se deben transferir los elementos que queden en el otro fragmento.

Finalmente, el caso G se da cuando los dos fragmentos se ha agotado. En ese caso comenzamos un nuevo tramo en la lista de salida. Si queda algún fragmento en alguna de las listas, volveremos a estar en el caso A, si no, habremos terminado.

La implementación es la siguiente:

```
// Ordenar usando el método de mezcla natural
// Octubre de 2009 Con Clase, Salvador Pozo

#include <iostream>

using namespace std;

void Mostrar(int*, int);
int MezclaNatural(int*, int);

int main() {
    int vector[12] = { 34, 33, 23, 3, 54, 21, 1, 99, 12, 32,
51, 64 };

    cout << "Vector inicial:" << endl;
    Mostrar(vector, 12);
    while(MezclaNatural(vector, 12) > 1);
    cout << "Vector final:" << endl;
    Mostrar(vector, 12);
    return 0;
}
```

```

}

void Mostrar(int *v, int n) {
    int *f = &v[n]; // Puntero a posición siguiente al
    último elemento
    while(v < f) {
        cout << *v << " ";
        v++;
    }
    cout << endl;
}

int MezclaNatural(int *v, int n) {
    int *aux[2];
    int *iaux[2];
    int *iaux2[2];
    int *i, *f;
    int activa=0; // Activamos el primer vector auxiliar
    int tramo;
    bool tramonuevo;

    aux[0] = iaux[0] = new int[12];
    aux[1] = iaux[1] = new int[12];
    i = v;
    f = &v[n];
    // El primer elemento se copia siempre al primer vector:
    *iaux[activa]++ = *i++;
    // Separar vector en auxiliares:
    while(i < f) {
        if(*i < *(i-1)) activa++;
        if(activa >=2) activa -= 2;
        *iaux[activa]++ = *i++;
    }

    // Fundir vectores auxiliares:
    iaux2[0] = aux[0];
    iaux2[1] = aux[1];
    i = v;
    tramo = 0;
    tramonuevo = true;
    while(iaux2[0] < iaux[0] || iaux2[1] < iaux[1]) {
        if(tramonuevo) { // caso A
            // El primer elemento lo añadimos directamente:
            if(*iaux2[0] < *iaux2[1]) { *i++ = *iaux2[0]++;
        }

        else { *i++ = *iaux2[1]++;

        }

        tramo++;
    }

```

```

        tramonuevo = false;
    } else // caso B
    if(iaux2[0] == iaux[0]) {
        while(iaux2[1] < iaux[1]) {
            if(*iaux2[1] < i[-1]) tramo++;
            *i++ = *iaux2[1]++;
        }
    } else // caso C
    if(iaux2[1] == iaux[1]) {
        while(iaux2[0] < iaux[0]) {
            if(*iaux2[0] < i[-1]) tramo++;
            *i++ = *iaux2[0]++;
        }
    } else // caso D
    if(*iaux2[0] > i[-1] && *iaux2[1] > i[-1]) {
        if(*iaux2[0] < *iaux2[1]) { *i++ = *iaux2[0]++;
    }
        else { *i++ = *iaux2[1]++;
    }

    } else // caso E
    if(*iaux2[0] > i[-1])
        while(iaux2[0] < iaux[0] && *iaux2[0] > i[-1]) {
            *i++ = *iaux2[0]++;
        }
    else // caso F
    if(*iaux2[1] > i[-1])
        while(iaux2[1] < iaux[1] && *iaux2[1] > i[-1]) {
            *i++ = *iaux2[1]++;
        }
    else { // caso G
        tramonuevo = true;
    }
}
delete[] aux[1];
delete[] aux[0];
return tramo;
}

```

Ejecutar este código en [codepad](#).

13 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>
```

Por ejemplo:

```
int *punt;  
int x = 10;  
  
punt = &x;
```

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

Sintaxis:

```
*<puntero>
```

Ejemplo:

```
int *punt;  
int x;  
  
x = *punt;
```

Operadores . y ->

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

Sintaxis:

```
<variable_estructura>.<nombre_de_variable>
```

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

Sintaxis:

```
<puntero_a_estructura>-><nombre_de_variable>
```

Ejemplo:

```
struct punto {  
    int x;  
    int y;  
};  
  
punto p1;  
punto *p2;  
  
p1.x = 10;  
p1.y = 20;  
p2->x = 30;  
p2->y = 40;
```

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.

El preprocesador es un programa auxiliar, que forma parte del compilador, y que procesa el fichero fuente antes de que sea compilado. En realidad se limita a seguir las órdenes expresadas en forma de directivas del preprocesador, modificando el programa fuente antes de que sea compilado.

Veremos, sin embargo dos de las más usadas.

Directiva define

La directiva **define**, sirve para definir macros. Cada macro es una especie de fórmula para la sustitución de texto dentro del fichero fuente, y puede usar, opcionalmente parámetros.

Sintaxis:

```
#define <identificador_de_macro> <secuencia>
```

El preprocesador sustituirá cada ocurrencia del <identificador_de_macro> en el fichero fuente, por la <secuencia>, (aunque con algunas excepciones). Cada sustitución se denomina "expansión de la macro", y la secuencia se suele conocer como "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>
using namespace std;

#define mult1(a,b) a*b
#define mult2(a,b) ((a)*(b))

int main() {
    // En este caso ambas macros funcionan bien: (1)
    cout << mult1(4,5) << endl;
    cout << mult2(4,5) << endl;
    // En este caso la primera macro no funciona, ¿por qué?:
    (2)
    cout << mult1(2+2,2+3) << endl;
    cout << mult2(2+2,2+3) << endl;

    return 0;
}
```

¿Por qué falla la macro `mult1` en el segundo caso?. Para averiguarlo, 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, ¿por qué?:
    cout << 2+2*2+3 << endl;
    cout << ((2+2)*(2+3)) << endl;

    return 0;
}

```

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.

El tercer caso es "raro", no he encontrado ningún ejemplo que lo use, y yo no he recurrido nunca a él. Pero el caso es que se puede usar, por ejemplo:

```
#define FICHERO "trabajo.h"

#include FICHERO

int main()
{
    ...
}
```

Es un ejemplo simple, pero en el {cc:023:capítulo 23} veremos más directivas del preprocesador, y verás el modo en que se puede definir FICHERO de forma condicional, de modo que el fichero a incluir puede depender de variables de entorno, de la plataforma, etc.

Por supuesto la macro puede ser una fórmula, y el nombre del fichero puede crearse usando esa fórmula.

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>] <tipo> [(<inicialización>)]  
[::]new [<emplazamiento>] (<tipo>) [(<inicialización>)]  
[::]new [<emplazamiento>] <tipo>[<número_elementos>]  
[::]new [<emplazamiento>] (<tipo>)[<número_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* dinámicos. 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 muy 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* dinámicos.

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

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.

De todos modos, es buena idea asignar el valor 0 a los punteros que no han sido inicializados y a los que han sido liberados. También es bueno preguntar si un puntero es nulo antes de intentar liberar la memoria dinámica que le fue asignada.

Nota: los operadores **new** y **delete** son propios de C++. En C se usan funciones, como **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;

    // Cadena de 122 más el nulo:
    c = new char[123];
    // Array de 10 punteros a float:
    f = new float *[10]; (1)
    // Cada elemento del array es un array de 10 float
    for(n = 0; n < 10; n++) f[n] = new float[10]; (2)
    // 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;
}
```

Nota: f es un puntero que apunta a un puntero que a su vez apunta a un **float**. Un puntero puede apuntar a cualquier tipo de

variable, incluidos otros punteros.

Este ejemplo nos permite crear *arrays* dinámicos de dos dimensiones. La línea (1) crea un *array* de 10 punteros a **float**. La (2) crea 10 *arrays* de **floats**. El comportamiento final de *f* es el mismo que si lo hubiéramos declarado como:

```
float f[10][10];
```

Otro ejemplo:

```
#include <iostream>
using namespace std;

int main() {
    int *x;

    x = new int(67);
    cout << *x << endl;
    delete x;
}
```

En este caso, reservamos memoria para un entero, se asigna la dirección de la memoria obtenida al puntero *x*, y además, se asigna el valor 67 al contenido de esa memoria.

Palabras reservadas usadas en este capítulo

delete, new.

14 Operadores III: Precedencia

Normalmente, las expresiones con operadores se evalúan de izquierda a derecha, aunque no todos, ciertos operadores que se evalúan y se asocian de derecha a izquierda. Además no todos los operadores tienen la misma prioridad, algunos se evalúan antes que otros, de hecho, existe un orden muy concreto en los operadores en la evaluación de expresiones. Esta propiedad de los operadores se conoce como precedencia o prioridad.

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 unitarios: ! ~ + - ++ -- & (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 unitarios 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.

15 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 "objetos" locales de la función, estos "objetos" son de hecho los propios parámetros.

Lo veremos mucho mejor con un ejemplo:

```
#include <iostream>
using namespace std;

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 + 2;  
    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 los objetos a y b como parámetros. Dentro de "funcion" esos parámetros se llaman n y m , y sus valores son modificados. Sin embargo al retornar a *main*, a y b conservan sus valores originales. ¿Por qué?

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

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

Si los parámetros por valor no funcionasen así, no sería posible llamar a una función con valores constantes o literales.

Referencias a objetos

Las referencias sirven para definir "alias" o nombres alternativos para un mismo objeto. Para ello se usa el operador de referencia (&).

Sintaxis:

```
<tipo> &<alias> = <objeto de referencia>  
<tipo> &<alias>
```

La primera forma es la que se usa para declarar objetos que son referencias, la asignación es obligatoria ya que no pueden definirse referencias indeterminadas.

La segunda forma es la que se usa para definir parámetros por referencia en funciones, en estos casos, las asignaciones son implícitas.

Ejemplo:

```
#include <iostream>
using namespace std;

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

    a = 10;
    cout << r << endl;

    return 0;
}
```

En este ejemplo los identificadores `a` y `r` se refieren al mismo objeto, cualquier cambio en una de ellos se produce en el otro, ya que son, de hecho, el mismo objeto.

El compilador mantiene una tabla en la que se hace corresponder una dirección de memoria para cada identificador de objeto. A cada nuevo objeto declarado se le reserva un espacio de memoria y se almacena su dirección. En el caso de las referencias, se omite ese paso, y se asigna la dirección de otro objeto que ya existía previamente.

De ese modo, podemos tener varios identificadores que hacen referencia al mismo objeto, pero sin usar punteros.

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 objetos. Por ejemplo:

```

#include <iostream>
using namespace std;

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; // (1)
    es ilegal pasar constantes como parámetros cuando
    estos son referencias */

    return 0;
}

int funcion(int &n, int &m) {
    n = n + 2;
    m = m - 5;
    return n+m;
}

```

En este caso, los objetos "a" y "b" tendrán valores distintos después de llamar a la función. Cualquier cambio de valor que realicemos en los parámetros dentro de la función, se hará también en los objetos referenciadas.

Esto quiere decir que no podremos llamar a la función con parámetros constantes, como se indica en (1), ya que aunque es posible definir referencias a constantes, en este ejemplo, la función tiene como parámetros referencias a objetos variables.

Y si bien es posible hacer un *casting* implícito de un objeto variable a uno constante, no es posible hacerlo en el sentido inverso. Un objeto constante no puede tratarse como objeto variable.

Punteros como parámetros de funciones

Esto ya lo hemos dicho anteriormente, pero no está de más repetirlo: los punteros son objetos como cualquier otro en C++, por lo tanto, tienen las mismas propiedades y limitaciones que el resto de los objetos.

Cuando pasamos un puntero como parámetro de una función por valor pasa lo mismo que con cualquier otro objeto.

Dentro de la función trabajamos con una copia del parámetro, que en este caso es un puntero. Por lo tanto, igual que pasaba con el ejemplo anterior, las modificaciones en el valor del parámetro serán locales a la función y no se mantendrán después de retornar.

Sin embargo, no sucede lo mismo con el objeto apuntado por el puntero, puesto que en ambos casos será el mismo, ya que tanto el puntero como el parámetro tienen como valor la misma dirección de memoria. Por lo tanto, los cambios que hagamos en los objetos apuntados por el puntero se conservarán al abandonar la función.

Ejemplo:

```
#include <iostream>
using namespace std;

void funcion(int *q);

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

    a = 100;
    p = &a;
    // Llamamos a funcion con un puntero
    funcion(p); // (1)
    cout << "Variable a: " << a << endl;
    cout << "Variable *p: " << *p << endl;
    // Llamada a funcion con la dirección de "a" (constante)
    funcion(&a); // (2)
    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++;
}

```

Dentro de la función se modifica el valor apuntado por el puntero, y los cambios permanecen al abandonar la función. Sin embargo, los cambios en el propio puntero son locales, y no se conservan al regresar.

Analogamente a como lo hicimos antes al pasar una constante literal, podemos pasar punteros variables o constantes como parámetro a la función. En (1) usamos un variable de tipo puntero, en (2) usamos un puntero constante.

De modo que con este tipo de declaración de parámetro para función estamos pasando 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.

En esta versión de la función, las modificaciones que se hagan para el valor del puntero pasado como parámetro, se mantendrán al regresar al punto de llamada.

Nota:

En C no existen referencias de este tipo, y la forma de pasar parámetros por referencia es usar un puntero. Por

supuesto, para pasar una referencia a un puntero se usa un puntero a puntero, etc.

La idea original de la implementación de referencias en C++ no es la de crear parámetros variables (algo que existe, por ejemplo, en PASCAL), sino ahorrar recursos a la hora de pasar como parámetros objetos de gran tamaño.

Por ejemplo, supongamos que necesitamos pasar como parámetro a una función un objeto que ocupe varios miles de bytes. Si se pasa por valor, en el momento de la llamada se debe copiar en la pila todo el objeto, y la función recupera ese objeto de la pila y se lo asigna al parámetro. Sin embargo, si se usa una referencia, este paso se limita a copiar una dirección de memoria.

En general, se considera mala práctica usar referencias en parámetros con el fin de modificar su valor en la función. La explicación es que es muy difícil, durante el análisis y depuración, encontrar errores si no estamos seguros del valor de los parámetros después de una llamada a función. Del mismo modo, se complica la actualización si los valores de ciertas variables pueden ser diferentes, dependiendo dónde se inserte nuevo código.

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 permanentes aún después de retornar.

Sin embargo, si sólo pasamos el nombre del *array* de más de una dimensión no podremos acceder a los elementos del *array* mediante subíndices, 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>
using namespace std;

#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;
    ...
}
```

Otro problema es que, a no ser que diseñemos nuestra función para que trabaje con un *array* de un tamaño fijo, en la función nunca nos será posible calcular el número de elementos del *array*.

En este último ejemplo, la tabla siempre será de NxM elementos, pero la misma función admite como parámetros arrays donde la primera dimensión puede tener cualquier valor. El problema es cómo averiguar cual es ese valor.

El operador **sizeof** no nos sirve en este caso, ya que nos devolverá siempre el tamaño de un puntero, y no el del *array* completo.

Por lo tanto, deberemos crear algún mecanismo para poder calcular ese tamaño. El más evidente es usar otro parámetro para eso. De hecho, debemos usar uno para cada dimensión. Pero de momento veamos cómo nos las arreglamos con una:

```
#include <iostream>
using namespace std;

#define N 10
#define M 20

void funcion(int tabla[][M], int n);

int main() {
    int Tabla[N][M];
    int Tabla2[50][M];

    funcion(Tabla, N);
    funcion(Tabla2, 50);
    return 0;
}

void funcion(int tabla[][M], int n) {
    cout << n*M << endl;
}
```

Generalizando más, si queremos que nuestra función pueda trabajar con cualquier *array* de dos dimensiones, deberemos prescindir de la declaración como *array*, y declarar el parámetro como un puntero. Ahora, para acceder al *array* tendremos que tener en cuenta que los elementos se guardan en posiciones de memoria consecutivas, y que a dos índices consecutivos de la dimensión más a la derecha, le corresponden posiciones de memoria adyacentes.

Por ejemplo, en un array declarado como `int tabla[3][4]`, las posiciones de `tabla[1][2]` y `tabla[1][3]` son consecutivas. En memoria se almacenan los valores de `tabla[0][0]` a `tabla[0][3]`, a continuación los de `tabla[1][0]` a `tabla[1][3]` y finalmente los de `tabla[2][0]` a `tabla[2][3]`.

Si sólo disponemos del puntero al primer elemento de la tabla, aún podemos acceder a cualquier elemento, pero tendremos que hacer nosotros las cuentas. Por ejemplo, si "t" es un puntero al primer elemento de tabla, para acceder al elemento `tabla[1][2]` usaremos la expresión `t[1*4+2]`, y en general para acceder al elemento `tabla[x][y]`, usaremos la expresión `t[x*4+y]`.

El mismo razonamiento sirve para arrays de más dimensiones. En un array de cuatro, por ejemplo, `int array[N][M][O][P];`, para acceder al elemento `array[n][m][o][p]`, siendo "a" un puntero al primer elemento, usaremos la expresión: `a[p+o*P+m*O*P+n*M*O*P]` o también `a[p+P*(n+m+o)+O*(m+n)+M*n]`.

Por ejemplo:

```
#include <iostream>
using namespace std;

#define N 10
#define M 20
#define O 25
#define P 40

void funcion(int *tabla, int n, int m, int o, int p);

int main() {
    int Tabla[N][M][O][P];

    Tabla[3][4][12][15] = 13;
    cout << "Tabla[3][4][12][15] = " <<
        Tabla[3][4][12][15] << endl;
    funcion((int*)Tabla, N, M, O, P);
    return 0;
}

void funcion(int *tabla, int n, int m, int o, int p) {
    cout << "tabla[3][4][12][15] = " <<
        tabla[3*m*o*p+4*o*p+12*p+15] << 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.

En el caso de las estructuras, los objetos pueden ser muy grandes, ocupando mucha memoria. Es por eso que es frecuente enviar referencias como parámetros, aunque no se vayan a modificar los valores de la estructura. Esto evita que el valor del objeto deba ser depositado en la pila para ser recuperado por la función posteriormente.

Funciones que devuelven referencias

También es posible devolver referencias desde una función, para ello basta con declarar el valor de retorno como una referencia.

Sintaxis:

```
<tipo> &<identificador_función>(<lista_parámetros>);
```

Esto nos permite que la llamada a una función se comporte como un objeto, ya que una referencia se comporta exactamente igual que el objeto al que referencia, y podremos hacer cosas como usar esa llamada en expresiones de asignación. Veamos un ejemplo:

```
#include <iostream>
using namespace std;

int &Acceso(int*, int);
```

```
int main() {
    int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    Acceso(array, 3)++;
    Acceso(array, 6) = Acceso(array, 4) + 10;

    cout << "Valor de array[3]: " << array[3] << endl;
    cout << "Valor de array[6]: " << array[6] << endl;

    return 0;
}

int &Acceso(int* vector, int indice) {
    return vector[indice];
}
```

Este uso de las referencias es una herramienta muy potente y útil que, como veremos más adelante, tiene múltiples aplicaciones.

Por ejemplo, veremos en el capítulo sobre sobrecarga que este mecanismo es imprescindible.

16 Tipos de variables V:

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 [<identificador>] {  
    [<tipo> <nombre_variable>[,<nombre_variable>,...]];  
} [<variable_union>[,<variable_union>,...];
```

El identificador de la unión es un nombre opcional para referirse a la unión.

Las variables de unión son objetos declarados del tipo de la unión, y su inclusión también es opcional.

Sin embargo, como en el caso de las estructuras, 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 los objetos. La particularidad es que cada elemento se almacenará comenzando 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 objetos del tipo de la unión en cualquier parte del programa, de la siguiente forma:

```
[union] <identifiador_de_unión> <variable>[,<variable>...];
```

La palabra clave **union** es opcional en la declaración de objetos en C++. Aunque en C es obligatoria.

Ejemplo:

```
#include <iostream>
using namespace std;

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(UnionEjemplo.A) << endl;
    cout << sizeof(UnionEjemplo.B) << endl;
    cout << sizeof(UnionEjemplo.C) << endl;

    return 0;
}
```

Supongamos que en nuestro ordenador, **int** ocupa cuatro bytes, **char** un byte y **double** ocho bytes. La forma en que se almacena la información en la unión del ejemplo sería la siguiente:

Byte Byte Byte Byte Byte Byte Byte Byte

0 1 2 3 4 5 6 7

A

B

C

Por el contrario, los mismos objetos almacenados en una estructura tendrían la siguiente disposición:

Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7	8	9	10	11	12	
A				B	C								

Nota:

Unas notas sobre el ejemplo:

- Observa que hemos hecho un "casting" del puntero al elemento B de la unión. Si no lo hiciéramos 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.
- Observa que el tamaño de la unión es el del elemento más grande.

Veamos otro ejemplo, pero 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 resultar útil:

```
struct stCoor3D {
    int X, Y, Z;
};

union unCoor3D {
    struct stCoor3D N;
    int Coor[3];
} Punto;
```

Con estas declaraciones, en nuestros programas podremos referirnos a la coordenada Y de estas dos formas:

```
Punto.N.Y  
Punto.Coor[1]
```

Estructuras anónimas

Como ya vimos en el capítulo sobre estructuras, una {cc:011b#STR_Anonimas:estructura anónima} es la que carece de identificador de tipo de estructura y de identificador 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]
```

Usar estructuras anónimas dentro de una unión tiene la ventaja de que nos ahorramos escribir el identificador de la estructura para acceder a sus campos. Esto no sólo es útil por el ahorro de código, sino sobre todo, porque el código es mucho más claro.

Inicialización de uniones

Las uniones solo pueden ser inicializadas en su declaración mediante su primer miembro.

Por ejemplo, en la primera unión:

```
union unEjemplo {  
    int A;  
    char B;  
    double C;  
} UnionEjemplo;
```

Podemos iniciar objetos de este tipo asignando un entero:

```
unEjemplo x = {10}; // int  
unEjemplo y = {'a'}; // char  
unEjemplo z = {10.02}; // double
```

Si usamos un carácter, como en el caso de 'y', se hará una conversión de tipo a **int** de forma automática, y se asignará el valor correcto. En el caso de 'z', se produce un aviso, por democión automática.

Quiero llamar tu atención sobre el modo de inicializar uniones. Al igual que pasa con otros tipos agregados, como *arrays* y estructuras, hay que usar llaves para incluir una lista de valores iniciales. En el caso de la unión, esa lista tendrá un único elemento, ya que todos los miembros comparten la misma zona de memoria, y sólo está permitido usar el primero para las inicializaciones.

Discriminadores

Supongamos que necesitamos almacenar en un *array* datos de diferentes tipos, nos importa más almacenarlos todos en la misma estructura. Por ejemplo, en la gestión de una biblioteca queremos crear una tabla que contenga, de forma indiscriminada, libros, revistas y películas.

Podemos crear una unión, ejemplar, que contenga un elemento de cada tipo, y después un *array* de ejemplares.

```
struct tipoLibro {
    int codigo;
    char autor[80];
    char titulo[80];
    char editorial[32];
    int anno;
};

struct tipoRevista {
    int codigo;
    char nombre[32];
    int mes;
    int anno;
};

struct tipoPelicula {
    int codigo;
    char titulo[80];
    char director[80];
    char productora[32];
    int anno;
};

union tipoEjemplar {
    tipoLibro l;
    tipoRevista r;
    tipoPelicula p;
};

tipoEjemplar tabla[100];
```

Pero hay un problema, del que quizás ya te hayas percatado...

Cuando accedamos a un elemento de la tabla, ¿cómo sabemos si contiene un libro, una revista o una película?

Lo que se suele hacer es añadir un elemento más que indique qué tipo de dato contiene la unión. A ese elemento se le llama **discriminador**:

```
enum eEjemplar { libro, revista, pelicula };

struct tipoEjemplar {
    eEjemplar tipo;
    union {
        tipoLibro l;
        tipoRevista r;
        tipoPelicula p;
    };
};
```

Usando el discriminador podemos averiguar qué tipo de publicación estamos manejando, y mostrar o asignar los valores adecuados.

Funciones dentro de uniones

Como en el caso de las estructuras, en las uniones también se pueden incluir como miembros funciones, *constructores* y *destructores*.

Del mismo modo, es posible crear tantos *constructores* como se necesiten. En cuanto a este aspecto, las estructuras y uniones son equivalentes.

Según la norma ANSI, todos los campos de las uniones deben ser públicos, y no se permiten los modificadores **private** y **protected**.

Un objeto que tenga constructor o destructor no puede ser utilizado como miembro de una unión. Esta limitación tiene su lógica, puesto que la memoria de cada elemento de una unión se comparte, no tendría sentido que los constructores de algunos elementos modificasen el contenido de esa memoria, ya que afectan directamente a los valores del resto de los elementos.

Una unión no puede participar en la jerarquía de clases; no puede ser derivada de ninguna clase, ni ser una clase base. Aunque sí pueden tener un constructor y ser miembros de clases.

Palabras reservadas usadas en este capítulo

`union`.

Ejemplos capítulo 16

Ejemplo 16.1

Una aplicación clásica de las uniones es ofrecer la posibilidad de manipular los mismos datos de formas diferentes. Por ejemplo, podemos crear una unión para manipular un byte a tres niveles: completo, bit a bit o nibble a nibble. (Un nibble es un conjunto de cuatro bits).

```
#include <iostream>
#include <cstdio>

using namespace std;

union byte {
    unsigned char b;
    struct {
        unsigned char bit8:1;
        unsigned char bit7:1;
        unsigned char bit6:1;
        unsigned char bit5:1;
        unsigned char bit4:1;
        unsigned char bit3:1;
        unsigned char bit2:1;
        unsigned char bit1:1;
    };
    struct {
        unsigned char nibble2:4;
        unsigned char nibble1:4;
    };
};

int main() {
    byte x;

    x.b = 0x2a;
```



```

printf("%d\n", x.bit1);
printf("%d\n", x.bit2);
printf("%d\n", x.bit3);
printf("%d\n", x.bit4);
printf("%d\n", x.bit5);
printf("%d\n", x.bit6);
printf("%d\n", x.bit7);
printf("%d\n", x.bit8);

printf("%x\n", x.nibble1);
printf("%x\n", x.nibble2);

x.bit2 = 1;
x.bit3 = 0;
printf("%02x\n", x.b);

return 0;
}

```

Ejemplo 16.2



Síntesis aditiva

En los ordenadores, por norma general, se usan números para codificar colores. Los dispositivos gráficos usan la síntesis aditiva de color. Partiendo de los colores básicos: rojo, azul y verde, se puede obtener cualquier color combinándolos en diferentes proporciones.

También por norma general, se suelen usar ocho bits para cada color, como hay tres componentes, eso da un total de 24 bits.

En una palabra de 32 bits sobrarían ocho. A veces, esos ocho bits se usan como componente alfa, que indica la transparencia, o cómo se combina el color con el que existía previamente en ese punto.

En un valor entero de 32 bits, se usan los ocho de menor peso para codificar el valor del color rojo. Se suele usar la letra 'R' (Red=Rojo). En los ocho bits siguientes se codifica el color verde. Se usa la letra 'G' (Green=Verde). En los ocho siguientes se codifica el

azul. Se usa la letra 'B' (Blue=Azul). Estos tres valores codifican un color en formato RGB.

Si se usan los 32 bits, los ocho restantes codifican la componente Alfa (Alpha), de transparencia.

Algunas aplicaciones y funciones de los APIs trabajan con enteros de 32 bits para manejar colores, pero a menudo es interesante acceder los componentes básicos directamente.

Podemos crear una unión para codificar colores según estas reglas, y que además de que nos permitan manipular el color como un valor único, podamos acceder a los componentes por separado:

```
#include <iostream>

using namespace std;

union color {
    unsigned int c;
    struct {
        unsigned char red;
        unsigned char green;
        unsigned char blue;
        unsigned char alpha;
    };
};

int main() {
    color c1 = { 0x80fedc12 };

    cout << "Color: " << dec << c1.c << " - " << hex << c1.c
    << endl;
    cout << "Rojo:  " << dec << (int)c1.red << " - " << hex
    << (int)c1.red << endl;
    cout << "Verde: " << dec << (int)c1.green << " - " << hex
    << (int)c1.green << endl;
    cout << "Azul:  " << dec << (int)c1.blue << " - " << hex
    << (int)c1.blue << endl;
    cout << "Alfa:  " << dec << (int)c1.alpha << " - " << hex
    << (int)c1.alpha << endl;

    c1.red = 0x42;
    c1.green = 0xde;
    cout << "Color: " << dec << c1.c << " - " << hex << c1.c
    << endl;
```

```

        cout << "Rojo:  " << dec << (int)c1.red << " - " << hex
<< (int)c1.red << endl;
        cout << "Verde: " << dec << (int)c1.green << " - " << hex
<< (int)c1.green << endl;
        cout << "Azul:  " << dec << (int)c1.blue << " - " << hex
<< (int)c1.blue << endl;
        cout << "Alfa:  " << dec << (int)c1.alpha << " - " << hex
<< (int)c1.alpha << endl;

        return 0;
}

```

Ejemplo 16.3

Veamos ahora cómo usar funciones dentro de uniones.

Para este ejemplo crearemos un constructor para cada tipo de dato que contenga la unión. Esto nos permitirá evitar la limitación de la inicialización de objetos de este tipo, ya que el compilador elegirá el constructor adecuado en función del valor suministrado.

Podemos reinterpretar, entonces, la regla que dice que sólo podemos inicializar uniones usando el primer elemento.

En realidad, lo que pasa es que el compilador sólo crea un constructor por defecto para las uniones y que el parámetro elegido para ese constructor es el primero. Nada nos impide, pues, crear nuestros propios constructores para modificar el comportamiento predefinido.

```

#include <iostream>
#include <cstring>

using namespace std;

union ejemplo {
    int x;
    double d;
    char cad[8];
    ejemplo(int i) : x(i) {}
    ejemplo(double n) : d(n) {}
    ejemplo(const char *c) {
        strncpy(cad, c, 7);
    }
}

```

```

        cad[7] = 0;
    }
};

int main() {
    ejemplo A(23);
    ejemplo B(123.323);
    ejemplo C("hola a todos");

    cout << "A: " << A.x << endl;
    cout << "B: " << B.d << endl;
    cout << "C: " << C.cad << endl;

    return 0;
}

```

Ejecutar este código en [codepad](#).

Vemos en el ejemplo que se puede invocar a los constructores de la forma normal, o implícitamente, como en el caso del objeto D, para el que se suministra una cadena, que evidentemente, no es el tipo del primer elemento de la unión.

Ejemplo 16.4

Vamos a completar el ejemplo de los discriminadores, añadiendo código para iniciar y visualizar elementos del *array*.

```

// Ejemplo de unión con discriminador
// 2009 Con Clase, Salvador Pozo
#include <iostream>

using namespace std;

struct tipoLibro {
    int codigo;
    char autor[80];
    char titulo[80];
    char editorial[32];
    int anno;
};

struct tipoRevista {

```

```

    int codigo;
    char nombre[32];
    int mes;
    int anno;
};

struct tipoPelicula {
    int codigo;
    char titulo[80];
    char director[80];
    char productora[32];
    int anno;
};

enum eEjemplar { libro, revista, pelicula };

struct tipoEjemplar {
    eEjemplar tipo;
    union {
        tipoLibro l;
        tipoRevista r;
        tipoPelicula p;
    };
};

tipoEjemplar tabla[100];

int main() {
    tabla[0].tipo = libro;
    tabla[0].l.codigo = 3;
    strcpy(tabla[0].l.titulo, "El señor de los anillos");
    strcpy(tabla[0].l.autor, "J.R.R. Tolkien");
    tabla[0].l.anno = 1954;

    tabla[1].tipo = revista;
    tabla[1].r.codigo = 12;
    strcpy(tabla[1].r.nombre, "National Geographic");
    tabla[1].r.mes = 11;
    tabla[1].r.anno = 2009;

    tabla[2].tipo = pelicula;
    tabla[2].p.codigo = 43;
    strcpy(tabla[2].p.titulo, "Blade Runner");
    strcpy(tabla[2].p.director, "Ridley Scott");
    strcpy(tabla[2].p.productora, "Warner Bros. Pictures");
    tabla[2].l.anno = 1982;

    for(int i=0; i < 3; i++) {

```

```
        switch(tabla[i].tipo) {
            case libro:
                cout << "[" << tabla[i].l.codigo << "]" Libro
titulo: " << tabla[i].l.titulo << endl;
                break;
            case revista:
                cout << "[" << tabla[i].r.codigo << "]"
Revista nombre: " << tabla[i].r.nombre << endl;
                break;
            case pelicula:
                cout << "[" << tabla[i].p.codigo << "]"
Película titulo: " << tabla[i].p.titulo << endl;
                break;
        }
    }
    return 0;
}
```

Ejecutar este código en [codepad](#).

17 Tipos de variables VI:

Punteros 2

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. Usaremos un puntero normal para crear vectores dinámicos, uno doble para tablas, etc.

Por ejemplo, crearemos una tabla dinámicamente. Para ello se usan los punteros a punteros.

Veamos la declaración de un puntero a puntero:

```
int **tabla;
```

"tabla" es un puntero que apunta a un objeto 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;
```

Otra diferencia con los *arrays* normales es que 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>
using namespace std;

int main() {
    int **tabla;
    int n = 134;
    int m = 231;
    int i;
```



```

// Array de punteros a int:
tabla = new int*[n];
// n arrays de m ints
for(i = 0; i < n; i++)
    tabla[i] = new int[m];
tabla[21][33] = 123;
cout << tabla[21][33] << endl;
// Liberar memoria:
for(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 implementar esta tabla:

```
#include <iostream>
using namespace std;

#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() {
    int i;

    // 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(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(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. Aunque en C++ es preferible usar variables constantes, como este tema aún no lo hemos visto, seguiremos usando macros.

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 $100 \times 100 = 10000$ elementos.

Tabla triangular $100 \times 99 / 2 = 4950$ 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.

Problemas

1. Usando como base el ejemplo anterior, realizar los siguientes cambios:

- Modificar el código para que "tabla" sea una variable local de *main*.

- Definir una función con el prototipo `void AsignarDistancia(int**, int, int, int);`, para asignar valores a distancias entre dos ciudades. El primer parámetro será la tabla de distancias, los dos siguientes parámetros serán identificadores de dos ciudades y el cuarto la distancia entre dichas ciudades.

Por ejemplo `AsignarDistancia(tabla, CIUDAD_E, CIUDAD_B, 123);`.

- Definir una función con el prototipo `void Intercambio(int &, int &);`, para intercambiar los contenidos de dos variables enteras.

Realizar los cambios necesarios en el programa para que se usen estas nuevas funciones siempre que sea posible.

2. Ordenar un array de float aleatorios, para ello, crear un array dinámico con el mismo número de elementos que contenga valores enteros, cada uno de ellos será un índice del array de floats a ordenar.

Ordenar los índices en este segundo array según el orden ascendente del array de números, pero sin modificar el orden ni el contenido del array de floats, que debe permanecer constante.

Por ejemplo, si el array dado contiene los valores: 1.32, 4.21, 2.33, 0.23, 8.35, 2.32, se debe crear un segundo array de enteros dinámico, que una vez ordenado debe contener los valores: 3, 0, 5, 2, 1, 4.

Para ordenar el array de enteros se debe usar la función [qsort](#).

3. Modificar el programa anterior para añadir un segundo array de enteros que contenga los índices del array de floats ordenados de mayor a menor, además del que ya tenía, con los índices de los floats ordenados de menor a mayor.
4. Concatenar dos arrays de enteros ordenados en un tercero, de modo que contenga los elementos de los dos, mezclados de modo que se mantenga el orden.

Por ejemplo:

```
int a1[] = {1,3,4,7,8,9,12,15,16,17,21,23,25};
int a2[] =
{2,5,6,10,11,13,14,18,19,20,22,24,26,27,28};
```

El resultado debe ser un tercer array con todos los elementos presentes en los dos arrays dados, y manteniendo el orden ascendente.

18 Operadores IV: Más operadores

Alguien dijo una vez que C prácticamente tiene un operador para cada instrucción de ensamblador. De hecho C y mucho más 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 de bits, cinco binarios y uno unitario: "&", "|", "^", "~", ">>" 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 "0", en caso contrario "1".

El operador "|" corresponde a la operación lógica "OR", o en álgebra de Boole 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, o bien con la combinación de teclas AltGr+4 seguido de un espacio), 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.

De todos modos, no es demasiado complicado. Existe una correspondencia directa entre los dígitos hexadecimales y combinaciones de cuatro dígitos binarios, veamos una tabla:

Hexadecimal Binario Hexadecimal Binario

0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

```
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 $E1 \text{ ? } E2 : E3$, 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 puede definir la macro "max":

```
#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".

Otros ejemplos prácticos:

Cuando el número a mostrar sea uno, el texto será un singular "elemento", si es otro número, será el plural "elementos".

```
cout << "Tenemos " << n << ((n!=1) ? "elementos" :  
"elemento") << endl;
```

Mostrar los valores de un *array* separados con comas, en seis columnas:

```
int v[] = {
```

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,  
24 };  
    int N = 24;  
    for(int i = 0; i < N; i++)  
        cout << v[i] << ((i%6==5 || i==N-1) ? "\n" : ",");
```

19 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++ dispone 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 struct {  
    int x;  
    int y;  
    int z;  
} Punto3D;
```

Define un nuevo tipo Punto3D, partiendo de una estructura.

```
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.

Palabras reservadas usadas en este capítulo

typedef.

Ejemplos capítulo 18 y 19

Ejemplo 19.1

Siguiendo con el problema de trabajar con enteros grandes, y aprovechando que ya sabemos manejar operadores de bits, veamos otra alternativa a la de los ejemplos del capítulo 9.

Hay una solución intermedia, que consiste en usar codificación BCD. No aprovecha tanto la capacidad de almacenamiento como la forma binaria pura y dificulta un poco más las operaciones para hacer las sumas, pero como contrapartida es mucho más fácil mostrar los resultados.

Nota:

La codificación BCD (Binary-Coded Decimal) consiste en usar cuatro bits para cada dígito decimal, concretamente, los códigos 0000 al 1001, es decir, del 0 al 9. Como usamos cuatro bits por dígito, en un byte podemos almacenar dos dígitos BCD.


```

    numero n2={
        0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
        0x00,0x09,0x98,0x12 };
    numero suma;

    Sumar(n1, n2, suma);
    MostrarBCD(n1);
    cout << " + ";
    MostrarBCD(n2);
    cout << " = ";
    MostrarBCD(suma);
    cout << endl;
    return 0;
}

bool Sumar(numero n1, numero n2, numero r) {
    int acarreo = 0;
    int c;

    for(int i = cadmax-1; i >= 0; i--) {
        // Sumar digito de menor peso:
        c = acarreo+(n1[i] & 0x0f) + (n2[i] & 0x0f);
        if(c > 9) { c-=10; acarreo=1; }
        else acarreo=0;
        r[i] = c;
        // Sumar digito de mayor peso:
        c = acarreo+((n1[i] >> 4) & 0x0f) + ((n2[i] >> 4) &
0x0f);
        if(c > 9) { c-=10; acarreo=1; }
        else acarreo=0;
        r[i] |= (c << 4);
    }
    return !acarreo;
}

void MostrarBCD(numero n) {
    char c;
    bool cero=true;

    for(unsigned int i = 0; i < cadmax; i++) {
        c = '0' + ((n[i] >> 4) & 0x0f);
        if(c != '0') cero=false;
    }
}

```

```
        if(!cero) cout << c;
        c = '0' + (n[i] & 0x0f);
        if(c != '0') cero=false;
        if(!cero) cout << c;
    }
}
```

Ejecutar este código en [codepad](#).

El método para extraer cada uno de los dígitos almacenados en un byte es el siguiente.

Para el de menor peso se usan los cuatro bits de la derecha, de modo que basta con hacer una operación AND de bits con la máscara 00001111, es decir, el valor 0x0f.

Nota:

Llamámos máscara a un valor binario que al combinarlo mediante una operación AND con otro, deja sólo los bits que queremos preservar, enmascarando el resto. En el valor usado como máscara, los bits con valor "1" preservan el valor de los bits, y los bits con valor "0" no, ya que cualquier bit AND "0" da como resultado "0" y cualquier bit AND "1" no cambia de valor.

Para el de mayor peso hacemos una rotación de cuatro bits a la derecha y después, enmascaramos el valor resultante con 0x0f.

En rigor, para este segundo caso no es necesario enmascarar el resultado, ya que las rotaciones de bits a la derecha introducen ceros por la izquierda.

Para volver a "empaquetar" los dos dígitos en un valor byte hacemos el proceso contrario. Rotamos a la izquierda el dígito de mayor peso cuatro bits y lo combinamos con el de menor peso usando el operador de bits OR.

Ejemplo 19.2

Veamos otro ejemplo de aplicación de operadores de bits.

Probablemente hayas oído hablar de algo llamado CRC. Seguramente en mensajes de error *Error de CRC*, al copiar ficheros, etc.

El CRC es una función que se aplica a un conjunto de datos de longitud indeterminada y produce un valor de longitud fija. Se usa para detectar errores durante la transmisión o el almacenamiento de datos.

La mecánica es sencilla:

1. El emisor de un mensaje (o fichero) calcula su CRC y lo añade al mensaje.
2. El receptor lee el mensaje (o fichero) y también calcula su CRC.
3. El receptor posee ahora dos valores de CRC, el que ha recibido con el mensaje, y el que ha calculado. Si ambos son iguales significa que el mensaje no ha sido alterado durante la transmisión. Si son diferentes, es que ha habido un error.

El algoritmo para calcular el CRC admite una entrada de tipo *stream*, es decir, se puede ir calculando el CRC a medida que los datos se generan o se reciben.

Si te interesa la parte teórica del CRC, mira este enlace: [Wikipedia.org](https://es.wikipedia.org/wiki/Algoritmo_CRC).

El algoritmo consiste en:

- Elegimos el valor del polinomio adecuado, dependerá del tamaño del CRC.
- Partimos de un valor de CRC semilla, normalmente 0xFFFF.
- Bucle: - Para cada bit del valor de entrada (empezando por la izquierda). - Si es "1" el nuevo valor del CRC será el resultado de aplicar la operación OR exclusivo entre el valor anterior del CRC y el polinomio. - Cerrar el bucle.

Se suelen usar tres valores diferentes para los polinomios, que han demostrado que restringen al máximo la probabilidad de no

detectar errores. Elegiríamos el adecuado en función del tamaño del CRC o de los datos.

- CRC-12: $X^{12}+X^{11}+X^3+X^2+X+1$
- CRC-16: $X^{16}+X^{15}+X^2+1$
- CRC-CCITT: $X^{16}+X^{12}+X^5+1$

El CRC-12 se utiliza cuando la longitud del carácter es de 6 bits, mientras que los otros dos se utilizan con caracteres de 8 bits.

```
// Calculo de CRC
// (C) 2009 Con Clase
// Salvador Pozo
#include <iostream>

using namespace std;

const unsigned int CCITT_POLYNOM = 0x1021;
typedef unsigned short int WORD;
typedef unsigned char BYTE;

WORD CalcCRC(WORD uCRC, BYTE bData);

int main() {
    WORD uCRC = 0xffff;

    char cadena[] = "Cadena de prueba para calculo de CRC";

    for(int i = 0; cadena[i]; i++)
        uCRC = CalcCRC(uCRC, (BYTE)cadena[i]);
    cout << "CRC: " << hex << uCRC << endl;
    return 0;
}

WORD CalcCRC(WORD uCRC, BYTE bData) {
    int i;
    BYTE bD = bData;
    for(i = 0; i < 8; i++) {
        uCRC <<= 1;
        if(bD & 0x0080) uCRC ^= CCITT_POLYNOM;
        bD <<= 1;
    }
}
```

```
    return uCRC;
}
```

Ejecutar este código en [codepad](#).

Este programa ilustra el uso de **typedef** para la definición de tipos como WORD o BYTE.

En la función CalcCRC se usa aritmética de bits, en desplazamientos y en la función XOR.

Ejemplo 19.3

Retomemos el ejemplo del capítulo 16 sobre la codificación de colores. En esa ocasión usamos una unión para acceder a las componentes de color empaquetadas en un entero largo sin signo. Veamos ahora cómo podemos usar operadores de bits para lo mismo:

```
// Codificación de colores
// aritmética de bits
// (C) 2009 Con Clase
// Salvador Pozo
#include <iostream>

using namespace std;

typedef unsigned long int color ;
typedef unsigned char BYTE;

inline BYTE ObtenerValorRojo(color);
inline BYTE ObtenerValorVerde(color);
inline BYTE ObtenerValorAzul(color);
inline BYTE ObtenerValorAlfa(color);
inline color ModificarValorRojo(color, BYTE);
inline color ModificarValorVerde(color, BYTE);
inline color ModificarValorAzul(color, BYTE);
inline color ModificarValorAlfa(color, BYTE);

int main() {
    color c1 = 0x80fedc12;
```

```

    cout << "Color: " << dec << c1 << " - " << hex << c1 <<
endl;
    cout << "Rojo: " << dec << (int)ObtenerValorRojo(c1) <<
" - " << hex << (int)ObtenerValorRojo(c1) << endl;
    cout << "Verde: " << dec << (int)ObtenerValorVerde(c1)
<< " - " << hex << (int)ObtenerValorVerde(c1) << endl;
    cout << "Azul: " << dec << (int)ObtenerValorAzul(c1) <<
" - " << hex << (int)ObtenerValorAzul(c1) << endl;
    cout << "Alfa: " << dec << (int)ObtenerValorAlfa(c1) <<
" - " << hex << (int)ObtenerValorAlfa(c1) << endl;

    c1 = ModificarValorRojo(c1, 0x42);
    c1 = ModificarValorVerde(c1, 0xde);
    cout << "Color: " << dec << c1 << " - " << hex << c1 <<
endl;
    cout << "Rojo: " << dec << (int)ObtenerValorRojo(c1) <<
" - " << hex << (int)ObtenerValorRojo(c1) << endl;
    cout << "Verde: " << dec << (int)ObtenerValorVerde(c1)
<< " - " << hex << (int)ObtenerValorVerde(c1) << endl;
    cout << "Azul: " << dec << (int)ObtenerValorAzul(c1) <<
" - " << hex << (int)ObtenerValorAzul(c1) << endl;
    cout << "Alfa: " << dec << (int)ObtenerValorAlfa(c1) <<
" - " << hex << (int)ObtenerValorAlfa(c1) << endl;

    return 0;
}

inline BYTE ObtenerValorRojo(color c) {
    return (BYTE)(c);
}

inline BYTE ObtenerValorVerde(color c) {
    return (BYTE)(c >> 8);
}

inline BYTE ObtenerValorAzul(color c) {
    return (BYTE)(c >> 16);
}

inline BYTE ObtenerValorAlfa(color c) {
    return (BYTE)(c >> 24);
}

inline color ModificarValorRojo(color c, BYTE r) {
    return (c & 0xffffffff00) | (color)r;
}

inline color ModificarValorVerde(color c, BYTE g) {

```

```
    return (c & 0xffff00ff) | (color)(g << 8);  
}  
  
inline color ModificarValorAzul(color c, BYTE b) {  
    return (c & 0xff00ffff) | (color)(b << 16);  
}  
  
inline color ModificarValorAlfa(color c, BYTE a) {  
    return (c & 0x00ffffff) | (color)(a << 24);  
}
```

Ejecutar este código en [codepad](#).

Para extraer una de las componentes de color, rotamos a la derecha tantos bits como sean necesarios y retornamos el byte de menor peso. Esto lo podemos hacer, como en este ejemplo, mediante un *casting* o aplicando la máscara 0x000000ff mediante el operador AND (&).

Para modificar una de las componentes, el proceso es el contrario. Primero eliminamos la componente original, aplicando la máscara adecuada, por ejemplo, para el color verde es 0xffff00ff, después rotamos el valor de la nueva componente a la izquierda tantos bits como sea necesario. Finalmente, unimos ambos valores usando el operador OR (|).

20 Funciones II: más cosas

Aún quedan algunas cosas interesantes por explicar sobre las funciones en C++.

Parámetros con valores por defecto

En algunas funciones suele suceder que para ciertos parámetros se repiten frecuentemente los mismos valores cada vez que necesitamos invocarlas.

Por ejemplo, podemos tener una función que calcule la velocidad final de un cuerpo en caída libre. Para hacer el [cálculo](#) necesitaremos algunos parámetros: velocidad inicial, altura...

$$h = v_0 \cdot t + \frac{1}{2} \cdot g \cdot t^2$$

$$v_f = v_0 + g \cdot t$$

donde:

h es altura, o espacio recorrido

g es la fuerza de la gravedad, o aceleración

v_0 es la velocidad inicial

v_f es la velocidad final y

t es el tiempo

La fuerza de la gravedad también es un parámetro que hay que tener en cuenta, aunque podemos considerar que casi siempre nuestros cálculos se referirán a caídas que tendrán lugar en el planeta Tierra, por lo que podemos estar tentados de considerar la gravedad como una constante. Sin embargo, C++ nos permite tomar una solución intermedia. Podemos hacer que la fuerza de la gravedad sea uno de los parámetros de la función, y darle como valor por defecto el que tiene en la Tierra. De este modo, cuando no proporcionemos un valor para ese parámetro en una llamada, se

tomará el valor por defecto, y en caso contrario, se usará el valor proporcionado.

Durante el programa, cuando se llama a la función incluyendo valores para esos 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 para esos parámetros que hemos definido.

Por ejemplo:

```
#include <iostream>
#include <cmath>

using namespace std;

double VelocidadFinal(double h, double v0=0.0, double
g=9.8);

int main() {
    cout << "Velocidad final en caida libre desde 100
metros,\n" <<
        "partiendo del reposo en la Tierra" <<
        VelocidadFinal(100) << "m/s" << endl;
    cout << "Velocidad final en caida libre desde 100
metros,\n" <<
        "con una velocidad inicial de 10m/s en la Tierra" <<
        VelocidadFinal(100, 10) << "m/s" << endl;
    cout << "Velocidad final en caida libre desde 100
metros,\n" <<
        "partiendo del reposo en la Luna" <<
        VelocidadFinal(100, 0, 1.6) << "m/s" << endl;
    cout << "Velocidad final en caida libre desde 100
metros,\n" <<
        "con una velocidad inicial de 10m/s en la Luna" <<
        VelocidadFinal(100, 10, 1.6) << "m/s" << endl;
    return 0;
}

double VelocidadFinal(double h, double v0, double g) {
    double t = (-v0+sqrt(v0*v0 + 2*g*h))/g;
    return v0 + g*t;
}
```

Nota: En este programa hemos usado la función `sqrt`. Se trata de una función ANSI C, que está declarada en el fichero de cabecera `math`, y que, evidentemente, sirve para calcular raíces cuadradas.

La salida de este programa será:

```
Velocidad final en caída libre desde 100 metros,  
partiendo del reposo en la Tierra 44.2719 m/s  
Velocidad final en caída libre desde 100 metros,  
con una velocidad inicial de 10m/s en la Tierra 45.3872 m/s  
Velocidad final en caída libre desde 100 metros,  
partiendo del reposo en la Luna 17.8885 m/s  
Velocidad final en caída libre desde 100 metros,  
con una velocidad inicial de 10m/s en la Luna 20.4939 m/s
```

La primera llamada a la función "VelocidadFinal" dará como salida la velocidad final para una caída libre desde 100 metros de altura. Como no hemos indicado ningún valor para la velocidad inicial, se tomará el valor por defecto de 0 m/s. Y como tampoco hemos indicado un valor para la gravedad, se tomará el valor por defecto, correspondiente a la fuerza de la gravedad en la Tierra.

En la segunda llamada hemos indicado explícitamente un valor para el segundo parámetro, dejando sin especificar el tercero. Como en el caso anterior, se tomará el valor por defecto para la fuerza de la gravedad, que es 9.8.

Este método tiene algunas limitaciones, por otra parte, muy lógicas:

- Sólo los últimos argumentos de las funciones pueden tener valores por defecto.
- De estos, sólo los últimos argumentos pueden ser omitidos en una llamada.
- Cada uno de los valores por defecto debe especificarse bien en los prototipos, o bien en las declaraciones, pero no en ambos.

En la tercera y cuarta llamadas hemos tenido que especificar los tres parámetros, a pesar de que en la tercera el valor de la velocidad

inicial es cero. Si sólo especificamos dos parámetros, el programa interpretará que el que falta es el último, y no el segundo. El compilador no puede adivinar qué parámetro queremos omitir, por eso es necesario aplicar reglas estrictas.

Cuando se declaren valores de parámetros por defecto en prototipos, no es necesario indicar el nombre de los parámetros.

Por ejemplo:

```
void funcion(int = 1); // Legal
void funcion1(int a, int b=0, int c=1); // Legal
void funcion2(int a=1, int b, int c); // Ilegal
void funcion3(int, int, int=1); // Legal
...
void funcion3(int a, int b=3, int c) // Legal
{
}
```

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" es 1
funcion1(12); // Legal, los valores para "b" y "c" son 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 declararemos los parámetros conocidos del modo normal, debe existir al menos un parámetro de este tipo. Los parámetros desconocidos se sustituyen por tres puntos (...), del siguiente modo:

```
<tipo_valor_retorno> <identificador>  
(<lista_parámetros_conocidos>, ...);
```

Los parámetros se pasan usando la pila, (esto es siempre así con todos los parámetros, pero normalmente no tendremos que prestar atención a este hecho). Además es el programador el responsable de decidir el tipo de cada argumento, lo cual limita algo 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 "[cstdarg](#)", estas macros permiten manejar más *fácilmente* las listas de argumentos desconocidos.

Tipos

En el fichero de cabecera "[cstdarg](#)" se define el tipo [va_list](#):

```
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 de parámetros desconocidos.

```
<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.

Leer la lista de parámetros

```
<tipo> 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  
}
```

Es necesario diseñar un sistema que permita determinar cuál es el último valor de la lista de parámetros, de modo que no queden parámetros por procesar o que no se procesen más de la cuenta.

Una forma es hacer que el último valor de la lista de parámetros en la llamada a la función sea un 0, (o de forma más general, un valor conocido).

También puede usarse uno de los parámetros conocidos para pasar a la función la cuenta de los parámetros desconocidos.

Además de esto, es necesario que el programador conozca el tipo de cada parámetro, para así poder leerlos adecuadamente. Una forma es que todos los parámetros sean del mismo tipo. Otra, que se use un mecanismo como el de la función "printf", donde analizando el primer parámetro se pueden deducir el tipo de todos los demás. Este último sistema tiene la ventaja de que también sirve para saber el número de parámetros.

Ejemplos:

```
#include <iostream>
#include <cstdarg>
using namespace std;

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>
#include <cstring>
#include <cstdarg>
using namespace std;

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 cada parámetro */
    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 pueden 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" (*argument counter*), es el número de argumentos que se han especificado en la línea de comandos. El segundo, "argv", (*argument values*) 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: "C:\programasc\programa", "arg1", "arg2", "arg3" y "arg4".

Ejemplo:

```
#include <iostream>
using namespace std;

int main(int argc, char **argv) {
    for(int i = 0; i < argc; i++)
        cout << argv[i] << " ";
    cout << endl;
    return 0;
}
```

Funciones inline

Cuando usamos el nombre de una función, indicando valores para sus argumentos, dentro de un programa, decimos que *llamamos* o *invocamos* a esa función. Esto quiere decir que el procesador guarda la dirección actual, "salta" a la dirección donde comienza el código de la función, la ejecuta, recupera la dirección guardada previamente, y retorna al punto desde el que fue llamada.

Esto es cierto para las funciones que hemos usado hasta ahora, pero hay un tipo especial de funciones que trabajan de otro modo. En lugar de existir una única copia de la función dentro del código, si se declara una función como **inline**, lo que se hace es insertar el código de la función, en el lugar (y cada vez) que sea llamada.

Sintaxis:

```
inline <tipo> <nombre_de_funcion>(<lista_de_parámetros>);
```

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** cuando éstas son de gran tamaño, aunque con funciones pequeñas puede ser recomendable, ya que pueden producir programas más rápidos. 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. De hecho, es incluso posible obtener un puntero a una función declarada **inline**.

Nota: **inline** es exclusivo de C++, y no está disponible en C.

Ejemplos:

```
#include <iostream>
using namespace std;

inline int mayor(int a, int b) {
    if(a > b) return a;
    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;
}
```

Punteros a funciones

Tanto en C como en C++ se pueden declarar punteros a funciones.

Sintaxis:

```
<tipo> (*<identificador>)(<lista_de_parámetros>);
```

De esta forma se declara un puntero a una función que devuelve un valor de tipo <tipo> y acepta la lista de parámetros especificada.

Es muy importante usar los paréntesis para agrupar el asterisco con el identificador, ya que de otro modo estaríamos declarando una función que devuelve un puntero al tipo especificado y que admite la lista de parámetros indicada.

No tiene sentido declarar variables de tipo función, es decir, la sintaxis indicada, prescindiendo del '*' lo que realmente declara es un prototipo, y no es posible asignarle un valor a un prototipo, como se puede hacer con los punteros, sino que únicamente podremos definir la función.

Ejemplos:

```
int (*pfuncion1)(); (1)
void (*pfuncion2)(int); (2)
float *(*pfuncion3)(char*, int); (3)
void (*pfuncion4)(void (*)(int)); (4)
int (*pfuncion5[10])(int); (5)
```

El ejemplo 1 declara un puntero, "pfuncion1" a una función que devuelve un **int** y no acepta parámetros.

El ejemplo 2 declara un puntero, "pfuncion2" a una función que no devuelve valor y que acepta un parámetro de tipo **int**.

El ejemplo 3 a una función que devuelve un puntero a **float** y admite dos parámetros: un puntero a **char** y un **int**.

El 4, declara una función "pfuncion4" que no devuelve valor y acepta un parámetro. Ese parámetro debe ser un puntero a una función que tampoco devuelve valor y admite como parámetro un **int**.

El 5 declara un *array* de punteros a función, cada una de ellas devuelve un **int** y admite como parámetro un **int**.

Este otro ejemplo:

```
int *(pfuncionx)();
```

Equivale a:

```
int *pfuncionx();
```

Que, claramente, es una declaración de un prototipo de una función que devuelve un puntero a **int** y no admite parámetros.

Utilidad de los punteros a funciones

La utilidad de los punteros a funciones se manifiesta sobre todo cuando se personalizan ciertas funciones de biblioteca. Podemos por ejemplo, diseñar una función de biblioteca que admita como parámetro una función, que debe crear el usuario (en este caso otro programador), para que la función de biblioteca complete su funcionamiento.

Este es el caso de la función [qsort](#), declarada en [cstdlib](#). Si nos fijamos en su prototipo:

```
void qsort(void *base, size_t nmemb, size_t tamanyo,  
           int (*comparar)(const void *, const void *));
```

Vemos que el cuarto parámetro es un puntero a una función *comparar* que devuelve un **int** y admite dos parámetros de tipo puntero genérico.

Esto permite a la biblioteca [cstdlib](#) definir una función para ordenar *arrays* independientemente de su tipo, ya que para comparar elementos del *array* se usa una función definida por el usuario, y [qsort](#) puede invocarla después.

Asignación de punteros a funciones

Una vez declarado uno de estos punteros, se comporta como una variable cualquiera, podemos por lo tanto, usarlo como

parámetro en funciones, o asignarle valores, por supuesto, del mismo tipo.

```
int funcion();
...
int (*pf1)(); // Puntero a función sin argumentos
               // que devuelve un int.
pf1 = funcion; // Asignamos al puntero pf1 la
               // función "funcion"
...
int funcion() {
    return 1;
}
```

La asignación es tan simple como asignar el nombre de la función.

Nota: Aunque muchos compiladores lo admiten, no es recomendable aplicar el operador de dirección (&) al nombre de la función

pf1 = &funcion;

La forma propuesta en el ejemplo es la recomendable.

Llamadas a través de un puntero a función

Para invocar a la función usando el puntero, sólo hay que usar el identificador del puntero como si se tratase de una función. En realidad, el puntero se comporta exactamente igual que un "alias" de la función a la que apunta.

```
int x = pf1();
```

De este modo, llamamos a la función "funcion" previamente asignada a *pf1.

Ejemplo completo:

```

#include <iostream>
using namespace std;

int Muestra1();
int Muestra2();
int Muestra3();
int Muestra4();

int main() {
    int (*pf1)();
    // Puntero a función sin argumentos que devuelve un int.
    int num;

    do {
        cout << "Introduce un número entre 1 y 4, "
              << "0 para salir: ";
        cin >> num;
        if(num >= 1 && num <=4) {
            switch(num) {
                case 1:
                    pf1 = Muestra1;
                    break;
                case 2:
                    pf1 = Muestra2;
                    break;
                case 3:
                    pf1 = Muestra3;
                    break;
                case 4:
                    pf1 = Muestra4;
                    break;
            }
            pf1();
        }
    } while(num != 0);

    return 0;
}

int Muestra1() {
    cout << "Muestra 1" << endl;
    return 1;
}

int Muestra2() {
    cout << "Muestra 2" << endl;

```

```

        return 2;
    }

    int Muestra3() {
        cout << "Muestra 3" << endl;
        return 3;
    }

    int Muestra4() {
        cout << "Muestra 4" << endl;
        return 4;
    }

```

Otro ejemplo:

```

#include <iostream>
using namespace std;

int Fun1(int);
int Fun2(int);
int Fun3(int);
int Fun4(int);

int main() {
    int (*pf1[4])(int); // Array de punteros a función con
    un                  // argumento int que devuelven un
    int.
    int num;
    int valores;

    pf1[0] = Fun1;
    pf1[1] = Fun2;
    pf1[2] = Fun3;
    pf1[3] = Fun4;
    do {
        cout << "Introduce un número entre 1 y 4, "
              << "0 para salir: ";
        cin >> num;
        if(num >= 1 && num <=4) {
            cout << "Introduce un número entre 1 y 10: ";
            cin >> valores;
            if(valores > 0 && valores < 11)
                pf1[num-1](valores);
        }
    }

```

```
    } while(num != 0);

    return 0;
}

int Fun1(int v) {
    while(v--) cout << "Muestra 1" << endl;
    return 1;
}

int Fun2(int v) {
    while(v--) cout << "Muestra 2" << endl;
    return 2;
}

int Fun3(int v) {
    while(v--) cout << "Muestra 3" << endl;
    return 3;
}

int Fun4(int v) {
    while(v--) cout << "Muestra 4" << endl;
    return 4;
}
```

Palabras reservadas usadas en este capítulo

inline.

21 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 en C++ podemos definir varias funciones con el mismo nombre, con la única condición de que el número y/o el tipo de los argumentos 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.

Nota: Esta propiedad sólo existe en C++, no en C.

Ejemplo:

```
#include <iostream>
using namespace std;

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>
using namespace std;

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.

Resolución de sobrecarga

Las llamadas a funciones sobrecargadas se resuelven en la fase de compilación. Es el compilador el que decide qué versión de la función debe ser invocada, después de analizar, y en ciertos casos, tratar los argumentos pasados en la llamadas.

A este proceso se le llama *resolución de sobrecarga*.

Hay que tener presente que el compilador puede aplicar conversiones de tipo, promociones o demociones, para que la llamada se ajuste a alguno de los prototipos de la función sobrecargada. Pero esto es algo que también sucede con las funciones no sobrecargadas, de modo que no debería sorprendernos.

En nuestro último ejemplo, una llamada como:

```
...  
    cout << mayor('A', 'v', 'r') << endl;  
...
```

Funcionaría igualmente bien, invocando a la versión de la función *mayor* con tres argumentos. El compilador hará la conversión implícita de **char** a **int**. Por supuesto, el valor retornado será un **int**, no un **char**.

Sin embargo, si usamos un valor **double** o **float** para alguno de los parámetros, obtendremos un aviso.

Problema

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.

22 Operadores V: Operadores sobrecargados

Al igual que sucede con las funciones, en C++ los operadores también pueden sobrecargarse.

En realidad la mayoría de los operadores en C++ ya 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 o para sus propios tipos.

Sintaxis:

Prototipo:

```
<tipo> operator <operador> (<argumentos>);
```

Definición:

```
<tipo> operator <operador> (<argumentos>)  
{  
    <sentencias>;  
}
```

También existen algunas limitaciones para la sobrecarga de operadores:

- 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.
- El número de argumentos viene predeterminado dependiendo del operador.

Operadores binarios

Antes de nada, mencionar que el tipo del valor de retorno y el de los parámetros no está limitado. Aunque la lógica de cada operador nos imponga ciertos tipos, hay que distinguir entre las limitaciones y obligaciones del lenguaje y las de las operaciones que estemos programando.

Por ejemplo, si queremos sobrecargar el operador suma para complejos, tendremos que sumar dos números complejos y el resultado será un número complejo.

Sin embargo, C++ nos permite definir el operador suma de modo que tome un complejo y un entero y devuelva un valor en coma flotante, por ejemplo.

Las limitaciones de C++ para operadores binarios es que uno de los parámetros debe ser de tipo estructura, clase o enumerado y que debe haber uno o dos parámetros.

Esta flexibilidad nos permite definir operadores que funcionen de forma diferente dependiendo de los tipos de los operandos. Podemos, por ejemplo, para los números complejos definir un operador que de resultados diferentes si se suma un complejo con un entero o con un número en coma flotante o con otro complejo.

Ejemplo:

```
/* Definición del operador + para complejos */
complejo operator +(complejo a, complejo b) {
    complejo temp = {a.a+b.a, a.b+b.b};
    return temp;
```

```

}

/* Definición del operador + para un complejo y un float */
complejo operator +(complejo a, float b) {
    complejo temp = {a.a+b, a.b};
    return temp;
}

/* Definición del operador + para un complejos y un entero
(arbitrariamente) */
int operator +(complejo a, int b) {
    return int(a.b)+b;
}

```

Al igual que con las funciones sobrecargadas, la versión del operador que se usará se decide durante la fase de compilación, después del análisis de los argumentos.

Operadores unitarios

También es posible sobrecargar los operadores de preincremento, postincremento, predecremento y postdecremento.

De hecho, es posible sobrecargar otros operadores de forma unitaria, como el +, -, * o & de forma prefija.

En estos casos, el operador sólo afecta a un operando. Veamos primero los de prefijos:

Forma prefija

```

/* Definición del operador ++ prefijo para complejos */
complejo operator ++(complejo &c) {
    c.a++;
    return c;
}

```

Evidentemente, el operador afecta al operando, modificando su valor, por lo tanto, tendremos que pasar una referencia.

Hemos definido el operador de preincremento para *complejo* de modo que sólo se incremente la parte real, dejando la imaginaria con el mismo valor.

Forma sufija

En el caso de los operadores sufijos tenemos un problema. El operador es el mismo, por lo tanto, no hay forma de distinguir qué versión estamos sobrecargando. Lo que está claro es que la definición anterior corresponde a la versión prefija, ya que el valor del operando cambia antes de que se evalúe cualquier expresión donde aparezca este operador:

```
complejo a, b, c;  
...  
c = ++a + b;
```

El valor de *a* cambia antes de que se calcule el valor de *c*.

Para resolver este inconveniente se creó una regla arbitraria que consiste en añadir un parámetro de tipo **int** a la declaración del operador, cuando se trate de la versión sufija:

```
/* Definición del operador ++ sufijo para complejos */  
complejo operator ++(complejo &c, int) {  
    complejo temp = {c.a, c.b};  
    c.a++;  
    return temp;  
}
```

Vemos que este segundo parámetro no se usa, de hecho, ni siquiera le asignamos un identificador. Sólo sirve para que el compilador sepa que estamos definiendo (o en el caso de un prototipo, declarando) la versión sufija del operador.

La forma de definir estos operadores es siempre similar, si es que queremos mantener un funcionamiento análogo al predefinido, claro:

- Creamos un objeto temporal copia del valor inicial.
- Modificamos el valor del objeto, que como lo hemos recibido por referencia, mantendrá el valor al regresar.
- Retornamos el objeto temporal.

De este modo, el valor del objeto será modificado, pero en la expresión donde aparezca se tomará el valor antes de modificarse.

Ejemplo completo:

```
// Sobrecarga de operadores
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>
using namespace <i>std</i>;

struct complejo {
    float a,b;
};

/* Prototipo del operador + para complejos */
complejo operator +(complejo a, complejo b);
/* Prototipo del operador ++ prefijo para complejos */
complejo operator ++(complejo &a);
/* Prototipo del operador ++ sufijo para complejos */
complejo operator ++(complejo &a, int);

void Mostrar(complejo);

int main() {
    complejo x = {10,32};
    complejo y = {21,12};

    complejo z;
    /* Uso del operador sobrecargado + con complejos */
    z = x + y;
    cout << "z = (x + y) = ";
    Mostrar(z);
}
```

```

    cout << "++z = ";
    Mostrar(++z);
    cout << "z++ = ";
    Mostrar(z++);
    cout << "z = ";
    Mostrar(z);

    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;
}

/* Definición del operador ++ prefijo para complejos */
complejo operator ++(complejo &c) {
    c.a++;
    return c;
}

/* Definición del operador ++ sufijo para complejos */
complejo operator ++(complejo &c, int) {
    complejo temp = {c.a, c.b};
    c.a++;
    return temp;
}

void Mostrar(complejo c) {
    cout << "(" << c.a << "," << c.b << ")" << endl;
}

```

Ejecutar este código en [codepad](#).

La salida de este programa es la siguiente:

```

z = (x + y) = (31,44)
++z = (32,44)
z++ = (32,44)
z = (33,44)

```

Donde podemos apreciar que los operadores se comportan tal como se espera que lo hagan.

Operador de asignación

Consideremos un caso hipotético.

Tenemos una estructura donde uno de los miembros es un puntero que apuntará a una zona de memoria obtenida dinámicamente, y trabajaremos con esos objetos en nuestro programa:

```
#include <iostream>
using namespace std;

struct tipo {
    int *mem;
};

int main() {
    tipo a, b;

    a.mem = new int[10];
    for(int i = 0; i < 10; i++) a.mem[i] = 0;

    b = a; // (1)

    cout << "b: ";
    for(int i = 0; i < 10; i++) cout << b.mem[i] << ",";
    cout << endl;

    b.mem[2] = 1; // (2)

    cout << "a: ";
    for(int i = 0; i < 10; i++) cout << a.mem[i] << ",";
    cout << endl;
    cout << "b: ";
    for(int i = 0; i < 10; i++) cout << b.mem[i] << ",";
    cout << endl;

    delete[] a.mem;
    // delete[] b.mem; // (3)
    return 0;
}
```

Veamos la salida de este programa:

```
b: 0,0,0,0,0,0,0,0,0,0,0,0,  
a: 0,0,1,0,0,0,0,0,0,0,0,0,  
b: 0,0,1,0,0,0,0,0,0,0,0,0,
```

¿Notas algo extraño?

En (2) hemos modificado el valor de la posición 2 del vector *mem* del objeto *b*. Sin embargo, cuando mostramos los valores de los dos vectores, *a* y *b*, vemos que se han modificado las posiciones 2 en ambos. ¿por qué?

La respuesta está en la línea (1). Aquí hemos asignado a *b* el valor del objeto *a*. Pero, ¿cómo funciona esa asignación?

Evidentemente, se copian los valores de los campos de la estructura *a* en la estructura *b*. El problema es que *mem* es un puntero, y lo que copiamos es una dirección de memoria. Es decir, después de la asignación, *a.mem* y *b.mem* apuntan a la misma dirección de memoria, por lo tanto, las modificaciones que hagamos en uno de los objetos, se reflejan en ambos.

El mayor peligro está en sentencias como la (3), donde podríamos intentar liberar una memoria que ya ha sido liberada al hacerlo con el objeto *a*.

Pero estaremos de acuerdo en que este no es el comportamiento deseado cuando se asigna a un objeto el valor de otro. En este caso esperaríamos que las modificaciones en *a* y *b* fueran independientes.

El origen de todo está en que hemos usado el operador de asignación sin haberlo sobrecargado. El compilador no se ha quejado, porque este operador se define automáticamente para cualquier tipo declarado en el programa, pero la definición por defecto es copiar los valores de toda la memoria ocupada por el objeto, sin discriminar tipos, ni tener en cuenta si se trata de punteros o de otros valores.

Lo normal sería que pudiéramos sobrecargar el operador de asignación, y evitar estas situaciones. De hecho, deberíamos poder hacerlo siempre que vayamos a usarlo sobre objetos que contengan memoria dinámica.

Lamentablemente, no se puede sobrecargar este operador fuera de una clase (veremos que sí se puede hacer esto con clases en el {cc:035#inicio:capítulo 35}).

Pero entonces, ¿qué hacemos con este problema? La solución es sustituir el operador de asignación por una función:

```
// Asignación de arrays
// (C) 2009 Con Clase
// Salvador Pozo
#include <iostream>
using namespace std;

struct tipo {
    int *mem;
};

void asignar(tipo&, tipo&);

int main() {
    tipo a, b;

    a.mem = new int[10];
    b.mem = 0;

    for(int i = 0; i < 10; i++) a.mem[i] = 0;

    asignar(b, a);

    cout << "b: ";
    for(int i = 0; i < 10; i++) cout << b.mem[i] << ",";
    cout << endl;

    b.mem[2] = 1;

    cout << "a: ";
    for(int i = 0; i < 10; i++) cout << a.mem[i] << ",";
    cout << endl;
    cout << "b: ";
```

```

        for(int i = 0; i < 10; i++) cout << b.mem[i] << ",";
        cout << endl;

        delete[] a.mem;
        delete[] b.mem;
        return 0;
    }

    void asignar(tipo &a, tipo &b) {
        if(&a != &b) {
            if(a.mem) delete[] a.mem;
            a.mem = new int[10];
            for(int i = 0; i < 10; i++) a.mem[i] = b.mem[i];
        }
    }
}

```

Ejecutar este código en [codepad](#).

Hay dos precauciones básicas que debemos tener:

- Verificar si los objetos origen y destino son el mismo. En ese caso, no hay nada que hacer.
- Liberar la memoria dinámica que pudiera tener el objeto de destino antes de asignarle una nueva.

Notación funcional de los operadores

Los operadores sobrecargados son formas alternativas de invocar a ciertas funciones, de modo que sean más fácilmente interpretables.

Tanto es así que para cada operador es posible usarlos en su forma de función, esta forma de usar los operadores se conoce como *notación funcional*:

```
z = operator+(x,y);
```

Pero donde veremos mejor toda la potencia de los operadores sobrecargados será cuando estudiemos las clases. En el

{cc:035#inicio:capítulo 35} veremos este tema con mayor detalle.

Palabras reservadas usadas en este capítulo

operator.

Problemas

1. Dada la siguiente estructura para almacenar ángulos en grados, minutos y segundos:

```
struct stAngulo {  
    int grados;  
    int minutos;  
    int segundos;  
};
```

Sobrecargar los operadores de suma y resta para sumar y restar ángulos, y los operadores de incremento y decremento, tanto en sus formas sufijas como prefijas.

Hay que tener en cuenta que tanto los valores para minutos como para los segundos están limitados entre 0 y 59. Para este ejercicio, en el caso de los grados podemos limitar esos valores entre 0 y 359, aunque en general se entienden los grados negativos y los valores fuera de ese rango para indicar sentidos en los giros y para indicar múltiples vueltas.

2. Sobrecargar el operador de resta para calcular la diferencia en días entre dos fechas.

La estructura para las fechas será:

```
struct fecha {  
    int dia;  
    int mes;
```

```
int anno;  
};
```

Ejemplos capítulo 22

Ejemplo 22.1

Vamos a ver con qué tipo de objetos podemos hacer operaciones.

Lo principal, a la hora de operarar con cualquier tipo de objeto, es definir claramente cada operación.

Esta definición incluye el número y tipo de cada operando, el tipo del resultado y el modo en que se combinan los operandos para obtener el resultado.

En este ejemplo sumaremos dos *arrays* de enteros. La primera condición afecta a los parámetros y valor de retorno, e implica que sumaremos dos *arrays* y el resultado será un *array*.

Podemos restringir la operación, de modo que sólo sea posible sumar *arrays* del mismo tamaño, es decir, con el mismo número de elementos, de forma que se sumen los elementos en la misma posición, y el resultado sea un *array* del mismo tamaño.

También podemos hacer que se puedan sumar *arrays* de diferente longitud, de modo que se sumen los elementos que existan en los dos *arrays*. Para el resto de los elementos tenemos dos opciones, una es sumarlos con cero, de modo que el *array* de salida tenga tantos elementos como el mayor de los de entrada. La otra opción es ignorar los sobrantes, de modo que el *array* de salida tenga tantos elementos como el menor de los de entrada.

Otra posibilidad es que la suma de dos *array* sea un tercer *array* con los elementos de ambos. Es decir, si sumamos dos *arrays* de 4 y 8 elementos, el resultado será un *array* con 12 elementos, que contendrá los elementos de los dos iniciales.

Para este ejemplo tomaremos la primera opción. Cuando los *arrays* sean de diferente tamaño, retornaremos un *array* nulo, es

decir, vacío.

```
// Suma de arrays
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>
using namespace std;

struct array {
    int *v; // Elementos
    int n;  // Número de elementos
};

array operator +(array, array);
void Mostrar(array);

int main() {
    array v1, v2, v3;

    v1.n = v2.n = 10;
    v1.v = new int[v1.n];
    v2.v = new int[v2.n];

    for(int i = 0; i < 10; i++) {
        v1.v[i] = i;
        v2.v[i] = 2*i-4;
    }
    v3 = v1 + v2;

    cout << "v1: ";
    Mostrar(v1);
    cout << "v2: ";
    Mostrar(v2);
    cout << "v3 = v1+v2: ";
    Mostrar(v3);

    delete[] v1.v;
    delete[] v2.v;
    delete[] v3.v;
    return 0;
}

array operator +(array a, array b) {
    array temp;
```

```

        if(a.n == b.n) {
            temp.n = a.n;
            temp.v = new int[temp.n];
            for(int i = 0; i < temp.n; i++) temp.v[i] =
a.v[i]+b.v[i];
        } else {
            temp.v = 0;
            temp.n = 0;
        }
        return temp;
    }

void Mostrar(array v) {
    for(int i = 0; i < v.n; i++) {
        cout << v.v[i] << ((i < v.n-1) ? "," : "");
    }
    cout << endl;
}

```

Ejecutar este código en [codepad](#).

Hay que tener cuidado con esta forma de trabajar. Hay un peligro en el uso del operador de asignación con estructuras que contengan punteros y memoria dinámica, que es el que comentamos antes. El compilador crea un operador por defecto, pero no se preocupa de las posibles fugas de memoria.

Si en este ejemplo usamos una expresión de suma, asignando el resultado a un objeto que previamente tenía un valor válido, el valor previo se pierde, y será imposible acceder a él ya sea para trabajar con él o para liberar la memoria utilizada.

Otro peligro es usar estas expresiones sin asignar el valor a ningún objeto, por ejemplo:

```

cout << "v1+v2: ";
Mostrar(v1+v2);

```

En este caso, tampoco podremos liberar la memoria asignada al objeto temporal.

La sobrecarga de operadores fuera de las clases es imperfecta. Tendremos que ser muy cuidadosos si la usamos.

Veremos un mejor uso más adelante.

Ejemplo 22.2

Sobrecargar operadores para objetos con memoria dinámica no es práctico, veremos que es mejor usar clases para eso.

Ahora vamos a sumar un número de días a una fecha. En este caso, el primer argumento es una fecha y el segundo un entero, que indica el número de días.

Se trata de un ejemplo, en un caso real creo que sería más intuitivo usar una función para este propósito, ya que una de las reglas de la suma es que no se deben sumar objetos de distinto tipo, ya sabes, no se pueden sumar peras y manzanas...

La aritmética de fechas es parecida a la aritmética de punteros, se pueden restar fechas, con lo que se obtiene un entero, y se pueden sumar enteros a fechas, con lo que se obtiene una fecha.

```
// Suma de fechas
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>
using namespace std;

struct fecha {
    int dia;
    int mes;
    int anno;
};

fecha operator +(fecha, int);
bool bisiestro(int);

int main() {
    fecha f1 = { 12, 11, 2009 };
    fecha f2;
```

```

        f2 = f1 + 1485;

        cout << "fecha 1: " << f1.dia << "/" << f1.mes << "/" <<
f1.anno << endl;
        cout << "fecha 2: " << f2.dia << "/" << f2.mes << "/" <<
f2.anno << endl;
        return 0;
    }

    fecha operator +(fecha f1, int d) {
        int dm[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31 };
        fecha temp = f1;

        temp.dia += d;
        if(bisiesto(temp.anno)) dm[1] = 29; else dm[1] = 28;
        while(temp.dia > dm[temp.mes-1]) {
            temp.dia -= dm[temp.mes-1];
            temp.mes++;
            if(temp.mes > 12) {
                temp.mes = 1;
                temp.anno++;
                if(bisiesto(temp.anno)) dm[1] = 29; else dm[1] =
28;
            }
        }

        return temp;
    }

    bool bisiesto(int a) {
        return !(a%4) && ((a%100) || !(a%400));
    }

```

Ejecutar este código en [codepad](#).

23 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. El preprocesador también elimina 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. Las macros suministran 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 como veremos, hay algunas excepciones. Cada sustitución se conoce como una expansión de la macro. La secuencia es llamada a menudo cuerpo de la macro.

Si no se especifica una *secuencia*, el *identificador_de_macro* sencillamente, 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 queda 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 este 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 procesará la sección-2, y después el control pasa al correspondiente **#endif**. Por el contrario, si el resultado 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, aunque se trate de una sección 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 <identificador>  
#ifndef <identificador>
```

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 comprueba 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 n°_línea : Error directive:  
mensaje_de_error
```

Este ejemplo está extraído de uno de los ficheros de cabecera del compilador GCC:

```
#ifndef BFD_HOST_64_BIT  
  #error No 64 bit integer type available  
#endif /* ! defined (BFD_HOST_64_BIT) */
```

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 interferirán 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.

Teniendo lo anterior en cuenta, veamos una de las directivas *pragma* más extendidas en compiladores, pero teniendo en cuenta que no tienen por qué estar disponibles en el compilador que uses.

Tampoco es bueno usar estas directivas, ya que suelen hacer que el código no sea portable, es mejor buscar alternativas.

Directiva **#pragma pack()**

Esta directiva se usa para cambiar la alineación de bytes cuando se declaran objetos o estructuras.

Recordemos lo que nos pasaba al aplicar el {cc:011b#STR_sizeof:operador **sizeof** a estructuras} con el mismo número y tipo de campos, aunque en distinto orden.

Algunas veces puede ser conveniente alterar el comportamiento predefinido del compilador en lo que respecta al modo de empaquetar los datos en memoria. Por ejemplo, si tenemos que leer un objeto de una estructura y con un alineamiento determinados, deberemos asegurarnos de que nuestro programa almacena esos objetos con la misma estructura y alineamiento.

La directiva `#pragma pack()` nos permite alterar ese alineamiento a voluntad, indicando como parámetro el valor deseado. Por ejemplo:

```
#include <iostream>
using namespace std;

#pragma pack(1)

struct A {
    int x;
    char a;
    int y;
    char b;
};

#pragma pack()

struct B {
    int x;
    int y;
    char a;
    char b;
};

int main() {
    cout << "Tamaño de int: "
          << sizeof(int) << endl;
    cout << "Tamaño de char: "
```

```
        << sizeof(char) << endl;
    cout << "Tamaño de estructura A: "
        << sizeof(A) << endl;
    cout << "Tamaño de estructura B: "
        << sizeof(B) << endl;

    return 0;
}
```

La salida, en este caso es:

```
Tamaño de int: 4
Tamaño de char: 1
Tamaño de estructura A: 10
Tamaño de estructura B: 12
```

Vemos que ahora la estructura A ocupa exactamente lo mismo que la suma de los tamaños de sus componentes, es decir, 10 bytes.

Esta directiva funciona como un conmutador, y permanece activa hasta que se cambie el alineamiento con otra directiva, o se desactive usando la forma sin parámetros.

Pero existe una alternativa, aparentemente más engorrosa, pero más portable.

Atributos

Se pueden especificar ciertos atributos para todas las variables, ya sea en la declaración de tipos o en la declaración de los objetos.

Para ello se usa la palabra `__attribute__`, que admite varios parámetros:

- `__aligned__` que permite especificar el número del que tiene que ser múltiplo la dirección de memoria.
- `__packed__` que permite especificar el formato empaquetado, es decir, el alineamiento será de un byte.

Por ejemplo:

```
struct __attribute__((__packed__)) A {  
    int x;  
    char a;  
    int y;  
    char b;  
};
```

Esta forma tiene el mismo efecto que el ejemplo anterior, pero tiene algunas ventajas.

Para empezar, se aplica sólo a la estructura, unión o clase especificada. Además, nos aseguramos de que o bien el compilador tiene en cuenta este atributo, o nos da un mensaje de error. Con la directiva `#pragma`, si el compilador no la reconoce, la ignora sin indicar un mensaje de error.

El otro atributo, `__aligned__` requiere un número entero como parámetro, que es el número del que las direcciones de memoria han de ser múltiplos:

```
struct __attribute__((__aligned__(4))) A {  
    int x;  
    char a;  
    int y;  
    char b;  
};
```

Directiva `#warning`

Sintaxis:

```
#warning mensaje_de_aviso
```

Sirve para enviar mensajes de aviso cuando se compile un fichero fuente C o C++. Esto nos permite detectar situaciones potencialmente peligrosas y avisar al programador de posibles errores.

Este ejemplo está extraído de uno de los ficheros de cabecera del compilador GCC:

```
#ifdef __DEPRECATED
#warning This file includes at least one deprecated or
antiquated header. \
Please consider using one of the 32 headers found in section
17.4.1.2 of the \
C++ standard. Examples include substituting the <X> header
for the <X.h> \
header for C++ includes, or <iostream> instead of the
deprecated header \
<iostream.h>. To disable this warning use -Wno-deprecated.
#endif
```

Aprovecho para comentar que una línea en el editor se puede dividir en varias añadiendo el carácter '\' al final de cada una. En el ejemplo anterior, todo el texto entre la segunda y la sexta línea se considera una única línea por el compilador.

24 Funciones V: Recursividad

Se dice que una función es recursiva cuando se define en función de si misma.

No todas las funciones pueden llamarse a si mismas, sino que deben estar diseñadas especialmente para que sean recursivas, de otro modo podrían conducir a bucles infinitos, o a que el programa termine inadecuadamente.

Tampoco todos los lenguajes de programación permiten usar recursividad.

C++ permite la recursividad. Cada vez que se llama a una función, se crea un juego de variables locales, de este modo, si la función hace una llamada a si misma, se guardan sus variables y parámetros, usando 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:

Prodríamos crear una 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:

$$n! = n * (n-1) * (n-2) * \dots * 1$$

Hay algunas limitaciones:

- No es posible calcular el factorial de números negativos, no está definido.
- El factorial de cero es 1.

De modo que una función bien hecha para cálculo de factoriales debería incluir un control para esos casos:

```
/* Función recursiva para cálculo de factoriales */  
int factorial(int n) {  
    if(n < 0) return 0;  
    else if(n > 1) return n*factorial(n-1); /* Recursividad */  
    return 1; /* Condición de terminación, n == 1 */  
}
```

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 $2 \cdot 3 = 6$

1ª instancia
(recupera n=4 de la pila) retorna $6 \cdot 4 = 24$
Valor de retorno $\rightarrow 24$

Aunque la función *factorial* es un buen ejemplo para demostrar cómo funciona una función recursiva, la recursividad no es un buen modo de resolver esta función, que sería más sencilla y rápida con un simple bucle **for**.

La recursividad consume muchos recursos de memoria y tiempo de ejecución, y se debe aplicar a funciones que realmente le saquen partido.

Veamos otro ejemplo: visualizar las permutaciones de n elementos.

Las permutaciones de un conjunto son 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>
using namespace std;

/* Prototipo de función */
void Permutaciones(char *, int l=0);

int main(int argc, char *argv[]) {
    char palabra[] = "ABCDE";

    Permutaciones(palabra);

    cin.get();
    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-1; i++) {
    if(n-1 > 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 permutación, 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.

Otras formas de recursividad

Existen otras formas de implementar algoritmos recursivos, no es necesario que una función se invoque a si misma.

Por ejemplo, un par de funciones A y B pueden crear un algoritmo recursivo si la función A invoca a la función B, y esta a su vez invoca a la función A.

Este mismo mecanismo se puede implementar con tres, cuatro o con cualquier número de funciones.

Veamos un ejemplo. Partamos de la siguiente serie:

$$1 - 1/2 + 1/3 - 1/4 + 1/5 - \dots - 1/2*n + 1/2*n+1 - \dots$$

Podemos diseñar un procedimiento recursivo para calcular la suma de los n primeros elementos de la serie, de modo que usemos una función diferente para los elementos pares e impares.

```
// Suma de la serie 1-1/2+1/3-1/4+1/5...
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>

using namespace std;

double par(int);
double impar(int);
double suma(int);

int main() {

    cout << suma(3) << endl;
    cout << suma(13) << endl;
    cout << suma(23) << endl;
    cout << suma(87) << endl;
    cout << suma(250) << endl;
    cout << suma(450) << endl;
    return 0;
}

double suma(int n) {
    if(n % 2) return impar(n);
    else return par(n);
}

double par(int n) {
    return impar(n-1)-1/double(n);
}

double impar(int n) {
    if(n == 1) return 1;
    return par(n-1)+1/double(n);
}
```

Veremos más aplicaciones de recursividad en el tema de [estructuras dinámicas de datos](#).

Problemas

1. La sucesión de Fibonacci se define como una serie infinita de números naturales.

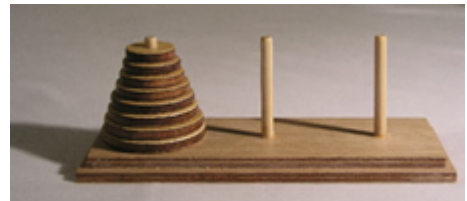
El primer término, para $n = 0$, es 0 y el segundo, para $n = 1$ es 1. Los sucesivos se calculan como la suma de los dos términos anteriores. Por ejemplo, el término 5 es la suma de los términos 3 y 4.

Los primeros términos son: 0, 1, 1, 2, 3, 5, 8...

Hacer un programa que calcule el término n de la sucesión de Fibonacci de forma recursiva.

2. Volvamos al problema de los palíndromos. Pero ahora usaremos una función recursiva para determinar si una cadena determinada es o no palíndroma.
3. Veamos ahora un problema clásico: las torres de Hanói.

El juego consiste en tres varillas verticales. En una de ellas están apiladas un número de discos, generalmente ocho, de diámetros diferentes, ordenados de mayor a menor (el de mayor diámetro abajo).



Torres de Hanói

Las otras dos varillas están vacías. El juego consiste en pasar todos los discos de la varilla ocupada a una de las varillas libres.

Para llegar a ese objetivo hay que respetar tres reglas:

1. Sólo se puede mover un disco cada vez.
2. Un disco de mayor tamaño no se puede colocar encima de uno más pequeño.
3. Sólo se puede mover el disco que se encuentre en la parte superior de cada varilla.

Resolver el juego usando algoritmos recursivos.

Ejemplos capítulo 24

Ejemplo 24.1

En el [capítulo 11](#) sobre los estructuras vimos un programa de ejemplo para implementar el método de "Búsqueda binaria" o "Busca dicotómica". También mencionamos que volveríamos a ver ese problema usando recursividad.

Bien, ese momento ha llegado.

Recordemos la idea: se elige el elemento central del rango en el que debemos buscar. Pueden pasar tres cosas:

- Que el elemento elegido sea el buscado, con lo que ha terminado la búsqueda.
- Que el elemento elegido sea menor que el buscado, en ese caso, tomaremos el elemento siguiente al elegido como límite inferior de un nuevo rango, y repetiremos el proceso.
- Que el elemento elegido sea mayor. Ahora tomaremos el elemento anterior al elegido como nuevo límite superior de un nuevo rango, y repetiremos el proceso.

El proceso termina cuando encontramos el elemento, o cuando el rango de búsqueda resulte nulo, y la búsqueda habrá fracasado.

```
// Búsqueda binaria
// Noviembre de 2009 Con Clase, Salvador Pozo
#include <iostream>
using namespace std;

int Binaria(int*, int, int, int);

int tabla[] = {
    1,   3,  12,  33,  42,  43,  44,  45,  54,  55,
    61,  63,  72,  73,  82,  83,  84,  85,  94,  95,
    101, 103, 112, 133, 142, 143, 144, 145, 154, 155,
    161, 163, 172, 173, 182, 183, 184, 185, 194, 195
};
```

```

int main() {
    int pos;
    int valor=141;

    pos = Binaria(tabla, valor, 0,
sizeof(tabla)/sizeof(tabla[0])-1);
    if(pos > 0) cout << "Valor " << valor << " encontrado en
posicion: " << pos << endl;
    else cout << "Valor " << valor << " no encontrado" <<
endl;
    return 0;
}

/* Función de búsqueda binaria:
Busca el "valor" dentro del vector "vec" entre los
márgenes
inferior "i" y superior "s" */
int Binaria(int* vec, int valor, int i, int s) {
    int inferior = i;
    int superior = s;
    int central;

    central = inferior+(superior-inferior)/2;
    if(vec[central] == valor) return central;
    if(superior <= inferior) return -1;
    else if(vec[central] < valor) return Binaria(vec, valor,
central+1, superior);
    return Binaria(vec, valor, inferior, central-1);
}

```

Ejecutar este código en paiza.io.

Ejemplo 24.2

Veamos otro problema común que se suele resolver aplicando recursividad.

El [algoritmo de Euclides](#) sirve para calcular el máximo común divisor de dos números.

El mcd se define así:

Dados dos enteros a y b distintos de 0, decimos que el entero d mayor o igual a 1 es un mcd de a y b si d divide a a y a b , y para

cualquier otro entero c tal que c divida a a y a b , entonces c también divide a d .

El algoritmo de Euclides parte de la proposición de, siendo a y b dos enteros distintos de cero, el mcd de a y b es el mismo que el de b y r , donde r es un entero mayor o igual que cero y menor que b , se calcula como el resto de la división entera entre a y b .

La ventaja es que r es un entero de menor que a . El algoritmo aprovecha esto para calcular el mcd de forma recursiva.

```
mcd(a,b)
- si  $a < b$ , retornar  $\text{mcd}(b,a)$ 
- si  $b == 0$ , retornar  $a$ 
- retornar  $\text{mcd}(b, a \% b)$ 
```

La implementación es realmente sencilla:

```
// Algoritmo de Euclides
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>
using namespace std;

int mcd(int, int);

int main() {
    int a, b;

    a = 364332;
    b = 30252;

    cout << "mcd(" << a << ", " << b << ") = " << mcd(a,b) <<
endl;
    return 0;
}

int mcd(int a, int b) {
    if(a < b) return mcd(b,a);
    if(b == 0) return a;
    return mcd(b, a % b);
}
```


Ejecutar este código en [codepad](#).

Ejemplo 24.3

Con este algoritmo podemos recodificar el [ejemplo 11.4](#), para simplificar fracciones:

```
// Simplificación de fracciones
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>
using namespace std;

struct fraccion {
    int num;
    int den;
};

fraccion Simplificar(fraccion);

int mcd(int, int);

int main() {
    fraccion f, s;
    f.num = 1204;
    f.den = 23212;

    s = Simplificar(f);
    cout << "Simplificar(" << f.num << "/" << f.den << ") = ";
    cout << s.num << "/" << s.den << endl;

    return 0;
}

fraccion Simplificar(fraccion f) {
    int d= mcd(f.num,f.den);
    f.num /= d;
    f.den /= d;
    return f;
}
```

```
int mcd(int a, int b) {
    if(a < b) return mcd(b,a);
    if(b == 0) return a;
    return mcd(b, a % b);
}
```

Ejemplo 24.4

Veamos como hacer un cambio de base de decimal a cualquier otra base, usando un algoritmo recursivo.

```
// Decimal a binario
// (C) 2009 Con Clase
// Salvador Pozo
#include <iostream>

using namespace std;

void DecimalBinario(int, int);

int main() {

    DecimalBinario(65536,2);
    cout << endl;
    DecimalBinario(65536,3);
    cout << endl;
    DecimalBinario(65536,8);
    cout << endl;
    DecimalBinario(65536,10);
    cout << endl;
    DecimalBinario(65536,16);
    cout << endl;
    return 0;
}

void DecimalBinario(int n, int base) {
    if(n >= base) DecimalBinario(n/base, base);
    cout << char((n%base < 10) ? '0'+n%base : 'A'+n%base-
10);
}
```

Ejecutar este código en [codepad](#).

Ejemplo 24.5

Un último ejemplo, vamos a crear un programa para sumar los dígitos de un número entero, de forma recursiva, claro.

```
// Sumar los dígitos de un número
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>

using namespace std;

int SumaDigitos(int);

int main() {
    cout << 32890123 << ": " << SumaDigitos(32890123) <<
endl;
    return 0;
}

int SumaDigitos(int n) {
    if(n < 10) return n;
    return n%10+SumaDigitos(n/10);
}
```

Ejecutar este código en [codepad](#).

25 Tipos de Variables VII: 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.

En el [capítulo 6](#) ya comentamos algo sobre el ámbito de las variables. Los tipos de almacenamiento son los que definen el ámbito de las variables u objetos. Los modificadores, por su parte, permiten alterar el comportamiento de esas variables.

Se distinguen dos aspectos en cuanto al ámbito: el temporal, o *duración* y el de acceso o *alcance* o *visibilidad*.

El ámbito de acceso es la parte de código desde el que un objeto es accesible.

El ámbito temporal se refiere al periodo de ejecución en que existe un objeto.

Ambos quedan definidos por el tipo de almacenamiento, y este tipo, a su vez, queda definido por los modificadores.

C++ dispone de los siguientes especificadores de tipo de almacenamiento: **auto**, **extern**, **static** y **register**.

Hay además algunos modificadores: **const** y **volatile**.

Por último, hay otra palabra reservada: **mutable**, que se suele considerar un especificador, aunque en mi opinión, es más bien un modificador.

Almacenamiento automático

Para especificar el tipo de almacenamiento automático se usa el especificador **auto**.

Sintaxis:

```
[auto] <tipo> <nombre_variable>;
```

Sirve para declarar variables automáticas o locales. Es el modificador por defecto cuando se declaran variables u objetos locales, es decir, si no se especifica ningún modificador, se creará una variable automática.

Estas variables se crean durante la ejecución, y se elige el tipo de memoria a utilizar en función del ámbito temporal de la variable. Una vez cumplido el ámbito, la variable es destruida. Es decir, una variable automática local de una función se creará cuando sea declarada, y se destruirá al terminar la función. Una variable local automática de un bucle será destruida cuando el bucle termine.

Debido a que estos objetos serán creados y destruidos cada vez que sea necesario, usándose, en general, diferentes posiciones de memoria, su valor se perderá cada vez que sean creadas, perdiéndose el valor previo en cada caso.

Por supuesto, no es posible crear variables automáticas globales, ya que son conceptos contradictorios.

Almacenamiento estático

Para especificar este tipo de almacenamiento se usa el especificador **static**.

Sintaxis:

```
static <tipo> <nombre_variable>;  
static <tipo> <nombre_de_función>(<lista_parámetros>);
```

Cuando se usa en la declaración de objetos, este especificador hace que se asigne una dirección de memoria fija para el objeto

mientras el programa se esté ejecutando. Es decir, su ámbito temporal es total. En cuanto al ámbito de acceso conserva el que le corresponde según el punto del código en que aparezca la declaración.

Debido a que el objeto tiene una posición de memoria fija, su valor permanece, aunque se trate de un objeto declarado de forma local, entre distintas reentradas en el ámbito del objeto. Por ejemplo si se trata de un objeto local a una función, el valor del objeto se mantiene entre distintas llamadas a la función.

Hay que tener en cuenta que los objetos estáticos no inicializados toman un valor nulo.

Por el contrario, si se le da un valor inicial a una variable estática, la asignación sólo afecta a la primera vez que es declarada.

```
#include <iostream>

using namespace std;

int funcion();

int main() {
    for(int i = 0; i < 10; i++)
        cout << "Llamada " << i+1 << ": " << funcion() <<
endl;
    return 0;
}

int funcion() {
    static int x=10;

    x++;
    return x;
}
```

La salida de este programa será:

```
Llamada 1: 11
Llamada 2: 12
Llamada 3: 13
```

```
Llamada 4: 14  
Llamada 5: 15  
Llamada 6: 16  
Llamada 7: 17  
Llamada 8: 18  
Llamada 9: 19  
Llamada 10: 20
```

Nota:

En realidad, el compilador analiza el código fuente y crea el código ejecutable necesario para crear todas las variables estáticas antes de empezar la ejecución el programa, asignando sus valores iniciales, si se indican o cero en caso contrario. De modo que cuando se llama a la *funcion* no se crea el objeto x, sino que se usa directamente.

En este caso, desde el punto de vista del ámbito temporal, x se comporta como un objeto global, pero para el ámbito de acceso se trata de un objeto local de *funcion*.

Este tipo de almacenamiento se usa con el fin 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.

Parecería lógico que, análogamente a lo que sucede con el especificador **auto**, no tenga sentido declarar objetos globales como estáticos, ya que lo son por defecto. Sin embargo, el especificador **static** tiene un significado distinto cuando se aplica a objetos globales. En ese caso indica que el objeto no es accesible desde otros ficheros fuente del programa.

En el caso de las funciones, el significado es el mismo, las funciones estáticas sólo son accesibles desde el fichero en que están declaradas.

Nota:

Veremos en el capítulo siguiente que esto se puede conseguir en C++ de una forma más clara usando espacios con nombre.

Almacenamiento externo

Para especificar este tipo de almacenamiento se usa la palabra **extern**.

Sintaxis:

```
extern <tipo> <nombre_variable>;  
[extern] <tipo> <nombre_de_función>(<lista_parámetros>);
```

De nuevo tenemos un especificador que se puede aplicar a funciones y a objetos. O más precisamente, a prototipos de funciones y a declaraciones de objetos.

Este especificador se usa para indicar que el almacenamiento y valor de una variable o la definición de una función están definidos en otro módulo o fichero fuente. Las funciones declaradas con **extern** son visibles por todos los ficheros fuente del programa, salvo que (como vimos más arriba) se defina la función como **static**.

El especificador **extern** sólo puede usarse con objetos y funciones globales.

En el caso de las funciones prototipo, el especificador **extern** es opcional. Las declaraciones de prototipos son externas por defecto.

Este especificador lo usaremos con programas que usen varios ficheros fuente, que será lo más normal con aplicaciones que no sean ejemplos o aplicaciones simples.

Veamos un ejemplo de un programa con dos ficheros fuente:

```
// principal.cpp
```



```

#include <iostream>

using namespace std;

int x=100; // Declaración e inicialización

int funcion(); // extern por defecto

int main() {
    cout << funcion() << endl;
    return 0;
}

```

Fichero fuente de "modulo.cpp":

```

// modulo.cpp

extern int x; // x es externa

int funcion() { // definición de la función
    return x;
}

```

En este ejemplo vemos que no es necesario declarar el prototipo como **extern**. En el fichero "modulo.cpp" declaramos x como externa, ya que está declarada y definida en el fichero "principal.cpp", y porque necesitamos usar su valor.

Se puede usar **extern "c"** con el fin de prevenir que algún nombre de función escrita en C pueda ser ocultado por funciones de programas C++. Este especificador no se refiere al tipo de almacenamiento, ya que sabemos que en el caso de prototipos de funciones es el especificador por defecto. En realidad es una directiva que está destinada al enlazador, y le instruye para que haga un enlazado "C", distinto del que se usa para funciones en C++.

Almacenamiento en registro

Para especificar este tipo de almacenamiento se usa el especificador **register**.

Sintaxis:

```
register <tipo> <nombre_variable>;
```

Indica al compilador una *preferencia* para que el objeto se almacene en un registro de la CPU, si es posible, con el fin de optimizar su acceso, consiguiendo una mayor velocidad de ejecución.

Los datos declarados con el especificador **register** tienen el mismo ámbito que las automáticas. De hecho, sólo se puede usar este especificador con parámetros y con objetos locales.

El compilador puede ignorar la petición de almacenamiento en registro, que se acepte o no estará basado en el análisis que realice el compilador sobre cómo se usa la variable.

Un objeto de este tipo no reside en memoria, y por lo tanto no tiene una dirección de memoria, es decir, no es posible obtener una referencia a un objeto declarado con el tipo de almacenamiento en registro.

Se puede usar un registro para almacenar objetos de tipo **char**, **int**, **float**, punteros. En general, objetos que quepan en un registro.

```
#include <iostream>

using namespace std;

void funcion(register int *x);

int main() {
    int s[10] = {1, 2, 1, 5, 2, 7, 3, 1, 3, 0};
    funcion(s);
    return 0;
}

void funcion(register int *x) {
    register char a = 'a';

    for(register int i=0; i < 10; i++) {
        cout << *x++ << " " << a++ << endl;
    }
}
```

```
}  
}
```

Modificador de almacenamiento constante

El modificador **const** crea nuevos tipos de objetos, e indica que el valor de tales objetos no puede ser modificado por el programa. Los tipos son nuevos en el sentido de que **const int** es un tipo diferente de **int**. Veremos que en algunos casos no son intercambiables.

Sintaxis:

```
const <tipo> <variable> = <inicialización>;  
const <tipo> <variable_agregada> = {<lista_inicialización>};  
<tipo> <nombre_de_función> (const <tipo>*<nombre-de-  
variable> );  
const <tipo> <nombre_de_función>(<lista_parámetros>);  
<tipo> <nombre_de_función_miembro>(<lista_parámetros>)  
const;
```

Lo primero que llama la atención es que este modificador se puede aparecer en muchas partes diferentes de un programa C++. En cada una de las sintaxis expuestas el significado tiene matices diferentes, que explicaremos a continuación.

De las dos primeras se deduce que es necesario inicializar siempre, los objetos declarados como constantes. Puesto que el valor de tales objetos no puede ser modificado por el programa posteriormente, será imprescindible asignar un valor en la declaración. C++ no permite dejar una constante indefinida.

Cuando se trata de un objeto de un tipo agregado: *array*, estructura o unión, se usa la segunda forma.

En C++ es preferible usar este tipo de constantes en lugar de constantes simbólicas (macros definidas con **#define**). El motivo es que estas constantes tienen un tipo declarado, y el compilador

puede encontrar errores por el uso inapropiado de constantes que no podría detectar si se usan constantes simbólicas.

Hay una excepción, que en realidad no es tal, y es que la declaración tenga además el especificador **extern**. En ese caso, como vimos antes, no estamos haciendo una definición, no se crea espacio para el objeto, y por lo tanto, no necesitamos asignar un valor. Una declaración **extern** indica al compilador que la definición del objeto está en otra parte del programa, así como su inicialización.

Cuando se usa con parámetros de funciones, como en el caso tercero, impide que el valor de los parámetros sea modificado por la función.

Sabemos que los parámetros son pasados por valor, y por lo tanto, aunque la función modifique sus valores, estos cambios no afectan al resto del programa fuera de la función, salvo que se trate de punteros, *arrays* o referencias. Es precisamente en estos tres casos cuando el modificador tiene aplicación, impidiendo que el código de la función pueda modificar el valor de los objetos referenciados por los parámetros.

Los intentos de hacer estas modificaciones se detectan en la fase de compilación, de modo que en realidad, a quien se impide que haga estas modificaciones es al programador. En ese sentido, la declaración de un parámetro constante nos compromete como programadores a no intentar modificar el valor de los objetos referenciados.

```
#include <iostream>

using namespace std;

void funcion(const int *x);

int main() {
    int s = 100;
    funcion(&s);
    return 0;
}
```

```
}  
  
void funcion(const int *x) {  
    (*x)++; // ERROR: intento de modificar un valor  
    constante  
}
```

El compilador dará un error al intentar compilar este ejemplo.

Nota:

El compilador puede referirse a estos objetos como de "sólo lectura", o "read-only". Viene a significar lo mismo: ya que podemos leer el valor de una constante, pero no escribirlo (o modificarlo).

Con las referencias pasa algo similar:

```
void funcion(const int &x);
```

En este caso no podremos modificar el valor del objeto referenciado por x.

Esto tiene varias utilidades prácticas:

- Imaginemos que sólo disponemos del prototipo de la función. Por ejemplo, la función `strlen` tiene el siguiente prototipo: `int strlen(const char*)`. Esto nos dice que podemos estar seguros de que la función no alterará el valor de la cadena que pasamos como parámetro, y por lo tanto no tendremos que tomar ninguna medida extraordinaria para preservar ese valor. Hay que tener en cuenta que no podemos pasar *arrays* por valor.
- Otro caso es cuando queremos pasar como parámetros objetos que no queremos que sean modificados por la función. En ese caso tenemos la opción de pasarlos por valor, y de este modo protegerlos. Pero si se trata de objetos de gran tamaño, resulta

muy costoso (en términos de memoria y tiempo de proceso) hacer copias de tales objetos, y es preferible usar una referencia. Si queremos dejar patente que la función no modificará el valor del objeto declararemos el parámetro como una referencia constante.

Cuando se aplica al valor de retorno de una variable, como en el cuarto caso, el significado es análogo. Evidentemente, cuando el valor de retorno no es una referencia, no tiene sentido declararlo como constante, ya que lo será siempre. Pero cuando se trate de referencias, este modificador impide que la variable referenciada sea modificada.

```
#include <iostream>
using namespace std;

int y;

const int &funcion();

int main() {
    // funcion()++; // Ilegal (1)
    cout << ", " << y << endl;
    return 0;
}

const int &funcion() {
    return y;
}
```

Como vemos en (1) no nos es posible modificar el valor de la referencia devuelta por "funcion".

El último caso, cuando el modificador se añade al final de un prototipo de función de una clase o estructura, indica que tal función no modifica el valor de ningún dato miembro del objeto. Cuando veamos con detalle clases, veremos que tiene gran utilidad.

Punteros constantes y punteros a constantes

Este matiz es importante, no es lo mismo un puntero constante que un puntero a una constante.

Declaración de un puntero constante:

```
<tipo> *const <identificador>=<valor inicial>
```

Declaración de un puntero a una constante:

```
const <tipo> *<identificador>[=<valor inicial>]
```

En el primero caso estaremos declarando un objeto constante de tipo puntero. Por lo tanto, deberemos proporcionar un valor inicial, y este puntero no podrá apuntar a otros objetos durante toda la vida del programa.

En el segundo caso estaremos declarando un puntero a un objeto constante. El puntero podrá apuntar a otros objetos, pero ningún objeto apuntado mediante este puntero podrá ser modificado.

A un puntero a constante se le pueden asignar tanto direcciones de objetos constantes como de objetos no constantes:

```
...
    int x = 100;          // Objeto entero
    const int y = 200;    // Objeto entero constante

    const int *cp;        // Puntero a entero constante
    cp = &x;              // Asignamos a cp la dirección de un
objeto no constante
    cp = &y;              // Asignamos a cp la dirección de un
objeto constante
...
```

Lo que está claro es que cualquier objeto referenciado por *cp* nunca podrá ser modificado mediante ese puntero:

```

...
    int x = 100;          // Objeto entero
    const int y = 200;    // Objeto entero constante

    const int *cp;        // Puntero a entero constante
    cp = &x;              // Asignamos a cp la dirección de un
objeto no constante
    (*cp)++;              // Ilegal, cp apunta a un objeto
constante
    x++;                  // Legal, x no es constante, ahora
*cp contendrá el valor 101
...

```

Modificador de almacenamiento volatile

Sintaxis:

```

volatile <tipo> <nombre_variable>;
<identificador_función> ( volatile <tipo> <nombre_variable>
);
<identificador_función> volatile;

```

Este modificador se usa con objetos que pueden ser modificados desde el exterior del programa, mediante procesos externos. Esta situación es común en programas multihilo o cuando el valor de ciertos objetos puede ser modificado mediante interrupciones o por hardware.

El compilador usa este modificador para omitir optimizaciones de la variable, por ejemplo, si se declara una variable sin usar el modificador **volatile**, el compilador o el sistema operativo puede almacenar el valor leído la primera vez que se accede a ella, bien en un registro o en la memoria caché. O incluso, si el compilador sabe que no ha modificado su valor, no actualizarla en la memoria normal. Si su valor se modifica externamente, sin que el programa sea notificado, se pueden producir errores, ya que estaremos trabajando con un valor no válido.

Usando el modificador **volatile** obligamos al compilador a consultar el valor de la variable en memoria cada vez que se deba acceder a ella.

Por esta misma razón es frecuente encontrar juntos los modificadores **volatile** y **const**: si la variable se modifica por un proceso externo, no tiene mucho sentido que el programa la modifique.

Las formas segunda y tercera de la sintaxis expuesta sólo se aplica a clases, y las veremos más adelante.

Modificador de almacenamiento mutable

Sintaxis:

```
class <identificador_clase> {  
    ...  
    mutable <tipo> <nombre_variable>;  
    ...  
};  
  
struct <identificador_estructura> {  
    ...  
    mutable <tipo> <nombre_variable>;  
    ...  
};
```

Sirve para que determinados miembros de un objeto de una estructura o clase declarado como constante, puedan ser modificados.

```
#include <iostream>  
  
using namespace std;  
  
struct stA {  
    int y;  
    int x;  
};
```

```
struct stB {
    int a;
    mutable int b;
};

int main() {
    const stA A = {1, 2}; // Obligatorio inicializar
    const stB B = {3, 4}; // Obligatorio inicializar

    //    A.x = 0; // Ilegal (1)
    //    A.y = 0;
    //    B.a = 0;
    B.b = 0; // Legal (2)
    return 0;
}
```

Como se ve en (2), es posible modificar el miembro "b" del objeto "B", a pesar de haber sido declarado como constante. Ninguno de los otros campos, ni en "A", ni en "B", puede ser modificado.

Palabras reservadas usadas en este capítulo

auto, const, extern, mutable, register, static y volatile.

26 Espacios con nombre

Ya hemos usado espacios con nombre en los ejemplos, pero aún no hemos explicado por qué lo hacemos, qué significan o para qué sirven.

Un espacio con nombre, como indica su denominación, es una zona separada donde se pueden declarar y definir objetos, funciones y en general, cualquier identificador de tipo, clase, estructura, etc; al que se asigna un nombre o identificador propio.

Hasta ahora, en nuestros programas, siempre habíamos declarado y definido nuestros identificadores fuera de cualquier espacio con nombre, en lo que se denomina el *espacio global*. En este capítulo veremos que podemos crear tantos espacios para identificadores como necesitemos.

En cuanto a la utilidad los espacios con nombre, veremos nos ayudan a evitar problemas con identificadores en grandes proyectos o cuando se usan bibliotecas externas. Nos permite, por ejemplo, que existan objetos o funciones con el mismo nombre, declarados en diferentes ficheros fuente, siempre y cuando se declaren en distintos espacios con nombre.

Declaraciones y definiciones

Sintaxis para crear un espacio con nombre:

```
namespace [<identificador>] {  
    ...  
    <declaraciones y definiciones>  
    ...  
}
```

Veamos un ejemplo:

```
// Fichero de cabecera "puntos.h"
namespace espacio_2D {
    struct Punto {
        int x;
        int y;
    };
}

namespace espacio_3D {
    struct Punto {
        int x;
        int y;
        int z;
    };
}
} // Fin de fichero
```

Este ejemplo crea dos versiones diferentes de la estructura *Punto*, una para puntos en dos dimensiones, y otro para puntos en tres dimensiones.

Sintaxis para activar un espacio para usar por defecto, esta forma se conoce como forma directiva de **using**:

```
using namespace <identificador>;
```

Ejemplo:

```
#include "puntos.h"
using namespace espacio_2D; // Activar el espacio con nombre
espacio_2D

Punto p1; // Define la variable p1 de tipo
espacio_2D::Punto
espacio_3D::Punto p2; // Define la variable p2 de tipo
espacio_3D::Punto
...
```

Sintaxis para activar un identificador concreto dentro de un espacio con nombre, esta es la forma declarativa de **using**:

```
using <nombre_de_espacio>::<identificador>;
```

Ejemplo:

```
#include "puntos.h"

using espacio_3D::Punto; // Usar la declaración de Punto
                        // del espacio con nombre
espacio_3D
...

Punto p2;                // Define la variable p2 de tipo
espacio_3D::Punto (el espacio activo)
espacio_2D::Punto p1; // Define la variable p1 de tipo
espacio_2D::Punto
...
```

Sintaxis para crear un alias de un espacio con nombre:

```
namespace <alias_de_espacio> = <nombre_de_espacio>;
```

Ejemplo:

```
namespace nombredemasiadolargoycomplicado {
...
declaraciones
...
}
...
namespace ndlyc = nombredemasiadolargoycomplicado; // Alias
...
```

Utilidad

Este mecanismo permite reutilizar el código en forma de bibliotecas, que de otro modo no podría usarse. Es frecuente que

diferentes diseñadores de bibliotecas usen los mismos nombres para cosas diferentes, de modo que resulta imposible integrar esas bibliotecas en la misma aplicación.

Por ejemplo un diseñador crea una biblioteca matemática con una clase llamada "Conjunto" y otro una biblioteca gráfica que también contenga una clase con ese nombre. Si nuestra aplicación incluye las dos bibliotecas, obtendremos un error al intentar declarar dos clases con el mismo nombre.

El nombre del espacio funciona como un prefijo para las variables, funciones o clases declaradas en su interior, de modo que para acceder a una de esas variables se tiene que usar el operador de especificador de ámbito (::), o activar el espacio con nombre adecuado.

Por ejemplo:

```
#include <iostream>

namespace uno {
    int x;
}

namespace dos {
    int x;
}

using namespace uno;

int main() {
    x = 10;
    dos::x = 30;

    std::cout << x << ", " << dos::x << std::endl;
    std::cin.get();
    return 0;
}
```

En este ejemplo hemos usado tres espacios con nombre diferentes: "uno", "dos" y "std". El espacio "std" se usa en todas las

bibliotecas estándar, de modo que todas las funciones y clases estándar se declaran y definen en ese espacio.

Hemos activado el espacio "uno", de modo que para acceder a clases estándar como *cout*, tenemos que especificar el nombre: *std::cout*.

También es posible crear un espacio con nombre a lo largo de varios ficheros diferentes, de hecho eso es lo que se hace con el espacio *std*, que se define en todos los ficheros estándar.

Espacios anónimos

¿Espacios con nombre sin nombre?

Si nos fijamos en la sintaxis de la definición de un espacio con nombre, vemos que el nombre es opcional, es decir, podemos crear espacios con nombre anónimos.

Pero, ¿para qué crear un espacio anónimo? Su uso es útil para crear identificadores accesibles sólo en determinadas zonas del código. Por ejemplo, si creamos una variable en uno de estos espacios en un fichero fuente concreto, la variable sólo será accesible desde ese punto hasta el final del fichero.

```
namespace Nombre {
    int f();
    char s;
    void g(int);
}

namespace {
    int x = 10;
}
// x sólo se puede desde este punto hasta el final del
// fichero
// Resulta inaccesible desde cualquier otro punto o fichero

namespace Nombre {
    int f() {
        return x;
    }
}
```

```
}  
}
```

Este mecanismo nos permite restringir el ámbito de objetos y funciones a un fichero determinado dentro de un proyecto con varios ficheros fuente. Si recordamos el {cc:025#VARV_AMBITO:capítulo anterior}, esto se podía hacer con el especificador **static** aplicado a funciones o a objetos globales.

De hecho, la especificación de C++ aconseja usar espacios con nombre anónimos para esto, en lugar del especificador **static**, con el fin de evitar la confusión que puede producir este doble uso del especificador.

Espacio global

Cualquier declaración hecha fuera de un espacio con nombre pertenece al *espacio global*.

Precisamente porque las bibliotecas estándar declaran todas sus variables, funciones, clases y objetos en el espacio *std*, es necesario usar las nuevas versiones de los ficheros de cabecera estándar: "iostream", "fstream", etc. Y en lo que respecta a las procedentes de C, hay que usar las nuevas versiones que comienzan por 'c' y no tienen extensión: "cstdio", "cstdlib", "cstring", etc...". Todas esas bibliotecas han sido rescritas en el espacio con nombre *std*.

Si decidimos usar los ficheros de cabecera de C en nuestros programas C++, estaremos declarando las variables, estructuras y funciones en el espacio global.

Espacios anidados

Los espacios con nombre también se pueden anidar:

```
#include <iostream>
```



```

namespace uno {
    int x;
    namespace dos {
        int x;
        namespace tres {
            int x;
        }
    }
}

using std::cout;
using std::endl;
using uno::x;

int main() {
    x = 10; // Declara x como uno::x
    uno::dos::x = 30;
    uno::dos::tres::x = 50;

    cout << x << ", " << uno::dos::x <<
        ", " << uno::dos::tres::x << endl;
    std::cin.get();
    return 0;
}

```

Palabras reservadas usadas en este capítulo

namespace y using.

27 Clases I: Definiciones

Aunque te parezca mentira, hasta ahora no hemos visto casi nada de C++ que le sea exclusivo a este lenguaje. La mayor parte de lo incluido hasta el momento también forma parte de C.

Ahora vamos a entrar a fondo en lo que constituye el mayor añadido de C++ a C: las clases, y por tanto, la programación orientada a objetos. 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.

Definiciones

Lo mejor es definir algunos conceptos clave en la programación orientada a objetos, más que nada para que todos sepamos de qué estamos hablando.

POO

Siglas de "Programación Orientada a Objetos". En inglés se escribe al revés "OOP". La idea básica de este *paradigma* de programación es agrupar los datos y los procedimientos para manejarlos en una única entidad: el objeto. Cada programa es un objeto, que a su vez está formado de objetos que se relacionan entre ellos.

Esto no significa que la idea de la programación estructurada haya desaparecido, de hecho se refuerza y resulta más evidente, como comprobarás cuando veamos conceptos como la herencia.

Nota:

Un paradigma es un modelo de trabajo o patrón compartido por una comunidad científica cuyos miembros están de acuerdo en qué es un problema legítimo y cuál es una solución legítima del problema, por lo que se comparten conceptos básicos, procedimientos, etc.

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 e interrelacionan 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 de ese objeto.

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 una instancia (un ejemplar) 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

Se trata de la parte del objeto que es visible para el resto de los objetos. Es decir, que es el conjunto de métodos (y a veces datos) de que dispone un objeto para comunicarse con él.

Las clases y por lo tanto también los objetos definidos a partir de ellas, tienen partes públicas y partes privadas. Generalmente, llamaremos a la parte pública de una clase u objeto su interfaz.

Herencia

Es la capacidad de crear nuevas clases basándose en clases previamente definidas, de las que se aprovechan ciertos datos y métodos, se desechan otros y se añaden nuevos.

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 otras, 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.

Jerarquía

Definición del orden de subordinación de un sistema clases.

En POO la herencia siempre se produce en un mismo sentido: se toman una o varias clases base y se crean clases nuevas que heredan parte de las características de las clases de las que procede.

Este procedimiento crea estructuras de clases en forma de árbol, en las que siempre es posible retroceder hasta una o varias clases base.

Veremos que la jerarquía es importante cuando hablemos de conceptos como polimorfismo.

Polimorfismo

La definición del diccionario es: "Cualidad de lo que tiene o puede tener distintas formas".

Esto ya nos da alguna pista. En POO este concepto es similar:

Propiedad según la cual un mismo objeto puede considerarse como perteneciente a distintas clases.

Esta propiedad se presenta sólo en objetos declarados a partir de una clase derivada. Nos permite tratar el mismo objeto como si hubiese sido declarado a partir de alguna de las clases base.

Es un concepto avanzado, y lo veremos con mayor detalle y claridad en el capítulo 37.

La idea básica es bastante simple. Por ejemplo, en una jerarquía como: "ser vivo", "vertebrado", "mamífero", "felino", "gato común"; un "objeto" creado de la clase "gato común" podría tratarse, según nos convenga, como un "vertebrado", un "felino", o como un objeto de cualquier clase de la jerarquía.

Si en la misma jerarquía derivamos de la clase "felino" otras clases como "tigre", "león" y "pantera", en determinadas circunstancias podríamos considerar de la misma clase objetos de las clases "gato doméstico", "tigre", "león", o "pantera". Podríamos, por ejemplo, almacenarlos todos en un *array* de "felinos".

28 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 definir una clase y para declarar objetos de esa clase. Su uso es parecido a la ya conocida **struct**:

```
class <identificador de clase> [<:lista de clases base>] {  
    <lista de miembros>  
} [<lista de identificadores 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 pueden ser 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>
```

```

using namespace std;

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() {
    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, y también que si no especificamos que estamos definiendo una función de la clase "pareja", en realidad estaremos definiendo una función corriente.

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 este 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.

Especificadores 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.

Palabras reservadas usadas en este capítulo

class, private, protected y public.

29 Constructores

Los constructores son funciones miembro especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara.

Los constructores son especiales por varios motivos:

- Tienen el mismo nombre que la clase a la que pertenecen.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- No pueden ser heredados.
- Por último, 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.

Sintaxis:

```
class <identificador de clase> {  
    public:  
        <identificador de clase>(<lista de parámetros>) [:  
<lista de constructores>] {  
            <código del constructor>  
        }  
        ...  
}
```

Añadamos un constructor a nuestra clase pareja:

```
#include <iostream>  
using namespace std;  
  
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;
};

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() {
    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 no definimos un constructor el compilador creará uno por defecto, sin parámetros, que no hará absolutamente nada. Los datos miembros de los objetos declarados en el programa contendrán basura.

Si una clase posee constructor, será llamado siempre que se declare un objeto de esa clase. Si ese constructor requiere argumentos, como en este caso, es obligatorio suministrarlos.

Por ejemplo, las siguientes declaraciones son ilegales:

```
pareja par1;  
pareja par1();
```

La primera porque el constructor de "pareja" requiere dos parámetros, y no se suministran.

La segunda es ilegal por otro motivo más complejo. Aunque existiese un constructor sin parámetros, no se debe usar esta forma para declarar el objeto, ya que el compilador lo considera como la declaración de un prototipo de una función que devuelve un objeto de tipo "pareja" y no admite parámetros.

Cuando se use un constructor sin parámetros para declarar un objeto no se deben escribir los paréntesis.

Y las siguientes declaraciones son válidas:

```
pareja par1(12,43);  
pareja par2(45,34);
```

Constructor por defecto

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.

Para declarar objetos usando el constructor por defecto o un constructor que hayamos declarado sin parámetros no se debe usar el paréntesis:

```
pareja par2();
```

Se trata de un error frecuente cuando se empiezan a usar clases, lo correcto es declarar el objeto sin usar los paréntesis:

```
pareja par2;
```

Inicialización de objetos

Hay un modo específico para inicializar los datos miembros de los objetos en los constructores, que consiste en invocar los constructores de los objetos miembro antes de las llaves de la definición del constructor.

En C++ incluso las variables de tipos básicos como **int**, **char** o **float** son objetos. En C++ cualquier variable (u objeto) tiene, al menos un constructor, el constructor por defecto, incluso aquellos que son de un tipo básico.

Sólo los constructores de las clases admiten inicializadores. Cada inicializador consiste en el nombre de la variable miembro a inicializar, seguida de la expresión que se usará para inicializarla entre paréntesis. Los inicializadores se añadirán a continuación del paréntesis cerrado que encierra a los parámetros del constructor, antes del cuerpo del constructor y separado del paréntesis por dos puntos ":".

Por ejemplo, en el caso anterior de la clase "pareja" teníamos este constructor:

```
pareja::pareja(int a2, int b2) {  
    a = a2;  
    b = b2;  
}
```

Podemos (y debemos) sustituir ese constructor por este otro:

```
pareja::pareja(int a2, int b2) : a(a2), b(b2) {}
```

Por supuesto, también pueden usarse inicializadores en línea, dentro de la declaración de la clase.

Ciertos miembros es obligatorio inicializarlos, ya que no pueden ser asignados, por ejemplo las constantes o las referencias. Es preferible usar la inicialización siempre que sea posible en lugar de asignaciones, ya que frecuentemente, es menos costoso y más predecible inicializar objetos en el momento de la creación que usar asignaciones.

Veremos más sobre este tema cuando veamos ejemplos de clases que tienen como miembros objetos de otras clases.

Sobrecarga de constructores

Los constructores son funciones, también pueden definirse varios constructores para cada clase, es decir, el constructor puede sobrecargarse. La única limitación (como en todos los casos de sobrecarga) 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) : a(a2), b(b2) {}
    pareja() : a(0), b(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;
};
```

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 dos datos miembros con cero.

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) : a(a2), b(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;
};
```

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:

```
#include <iostream>
using namespace std;
```



```

int main() {
    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.

En realidad, igual que pasa con los constructores, el compilador crea un operador de asignación por defecto, que copia los valores de todos los datos miembro de un objeto al otro. Veremos más adelante que podemos redefinir ese operador para nuestras clases, si lo consideramos necesario.

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) : a(a2), b(b2) {}
    // 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;
};

// Definición del constructor copia:
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;
}

```

Aunque pueda parecer confuso, el constructor copia en otras circunstancias:

```

int main() {
    pareja par1(12, 32)
    pareja par2 = par1; // Uso del constructor copia
    ...
}

```

En este caso se usa el constructor copia porque el objeto par2 se inicializa al mismo tiempo que se declara, por lo tanto, el compilador busca un constructor que tenga como parámetro un objeto del tipo de par1, es decir, busca un constructor copia.

Tanto es así que se invoca al constructor copia aunque el valor a la derecha del signo igual no sea un objeto de tipo pareja.

Disponemos de un constructor con valores por defecto para los parámetros, así que intentemos hacer esto:

```
int main() {  
    pareja par2 = 14; // Uso del constructor copia  
    ...  
}
```

Ahora el compilador intenta crear el objeto par2 usando el constructor copia sobre el objeto 14. Pero 14 no es un objeto de la clase pareja, de modo que el compilador usa el constructor de pareja con el valor 14, y después usa el constructor copia.

También para cualquier clase, 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 muchas ocasiones necesitaremos redefinir el constructor copia.

Problemas

1. Problemas de sudokus para resolver con clases:

A partir de este capítulo, y a lo largo de los siguientes, iremos creando una estructura de clases para resolver Sudokus ([enlace del juego en Wikipedia](#)).

Este problema se ajusta bien a un enfoque POO, ya que es lo suficientemente complejo (ya iremos viendo algunos

algoritmos para resolución), y en la estructura del juego es relativamente sencillo localizar objetos.

Empecemos por algunas clases auxiliares que usaremos más adelante y que nos ayudarán a clasificar distintas capas del juego.

Lo más evidente, cuando se ve un Sudoku clásico, es que se trata de una tabla de 9x9 casillas, en total 81. Una vez resuelto, en cada casilla debe haber un número entre 1 y 9, y se deben cumplir tres reglas:

1. En cada fila deben aparecer los nueve dígitos (1 a 9).
2. En cada columna deben aparecer los nueve dígitos (1 a 9).
3. En cada bloque cuadrado de 3x3 casillas deben aparecer los nueve dígitos (1 a 9).

De momento, el primer problema es diseñar una clase para contener la información de una única casilla. Entre los datos que debe tener, en esta primera versión, están los siguientes:

- valor: el valor de la casilla, entre 0 y 9. Usaremos un char para este dato.
- posible: un conjunto de 9 bits, cada uno indica si el número n es un posible valor para la casilla.
- opciones: número de valores posibles, es decir, un contador de los bits "posible" con valor 1.

Durante la resolución del Sudoku iremos aplicando reglas que eliminen valores posibles de ciertas casillas. Cuando sólo quede una posibilidad, esa será el valor de la casilla. En ese momento sólo uno de los bits de posible será 1, opciones tendrá el valor 1 y valor tendrá el de la única posibilidad viable.

También puede suceder, porque queremos ser previsores, que todos los bits de opciones sean 0, y que opciones valga 0. Eso significa que el Sudoku actual no tiene solución.

Otros datos miembro, que usaremos para aplicar las reglas más tarde de una forma más sencilla son:

- sigFila: puntero a la siguiente casilla de la misma fila.
- sigColumna: puntero a la siguiente casilla de la misma columna.

- sigBloque: puntero a la siguiente casilla del mismo bloque.

Usaremos estos punteros para recorrer cada fila, columna o bloque cuando nos interese.

Más adelante añadiremos más métodos, pero en esta primera versión debe tener los siguientes:

- Un constructor, que recibirá parámetro char con un valor entre 0 y 9, y con un valor por defecto de 0. El cero indica que la casilla está vacía, es decir, que aún no se le ha asignado un valor. Por supuesto, debe asignar valores iniciales para todos los datos miembro de la clase. Los que no se pueda, porque sean desconocidos, serán 0.
- Un método para modificar el valor de sigFila.
- Un método para modificar el valor de sigColumna.
- Un método para modificar el valor de sigBloque.
- Un método para obtener el valor de sigFila.
- Un método para obtener el valor de sigColumna.
- Un método para obtener el valor de sigBloque.

Definir una segunda clase Tablero, que debe contener las 81 casillas. Para ello, uno de los datos miembro será, precisamente, un array de Casillas de 9x9. De momento, no añadiremos más datos.

Añadir dos constructores. Uno de ellos sin parámetros, que creará un tablero con todas las casillas vacías. El segundo admitirá como parámetro una cadena de 81 caracteres, como un puntero a char. En este caso se iniciará cada casilla con el valor correspondiente de la cadena, fila a fila.

Para codificar cada casilla en la cadena se consideran válidos los valores '1' a '9', y para indicar casillas vacías serán válidos los caracteres siguientes: '.', '-', o un espacio. Cualquier error en la cadena, ya sea de longitud (cadenas de más o menos de 81 caracteres), o de contenido (caracteres no permitidos), debe crear un tablero vacío.

Los constructores deben inicializar los valores de los punteros sigFila, sigColumna y sigBloque de cada una de las 81 casillas.

2. Crear una clase Fecha que tenga un constructor, con tres parámetros: día, mes y año. Para evitar fracasos en la creación de la fecha (fechas no válidas), se añadirá un dato a la clase que indique si la fecha actual es o no válida.

En siguientes capítulos iremos añadiendo más métodos a esta clase, así que no la pierdas.

3. Crear una clase para representar números racionales (fracciones). Se debe implementar un constructor, un método para simplificar, y un método privado para calcular el máximo común divisor.

Consultar [Ejemplo 11-4](#), [Ejemplo 11-5](#) y [Ejemplo 24-2](#) sobre operaciones con fracciones y cálculo del máximo común divisor.

También se debe crear un método para mostrar el objeto en pantalla, en la forma "N/D", donde N es el numerador y D el denominador.

Conservar este programa para futuras ampliaciones.

30 destructores

Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase. El destructor realizará procesos necesarios cuando un objeto termine su ámbito temporal, por ejemplo liberando la memoria dinámica utilizada por dicho objeto o liberando recursos usados, como ficheros, dispositivos, etc.

Al igual que los constructores, los destructores también tienen algunas características especiales:

- También tienen el mismo nombre que la clase a la que pertenecen, pero tienen el símbolo ~ delante.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- No tienen parámetros.
- No pueden ser heredados.
- Deben ser públicos, no tendría ningún sentido declarar un destructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.
- No pueden ser sobrecargados, lo cual es lógico, puesto que no tienen valor de retorno ni parámetros, no hay posibilidad de sobrecarga.

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, cuando es necesario eliminarlo, hay que hacerlo explícitamente usando 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.

Ejemplo:

```
#include <iostream>
#include <cstring>
using namespace std;

class cadena {
public:
    cadena();           // Constructor por defecto
    cadena(const char *c); // Constructor desde cadena c
    cadena(int n);      // Constructor de cadena de n caracteres
    cadena(const cadena &); // Constructor copia
    ~cadena();          // Destructor

    void Asignar(const char *dest);
    char *Leer(char *c);
private:
    char *cad;          // Puntero a char: cadena de caracteres
};

cadena::cadena() : cad(NULL) {}

cadena::cadena(const char *c) {
    cad = new char[strlen(c)+1]; // Reserva memoria para
    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) {
    // Reservamos memoria para la nueva y la almacenamos
    cad = new char[strlen(Cad.cad)+1];
    // Reserva memoria para cadena
    strcpy(cad, Cad.cad);          // Almacena la cadena
}

cadena::~~cadena() {
    delete[] cad;          // Libera la memoria reservada a cad
}
```



```

void cadena::Asignar(const 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() {
    cadena Cadenal("Cadena de prueba");
    cadena Cadenal2(Cadenal);    // Cadenal2 es copia de Cadenal
    cadena *Cadenal3;           // Cadenal3 es un puntero
    char c[256];

    // Modificamos Cadenal:
    Cadenal.Asignar("Otra cadena diferente");
    // Creamos Cadenal3:
    Cadenal3 = new cadena("Cadena de prueba nº 3");

    // Ver resultados
    cout << "Cadena 1: " << Cadenal.Leer(c) << endl;
    cout << "Cadena 2: " << Cadenal2.Leer(c) << endl;
    cout << "Cadena 3: " << Cadenal3->Leer(c) << endl;

    delete Cadenal3; // Destruir Cadenal3.
    // Cadenal y Cadenal2 se destruyen automáticamente

    return 0;
}

```

Ejecutar este código en [codepad](#).

Voy a hacer varias observaciones sobre este programa:

1. Hemos implementado un constructor copia. Esto es necesario porque una simple asignación entre los datos miembro "cad" no copiaría la cadena de un objeto a otro, sino únicamente los punteros.

Por ejemplo, si definimos el constructor copia como:

```
cadena::cadena(const cadena &Cad) {  
    cad = Cad.cad;  
}
```

en lugar de cómo lo hacemos en el ejemplo, 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 del ejemplo, y es lo que haremos más adelante, y con más elegancia, sobrecargando el operador de asignación.

La definición del constructor copia que hemos creado en este último ejemplo es la equivalente a la del constructor copia por defecto.

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. Más adelante veremos cómo se puede conseguir mantener la seguridad sin crear más datos miembro.
3. La `Cadena3` debe ser destruida implícitamente usando el operador **`delete`**, que a su vez invoca al destructor de la clase. Esto es así porque `Cadena3` es un puntero, y la memoria que se usa en el objeto al que apunta no se libera automáticamente al destruirse el puntero `Cadena3`.

31 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 especial llamado **this**. Se trata de un puntero que tiene asociado cada objeto y que apunta a si mismo. Ese puntero se puede usar, y de hecho se usa, para acceder a sus miembros.

Volvamos al ejemplo de la clase pareja:

```
#include <iostream>
using namespace std;

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;
};
```

Para cada dato podemos referirnos de dos modos distintos, lo veremos con la función Guarda. Esta es la implementación que

usamos en el {cc:029#inicio:capítulo 29}, 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) {  
    this->a = a2;  
    this->b = b2;  
}
```

Veamos otro ejemplo donde podemos aplicar el operador **this**. Se trata de la aplicación más frecuente, como veremos al implementar el constructor copia, o al sobrecargar ciertos operadores.

A veces necesitamos invocar a una función de una clase con una referencia a un objeto de la misma clase, pero las acciones a tomar serán diferentes dependiendo de si la referencia que pasamos se refiere al mismo objeto o a otro diferente, veamos cómo podemos usar el puntero **this** para determinar esto:

```
#include <iostream>  
using namespace std;  
  
class clase {  
public:  
    clase() {}  
    void EresTu(clase& c) {  
        if(&c == this) cout << "Sí, soy yo." << endl;  
        else cout << "No, no soy yo." << endl;  
    }  
};
```

```
int main() {  
    clase c1, c2;  
  
    c1.EresTu(c2);  
    c1.EresTu(c1);  
  
    return 0;  
}
```

La función "EresTu" recibe una referencia a un objeto de la clase "clase". Para saber si se trata del mismo objeto, comparamos la dirección del objeto recibido con el valor de **this**, si son la misma, es que se trata del mismo objeto.

```
No, no soy yo.  
Sí, soy yo.
```

Este puntero nos resultará muy útil en próximos capítulos, en los que nos encontraremos situaciones en las que es imprescindible su uso.

Cómo funciona

Para intentar comprender algo mejor cómo funciona este puntero, veremos una forma de simularlo usando un ejemplo con estructuras.

Intentaremos crear una estructura sencilla que tenga un miembro con un puntero a un objeto de su mismo tipo. Además, haremos que ese puntero contenga la dirección del propio objeto. De modo que si creamos varios objetos de esa clase, cada uno de ellos apunte a si mismo.

Después de intentarlo un rato, comprobaremos que no es posible hacer esto dentro del constructor, ya que en ese momento no tenemos referencias al objeto. Tendremos que hacerlo, pues,

añadiendo un parámetro más al constructor o después de crear el objeto, añadiendo una línea `obj.esto = &obj;.`

```
#include <iostream>
using namespace std;

struct ejemplo {
    ejemplo(int v, ejemplo* e);
    int valor;
    ejemplo *esto;
};

ejemplo::ejemplo(int v, ejemplo *e) : valor(v), esto(e) {}

int main() {
    ejemplo e1(19, &e1);

    cout << &e1 << " " << (void*)e1.esto << endl;
    return 0;
}
```

Ejecutar este código en [codepad](#).

Bien, esto es lo que hace el compilador en nuestro lugar cuando *crea* el puntero *this*. Y lo pongo en cursiva porque en realidad ese puntero no se almacena junto con el objeto. No ocupa memoria, y por lo tanto no se tiene en cuenta al calcular el tamaño de la estructura con el operador `sizeof`, no se crea, ni se inicializa. El compilador *sabe* durante la fase de compilación dónde almacena cada objeto, y por lo tanto, puede sustituir las referencias al puntero *this* por sus valores en cada caso.

El puntero *this* nos permite hacer cosas como obtener una referencia al propio objeto como valor de retorno en determinadas funciones u operadores. En ocasiones puede resolver ambigüedades o aclarar algún código.

Palabras reservadas usadas en este capítulo

this.

Ejemplos capítulos 27 a 31

Ejemplo 31.1

Ahora estamos en disposición de empezar a usar clases para modelar algunos problemas cotidianos.

Empezaremos por las fracciones. Ya hemos hecho algunas aproximaciones usando estructuras, ahora usaremos una clase, y en sucesivos capítulos, iremos añadiendo más refinamientos.

```
// Clase fracción
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>

using namespace std;

class fraccion {
public:
    fraccion(int n=0, int d=0) : numerador(n),
denominador(d) {
        Simplificar();
    }
    void Simplificar();
    void Mostrar();
private:
    int numerador;
    int denominador;
    int MCD(int, int);
};

void fraccion::Simplificar() {
    int mcd = MCD(numerador, denominador);
    numerador /= mcd;
    denominador /= mcd;
}

void fraccion::Mostrar() {
```

```

        cout << numerador << "/" << denominador << endl;
    }

    int fraccion::MCD(int a, int b) {
        if(a < b) return MCD(b,a);
        if(b == 0) return a;
        return MCD(b, a % b);
    }

    int main() {
        fraccion f1(234, 2238);
        fraccion f2(64, 1024);

        f1.Mostrar();
        f2.Mostrar();
        return 0;
    }

```

Ejecutar este código en [codepad](#).

Hemos invocado al método *Simplificar* dentro del constructor, de ese modo, la fracción siempre se almacenará simplificada cuando se cree. Probablemente esto haga que sea innecesario que *Simplificar* sea pública, ya que cualquier llamada desde el exterior no afectará a la fracción. Podríamos haberla declarado como privada.

Ejemplo 31.2

Añadamos una función *Sumar* a esta clase, que nos sirva para sumar dos fracciones:

```

// Clase fracción
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>

using namespace std;

class fraccion {

```



```

    public:
        fraccion(int n=0, int d=0) : numerador(n),
denominador(d) {
            Simplificar();
        }
        void Simplificar();
        void Sumar(fraccion);
        void Mostrar();
    private:
        int numerador;
        int denominador;
        int MCD(int, int);
};

void fraccion::Simplificar() {
    int mcd = MCD(numerador, denominador);
    numerador /= mcd;
    denominador /= mcd;
}

void fraccion::Sumar(fraccion f2) {
    numerador =
numerador*f2.denominador+denominador*f2.numerador;
    denominador = denominador*f2.denominador;
    Simplificar();
}

void fraccion::Mostrar() {
    cout << numerador << "/" << denominador << endl;
}

int fraccion::MCD(int a, int b) {
    if(a &lt; b) return MCD(b,a);
    if(b == 0) return a;
    return MCD(b, a % b);
}

int main() {
    fraccion f1(234, 2238);
    fraccion f2(64, 1024);

    f1.Mostrar();
    f2.Mostrar();

    f1.Sumar(f2);
    f1.Mostrar();
}

```

```
    return 0;
}
```

Ejecutar este código en [codepad](#).

Más adelante veremos como sobrecargar el operador suma para implementar esta función.

Ejemplo 31.3

En este ejemplo crearemos una clase "tonta", que nos servirá para seguir la pista a constructores y destructores a medida que son invocados de forma implícita o explícita. En próximos capítulos añadiremos más cosas para monitorizar otras características.

```
// Clase tonta
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>

using namespace std;

class tonta {
public:
    tonta();
    tonta(int);
    tonta(const tonta&);
    ~tonta();
    int Modifica(int);
    int Lee();
private:
    int valor;
};

tonta::tonta() : valor(0) {
    cout << "Constructor sin parámetros (0)" << endl;
}

tonta::tonta(int v) : valor(v) {
    cout << "Constructor con un parámetro (" << v << ")" <<
endl;
```

```

}

tonta::tonta(const tonta &t) : valor(t.valor) {
    cout << "Constructor copia (" << t.valor << ")" << endl;
}

tonta::~~tonta() {
    cout << "Destructor (" << valor << ")" << endl;
}

int tonta::Modifica(int v) {
    int retval = valor;

    cout << "Modificar valor (" << valor << ") -> (" << v <<
    ")" << endl;
    valor = v;
    return retval;
}

int tonta::Lee() {
    return valor;
}

int main() {
    tonta obj1;
    tonta *obj2;
    tonta obj3 = obj1;

    obj2 = new tonta(2);

    obj1.Modifica(3);
    cout << "Objeto1: " << obj1.Lee() << endl;
    cout << "Objeto2: " << obj2->Lee() << endl;
    cout << "Objeto3: " << obj3.Lee() << endl;

    delete obj2;

    return 0;
}

```

Ejecutar este código en [codepad](#).

Viendo la salida de este programa:

```

Constructor sin parámetros (0)
Constructor copia (0)

```

```
Constructor con un parámetro (2)
Modificar valor (0) -> (3)
Objeto1: 3
Objeto2: 2
Objeto3: 0
Destructor (2)
Destructor (0)
Destructor (3)
```

vemos que para obj1 se invoca al constructor sin parámetros cuando es declarado, y que para obj2 se invoca al constructor con un parámetro cuando es invocado explícitamente junto al operador **new**.

Con los destructores pasa lo mismo, para obj2 se invoca al destruir el objeto mediante el operador **delete**, y para obj1 se invoca de forma implícita al terminar su ámbito temporal.

32 Sistema de protección

Ya sabemos que los miembros privados de una clase no son accesibles para funciones y clases exteriores a dicha clase.

Este es uno de los conceptos de POO, el *encapsulamiento*, que tiene como objetivo hacer que lo que pase en el interior de cada objeto sea inaccesible desde el exterior, y que el comportamiento de otros objetos no pueda influir en él. Cada objeto sólo responde a ciertos mensajes y proporciona determinadas salidas.

Pero, en ciertas ocasiones, necesitaremos tener acceso a determinados miembros de un objeto de una clase desde otros objetos de clases diferentes, pero sin perder ese encapsulamiento para el resto del programa, es decir, manteniendo esos miembros como privados.

C++ proporciona un mecanismo para sortear el sistema de protección. En otros capítulos veremos la utilidad de esta técnica, pero 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 de nuevo me temo que en realidad se trata de una situación muy frecuente, y normalmente A no sabe que B no se considera su amigo).

Funciones amigas externas

El caso más sencillo es el de una relación de amistad con una función externa.

Veamos un ejemplo muy sencillo:

```
#include <iostream>
using namespace std;

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() {
    A Na(10);

    Ver(Na); // Ver el valor de Na.a
    Na.Ver(); // Equivalente a la anterior

    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>
using namespace std;

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() {
    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;

    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>
using namespace std;

/* Clase para elemento de lista enlazada */
class Elemento {
public:
    Elemento(int t);                /* Constructor */
    int Tipo() { return tipo; }    /* Obtener tipo */
private:
    int tipo;                      /* Datos: */
    Elemento *sig;                 /* Tipo */
    Elemento *sig;                 /* Siguiete elemento */
}

```



```

*/
    friend class Lista;                                /* Amistad con lista
*/
};

/* Clase para lista enlazada de números*/
class Lista {
    public:
        Lista() : Cabeza(NULL) {}                        /* Constructor
*/
                                                    /* Lista vacía
*/
        ~Lista() { LiberarLista(); }                    /* Destructor
*/
        void Nuevo(int tipo);                            /* Insertar figura
*/
        Elemento *Primero()                             /* Obtener primer elemento
*/
        { return Cabeza; }
        /* Obtener el siguiente elemento a p */
        Elemento *Siguiente(Elemento *p) {
            if(p) return p->sig; else return p;};
        /* Si p no es NULL */
        /* Averiguar si la lista está vacía */
        bool EstaVacio() { return Cabeza == NULL; }

    private:
        Elemento *Cabeza;                                /* Puntero al primer elemento
*/
        void LiberarLista(); /* Función privada para borrar lista
*/
};

/* Constructor */
Elemento::Elemento(int t) : tipo(t), sig(NULL) {}
    /* Asignar datos desde lista de parámetros */

/* 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() {
    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.Primer();
    while(e) {
        cout << e->Tipo() << " , ";
        e = miLista.Siguiente(e);
    }
    cout << endl;

    return 0;
}

```

Ejecutar este código en [codepad](#).

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.

Palabras reservadas usadas en este capítulo

friend.

33 Modificadores para miembros

Existen varias alternativas a la hora de definir algunos de los miembros de las clases. Esto es lo que veremos en este 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.

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 este 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. Además proporciona ciertos mecanismos necesarios para mantener la protección de los datos.

Cuando una función miembro no deba modificar el valor de ningún dato de la clase, podemos y debemos declararla como constante. Esto evitará que la función intente modificar los datos del objeto; pero como, a fin de cuentas, el código de la función lo escribimos nosotros; el compilador nos ayudará generando un error durante la compilación si la función intenta modificar alguno de los datos miembro del objeto, ya sea directamente o de forma indirecta.

Por ejemplo:

```
#include <iostream>  
using namespace std;  
  
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;  
  
    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.

Evidentemente, si somos nosotros los que escribimos el código de la función, sabemos si la función modifica o no los datos, de modo que en rigor no necesitamos saber si es o no constante, pero frecuentemente otros programadores pueden usar clases definidas por nosotros, o nosotros las definidas por otros. En ese caso es frecuente que sólo se disponga de la declaración de la clase, y el modificador "const" nos dice si cierto método modifica o no los datos del objeto.

Consideremos también que la función puede invocar a otras funciones, y usar este modificador nos asegura que ni siquiera esas funciones modifican datos del objeto.

Valores de retorno constantes

Otra técnica muy útil y aconsejable en muchos casos es usar valores de retorno de las funciones constantes, en particular cuando se usen para devolver punteros miembro de la clase.

Por ejemplo, supongamos que tenemos una clase para cadenas de caracteres:

```
class cadena {
public:
    cadena();           // Constructor por defecto
    cadena(const char *c); // Constructor desde cadena c
    cadena(int n);      // Constructor para cadena de n
    caracteres
    cadena(const cadena &); // Constructor copia
    ~cadena();           // Destructor

    void Asignar(const char *dest);
    char *Leer(char *c) {
        strcpy(c, cad);
        return c;
    }
private:
    char *cad;          // Puntero a char: cadena de
    caracteres
};
```

Si te fijas en la función "Leer", verás que devuelve un puntero a la cadena que pasamos como parámetro, después de copiar el valor de cad en esa cadena. Esto es necesario para mantener la protección de cad, si nos limitáramos a devolver ese parámetro, el programa podría modificar la cadena almacenada a pesar de se cad un miembro privado:

```
char *Leer() { return cad; }
```

Para evitar eso podemos declarar el valor de retorno de la función "Leer" como constante:

```
const char *Leer() { return cad; }
```

De este modo, el programa que lea la cadena mediante esta función no podrá modificar ni el valor del puntero ni su contenido. Por ejemplo:

```
class cadena {  
    ...  
};  
...  
int main() {  
    cadena Cadenal("hola");  
  
    cout << Cadenal.Leer() << endl; // Legal  
    Cadenal.Leer() = cadena2;       // Ilegal  
    Cadenal.Leer()[1] = 'O';        // Ilegal  
}
```

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>  
using namespace std;  
  
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();
}

// Definición e 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;

    return 0;
}

```

Ejecutar este código en [codepad](#).

Observa que es necesario declarar e inicializar los miembros **static** de la clase, esto es por dos motivos. El primero es que los miembros **static** deben existir aunque no exista ningún objeto de la clase, declarar la clase no crea los datos miembro estáticos, es necesario hacerlo explícitamente. 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 las 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 acceso al 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>
using namespace std;

```

```

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;
    return 0;
}

```

Ejecutar este código en [codepad](#).

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.

Palabras reservadas usadas en este capítulo

`const`, `inline` y `static`.

34 Más sobre las funciones

Funciones sobrecargadas

Ya hemos visto que se pueden sobrecargar los constructores, y en el [capítulo 21](#) 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.

No hay mucho más que añadir, así que pondré un ejemplo:

```
// Sobrecarga de funciones
// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>
using namespace std;

struct punto3D {
    float x, y, z;
};

class punto {
public:
    punto(float xi, float yi, float zi) :
        x(xi), y(yi), z(zi) {}
    punto(punto3D p) : x(p.x), y(p.y), z(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();

    return 0;
}

```

Ejecutar este código en [codepad](#).

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 {cc:020#inicio:capítulo 20} 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:

```
// Parámetros con valores por defecto
```

```

// (C) 2009 Con Clase
// Salvador Pozo

#include <iostream>
using namespace std;

class punto {
public:
    punto(float xi, float yi, float zi) :
        x(xi), y(yi), z(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();

    return 0;
}

```

Ejecutar este código en [codepad](#).

Las reglas para definir parámetros con valores por defecto son las mismas que se expusieron en el capítulo 22.

35 Operadores sobrecargados

Ya habíamos visto el funcionamiento de los operadores sobrecargados en el [capítulo 22](#), aplicándolos a operaciones con estructuras. Ahora veremos todo su potencial, aplicándolos 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 22](#), 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>
using namespace std;

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)

    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 o la misma clase.

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:

```

C2 = C1;

```

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 C2 se ha perdido, por lo que su memoria no puede ser liberada.

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. Esta es una de las situaciones en las que el puntero **this** es imprescindible.

También hay que tener cuidado de que la cadena a copiar no sea 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 también 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 mixtos no necesitan valor de retorno, ya que es el propio objeto al que se aplican el que recibe el resultado de la operación y además, no pueden asociarse.

Con el resto de los 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 los 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 devolverlo como resultado en el caso adecuado.

Esta es otra de las aplicaciones del puntero **this**, si no dispusiéramos de él, sería imposible hacer referencia al propio objeto al que se aplica el operador.

También vemos que los operadores binarios deben seguir admitiendo la asociación 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 demasiado aconsejable.

En el caso del operador + las siguientes expresiones son equivalentes:

```
T1 = T1.operator+(Ahora);  
  
T1 = Ahora + T1;
```

Sobrecarga de operadores para 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. El significado, en este caso, será concatenar las cadenas sumadas:

```
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"))));
```

Si aplicamos la precedencia a esta expresión, tenemos la siguiente secuencia:

- 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.

Cuando hablamos del ámbito temporal de los objetos locales o automáticos, vimos que éstos son destruidos tan pronto se abandona el ámbito al que pertenecen. Esto es lo que pasa con el objeto temp en el operador +. Sin embargo, parece ser que ese es precisamente el objeto que devolvemos. En realidad, el compilador crea otro objeto temporal, y copia en él el valor de temp antes de destruirlo y abandonar su ámbito.

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 nombre 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.

¡Ah!, pero como nosotros no hemos definido un constructor copia, de modo que 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 ha 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

En este ejemplo es sencillo ya que disponemos del operador de asignación. No debemos olvidar que hay que inicializar los datos miembros, el constructor copia no deja de ser un constructor:

```
class Cadena {  
    ...  
    Cadena(const Cadena &c) : cadena(NULL) { *this = c; }  
    ...  
};
```

Si no tenemos cuidado de iniciar el valor de cadena, cuando se invoque al operador "=" el puntero cadena tendrá algún valor inválido, y al ejecutar el código del operador de asignación se producirá un error al intentar liberarlo.

```
Cadena &Cadena::operator=(const Cadena &c) {  
    if(this != &c) {  
        delete[] cadena; // (1)  
        if(c.cadena) {  
            cadena = new char[strlen(c.cadena)+1];  
            strcpy(cadena, c.cadena);  
        }  
        else cadena = NULL;  
    }  
    return *this;  
}
```

En (1), si cadena no es NULL, pero tampoco es un puntero válido, se producirá un error de ejecución. En general, si se usa el operador de asignación con objetos que existan no habrá problema, pero si se usa desde el constructor copia debemos asegurarnos de que el puntero es NULL.

Nota: La moraleja es que cuando nuestras clases tengan datos miembro que sean punteros a memoria dinámica debemos sobrecargar 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)

Notas sobre este tema

Esto es sólo un comentario sobre algunos aspectos curiosos de los compiladores.

Puede que el compilador optimice el código de modo que no se cree el objeto copia del objeto temporal creado en el operador suma. Esto pasará si el compilador lo permite, y si lo permite el modelo de memoria utilizado. Por ejemplo, en Windows, con un modelo de memoria virtual, en el que no se distingue la memoria del montón de la pila o de la memoria local, es posible usar las direcciones de memoria del objeto creado para un objeto automático de un operador o función para un objeto automático de otra función o para uno global.

Así pasará en nuestro ejemplo con el objeto temp creado en el operador suma. Se trata de un objeto automático, y por lo tanto, local al operador. En algunos sistemas operativos este objeto se creará en una zona de memoria local (por ejemplo, la pila), y no estará accesible al retornar a la función *main*, desde el que fue invocado. En ese caso, el compilador creará una copia, invocando al constructor copia, y después destruirá el objeto temporal.

Con un modelo de memoria como el que usar Windows de 32 bits, por ejemplo, esto no es necesario, y el objeto puede ser persistente, aunque haya terminado su ámbito.

Pero hay que tener presente que esto es una *optimización*, y que en ningún caso puede ser tomado como una comportamiento general.

El siguiente ejemplo funcionará correctamente en Windows XP, y en otros sistemas operativos, a pesar de que se ha omitido intencionadamente el código para el constructor copia:

```

#include <cstring>
#include <iostream>

using namespace std;

class Cadena {
public:
    Cadena(const char *cad);
    Cadena() : cadena(0) { cout << "Constructor sin
parametros" << endl; };
    Cadena(const Cadena &c);
    ~Cadena();
    Cadena &operator=(const Cadena &c);
    Cadena operator+(const Cadena &);

    void Mostrar() const;
private:
    char *cadena;
};

Cadena::Cadena(const char *cad) {
    cadena = new char[strlen(cad)+1];
    strcpy(cadena, cad);
    cout << "Constructor (" << cad <<") [" << (void*)cadena
<< "]" << endl;
}

Cadena::Cadena(const Cadena &) {} // NO HACE NADA

Cadena::~~Cadena() {
    cout << "Destructor (" << cadena << ") [" <<
(void*)cadena << "]" << endl;
    delete[] cadena;
}

void Cadena::Mostrar() const {
    cout << cadena << endl;
}

Cadena &Cadena::operator=(const Cadena &c) {
    cout << "Asignacion" << endl;
    if(this != &c) {
        delete[] cadena;
        if(c.cadena) {
            cadena = new char[strlen(c.cadena)+1];
            strcpy(cadena, c.cadena);
        }
    }
}

```

```

        }
        else cadena = 0;
    }
    return *this;
}

Cadena Cadena::operator+(const Cadena &c) {
    cout << "suma-";
    Cadena temp;

    temp.cadena = new char[strlen(c.cadena)+strlen(this-
>cadena)+1];
    strcpy(temp.cadena, this->cadena);
    strcat(temp.cadena, c.cadena);
    return temp;
}

int main() {
    Cadena C1("Hola"), C2("Adios"), C3("");

    C3 = C1 + ", mundo!";
    C3.Mostrar();
    (C2+", mundo").Mostrar();
    return 0;
}

```

La salida puede ser:

Constructor (Hola) [0x3e3d90]	<- C1 en main
Constructor (Adios) [0x3e3df8]	<- C2 en main
Constructor () [0x3e3e08]	<- C3 en main
Constructor (, mundo!) [0x3e3e18]	<- C4 en main
suma-Constructor sin parametros	<- temp en
operator+	
Asignacion	<- Asignación a
C3	
Destructor (Hola, mundo!) [0x3e3e30]	<- Destrucción
de objeto temporal, una vez asignado	
Destructor (, mundo!) [0x3e3e18]	<- Destrucción
de objeto temporal de suma	
Hola, mundo!	<- Mostrar()
Constructor (, mundo) [0x3e3e08]	<- C2+",
mundo"; // Segundo operando en suma	
suma-Constructor sin parametros	<- temp en
operator+	

Adios, mundo	<- Mostrar()
Destructor (Adios, mundo) [0x3e3e18]	<- Destrucción
de objeto temporal Cadena(C2+", mundo")	
Destructor (, mundo) [0x3e3e08]	<- Destrucción
de objeto temporal de suma	
Destructor (Hola, mundo!) [0x3e3e48]	<- Destrucción
de C3	
Destructor (Adios) [0x3e3df8]	<- Destrucción
de C2	
Destructor (Hola) [0x3e3d90]	<- Destrucción
de C1	

Sin embargo, en otros compiladores (por ejemplo: <http://codepad.org>), es imprescindible crear correctamente el constructor copia, y se usará para pasar los objetos temporales automáticos entre distintos ámbitos de acceso y duración.

El código para el constructor copia podría ser:

```
Cadena::Cadena(const Cadena &c) {
    cadena = new char[strlen(c.cadena)+1];
    strcpy(cadena, c.cadena);
    cout << "Constructor copia (" << cadena << ") [" <<
(void*)cadena << "]" << endl;
}
```

Ejecutar este código en [codepad](http://codepad.org).

La salida de este programa en el compilador de codepad sería:

Constructor (Hola) [0x8051438]	<- C1 en main
Constructor (Adios) [0x8051568]	<- C2 en main
Constructor () [0x8051590]	<- C3 en main
Constructor (, mundo!) [0x80515b0]	<- C1+Cadena(",
mundo"); // Segundo operando en suma en main	
suma-Constructor sin parametros	<- temp en
operator+	
Constructor copia (Hola, mundo!) [0x80515d8]	<- copia para
cambio de ámbito	
Destructor (Hola, mundo!) [0x80514e8]	<- destrucción
de temp	
Asignacion	<- Asignación a

```

C3
Destructor (Hola, mundo!) [0x80515d8]      <- Destrucción
de objeto temporal, una vez asignado
Destructor (, mundo!) [0x80515b0]          <- Destrucción
de objeto temporal de suma
Hola, mundo!                              <- Mostrar()
Constructor (, mundo) [0x80515b0]          <- C2+",
mundo"; // Segundo operando en suma
suma-Constructor sin parametros           <- temp en
operator+
Constructor copia (Adios, mundo) [0x8051608] <- copia para
cambio de ámbito
Destructor (Adios, mundo) [0x80515d8]      <- destrucción
de temp
Adios, mundo                              <- Mostrar()
Destructor (Adios, mundo) [0x8051608]      <- Destrucción
de objeto temporal Cadena(C2+",mundo")
Destructor (, mundo) [0x80515b0]          <- Destrucción
de objeto temporal de suma
Destructor (Hola, mundo!) [0x80514e8]      <- Destrucción
de C3
Destructor (Adios) [0x8051568]             <- Destrucción
de C2
Destructor (Hola) [0x8051438]             <- Destrucción
de C1

```

He resaltado en negrita las diferencias entre un programa y el otro, que corresponden con las creaciones y destrucciones de los objetos temporales necesarios al cambiar de ámbito, que no se hacen en el primer ejemplo.

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 parámetros

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 evaluada la expresión, se aplica el operador. Si nosotros queremos que nuestro operador actúe 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); // Constructor copia

    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 mostrar 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 resultado es que se suman 432 horas, cuando 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 asignar 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, puesto que 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 estructura 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, cuando se sobrecarga este operador no existe esa limitación.

Donde más útil resulta este operador es cuando se usa con estructuras dinámicas de datos: listas y árboles. Pero también puede servirnos para crear *arrays* asociativos, donde los índices sean por ejemplo, palabras.

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.

Nota: Un histograma es un gráfico o una tabla utilizado en la representación de distribuciones de frecuencias de cualquier tipo de información o función. La clase de nuestro ejemplo podría usar los valores de la tabla para generar ese gráfico.

```
#include <iostream>
using namespace std;

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();

return 0;
}
```

Ejecutar este código en [codepad](#).

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 tratara 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.

36 Herencia

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.

Jerarquía, clases base y clases derivadas

Cada nueva clase obtenida mediante herencia 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 este último caso hablaremos de *derivación múltiple*.

Esto nos permite 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.

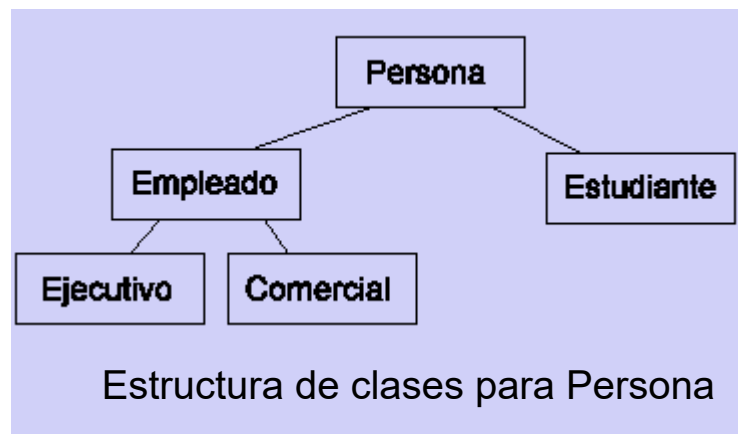
Un ejemplo muy socorrido es de las personas. Supongamos que nuestra clase base para clasificar a las personas en función de su profesión sea "Persona". Presta especial atención a la palabra "**clasificar**", es el punto de partida para buscar la solución de cualquier problema que se pretenda resolver 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 jerarquías 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 considerar 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 este 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] <base1> [, [public|private] <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);
    const char *LeerNombre(char *n) const;
    int LeerEdad() const;
    void CambiarNombre(const 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(const 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 jerárquicas 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 inaccesibles 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>
using namespace std;

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;

    return 0;
}

```

La salida es ésta:

```

Constructor de A
Constructor de B
a = 10, b = 20

```

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>  
using namespace std;  
  
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) { // (1)  
        cout << "Constructor de B" << endl;  
    }  
    int LeerB() const { return datoB; }  
  
protected:  
    int datoB;  
};  
  
int main() {  
    ClaseB objeto(5,15);  
  
    cout << "a = " << objeto.LeerA() << ", b = "  
        << objeto.LeerB() << endl;  
  
    return 0;  
}
```

La salida es esta:

```
Constructor de A  
Constructor de B  
a = 5, b = 15
```

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>  
using namespace std;  
  
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) { // (1)
```

```

        cout << "Constructor de B" << endl;
    }
    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;

    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.

Destrucción 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>
using namespace std;

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;

    return 0;
}
```

La salida es esta:

```
Constructor de A  
Constructor de B  
a = 10, b = 20  
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.

37 Funciones virtuales

Llegamos ahora a los conceptos más sutiles de la programación orientada a objetos.

La *virtualización* de funciones y clases nos permite implementar una de las propiedades más potentes de POO: el polimorfismo.

Pero vayamos con calma...

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>
using namespace std;

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();

    return 0;
}

```

La salida de este programa es:

```

a = 10, b = 20
a = 10

```

Decimos que la definición de la función "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>
using namespace std;

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() {
    ClaseB objeto;

    objeto.Incrementar();
    // objeto.Incrementar(10);
    objeto.ClaseA::Incrementar();
    objeto.ClaseA::Incrementar(10);

    return 0;
}

```

La salida sería:

```

Suma 2
Suma 1
Suma 10

```

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 punteros o con referencias.

C++ nos permite acceder a objetos de una clase derivada usando un puntero a la clase base. En esa capacidad es posible 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 inaccesibles.

Nota:

Podemos usar un microscopio como martillo, (probablemente podremos clavar algunos clavos antes de que se rompa y deje de funcionar como tal). Pero mientras sea un martillo, no nos permitirá observar a través de él. Es decir, los métodos propios de objetos de la clase "microscopio" serán inaccesibles mientras usemos una referencia a "martillo" para manejarlo.

De todos modos, no te procupes por el microscopio, no creo que en general usemos jerarquías de clases en las que objetos valiosos y delicados se deriven de otros tan radicalmente diferentes y toscos.

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>
#include <cstring>
using namespace std;
```

```

class Persona {
public:
    Persona(char *n) { strcpy(nombre, n); }
    void VerNombre() { cout << nombre << endl; }
protected:
    char nombre[30];
};

class Empleado : public Persona {
public:
    Empleado(char *n) : Persona(n) {}
    void VerNombre() {
        cout << "Emp: " << nombre << endl;
    }
};

class Estudiante : public Persona {
public:
    Estudiante(char *n) : Persona(n) {}
    void VerNombre() {
        cout << "Est: " << nombre << endl;
    }
};

int main() {
    Persona *Pepito = new Estudiante("Jose");
    Persona *Carlos = new Empleado("Carlos");

    Carlos->VerNombre();
    Pepito->VerNombre();
    delete Pepito;
    delete Carlos;

    return 0;
}

```

La salida es como ésta:

```

Carlos
Jose

```

Podemos comprobar que se ejecuta la versión de la función "VerNombre" que hemos definido para la clase base, y no la de las clases derivadas.

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 void VerNombre() {
        cout << nombre << endl;
    }
protected:
    char nombre[30];
};
```

Si ejecutemos el programa de nuevo, veremos que la salida es ahora diferente:


```
Emp: Carlos  
Est: Jose
```

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".

Nota:

Volvamos al ejemplo del microscopio usado como martillo. Supongamos que quien diseñó el microscopio se ha dado cuenta de que a menudo se usan esos aparatos como martillos, y decide protegerlos, añadiendo una funcionalidad específica (y, por supuesto, virtual) que permite clavar clavos sin que el microscopio se deteriore.

Cada vez que alguien use un microscopio para clavar un clavo, se activará esa función, y el clavo quedará bien clavado, al mismo tiempo que el microscopio queda intacto.

Por muy bruto que sea el que use el microscopio, nunca podrá acceder a la función "Clavar" de la clase base "martillo", ya que es virtual, y por lo tanto el microscopio sigue protegido.

Pero vayamos más lejos. La función virtual en la clase derivada no tiene por qué hacer lo mismo que en la clase base. Así, nuestro diseñador, puede diseñar la función "clavar" de modo que de una descarga de alta tensión al que la maneje, o siendo más civilizado, que emita un mensaje de error. :)

Lástima que esto no sea posible en la vida real...

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.

Veamos un ejemplo con referencias:

```
#include <iostream>
#include <cstring>
using namespace std;

class Persona {
public:
    Persona(const char *n) { strcpy(nombre, n); }
    virtual void VerNombre() {
        cout << nombre << endl;
    }
protected:
    char nombre[30];
};

class Empleado : public Persona {
public:
    Empleado(const char *n) : Persona(n) {}
    void VerNombre() {
        cout << "Emp: " << nombre << endl;
    }
};

class Estudiante : public Persona {
public:
    Estudiante(const char *n) : Persona(n) {}
    void VerNombre() {
        cout << "Est: " << nombre << endl;
    }
};

int main() {
    Estudiante Pepito("Jose");
    Empleado Carlos("Carlos");
    Persona &rPepito = Pepito; // Referencia como Persona
    Persona &rCarlos = Carlos; // Referencia como Persona

    rCarlos.VerNombre();
    rPepito.VerNombre();
}
```

```
    return 0;  
}
```

Ejecutar este código en [codepad](#).

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 este inconveniente se suele crear una función virtual "clonar" en la clase base que se superpondrá para cada clase derivada.

Por ejemplo:

```
#include <iostream>  
#include <cstring>  
using namespace std;  
  
class Persona {
```

```

public:
    Persona(const char *n) { strcpy(nombre, n); }
    Persona(const Persona &p);
    virtual void VerNombre() {
        cout << nombre << endl;
    }
    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(const char *n) : Persona(n) {}
    Empleado(const Empleado &e);
    void VerNombre() {
        cout << "Emp: " << nombre << endl;
    }
    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(const char *n) : Persona(n) {}
    Estudiante(const Estudiante &e);
    void VerNombre() {
        cout << "Est: " << nombre << endl;
    }
    virtual Persona* Clonar() {
        return new Estudiante(*this);
    }
};

Estudiante::Estudiante(const Estudiante &e) : Persona(e) {
    cout << "Est: constructor copia." << endl;
}

int main() {
    Persona *Pepito = new Estudiante("Jose");

```

```

Persona *Carlos = new Empleado("Carlos");
Persona *Gente[2];

Carlos->VerNombre();
Pepito->VerNombre();

Gente[0] = Carlos->Clonar();
Gente[0]->VerNombre();

Gente[1] = Pepito->Clonar();
Gente[1]->VerNombre();

delete Pepito;
delete Carlos;
delete Gente[0];
delete Gente[1];

return 0;
}

```

Ejecutar este código en [codepad](#).

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

```

Este método asegura que siempre se llama al constructor copia adecuado, ya que se hace desde una función virtual.

Nota:

Como puedes ver, C++ lleva algunos años de adelanto sobre la ingeniería genética, y ya ha resuelto el problema de clonar personas. :-).

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.

Palabras reservadas usadas en este capítulo

virtual.

38 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 clases 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>  
using namespace std;  
  
class ClaseA {  
public:  
    ClaseA() : valorA(10) {}  
    int LeerValor() const { return valorA; }  
protected:  
    int valorA;  
};  
  
class ClaseB {  
public:  
    ClaseB() : valorB(20) {}  
    int LeerValor() const { return valorB; }  
protected:  
    int valorB;
```

```

};

class ClaseC : public ClaseA, public ClaseB {};

int main() {
    ClaseC CC;

    // cout << CC.LeerValor() << endl;
    // Produce error de compilación por ambigüedad.
    cout << CC.ClaseA::LeerValor() << endl;

    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>
using namespace std;

class ClaseA {
public:
    ClaseA() : valorA(10) {}
    int LeerValor() const { return valorA; }
protected:
    int valorA;
};

class ClaseB {
public:
    ClaseB() : valorB(20) {}
    int LeerValor() const { return valorB; }
protected:
    int valorB;
};

class ClaseC : public ClaseA, public ClaseB {
public:
    int LeerValor() const { return ClaseA::LeerValor(); }
};

```



```

int main() {
    ClaseC CC;

    cout << CC.LeerValor() << endl;

    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>
using namespace std;

class ClaseA {
public:
    ClaseA() : valorA(10) {}
    ClaseA(int va) : valorA(va) {}
    int LeerValor() const { return valorA; }
protected:
    int valorA;
};

class ClaseB {
public:
    ClaseB() : valorB(20) {}
    ClaseB(int vb) : valorB(vb) {}
    int LeerValor() const { return valorB; }
protected:
    int valorB;
};

class ClaseC : public ClaseA, public ClaseB {
public:
    ClaseC(int va, int vb) : ClaseA(va), ClaseB(vb) {};
}

```

```

        int LeerValorA() const { return ClaseA::LeerValor(); }
        int LeerValorB() const { return ClaseB::LeerValor(); }
    };

    int main() {
        ClaseC CC(12,14);

        cout << CC.LeerValorA() << ", "
              << CC.LeerValorB() << endl;

        return 0;
    }

```

Sintaxis:

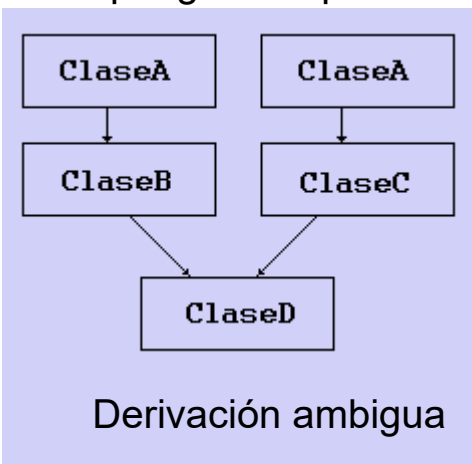
```

<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 {};

```

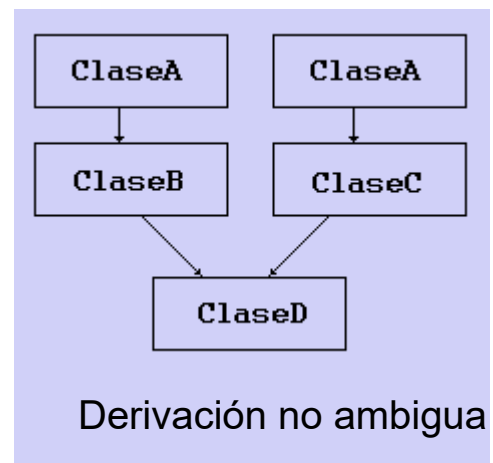
Desde el punto de vista de la ClaseB, no hay ninguna diferencia entre ésta declaración y la que hemos usado hasta ahora. La diferencia estará cuando declaramos la ClaseD. Veamos el ejemplo completo:

```
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 creamos una estructura de este 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>  
#include <cstring>  
using namespace std;  
  
class Persona {  
public:  
    Persona(char *n) { strcpy(nombre, n); }  
    const char *LeeNombre() const { return nombre; }  
protected:  
    char nombre[30];  
};
```

```

class Empleado : virtual public Persona {
public:
    Empleado(char *n, int s) : Persona(n), salario(s) {}
    int LeeSalario() const { 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() const { 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);

    cout << Fulanito.LeeNombre() << ", "
        << Fulanito.LeeSalario() << ", "
        << Fulanito.LeeNota() << endl;

    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 definir las 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>
#include <cstring>
using namespace std;

class Persona {
public:
    Persona(char *n) { strcpy(nombre, n); }
    virtual void Mostrar() const = 0;
protected:
    char nombre[30];
};

class Empleado : public Persona {
public:
    Empleado(char *n, int s) : Persona(n), salario(s) {}
    void Mostrar() const;
    int LeeSalario() const { return salario; }
    void ModificaSalario(int s) { salario = s; }
protected:
    int salario;
};

void Empleado::Mostrar() const {
    cout << "Empleado: " << nombre
         << ", Salario: " << salario
         << endl;
}

class Estudiante : public Persona {
public:
    Estudiante(char *n, float no) : Persona(n), nota(no) {}
    void Mostrar() const;
    float LeeNota() const { return nota; }
    void ModificaNota(float no) { nota = no; }
protected:
    float nota;
};

void Estudiante::Mostrar() const {
    cout << "Estudiante: " << nombre
         << ", Nota: " << nota << endl;
}

int main() {
    Persona *Pepito = new Empleado("Jose", 1000); // (1)
    Persona *Pablito = new Estudiante("Pablo", 7.56);
}

```

```
Pepito->Mostrar();  
Pablito->Mostrar();  
  
delete Pepito;  
delete Pablito;  
  
return 0;  
}
```

La salida será así:

```
Empleado: Jose, Salario: 1000  
Estudiante: Pablo, Nota: 7.56
```

En este 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.

39 Trabajar con ficheros

Usar streams facilita mucho el acceso a ficheros en disco, veremos que una vez que creamos un stream para un fichero, podremos trabajar con él igual que lo hacemos con *cin* o *cout*.

Mediante las clases *ofstream*, *ifstream* y *fstream* tendremos acceso a todas las funciones de las clases base de las que se derivan estas: *ios*, *istream*, *ostream*, *fstreambase*, y como también contienen un objeto *filebuf*, podremos acceder a las funciones de *filebuf* y *streambuf*.

En {cc:904#inicio:apendice E} hay una referencia bastante completa de las clases estándar de entrada y salida.

Evidentemente, muchas de estas funciones puede que nunca nos sean de utilidad, pero algunas de ellas se usan con frecuencia, y facilitan mucho el trabajo con ficheros.

Crear un fichero de salida, abrir un fichero de entrada

Empezaremos con algo sencillo. Vamos a crear un fichero mediante un objeto de la clase *ofstream*, y posteriormente lo leeremos mediante un objeto de la clase *ifstream*:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    char cadena[128];
    // Crea un fichero de salida
    ofstream fs("nombre.txt");
```



```

// Enviamos una cadena al fichero de salida:
fs << "Hola, mundo" << endl;
// Cerrar el fichero,
// para luego poder abrirlo para lectura:
fs.close();

// Abre un fichero de entrada
ifstream fe("nombre.txt");

// Leeremos mediante getline, si lo hiciéramos
// mediante el operador << sólo leeríamos
// parte de la cadena:
fe.getline(cadena, 128);

cout << cadena << endl;

return 0;
}

```

Este sencillo ejemplo crea un fichero de texto y después visualiza su contenido en pantalla.

Veamos otro ejemplo sencillo, para ilustrar algunas *limitaciones* del operador >> para hacer lecturas, cuando no queremos perder caracteres.

Supongamos que llamamos a este programa "streams.cpp", y que pretendemos que se autoimprima en pantalla:

```

#include <iostream>
#include <fstream>
using namespace std;

int main() {
    char cadena[128];
    ifstream fe("streams.cpp");

    while(!fe.eof()) {
        fe >> cadena;
        cout << cadena << endl;
    }
    fe.close();
}

```

```
    return 0;  
}
```

El resultado quizá no sea el esperado. El motivo es que el operador >> interpreta los espacios, tabuladores y retornos de línea como separadores, y los elimina de la cadena de entrada.

Ficheros binarios

Muchos sistemas operativos distinguen entre ficheros de texto y ficheros binarios. Por ejemplo, en MS-DOS, los ficheros de texto sólo permiten almacenar caracteres.

En otros sistemas no existe tal distinción, todos los ficheros son binarios. En esencia esto es más correcto, puesto que un fichero de texto es un fichero binario con un rango limitado para los valores que puede almacenar.

En general, usaremos ficheros de texto para almacenar información que pueda o deba ser manipulada con un editor de texto. Un ejemplo es un fichero fuente C++. Los ficheros binarios son más útiles para guardar información cuyos valores no estén limitados. Por ejemplo, para almacenar imágenes, o bases de datos. Un fichero binario permite almacenar estructuras completas, en las que se mezclen datos de cadenas con datos numéricos.

En realidad no hay nada que nos impida almacenar cualquier valor en un fichero de texto, el problema surge cuando se almacena el valor que el sistema operativo usa para marcar el fin de fichero en un archivo de texto. En MS-DOS ese valor es 0x1A. Si abrimos un fichero en modo de texto que contenga un dato con ese valor, no nos será posible leer ningún dato a partir de esa posición. Si lo abrimos en modo binario, ese problema no existirá.

Los ficheros que hemos usado en los ejemplos anteriores son en modo texto, veremos ahora un ejemplo en modo binario:

```

#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

struct tipoRegistro {
    char nombre[32];
    int edad;
    float altura;
};

int main() {
    tipoRegistro pepe;
    tipoRegistro pepe2;
    ofstream fsalida("prueba.dat",
        ios::out | ios::binary);

    strcpy(pepe.nombre, "Jose Luis");
    pepe.edad = 32;
    pepe.altura = 1.78;

    fsalida.write(reinterpret_cast<char *>(&pepe),
        sizeof(tipoRegistro));
    fsalida.close();

    ifstream fentrada("prueba.dat",
        ios::in | ios::binary);

    fentrada.read(reinterpret_cast<char *>(&pepe2),
        sizeof(tipoRegistro));
    cout << pepe2.nombre << endl;
    cout << pepe2.edad << endl;
    cout << pepe2.altura << endl;
    fentrada.close();

    return 0;
}

```

Al declarar streams de las clases ofstream o ifstream y abrirlos en modo binario, tenemos que añadir el valor *ios::out* e *ios::in*, respectivamente, al valor *ios::binary*. Esto es necesario porque los valores por defecto para el modo son *ios::out* e *ios::in*, también

respectivamente, pero al añadir el flag *ios::binary*, el valor por defecto no se tiene en cuenta.

Cuando trabajemos con streams binarios usaremos las funciones *write* y *read*. En este caso nos permiten escribir y leer estructuras completas.

En general, cuando usemos estas funciones necesitaremos hacer un casting, es recomendable usar el operador *reinterpret_cast*.

Ficheros de acceso aleatorio

Hasta ahora sólo hemos trabajado con los ficheros secuencialmente, es decir, empezamos a leer o a escribir desde el principio, y avanzamos a medida que leemos o escribimos en ellos.

Otra característica importante de los ficheros es la posibilidad de trabajar con ellos haciendo acceso aleatorio, es decir, poder hacer lecturas o escrituras en cualquier punto del fichero. Para eso disponemos de las funciones *seekp* y *seekg*, que permiten cambiar la posición del fichero en la que se hará la siguiente escritura o lectura. La 'p' es de *put* y la 'g' de *get*, es decir escritura y lectura, respectivamente.

Otro par de funciones relacionadas con el acceso aleatorio son *tellp* y *tellg*, que sirven para saber en qué posición del fichero nos encontramos.

```
#include <fstream>
using namespace std;

int main() {
    int i;
    char mes[][20] = {"Enero", "Febrero", "Marzo",
                     "Abril", "Mayo", "Junio", "Julio", "Agosto",
                     "Septiembre", "Octubre", "Noviembre",
                     "Diciembre"};
    char cad[20];
```

```

ofstream fsalida("meses.dat",
    ios::out | ios::binary);

// Crear fichero con los nombres de los meses:
cout << "Crear archivo de nombres de meses:" << endl;
for(i = 0; i < 12; i++)
    fsalida.write(mes[i], 20);
fsalida.close();

ifstream fentrada("meses.dat", ios::in | ios::binary);

// Acceso secuencial:
cout << "\nAcceso secuencial:" << endl;
fentrada.read(cad, 20);
do {
    cout << cad << endl;
    fentrada.read(cad, 20);
} while(!fentrada.eof());

fentrada.clear();
// Acceso aleatorio:
cout << "\nAcceso aleatorio:" << endl;
for(i = 11; i >= 0; i--) {
    fentrada.seekg(20*i, ios::beg);
    fentrada.read(cad, 20);
    cout << cad << endl;
}

// Calcular el número de elementos
// almacenados en un fichero:
// ir al final del fichero
fentrada.seekg(0, ios::end);
// leer la posición actual
pos = fentrada.tellg();
// El número de registros es el tamaño en
// bytes dividido entre el tamaño del registro:
cout << "\nNúmero de registros: " << pos/20 << endl;
fentrada.close();

return 0;
}

```

La función **seekg** nos permite acceder a cualquier punto del fichero, no tiene por qué ser exactamente al principio de un registro, la resolución de la funciones seek es de un byte.

Cuando trabajemos con nuestros propios streams para nuestras clases, derivándolas de *ifstream*, *ofstream* o *fstream*, es posible que nos convenga sobrecargar las funciones *seek* y *tell* para que trabajen a nivel de registro, en lugar de hacerlo a nivel de byte.

La función *seekp* nos permite sobrescribir o modificar registros en un fichero de acceso aleatorio de salida. La función *tellp* es análoga a *tellg*, pero para ficheros de salida.

Ficheros de entrada y salida

Ahora veremos cómo podemos trabajar con un stream simultáneamente en entrada y salida.

Para eso usaremos la clase *fstream*, que al ser derivada de *ifstream* y *ofstream*, dispone de todas las funciones necesarias para realizar cualquier operación de entrada o salida.

Hay que tener la precaución de usar la opción *ios::trunc* de modo que el fichero sea creado si no existe previamente.

```
#include <fstream>
using namespace std;

int main() {
    char l;
    long i, lon;
    fstream fich("prueba.dat", ios::in |
        ios::out | ios::trunc | ios::binary);

    fich << "abracadabra" << flush;

    fich.seekg(0L, ios::end);
    lon = fich.tellg();
    for(i = 0L; i < lon; i++) {
        fich.seekg(i, ios::beg);
        fich.get(l);
        if(l == 'a') {
            fich.seekp(i, ios::beg);
            fich << 'e';
        }
    }
}
```

```

    cout << "Salida:" << endl;
    fich.seekg(0L, ios::beg);
    for(i = 0L; i < lon; i++) {
        fich.get(l);
        cout << l;
    }
    cout << endl;
    fich.close();

    return 0;
}

```

Este programa crea un fichero con una palabra, a continuación lee todo el fichero e cambia todos los caracteres 'a' por 'e'. Finalmente muestra el resultado.

Básicamente muestra cómo trabajar con ficheros simultáneamente en entrada y salida.

Sobrecarga de operadores << y >>

Una de las principales ventajas de trabajar con streams es que nos permiten sobrecargar los operadores << y >> para realizar salidas y entradas de nuestros propios tipos de datos.

Por ejemplo, tenemos una clase:

```

#include <iostream>
#include <cstring>
using namespace std;

class Registro {
public:
    Registro(char *, int, char *);
    const char* LeeNombre() const {return nombre;}
    int LeeEdad() const {return edad;}
    const char* LeeTelefono() const {return telefono;}

private:
    char nombre[64];
    int edad;
    char telefono[10];
}

```

```
};

Registro::Registro(char *n, int e, char *t) : edad(e) {
    strcpy(nombre, n);
    strcpy(telefono, t);
}

ostream& operator<<(ostream &os, Registro& reg) {
    os << "Nombre: " << reg.LeeNombre() << "\nEdad: " <<
        reg.LeeEdad() << "\nTelefono: " << reg.LeeTelefono();

    return os;
}

int main() {
    Registro Pepe((char*)"José", 32, (char*)"61545552");

    cout << Pepe << endl;

    return 0;
}
```

Comprobar estado de un stream

Hay varios flags de estado que podemos usar para comprobar el estado en que se encuentra un stream.

Concretamente nos puede interesar si hemos alcanzado el fin de fichero, o si el stream con el que estamos trabajando está en un estado de error.

La función principal para esto es *good()*, de la clase *ios*.

Después de ciertas operaciones con streams, a menudo no es mala idea comprobar el estado en que ha quedado el stream. Hay que tener en cuenta que ciertos estados de error impiden que se puedan seguir realizando operaciones de entrada y salida.

Otras funciones útiles son *fail()*, *eof()*, *bad()*, *rdstate()* o *clear()*.

En el ejemplo de archivos de acceso aleatorio hemos usado *clear()* para eliminar el bit de estado *eofbit* del fichero de entrada, si no hacemos eso, las siguientes operaciones de lectura fallarían.

Otra condición que conviene verificar es la existencia de un fichero. En los ejemplos anteriores no ha sido necesario, aunque hubiera sido conveniente, verificar la existencia, ya que el propio ejemplo crea el fichero que después lee.

Cuando vayamos a leer un fichero que no podamos estar seguros de que existe, o que aunque exista pueda estar abierto por otro programa, debemos asegurarnos de que nuestro programa tiene acceso al stream. Por ejemplo:

```
#include <fstream>
using namespace std;

int main() {
    char mes[20];
    ifstream fich("meses1.dat", ios::in | ios::binary);

    // El fichero meses1.dat no existe, este programa es
    // una prueba de los bits de estado.

    if(fich.good()) {
        fich.read(mes, 20);
        cout << mes << endl;
    }
    else {
        cout << "Fichero no disponible" << endl;
        if(fich.fail()) cout << "Bit fail activo" << endl;
        if(fich.eof()) cout << "Bit eof activo" << endl;
        if(fich.bad()) cout << "Bit bad activo" << endl;
    }
    fich.close();

    return 0;
}
```

Ejemplo de fichero previamente abierto:

```
#include <fstream>
using namespace std;

int main() {
```

```
char mes[20];
ofstream fich1("meses.dat", ios::out | ios::binary);
ifstream fich("meses.dat", ios::in | ios::binary);

// El fichero meses.dat existe, pero este programa
// intenta abrir dos streams al mismo fichero, uno en
// escritura y otro en lectura. Eso no es posible, se
// trata de una prueba de los bits de estado.

fich.read(mes, 20);
if(fich.good())
    cout << mes << endl;
else {
    cout << "Error al leer de Fichero" << endl;
    if(fich.fail()) cout << "Bit fail activo" << endl;
    if(fich.eof())  cout << "Bit eof activo" << endl;
    if(fich.bad())  cout << "Bit bad activo" << endl;
}
fich.close();
fich1.close();

return 0;
}
```

40 Plantillas

Según va aumentando la complejidad de nuestros programas y sobre todo, de los problemas a los que nos enfrentamos, descubrimos que tenemos que repetir una y otra vez las mismas estructuras.

Por ejemplo, a menudo tendremos que implementar arrays dinámicos para diferentes tipos de objetos, o listas dinámicas, pilas, colas, árboles, etc.

El código es similar siempre, pero estamos obligados a rescribir ciertas funciones que dependen del tipo o de la clase del objeto que se almacena.

Las plantillas (templates) nos permiten parametrizar estas clases para adaptarlas a cualquier tipo de dato.

Vamos a desarrollar un ejemplo sencillo de un array que pueda almacenar cualquier objeto. Aunque más adelante veremos que se presentan algunas limitaciones y veremos cómo solucionarlas.

Con lo que ya sabemos, podemos crear fácilmente una clase que encapsule un array de, por ejemplo, enteros. Veamos el código de esta clase:

```
// TablaInt.cpp: Clase para crear Tablas de enteros
// C con Clase: Marzo de 2002

#include <iostream>
using namespace std;

class TablaInt {
public:
    TablaInt(int nElem);
    ~TablaInt();
    int& operator[](int indice) { return pInt[indice]; }

private:
```

```

    int *pInt;
    int nElementos;
};

// Definición:
TablaInt::TablaInt(int nElem) : nElementos(nElem) {
    pInt = new int[nElementos];
}

TablaInt::~~TablaInt() {
    delete[] pInt;
}

int main() {
    TablaInt TablaI(10);

    for(int i = 0; i < 10; i++)
        TablaI[i] = 10-i;

    for(int i = 0; i < 10; i++)
        cout << TablaI[i] << endl;

    return 0;
}

```

Bien, la clase TablaInt nos permite crear arrays de la dimensión que queramos, para almacenar enteros. Quizás pienses que para eso no hace falta una clase, ya que podríamos haber declarado sencillamente:

```
int TablaI[10];
```

Bueno, tal vez tengas razón, pero para empezar, esto es un ejemplo sencillo. Además, la clase TablaInt nos permite hacer cosas como esta:

```
int elementos = 24;
TablaInt TablaI(elementos);
```

Recordarás que no está permitido usar variables para indicar el tamaño de un array. Pero no sólo eso, en realidad esta podría ser una primera aproximación a una clase `TablaInt` que nos permitiría aumentar el número elementos o disminuirlo durante la ejecución, definir constructores copia, o sobrecargar operadores suma, resta, etc.

La clase para `Tabla` podría ser mucho más potente de lo que puede ser un array normal, pero dejaremos eso para otra ocasión.

Supongamos que ya tenemos esa maravillosa clase definida para enteros. ¿Qué pasa si ahora necesitamos definir esa clase para números en coma flotante?. Podemos cortar y pegar la definición y sustituir todas las referencias a `int` por `float`. Pero, ¿y si también necesitamos esta estructura para cadenas, complejos, o para la clase persona que implementamos en anteriores capítulos?, ¿haremos una versión para cada tipo para el que necesitemos una `Tabla` de estas características?.

Afortunadamente existen las plantillas y (aunque al principio no lo parezca), esto nos hace la vida más fácil.

Sintaxis

C++ permite crear plantillas de funciones y plantillas de clases.

La sintaxis para declarar una plantilla de función es parecida a la de cualquier otra función, pero se añade al principio una presentación de la clase que se usará como referencia en la plantilla:

```
template <class|typename <id>[,...]>
<tipo_retorno> <identificador>(<lista_de_parámetros>)
{
    // Declaración de función
};
```

La sintaxis para declarar una plantilla de clase es parecida a la de cualquier otra clase, pero se añade al principio una presentación

de la clase que se usará como referencia en la plantilla:

```
template <class|typename <id>[,...]>
class <identificador_de_plantilla>
{
    // Declaración de funciones
    // y datos miembro de la plantilla
};
```

Nota:

La lista de clases que se incluye a continuación de la palabra reservada `template` se escriben entre las llaves "<" y ">", en este caso esos símbolos no indican que se debe introducir un literal, sino que deben escribirse, no me es posible mostrar estos símbolos en negrita, por eso los escribo en rojo. Siento si esto causa algún malentendido.

Pero seguro que se ve mejor con un ejemplo:

```
template <class T1>
class Tabla {
public:
    Tabla();
    ~Tabla();
    ...
};
```

Del mismo modo, cuando tengamos que definir una función miembro fuera de la declaración de la clase, tendremos que incluir la parte del `template` y como nombre de la clase incluir la plantilla antes del operador de ámbito (::). Por ejemplo:

```
template <class T1>
```

```
Tabla<T1>::Tabla() {  
    // Definición del constructor  
}
```

Plantillas de funciones

Un ejemplo de plantilla de función puede ser esta que sustituye a la versión macro de max:

```
template <class T>  
T max(T x, T y) {  
    return (x > y) ? x : y;  
};
```

La ventaja de la versión en plantilla sobre la versión macro se manifiesta en cuanto a la seguridad de tipos. Por supuesto, podemos usar argumentos de cualquier tipo, no han de ser necesariamente clases. Pero cosas como estas darán error de compilación:

```
int a=2;  
char b='j';  
  
int c=max(a,b);
```

El motivo es que a y b no son del mismo tipo. Aquí no hay promoción implícita de tipos. Sin embargo, están permitidas todas las combinaciones en las que los argumentos sean del mismo tipo o clase, siempre y cuando que el operador > esté implementado para esa clase.

```
int a=3, b=5, c;  
char f='a', g='k', h;
```

```
c = max(a,b);  
h = max(f,g);
```

Plantilla para Tabla

Ahora estamos en disposición de crear una plantilla a partir de la clase Tabla que hemos definido antes. Esta vez podremos usar esa plantilla para definir Tablas de cualquier tipo de objeto.

```
template <class T>  
class Tabla {  
public:  
    Tabla(int nElem);  
    ~Tabla();  
    T& operator[](int indice) { return pT[indice]; }  
  
private:  
    T *pT;  
    int nElementos;  
};  
  
// Definición:  
template <class T>  
Tabla<T>::Tabla(int nElem) : nElementos(nElem) {  
    pT = new T[nElementos];  
}  
  
template <class T>  
Tabla<T>::~~Tabla() {  
    delete[] pT;  
}
```

Dentro de la declaración y definición de la plantilla, podremos usar los parámetros que hemos especificado en la lista de parámetros del template como si se tratase de comodines. Más adelante, cuando creamos instancias de la plantilla para diferentes tipos, el compilador sustituirá esos comodines por los tipos que especifiquemos.

Y ya sólo nos queda por saber cómo declarar Tablas del tipo que queramos. La sintaxis es:

```
<identificador_de_plantilla><<tipo/clase>>  
<identificador/constructor>;
```

Seguro que se ve mejor con un ejemplo, veamos como declarar Tablas de enteros, punto flotante o valores booleanos:

```
Tabla<int>    TablaInt(32);    // Tabla de 32 enteros  
Tabla<float> TablaFloat(12);  // Tabla de 12 floats  
Tabla<bool>   TablaBool(10);  // Tabla de 10 bools
```

Pero no es este el único modo de proceder. Las plantillas admiten varios parámetros, de modo que también podríamos haber especificado el número de elementos como un segundo parámetro de la plantilla:

```
template <class T, int nElementos>  
class Tabla {  
    public:  
        Tabla();  
        ~Tabla();  
        T& operator[](int indice) { return pT[indice]; }  
  
    private:  
        T *pT;  
};  
  
// Definición:  
template <class T, int nElementos>  
Tabla<T,nElementos>::Tabla() {  
    pT = new T[nElementos];  
}  
  
template <class T, int nElementos>  
Tabla<T, nElementos>::~~Tabla() {  
    delete[] pT;  
}
```

La declaración de tablas con esta versión difiere ligeramente:

```
Tabla<int,32>    TablaInt;    // Tabla de 32 enteros
Tabla<float,12> TablaFloat;  // Tabla de 12 floats
Tabla<bool,10>  TablaBool;   // Tabla de 10 bools
```

Esta forma tiene una limitación: el argumento `nElementos` debe ser una constante, nunca una variable. Esto es porque el valor debe conocerse durante la compilación del programa. Las plantillas no son definiciones de clases, sino plantillas que se usan para generar las clases que a su vez se compilarán para crear el programa:

```
#define N 12
...
const n = 10;
int i = 23;

Tabla<int,N>      TablaInt;    // Legal, Tabla de 12 enteros
Tabla<float,3*n> TablaFloat;  // Legal, Tabla de 30 floats
Tabla<char,i>     TablaFloat; // Ilegal
```

Ficheros de cabecera

En los siguientes ejemplos usaremos varios ficheros fuente. Más concretamente, crearemos algunos ficheros para definir plantillas que usaremos en programas de ejemplo.

Dado que algunas de esas plantillas se podrán usar en varios programas diferentes, es buena idea definir cada una en un fichero separado, que se pueda incluir desde otros ficheros fuente para hacer uso de esas plantillas.

Generalmente, un fichero de cabecera suele contener sólo declaraciones, de clases, tipos y/o funciones. En el caso de

plantillas, también se incluye el código, pero esto es sólo en apariencia.

Es un error ver la implementación de las plantillas como código, ya que el compilador usa las plantillas para generar código que posteriormente compila, pero el código de una plantilla no es compilable. En realidad es sólo un molde para crear clases, y esas son las clases que se pueden compilar.

De modo que el código de una plantilla se suele escribir en el mismo fichero de cabecera que el resto de la declaración de plantilla.

Esta filosofía, la de crear ficheros de cabecera, es la que se debe seguir para reutilizar el código. Si hemos diseñado una clase o la plantilla de una clase que nos puede ser útil en otros proyectos, es buena idea convertirla en un fichero de cabecera, y llegado el caso, en una biblioteca externa.

Hay una precaución que se debe tomar siempre cuando se escriben ficheros de cabecera.

Debido a que el mismo fichero de cabecera puede ser incluido desde varios ficheros fuente de un mismo proyecto, es imprescindible diseñar un mecanismo que evite que se declaren las mismas clases, tipos o funciones varias veces. Esto provocaría errores de compilación, y harían imposible compilar el proyecto completo.

Esto es sencillo, nos basamos en si una macro está o no definida, y haremos una compilación condicional del código en función de que la macro esté o no definida. Es importante elegir un nombre único para la macro. Lo normal es crear un identificador en función del nombre del fichero de cabecera.

Por ejemplo, si tenemos un fichero llamado "ccadena.h", el identificador de la macro podría ser CCADENA_H, o _CCADENA_H, o CCADENA, etc.

El fichero tendría esta estructura:

```
// Zona de comentarios, fechas, versiones, autor y propósito
del fichero
#ifndef CCADENA
#define CCADENA
// Zona de código
#endif
```

De este modo, la primera vez que se incluye este fichero la macro CCADENA no está definida. Lo primero que hacemos es definirla, y después incluimos el código del fichero de cabecera.

En posteriores inclusiones la macro sí está definida, de modo que no se incluye nada de código, y no necesitamos definir la macro, ya que ya lo estaba.

Ejemplo de uso de plantilla Tabla

Vamos a ver un ejemplo completo de cómo aplicar la plantilla anterior a diferentes tipos.

Fichero de cabecera que declara y define la plantilla Tabla:

```
// Tabla.h: definición de la plantilla tabla:
// C con Clase: Marzo de 2002

#ifndef T_TABLA
#define T_TABLA

template <class T>
class Tabla {
public:
    Tabla(int nElem);
    ~Tabla();
    T& operator[](int indice) { return pT[indice]; }
    int NElementos() const { return nElementos; }

private:
    T *pT;
    int nElementos;
};
```

```
// Definición:
template <class T>
Tabla<T>::Tabla(int nElem) : nElementos(nElem) {
    pT = new T[nElementos];
}

template <class T>
Tabla<T>::~~Tabla() {
    delete[] pT;
}
#endif
```

Fichero de aplicación de plantilla:

```
// Tabla.cpp: ejemplo de Tabla
// C con Clase: Marzo de 2002

#include <iostream>
#include "Tabla.h"

using namespace std;

const int nElementos = 10;

int main() {
    Tabla<int> TablaInt(nElementos);
    Tabla<float> TablaFloat(nElementos);

    for(int i = 0; i < nElementos; i++)
        TablaInt[i] = nElementos-i;

    for(int i = 0; i < nElementos; i++)
        TablaFloat[i] = 1/(1+i);

    for(int i = 0; i < nElementos; i++) {
        cout << "TablaInt[" << i << "] = "
              << TablaInt[i] << endl;
        cout << "TablaFloat[" << i << "] = "
              << TablaFloat[i] << endl;
    }

    return 0;
}
```

Posibles problemas

Ahora bien, supongamos que quieres usar la plantilla Tabla para crear una tabla de cadenas. Lo primero que se nos ocurre hacer probablemente sea:

```
Tabla<char*> TablaCad(15);
```

No hay nada que objetar, todo funciona, el programa compila, y no hay ningún error, pero... es probable que no funcione como esperas. Veamos otro ejemplo:

Fichero de aplicación de plantilla:

```
// Tablacad.cpp: ejemplo de Tabla con cadenas
// C con Clase: Marzo de 2002

#include <iostream>
#include <cstring>
#include <cstdio>
#include "Tabla.h"

using namespace std;

const int nElementos = 5;

int main() {
    Tabla<char *> TablaCad(nElementos);
    char cadena[20];

    for(int i = 0; i < nElementos; i++) {
        sprintf(cadena, "Numero: %5d", i);
        TablaCad[i] = cadena;
    }

    strcpy(cadena, "Modificada");

    for(int i = 0; i < nElementos; i++)
        cout << "TablaCad[" << i << "] = "
              << TablaCad[i] << endl;
```

```
    return 0;  
}
```

Si has compilado el programa y has visto la salida, tal vez te sorprenda algo el resultado: Efectivamente, parece que nuestra tabla no es capaz de almacenar cadenas, o al menos no más de una cadena. La cosa sería aún más grave si la cadena auxiliar fuera liberada, por ejemplo porque se tratase de una variable local de una función, o porque se tratase de memoria dinámica.

```
TablaCad[0] = Modificada  
TablaCad[1] = Modificada  
TablaCad[2] = Modificada  
TablaCad[3] = Modificada  
TablaCad[4] = Modificada
```

¿Cuál es el problema?

Lo que pasa es que nuestra tabla no es de cadenas, sino de punteros a char. De hecho eso es lo que hemos escrito `Tabla<char*>`, por lo tanto, no hay nada sorprendente en el resultado. Pero esto nos plantea un problema: ¿cómo nos las apañamos para crear una tabla de cadenas?

Tablas de cadenas

La solución es usar una clase que encapsule las cadenas y crear una tabla de objetos de esa clase.

Veamos una clase básica para manejar cadenas:

```
// CCadena.h: Fichero de cabecera de definición de cadenas  
// C con Clase: Marzo de 2002  
  
#ifndef CCADENA  
#define CCADENA  
#include <cstring>
```

```

using std::strcpy;
using std::strlen;

class Cadena {
public:
    Cadena(const char *cad) {
        cadena = new char[strlen(cad)+1];
        strcpy(cadena, cad);
    }
    Cadena() : cadena(NULL) {}
    Cadena(const Cadena &c) : cadena(NULL) {*this = c;}
    ~Cadena() { if(cadena) delete[] cadena; }
    Cadena &operator=(const Cadena &c) {
        if(this != &c) {
            if(cadena) delete[] cadena;
            if(c.cadena) {
                cadena = new char[strlen(c.cadena)+1];
                strcpy(cadena, c.cadena);
            }
            else cadena = NULL;
        }
        return *this;
    }
    const char* Lee() const {return cadena;}

private:
    char *cadena;
};

std::ostream& operator<<(std::ostream &os, const Cadena&
cad) {
    os << cad.Lee();
    return os;
}
#endif

```

Usando esta clase para cadenas podemos crear una tabla de cadenas usando nuestra plantilla:

```

// Tabla.cpp: ejemplo de Tabla de cadenas
// C con Clase: Marzo de 2002

#include <iostream>
#include <cstdio>

```



```

#include "Tabla.h"
#include "CCadena.h"

using namespace std;

const int nElementos = 5;

int main() {
    Tabla<Cadena> TablaCad(nElementos);
    char cadena[20];

    for(int i = 0; i < nElementos; i++) {
        sprintf(cadena, "Numero: %2d", i);
        TablaCad[i] = cadena;
    }

    strcpy(cadena, "Modificada");

    for(int i = 0; i < nElementos; i++)
        cout << "TablaCad[" << i << "] = "
              << TablaCad[i] << endl;

    return 0;
}

```

Ejecutar este código en [codepad](#).

La salida de este programa es:

```

TablaCad[0] = Numero:  0
TablaCad[1] = Numero:  1
TablaCad[2] = Numero:  2
TablaCad[3] = Numero:  3
TablaCad[4] = Numero:  4

```

Ahora funciona como es debido.

El problema es parecido al que surgía con el constructor copia en clases que usaban memoria dinámica. El funcionamiento es correcto, pero el resultado no siempre es el esperado. Como norma general, cuando apliquemos plantillas, debemos usar clases con constructores sin parámetros, y tener cuidado cuando las apliquemos a tipos que impliquen punteros o memoria dinámica.

Funciones que usan plantillas como parámetros

Es posible crear funciones que admitan parámetros que sean una plantilla. Hay dos modos de pasar las plantillas: se puede pasar una instancia determinada de la plantilla o la plantilla genérica.

Pasar una instancia de una plantilla

Si declaramos y definimos una función que tome como parámetro una instancia concreta de una plantilla, esa función no estará disponible para el resto de las posibles instancias. Por ejemplo, si creamos una función para una tabla de enteros, esa función no podrá aplicarse a una tabla de caracteres:

```
// F_Tabla.cpp: ejemplo de función de plantilla para
// Tabla para enteros:
// C con Clase: Marzo de 2002

#include <iostream>
#include "Tabla.h"

using namespace std;

const int nElementos = 5;

void Incrementa(Tabla<int> &t); // (1)

int main() {
    Tabla<int> TablaInt(nElementos);
    Tabla<char> TablaChar(nElementos);

    for(int i = 0; i < nElementos; i++) {
        TablaInt[i] = 0;
        TablaChar[i] = 0;
    }

    Incrementa(TablaInt);
    // Incrementa(TablaChar); // <-- Ilegal (2)
```

```

        for(int i = 0; i < nElementos; i++)
            cout << "TablaInt[" << i << "] = "
                << TablaInt[i] << endl;

        return 0;
    }

    void Incrementa(Tabla<int> &t) { // (3)
        for(int i = 0; i < t.NElementos(); i++)
            t[i]++;
    }
}

```

En (1) vemos que el argumento que especificamos es `Tabla<int>`, es decir, un tipo específico de plantilla: una instancia de `int`. Esto hace que sea imposible aplicar esta función a otros tipos de instancia, como en (2) para el caso de `char`, si intentamos compilar sin comentar esta línea el compilador dará error. Finalmente, en (3) vemos cómo se implementa la función, y cómo se usa el parámetro como si se tratase de una clase corriente.

Pasar una plantilla genérica

Si declaramos y definimos una función que tome como parámetro una instancia cualquiera, tendremos que crear una función de plantilla. Para ello, como ya hemos visto, hay que añadir a la declaración de la función la parte `template<class T>`.

Veamos un ejemplo:

```

// F_Tabla2.cpp: ejemplo de función de plantilla
// Tabla genérica:
// C con Clase: Marzo de 2002

#include <iostream>
#include <cstdio>
#include "Tabla.h"
#include "CCadena.h"

using namespace std;

```

```

const int nElementos = 5;

template<class T>
void Mostrar(Tabla<T> &t); // (1)

int main() {
    Tabla<int> TablaInt(nElementos);
    Tabla<Cadena> TablaCadena(nElementos);
    char cad[20];

    for(int i = 0; i < nElementos; i++) TablaInt[i] = i;
    for(int i = 0; i < nElementos; i++) {
        sprintf(cad, "Cad no.: %2d", i);
        TablaCadena[i] = cad;
    }

    Mostrar(TablaInt); // (2)
    Mostrar(TablaCadena); // (3)

    return 0;
}

template<class T>
void Mostrar(Tabla<T> &t) { // (4)
    for(int i = 0; i < t.NElementos(); i++)
        cout << t[i] << endl;
}

```

En (1) vemos la forma de declarar una plantilla de función que puede aplicarse a nuestra plantilla de clase Tabla. En este caso, la función sí puede aplicarse a cualquier tipo de instancia de la clase Tabla, como se ve en (2) y (3). Finalmente, en (4) vemos la definición de la plantilla de función.

Amigos de plantillas

Por supuesto, en el caso de las plantillas también podemos definir relaciones de amistad con otras funciones o clases. Podemos distinguir dos tipos de funciones o clases amigas de plantillas de clases:

Clase o función amiga de una plantilla

Tomemos el caso de la plantilla de función que vimos en el apartado anterior. La función que pusimos como ejemplo no necesitaba ser amiga de la plantilla porque no era necesario que accediera a miembros privados de la plantilla. Pero podemos escribirla de modo que acceda directamente a los miembros privados, y para que ese acceso sea posible, debemos declarar la función como amiga de la plantilla.

Modificación de la plantilla Tabla con la función Mostrar como amiga de la plantilla:

```
// Tabla.h: definición de la plantilla tabla:
// C con Clase: Marzo de 2002

#ifndef T_TABLA
#define T_TABLA

template <class T> class Tabla; // Declaración adelantada de
Tabla
template <class T> void Mostrar(Tabla<T> &t); // Para poder
declarar la función Mostrar

template <class T>
class Tabla {
public:
    Tabla(int nElem);
    ~Tabla();
    T& operator[](int indice) { return pT[indice]; }
    const int NElementos() {return nElementos;}

    friend void Mostrar<>(Tabla<T>&); // (1) Y poder
    declararla como amiga de Tabla

private:
    T *pT;
    int nElementos;
};

// Definición:
template <class T>
```

```

Tabla<T>::Tabla(int nElem) : nElementos(nElem) {
    pT = new T[nElementos];
}

template <class T>
Tabla<T>::~~Tabla() {
    delete[] pT;
}
#endif

```

Programa de ejemplo:

```

// F_Tabla2.cpp: ejemplo de función amiga de
// plantilla Tabla genérica:
// C con Clase: Marzo de 2002

#include <iostream>
#include <cstdio>
#include "Tabla.h"
#include "CCadena.h"

using namespace std;

const int nElementos = 5;

template<class T>
void Mostrar(Tabla<T> &t); // (2)

int main() {
    Tabla<int> TablaInt(nElementos);
    Tabla<Cadena> TablaCadena(nElementos);
    char cad[20];

    for(int i = 0; i < nElementos; i++) TablaInt[i] = i;
    for(int i = 0; i < nElementos; i++) {
        sprintf(cad, "Cad no.: %2d", i);
        TablaCadena[i] = cad;
    }

    Mostrar(TablaInt);
    Mostrar(TablaCadena);

    return 0;
}

```

```

template<class T>
void Mostrar(Tabla<T> &t) { // (3)
    for(int i = 0; i < t.nElementos; i++)
        cout << t.pT[i] << endl;
}

```

Nota (1):

aunque esto no sea del todo estándar, algunos compiladores, (sin ir más lejos los que usa Dev-C++), requieren que se incluya <> a continuación del nombre de la función que declaramos como amiga, cuando esa función sea una plantilla de función. Si te fijas en (2) y (3) verás que eso no es necesario cuando se declara el prototipo o se define la plantilla de función. En otros compiladores puede que no sea necesario incluir <>, por ejemplo, en Borland C++ no lo es.

Clase o función amiga de una instancia de una plantilla

Limitando aún más las relaciones de amistad, podemos declarar funciones amigas de una única instancia de la plantilla.

Dentro de la declaración de una plantilla podemos declarar una clase o función amiga de una determinada instancia de la plantilla, la relación de amistad no se establece para otras posibles instancias de la plantilla. Usando el mismo último ejemplo, pero sustituyendo la línea de la declaración de la clase que hace referencia a la función amiga:

```

template<class T>
class Tabla {
    ...
    friend void Mostrar<>(Tabla<int>&);
    ...
};

```

Esto hace que sólo se pueda aplicar la función Mostrar a las instancias <int> de la plantilla Tabla. Por supuesto, tanto el prototipo como la definición de la función "Mostrar" no cambian en nada, debemos seguir usando una plantilla de función idéntica que en el ejemplo anterior.

Miembros estáticos: datos y funciones

Igual que con las clases normales, es posible declarar datos miembro o funciones estáticas dentro de una plantilla. En este caso existirá una copia de cada uno de ellos para cada tipo de instancia que se cree.

Por ejemplo, si añadimos un miembro static en nuestra declaración de "Tabla", se creará una única instancia de miembro estático para todas las instancias de Tabla<int>, otro para las instancias de Tabla<Cadena>, etc.

Hay un punto importante a tener en cuenta. Tanto si trabajamos con clases normales como con plantillas, debemos reservar un espacio físico para las variables estáticas, de otro modo el compilador no les asignará memoria, y se producirá un error si intentamos acceder a esos miembros.

Nota:

Tanto una plantilla como una clase pueden estar definidas en un fichero de cabecera (.h), aunque es más frecuente con las plantillas, ya que toda su definición debe estar en el fichero de cabecera; bien dentro de la declaración, en forma de funciones inline o bien en el mismo fichero de cabecera, a continuación de la declaración de la plantilla. Estos ficheros de cabecera pueden estar incluidos en varios módulos diferentes de la misma aplicación, de modo que el espacio físico de los miembros estáticos debe crearse sólo una vez en la memoria

global de la aplicación, y por lo tanto, no debe hacerse dentro de los ficheros de cabecera.

Veamos un ejemplo:

```
// Fichero de cabecera: prueba.h
#ifndef T_PRUEBA
#define T_PRUEBA

template <class T>;
class Ejemplo {
public:
    Ejemplo(T obj) {objeto = obj; estatico++;}
    ~Ejemplo() {estatico--;}
    static int LeeEstatico() {return estatico;}

private:
    static int estatico; // (1)
    T objeto; // Justificamos el uso de la plantilla :-)
};

#endif
```

Ahora probemos los miembros estáticos de nuestra clase:

```
// Fichero de prueba: prueba.cpp
#include <iostream>
#include "prueba.h"

using namespace std;

// Esto es necesario para que exista
// una instancia de la variable:
template <class T> int Ejemplo<T>::estatico; // (2)

int main() {
    Ejemplo<int> EjemploInt1(10);
    cout << "Ejemplo<int>: " << EjemploInt1.LeeEstatico()
        << endl; // (3)
    Ejemplo<char> EjemploChar1('g');
    cout << "Ejemplo<char>: "
```

```

        << EjemploChar1.LeeEstatico() << endl;    // (4)
Ejemplo<int> EjemploInt2(20);
cout << "Ejemplo<int>: "
        << EjemploInt1.LeeEstatico() << endl;    // (5)
Ejemplo<float> EjemploFloat1(32.12);
cout << "Ejemplo<float>: "
        << Ejemplo<float>::LeeEstatico() << endl;    // (6)
Ejemplo<int> EjemploInt3(30);
cout << "Ejemplo<int>: "
        << EjemploInt1.LeeEstatico() << endl;    // (7)

return 0;
}

```

La salida de este programa debería ser algo así:

```

Ejemplo<int>: 1
Ejemplo<char>: 1
Ejemplo<int>: 2
Ejemplo<float>: 1
Ejemplo<int>: 3

```

Vamos a ver si explicamos algunos detalles de este programa.

Para empezar, en (1) vemos la declaración de la variable estática de la plantilla y en (2) la definición. En realidad se trata de una plantilla de declaración, en cierto modo, ya que adjudica memoria para cada miembro estático de cada tipo de instancia de la plantilla. Prueba a ver qué pasa si no incluyes esta línea.

El resto de los puntos muestra que realmente se crea un miembro estático para cada tipo de instancia. En (2) se crea una instancia de tipo `Ejemplo<int>`, en (3) una de tipo `Ejemplo<char>`, en (4) una segunda de tipo `Ejemplo<int>`, etc. Eso se ve también en los valores de salida.

Ejemplo de implementación de una plantilla para una pila

Considero que el tema es lo bastante interesante como para incluir algún ejemplo ilustrativo. Vamos a crear una plantilla para declarar pilas de cualquier tipo de objeto, y de ese modo demostraremos parte de la potencia de las plantillas.

Para empezar, vamos a ver la declaración de la plantilla, como siempre, incluida en un fichero de cabecera para ella sola:

```
// Pila.h: definición de plantilla para pilas
// C con Clase: Marzo de 2002

#ifndef TPILA
#define TPILA

// Plantilla para pilas genéricas:
template <class T>
class Pila {
    // Plantilla para nodos de pila, definición
    // local, sólo accesible para Pila:
    template <class Tn>
    class Nodo {
    public:
        Nodo(const Tn& t, Nodo<Tn> *ant) : anterior(ant) {
            pT = new Tn(t);
        }
        Nodo(Nodo<Tn> &n) { // Constructor copia
            // Invocamos al constructor copia de la clase de Tn
            pT = new Tn(*n.pT);
            anterior = n.anterior;
        }
        ~Nodo() { delete pT; }
        Tn *pT;
        Nodo<Tn> *anterior;
    };
    ///// Fin de declaración de plantilla de nodo /////

// Declaraciones de Pila:
public:
    Pila() : inicio(NULL) {} // Constructor
    ~Pila() { while(inicio) Pop(); }
    void Push(const T &t) {
        Nodo<T> *aux = new Nodo<T>(t, inicio);
        inicio = aux;
    }
}
```

```

T Pop() {
    T temp(*inicio->pT);
    Nodo<T> *aux = inicio;
    inicio = aux->anterior;
    delete aux;
    return temp;
}

bool Vacia() { return inicio == NULL; }

private:
    Nodo<T> *inicio;
};

#endif

```

Aquí hemos añadido una pequeña complicación: hemos definido una plantilla para *Nodo* dentro de la plantilla de *Pila*. De este modo definiremos instancias de *Nodo* locales para cada instancia de *Pila*. Aunque no sea necesario, ya que podríamos haber creado dos plantillas independientes, hacerlo de este modo nos permite declarar ambas clases sin necesidad de establecer relaciones de amistad entre ellas. De todos modos, el nodo que hemos creado para la estructura *Pila* no tiene uso para otras clases.

En cuanto a la definición de la *Pila*, sólo hemos declarado cinco funciones: el constructor, el destructor, las funciones *Push*, para almacenar objetos en la pila y *Pop* para recuperarlos y *Vacia* para comprobar si la pila está vacía.

En cuanto al constructor, sencillamente construye una pila vacía.

El destructor recupera todos los objetos almacenados en la pila. Recuerda que en las [pilas](#), leer un valor implica borrarlo o retirarlo de ella.

La función *Push* coloca un objeto en la pila. Para ello crea un nuevo nodo que contiene una copia de ese objeto y hace que su puntero "anterior" apunte al nodo que hasta ahora era el último. Después actualiza el inicio de la *Pila* para que apunte a ese nuevo nodo.

La función *Pop* recupera un objeto de la pila. Para ello creamos una copia temporal del objeto almacenado en el último nodo de la

pila, esto es necesario porque lo siguiente que hacemos es actualizar el nodo de inicio (que será el anterior al último) y después borrar el último nodo. Finalmente devolvemos el objeto temporal.

Ahora vamos a probar nuestra pila, para ello haremos un pequeño programa:

```
// Pru_Pila.cpp: Prueba de plantilla pila
// C con Clase: Marzo de 2002

#include <iostream>
#include "pila.h"
#include "CCadena.h"

using namespace std;

// Ejemplo de plantilla de función:
template <class T>
void Intercambia(T &x, T &y) {
    Pila<T> pila;

    pila.Push(x);
    pila.Push(y);
    x = pila.Pop();
    y = pila.Pop();
}

int main() {
    Pila<int> PilaInt;
    Pila<Cadena> PilaCad;

    int x=13, y=21;
    Cadena cadx("Cadena X");
    Cadena cady("Cadena Y ----");

    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
    Intercambia(x, y);
    cout << "x=" << x << endl;
    cout << "y=" << y << endl;

    cout << "cadx=" << cadx << endl;
    cout << "cady=" << cady << endl;
    Intercambia(cadx, cady);
    cout << "cadx=" << cadx << endl;
```

```

    cout << "cady=" << cady << endl;

    PilaInt.Push(32);
    PilaInt.Push(4);
    PilaInt.Push(23);
    PilaInt.Push(12);
    PilaInt.Push(64);
    PilaInt.Push(31);

    PilaCad.Push("uno");
    PilaCad.Push("dos");
    PilaCad.Push("tres");
    PilaCad.Push("cuatro");

    cout << PilaInt.Pop() << endl;
    cout << PilaInt.Pop() << endl;
    cout << PilaInt.Pop() << endl;
    cout << PilaInt.Pop() << endl;
    cout << PilaInt.Pop() << endl;
    cout << PilaInt.Pop() << endl;

    cout << PilaCad.Pop() << endl;
    cout << PilaCad.Pop() << endl;
    cout << PilaCad.Pop() << endl;
    cout << PilaCad.Pop() << endl;

    return 0;
}

```

Ejecutar este código en [codepad](#).

La salida demuestra que todo funciona:

```

x=13
y=21
x=21
y=13
cadx=Cadena X
cady=Cadena Y ----
cadx=Cadena Y ----
cady=Cadena X
31
64
12
23
4

```

```
32
cuatro
tres
dos
uno
```

Hemos aprovechado para crear una plantilla de función para intercambiar valores de objetos, que a su vez se basa en la plantilla de pila, y aunque esto no sería necesario, reconocerás que queda bonito :-).

Bibliotecas de plantillas

El ANSI de C++ define ciertas bibliotecas de plantillas conocidas como STL (Standard Template Library), que contiene muchas definiciones de plantillas para crear estructuras como listas, colas, pilas, árboles, tablas HASH, mapas, etc.

Está fuera del objetivo de este curso explicar estas bibliotecas, pero es conveniente saber que existen y que por su puesto, se suelen incluir con los compiladores de C++.

Aunque se trata de bibliotecas estándar, lo cierto es que existen varias implementaciones, que, al menos en teoría, deberían coincidir en cuanto a sintaxis y comportamiento.

Palabra typename

```
template <typename T> class Plantilla;
```

Es equivalente usar typename y class como parte de la declaración de T, en ambos casos T puede ser una clase o un tipo fundamental, como int, char o float. Sin embargo, usar typename puede ser mucho más claro como nombre genérico que class.

Palabras reservadas usadas en este capítulo

template, typename.

41 Punteros a miembros de clases o estructuras

C++ permite declarar punteros a miembros de clases, estructuras y uniones. Aunque en el caso de las clases, los miembros deben ser públicos para que pueda accederse a ellos.

La sintaxis para la declaración de un puntero a un miembro es la siguiente:

```
<tipo> <clase|estructura|unión>::*<identificador>;
```

De este modo se declara un puntero "identificador" a un miembro de tipo "tipo" de la clase, estructura o unión especificada.

Ejemplos:

```
struct punto3D {
    int x;
    int y;
    int z;
};

class registro {
public:
    registro();

    float v;
    float w;
};

int punto3D::*coordenada; // (1)
float registro::*valor;   // (2)
```

El primer ejemplo declara un puntero "coordenada" a un miembro de tipo int de la estructura "punto3D". El segundo declara un puntero "valor" a un miembro público de la clase "registro".

Asignación de valores a punteros a miembro

Una vez declarado un puntero, debemos asignarle la dirección de un miembro del tipo adecuado de la clase, estructura o unión. Podremos asignarle la dirección de cualquiera de los miembros del tipo adecuado. La sintaxis es:

```
<identificador> = &<clase|estructura|unión>::<campo>;
```

En el ejemplo anterior, serían válidas las siguientes asignaciones:

```
coordenada = &punto3D::x;  
coordenada = &punto3D::y;  
coordenada = &punto3D::z;  
valor = &registro::v;  
valor = &registro::w;
```

Operadores .* y ->*

Ahora bien, ya sabemos cómo declarar punteros a miembros, pero no cómo trabajar con ellos.

C++ dispone de dos operadores para trabajar con punteros a miembros: .* y ->*. A lo mejor los echabas de menos :-).

Se trata de dos variantes del mismo operador, uno para objetos y otro para punteros:

```
<objeto>.*<puntero>  
<puntero_a_objeto>->*<puntero>
```

La primera forma se usa cuando tratamos de acceder a un miembro de un objeto.

La segunda cuando lo hacemos a través de un puntero a un objeto.

Veamos un ejemplo completo:

```
#include <iostream>
using namespace std;

class clase {
public:
    clase(int a, int b) : x(a), y(b) {}

public:
    int x;
    int y;
};

int main() {
    clase uno(6,10);
    clase *dos = new clase(88,99);
    int clase::*puntero;

    puntero = &clase::x;
    cout << uno.*puntero << endl;
    cout << dos->*puntero << endl;

    puntero = &clase::y;
    cout << uno.*puntero << endl;
    cout << dos->*puntero << endl;

    delete dos;
    return 0;
}
```

La utilidad práctica no es probable que se presente frecuentemente, y casi nunca con clases, ya que no es corriente declarar miembros públicos. Sin embargo nos ofrece algunas posibilidades interesantes a la hora de recorrer miembros concretos de arrays de estructuras, aplicando la misma función o expresión a cada uno.

También debemos recordar que es posible declarar punteros a funciones, y las funciones miembros de clases no son una excepción. En ese caso sí es corriente que existan funciones públicas.

```
#include <iostream>
using namespace std;

class clase {
public:
    clase(int a, int b) : x(a), y(b) {}
    int funcion(int a) {
        if(0 == a) return x; else return y;
    }

private:
    int x;
    int y;
};

int main() {
    clase uno(6,10);
    clase *dos = new clase(88,99);
    int (clase::*pfun)(int);

    pfun = &clase::funcion;

    cout << (uno.*pfun)(0) << endl;
    cout << (uno.*pfun)(1) << endl;
    cout << (dos->*pfun)(0) << endl;
    cout << (dos->*pfun)(1) << endl;

    delete dos;
    return 0;
}
```

Para ejecutar una función desde un puntero a miembro hay que usar los paréntesis, ya que el operador de llamada a función "()" tiene mayor prioridad que los operadores "." y "->".

42 Castings en C++

Hasta ahora hemos usado sólo el *casting* que existe en C, que vimos en el [capítulo 9](#). Pero ese tipo de *casting* no es el único que existe en C++, de hecho, su uso está desaconsejado, ya que el por una parte los paréntesis se usan mucho en C++, además, este tipo de *casting* realiza conversiones diferentes dependiendo de cada situación. Se recomienda usar uno de los nuevos operadores de C++ diseñados para realizar esta tarea.

C++ dispone de cuatro operadores para realizar *castings*, algunos de ellos necesarios para realizar conversiones entre objetos de una misma jerarquía de clases.

Operador `static_cast<>`

La sintaxis de este operador es:

```
static_cast<tipo> (<expresión>);
```

Este operador realiza una conversión de tipo durante la compilación del programa, de modo que no crea más código durante la ejecución, descargando esa tarea en el compilador.

Este operador se usa para realizar conversiones de tipo que de otro modo haría el compilador automáticamente, por ejemplo, convertir un puntero a un objeto de una clase derivada a un puntero a una clase base pública:

```
#include <iostream>
using namespace std;

class Base {
```

```

public:
    Base(int valor) : x(valor) {}
    void Mostrar() { cout << x << endl; }

protected:
    int x;
};

class Derivada : public Base {
public:
    Derivada(int ivalor, float fvalor) :
        Base(ivalor), y(fvalor) {}
    void Mostrar() {
        cout << x << ", " << y << endl;
    }

private:
    float y;
};

int main() {
    Derivada *pDer = new Derivada(10, 23.3);
    Base *pBas;

    pDer->Mostrar(); // Derivada
    pBas = static_cast<Base *> (pDer);
    // pBas = pDer; // Igualmente legal, pero implícito
    pBas->Mostrar(); // Base

    delete pDer;
    return 0;
}

```

Otro ejemplo es cuando se usan operadores de conversión de tipo, como en el caso del [capítulo 35](#), en lugar de usar el operador de forma implícita, podemos usarlo mediante el operador `static_cast`:

```

#include <iostream>
using namespace std;

class Tiempo {
public:

```

```

    Tiempo(int h=0, int m=0) : hora(h), minuto(m) {}

    void Mostrar();
    operator int() {
        return hora*60+minuto;
    }
private:
    int hora;
    int minuto;
};

void Tiempo::Mostrar() {
    cout << hora << ":" << minuto << endl;
}

int main() {
    Tiempo Ahora(12,24);
    int minutos;

    Ahora.Mostrar();
    minutos = static_cast<int> (Ahora);
    // minutos = Ahora; // Igualmente legal, pero implícito

    cout << minutos << endl;

    return 0;
}

```

Este operador se usa en los casos en que el programador desea documentar las conversiones de tipo implícitas, con objeto de aclarar que realmente se desean hacer esas conversiones de tipo.

Operador `const_cast<>`

La sintaxis de este operador es:

```
const_cast<tipo> (<expresión>);
```

Se usa para eliminar o añadir los modificadores `const` y `volatile` de una expresión.

Por eso, tanto tipo como expresión deben ser del mismo tipo, salvo por los modificadores `const` o `volatile` que tengan que aparecer o desaparecer.

Por ejemplo:

```
#include <iostream>
using namespace std;

int main() {
    const int x = 10;
    int *x_var;

    x_var = const_cast<int*> (&x); // Válido
    // x_var = &x; // Ilegal, el compilador da error
    *x_var = 14;    // Indefinido

    cout << *x_var << ", " << x << endl;

    return 0;
}
```

En el ejemplo vemos que podemos hacer que un puntero apunte a una constante, e incluso podemos modificar el valor de la variable a la que apunta. Sin embargo, este operador se usa para obtener expresiones donde se necesite añadir o eliminar esos modificadores. Si se intenta modificar el valor de una expresión constante, el resultado es indeterminado.

El ejemplo anterior, compilado en Dev-C++ no modifica el valor de `x`, lo cual es lógico, ya que `x` es constante.

Operador `reinterpret_cast<>`

La sintaxis de este operador es:

```
reinterpret_cast<tipo> (<expresión>);
```


Se usa para hacer cambios de tipo a nivel de bits, es decir, el valor de la "expresión" se interpreta como si fuese un objeto del tipo "tipo". Los modificadores `const` y `volatile` no se modifican, permanecen igual que en el valor original de "expresión". Este tipo de conversión es peligrosa, desde el punto de vista de la compatibilidad, hay que usarla con cuidado.

Posibles usos son conseguir punteros a variables de distinto tipo al original, por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int x = 0x12dc34f2;
    int *pix = &x;
    unsigned char *pcx;

    pcx = reinterpret_cast<unsigned char *> (pix);

    cout << hex << " = "
         << static_cast<unsigned int> (pcx[0]) << ", "
         << static_cast<unsigned int> (pcx[1]) << ", "
         << static_cast<unsigned int> (pcx[2]) << ", "
         << static_cast<unsigned int> (pcx[3]) << endl;

    return 0;
}
```

La salida tiene esta forma:

```
12dc34f2 = f2, 34, dc, 12
```

Operador typeid

La sintaxis de este operador es:

```
const type_info typeid(<tipo>)  
const type_info typeid(<objeto>)
```

El tipo puede ser cualquiera de los fundamentales, derivados o una clase, estructura o unión. Si se trata de un objeto, también puede ser de cualquier tipo.

El valor de retorno es un objeto constante de tipo **type_info**.

La clase **type_info** se define en el fichero de cabecera estándar "typeinfo". Tiene la siguiente declaración:

```
class type_info {  
public:  
    virtual ~type_info();  
  
private:  
    type_info& operator=(const type_info&);  
    type_info(const type_info&);  
  
protected:  
    const char * __name;  
  
protected:  
    explicit type_info(const char * __n): __name(__n) { }  
  
public:  
    const char* name() const;  
    bool before(const type_info& __arg) const;  
    bool operator==(const type_info& __arg) const;  
    bool operator!=(const type_info& __arg) const;  
    ...  
};
```

Nos interesa, concretamente, la función "name", y tal vez, los operadores == y !=.

La función "name" nos permite mostrar el nombre del tipo a que pertenece un objeto, los operadores nos permiten averiguar si dos objetos son del mismo tipo, o clase o si dos clases o tipos son equivalentes, por ejemplo:

```

#include <iostream>
#include <typeinfo>
using namespace std;

struct punto3D {
    int x,y,z;
};

union Union {
    int x;
    float z;
    char a;
};

class Clase {
public:
    Clase() {}
};

typedef int Entero;

int main() {
    int x;
    float y;
    int z[10];
    punto3D punto3d;
    Union uni;
    Clase clase;
    void *pv;

    cout << "variable int: " << typeid(x).name() << endl;
    cout << "variable float: " << typeid(y).name() << endl;
    cout << "array de 10 int:" << typeid(z).name() << endl;
    cout << "estructura global: " << typeid(punto3d).name()
        << endl;
    cout << "unión global: " << typeid(uni).name() << endl;
    cout << "clase global: " << typeid(clase).name()
        << endl;
    cout << "puntero void: " << typeid(pv).name() << endl;
    cout << "typedef Entero: " << typeid(Entero).name()
        << endl;
    if(typeid(Entero) == typeid(int))
        cout << "int y Entero son el mismo tipo" << endl;

    return 0;
}

```

Ejecutar este código en [codepad](#).

La salida, en Dev-C++, tiene esta forma:

```
variable int: i
variable float: f
array de 10 int: A10_i
estructura global: 7punto3D
unión global: 5Union
clase global: 5Clase
puntero void: Pv
typedef Entero: i
int y Entero son el mismo tipo
```

La utilidad es detectar los tipos de objetos durante la ejecución, sobre todo en aplicaciones con frecuente uso de polimorfismo, y que requieran diferentes formas de manejar cada objeto en función de su clase.

Además de usar el operador `typeid` se puede usar el operador `dynamic_cast`, que se explica en el siguiente punto.

Operador `dynamic_cast<>`

La sintaxis de este operador es:

```
dynamic_cast<tipo> (<objeto>);
```

Se usa para hacer cambios de tipo durante la ejecución. Y se usa la base de datos formada por las estructuras *type_info* que vimos antes.

Este operador sólo puede usarse con objetos polimórficos, cualquier intento de aplicarlo a objetos de tipos fundamentales o agregados o de clases no polimórficas dará como resultado un error.

Hay dos modalidades de `dynamic_cast`, una usa punteros y la otra referencias:

```

class Base { // Clase base virtual
...
};

class Derivada : public Base { // Clase derivada
...
};

// Modalidad de puntero:
Derivada *p = dynamic_cast<Derivada *> (&Base);
// Modalidad de referencia:
Derivada &refd = dynamic_cast<Derivada &> (Base);

```

Lo que el programa intentará hacer, durante la ejecución, es obtener bien un puntero o bien una referencia a un objeto de cierta clase base en forma de clase derivada, pero puede que se consiga, o puede que no.

Esto es, en cierto modo, lo contrario a lo que hacíamos al usar polimorfismo. En lugar de obtener un puntero o referencia a la clase base a partir de la derivada, haremos lo contrario: *intentar* obtener un puntero o referencia a la clase derivada a partir de la clase base.

Volvamos a nuestro ejemplo de Personas.

Vamos a añadir un miembro "sueldo" a la clase "Empleado", y la modificaremos para poder inicializar y leer ese dato.

También crearemos una función "LeerSueldo" que aceptará como parámetro un puntero a un objeto de la clase "Persona":

```

#include <iostream>
#include <cstring>
using namespace std;

class Persona { // Virtual
public:
    Persona(char *n) { strcpy(nombre, n); }
    virtual void VerNombre() = 0; // Virtual pura
protected:
    char nombre[30];
};

```

```

class Empleado : public Persona {
public:
    Empleado(char *n, float s) : Persona(n), sueldo(s) {}
    void VerNombre() {
        cout << "Emp: " << nombre << endl;
    }
    void VerSueldo() {
        cout << "Salario: " << sueldo << endl;
    }
private:
    float sueldo;
};

class Estudiante : public Persona {
public:
    Estudiante(char *n) : Persona(n) {}
    void VerNombre() {
        cout << "Est: " << nombre << endl;
    }
};

void VerSueldo(Persona *p) {
    if(Empleado *pEmp = dynamic_cast<Empleado *> (p))
        pEmp->VerSueldo();
    else
        cout << "No tiene salario." << endl;
}

int main() {
    Persona *Pepito = new Estudiante("Jose");
    Persona *Carlos = new Empleado("Carlos", 1000.0);

    Carlos->VerNombre();
    VerSueldo(Carlos);

    Pepito->VerNombre();
    VerSueldo(Pepito);

    delete Pepito;
    delete Carlos;

    return 0;
}

```

La función "VerSueldo" recibe un puntero a un objeto de la clase base virtual "Persona". Pero a priori no sabemos si el objeto original era un empleado o un estudiante, de modo que no podemos prever si tendrá o no sueldo. Para averiguarlo hacemos un casting dinámico a la clase derivada "Empleado", y si tenemos éxito es que se trata efectivamente de un empleado y mostramos el salario. Si fracasamos, mostramos un mensaje de error.

Pero esto es válido sólo con punteros, si intentamos hacerlo con referencias tendremos un serio problema, ya que no es posible declarar referencias indefinidas. Esto quiere decir que, por una parte, no podemos usar la expresión del *casting* como una condición en una sentencia if. Por otra parte, si el casting fracasa, se producirá una excepción, ya que se asignará un valor nulo a una referencia durante la ejecución:

```
void VerSueldo(Persona &p) {  
    Empleado rEmp = dynamic_cast<Empleado &> p;  
    ...  
}
```

Por lo tanto tendremos que usar manejo de excepciones (que es el tema del siguiente capítulo), para usar referencias con el *casting* dinámico:

```
void VerSueldo(Persona &p) {  
    try {  
        Empleado rEmp = dynamic_cast<Empleado &> p;  
        rEmp.VerSueldo();  
    }  
    catch (std::bad_cast) {  
        cout << "No tiene salario." << endl;  
    }  
}
```

Castings cruzados

Cuando tenemos una clase producto de una derivación múltiple, es posible obtener punteros o referencias a una clase base a partir de objetos o punteros a objetos de otra clase base, eso sí, es necesario usar polimorfismo, no podemos usar un objeto de una clase base para obtener otra. Por ejemplo:

```
#include <iostream>
using namespace std;

class ClaseA {
public:
    ClaseA(int x) : valorA(x) {}
    void Mostrar() {
        cout << valorA << endl;
    }
    virtual void nada() {} // Forzar polimorfismo
private:
    int valorA;
};

class ClaseB {
public:
    ClaseB(float x) : valorB(x) {}
    void Mostrar() {
        cout << valorB << endl;
    }

private:
    float valorB;
};

class ClaseD : public ClaseA, public ClaseB {
public:
    ClaseD(int x, float y, char c) :
        ClaseA(x), ClaseB(y), valorD(c) {}
    void Mostrar() {
        cout << valorD << endl;
    }

private:
    char valorD;
};

int main() {
```



```
ClaseA *cA = new ClaseD(10,15.3,'a');  
ClaseB *cB = dynamic_cast<ClaseB*> (cA);  
  
cA->Mostrar();  
cB->Mostrar();  
  
return 0;  
}
```

43 Manejo de excepciones

Las excepciones son en realidad errores durante la ejecución. Si uno de esos errores se produce y no implementamos el manejo de excepciones, el programa sencillamente terminará abruptamente. Es muy probable que si hay ficheros abiertos no se guarde el contenido de los buffers, ni se cierren, además ciertos objetos no serán destruidos, y se producirán fugas de memoria.

En programas pequeños podemos prever las situaciones en que se pueden producir excepciones y evitarlos. Las excepciones más habituales son las de peticiones de memoria fallidas.

Veamos este ejemplo, en el que intentamos crear un *array* de cien millones de enteros:

```
#include <iostream>
using namespace std;

int main() {
    int *x = NULL;
    int y = 100000000;

    x = new int[y];
    x[10] = 0;
    cout << "Puntero: " << (void *) x << endl;
    delete[] x;

    return 0;
}
```

El sistema operativo se quejará, y el programa terminará en el momento que intentamos asignar un valor a un elemento del *array*.

Podemos intentar evitar este error, comprobando el valor del puntero después del new:

```

#include <iostream>
using namespace std;

int main() {
    int *x = 0;
    int y = 1000000000;

    x = new int[y];
    if(x) {
        x[10] = 0;
        cout << "Puntero: " << (void *) x << endl;
        delete[] x;
    } else {
        cout << "Memoria insuficiente." << endl;
    }
    return 0;
}

```

Pero esto tampoco funcionará, ya que es al procesar la sentencia que contiene el operador new cuando se produce la excepción. Sólo nos queda evitar peticiones de memoria que puedan fallar, pero eso no es previsible.

Sin embargo, C++ proporciona un mecanismo más potente para detectar errores de ejecución: las excepciones. Para ello disponemos de tres palabras reservadas extra: try, catch y throw, veamos un ejemplo:

```

#include <iostream>

using namespace std;

int main() {
    int *x;
    int y = 1000000000;

    try {
        x = new int[y];
        x[0] = 10;
        cout << "Puntero: " << (void *) x << endl;
        delete[] x;
    }
}

```

```

    }
    catch(std::bad_alloc&) {
        cout << "Memoria insuficiente" << endl;
    }

    return 0;
}

```

La manipulación de excepciones consiste en transferir la ejecución del programa desde el punto donde se produce la excepción a un manipulador que coincida con el motivo de la excepción.

Como vemos en este ejemplo, un manipulador consiste en un bloque try, donde se incluye el código que puede producir la excepción.

A continuación encontraremos uno o varios manipuladores asociados al bloque try, cada uno de esos manipuladores empiezan con la palabra catch, y entre paréntesis una referencia o un objeto.

En nuestro ejemplo se trata de una referencia a un objeto *bad_alloc*, que es el asociado a excepciones consecuencia de aplicar el operador new.

También debe existir una expresión throw, dentro del bloque try. En nuestro caso es implícita, ya que se trata de una excepción estándar, pero podría haber un throw explícito, por ejemplo:

```

#include <iostream>

using namespace std;

int main() {
    try {
        throw 'x'; // valor de tipo char
    }
    catch(char c) {
        cout << "El valor de c es: " << c << endl;
    }
    catch(int n) {
        cout << "El valor de n es: " << n << endl;
    }
}

```

```
    return 0;
}
```

El throw se comporta como un return. Lo que sucede es lo siguiente: el valor *devuelto* por el throw se asigna al objeto del catch adecuado. En este ejemplo, al tratarse de un carácter, se asigna a la variable 'c', en el catch que contiene un parámetro de tipo char.

En el caso del operador new, si se produce una excepción, se hace un throw de un objeto de la clase *std::bad_alloc*, y como no estamos interesados en ese objeto, sólo usamos el tipo, sin nombre.

El manipulador puede ser invocado por un throw que se encuentre dentro del bloque try asociado, o en una de las funciones llamadas desde él.

Cuando se produce una excepción se busca un manipulador apropiado en el rango del try actual. Si no se encuentra se retrocede al anterior, de modo recursivo, hasta encontrarlo. Cuando se encuentra se destruyen todos los objetos locales en el nivel donde se ha localizado el manipulador, y en todos los niveles por los que hemos pasado.

```
#include <iostream>

using namespace std;

int main() {
    try {
        try {
            try {
                throw 'x'; // valor de tipo char
            }
            catch(int i) {}
            catch(float k) {}
        }
        catch(unsigned int x) {}
    }
    catch(char c) {
        cout << "El valor de c es: " << c << endl;
    }
}
```

```
    return 0;
}
```

En este ejemplo podemos comprobar que a pesar de haber hecho el throw en el tercer nivel del try, el catch que lo procesa es el del primer nivel.

Los tipos de la expresión del throw y el especificado en el catch deben coincidir, o bien, el tipo del catch debe ser una clase base de la expresión del throw. La concordancia de tipos es muy estricta, por ejemplo, no se considera como el mismo tipo int que unsigned int.

Si no se encontrase ningún catch adecuado, se abandona el programa, del mismo modo que si se produce una excepción y no hemos hecho ningún tipo de manipulación de excepciones. Los objetos locales no se destruyen, etc.

Para evitar eso existe un catch general, que captura cualquier throw para el que no exista un catch concreto:

```
#include <iostream>

using namespace std;

int main() {
    try {
        throw 'x'; //
    }
    catch(int c) {
        cout << "El valor de c es: " << c << endl;
    }
    catch(...) {
        cout << "Excepción imprevista" << endl;
    }

    return 0;
}
```

La clase "exception"

Existe una clase base *exception* de la que podemos heredar nuestras propias clases derivadas para pasar objetos a los manipuladores. Esto nos ahorra cierto trabajo, ya que aplicando polimorfismo necesitamos un único catch para procesar todas las posibles excepciones.

Esta clase base se declara en el fichero de cabecera estándar "exception", y tiene la siguiente forma (muy sencilla):

```
class exception {
public:
    exception() throw() { }
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
```

Sólo contiene tres funciones: el constructor, el destructor y la función *what*, estas dos últimas virtuales. La función *what* debe devolver una cadena que indique el motivo de la excepción.

Verás que después del nombre de la función, en el caso del destructor y de la función *what* aparece un añadido throw(), esto sirve para indicar que estas funciones no pueden producir ningún tipo de excepción, es decir, que no contienen sentencias throw.

Podemos hacer una prueba sobre este tema, por ejemplo, crearemos un programa que copie un fichero.

```
#include <iostream>
#include <fstream>
using namespace std;

void CopiaFichero(const char* Origen, const char *Destino);

int main() {
    char Desde[] = "excepcion.cpt"; // Este fichero
    char Hacia[] = "excepcion.cpy";

    CopiaFichero(Desde, Hacia);
    cin.get();
}
```

```

        return 0;
    }

    void CopiaFichero(const char* Origen, const char *Destino) {
        unsigned char buffer[1024];
        int leido;
        ifstream fe(Origen, ios::in | ios::binary);
        ofstream fs(Destino, ios::out | ios::binary);

        do {
            fe.read(reinterpret_cast<char *> (buffer), 1024);
            leido = fe.gcount();
            fs.write(reinterpret_cast<char *> (buffer), leido);
        } while(leido);
        fe.close();
        fs.close();
    }
}

```

Ahora lo modificaremos para que se genere una excepción si el fichero origen no existe o el de destino no puede ser abierto:

```

#include <iostream>
#include <fstream>
using namespace std;

// Clase derivada de "exception" para manejar excepciones
// de copia de ficheros.
class CopiaEx: public exception {
public:
    CopiaEx(int mot) : exception(), motivo(mot) {}
    const char* what() const throw();
private:
    int motivo;
};

const char* CopiaEx::what() const throw() {
    switch(motivo) {
        case 1:
            return "Fichero de origen no existe";
        case 2:
            return "No es posible abrir el fichero de salida";
    }
    return "Error inesperado";
} // (1)

```



```

void CopiaFichero(const char* Origen, const char *Destino);

int main() {
    char Desde[] = "excepcion.cpp"; // Este fichero
    char Hacia[] = "excepcion.cpy";

    try {
        CopiaFichero(Desde, Hacia);
    }
    catch(CopiaEx &ex) {
        cout << ex.what() << endl;
    } // (2)
    return 0;
}

void CopiaFichero(const char* Origen, const char *Destino) {
    unsigned char buffer[1024];
    int leido;
    ifstream fe(Origen, ios::in | ios::binary);
    if(!fe.good()) throw CopiaEx(1); // (3)

    ofstream fs(Destino, ios::out | ios::binary);
    if(!fs.good()) throw CopiaEx(2); // (4)

    do {
        fe.read(reinterpret_cast<char *> (buffer), 1024);
        leido = fe.gcount();
        fs.write(reinterpret_cast<char *> (buffer), leido);
    } while(leido);
    fe.close();
    fs.close();
}

```

Espero que esté claro lo que hemos hecho. En (1) hemos derivado una clase de *exception*, para hacer el tratamiento de nuestras propias excepciones. Hemos redefinido la función virtual *what* para que muestre los mensajes de error que hemos predefinido para los dos posibles errores que queremos detectar.

En (2) hemos hecho el tratamiento de excepciones, propiamente dicho. Intentamos copiar el fichero, y si no es posible, mostramos el mensaje de error.

Dentro de la función "CopiaFichero" intentamos abrir el fichero de entrada, si fracasamos hacemos un throw con el valor 1, lo mismo con el de salida, pero con un valor 2 para throw.

Ahora podemos intentar ver qué pasa si el fichero de entrada no existe, o si el fichero de salida existe y está protegido contra escritura.

Hay que observar que el objeto que obtenemos en el catch es una referencia. Esto es recomendable, ya que evitamos copiar el objeto devuelto por el throw. Imaginemos que se trata de un objeto más grande, y que la excepción que maneja está relacionada con la memoria disponible. Si pasamos el objeto por valor estaremos obligando al programa a usar más memoria, y puede que no exista suficiente.

Orden en la captura de excepciones

Cuando se derivan clases desde la clase base "exception" hay que tener cuidado en el orden en que las capturamos. Debido que se aplica polimorfismo, cualquier objeto de la jerarquía se ajustará al catch que tenga por argumento un objeto o referencia de la clase base, y sucesivamente, con cada una de las clases derivadas.

Por ejemplo, podemos crear una clase derivada de "exception", "Excep2", otra derivada de ésta, "Excep3", para hacer un tratamiento de excepciones de uno de nuestros programas:

```
class Excep2 : public exception {}
class Excep3 :public Excep2 {}
...
    try {
        // Nuestro código
    }
    catch(Excep2&) {
        // tratamiento
    }
    catch(Excep1&) {
        // tratamiento
    }
```

```

    }
    catch(exception&) {
        // tratamiento
    }
    catch(...) {
        // tratamiento
    }
    ...

```

Si usamos otro orden, perderemos la captura de determinados objetos, por ejemplo, supongamos que primero hacemos el catch de "exception":

```

class Excep2 : public exception {}
class Excep3 :public Excep2 {}
...
    try {
        // Nuestro código
    }
    catch(exception&) {
        // tratamiento
    }
    catch(Excep2&) { // No se captura
        // tratamiento
    }
    catch(Excep1&) { // No se captura
        // tratamiento
    }
    catch(...) {
        // tratamiento
    }
    ...

```

En este caso jamás se capturará una excepción mediante "Excep2" y "Excep3".

Especificaciones de excepciones

Se puede añadir una especificación de las posibles excepciones que puede producir una función:

```
<tipo> <identificador>(<parametros>)  
throw(<lista_excepciones>);
```

De este modo indicamos que la función sólo puede hacer un throw de uno de los tipos especificados en la lista, si la lista está vacía indica que la función no puede producir excepciones.

El compilador no verifica si realmente es así, es decir, podemos hacer un throw con un objeto de uno de los tipos listados, y el compilador no notificará ningún error. Sólo se verifica durante la ejecución, de modo que si se produce una excepción no permitida, el programa sencillamente termina.

Veamos algunos ejemplos:

```
int Compara(int, int) throw();
```

Indica que la función "Compara" no puede producir excepciones.

```
int CrearArray(int) throw(std::bad_alloc);
```

Indica que la función "CrearArray" sólo puede producir excepciones por memoria insuficiente.

```
int MiFuncion(char *) throw(std::bad_alloc, ExDivCero);
```

Indica que la función "MiFuncion" puede producir excepciones por falta de memoria o por división por cero.

Excepciones en constructores y destructores

Uno de los lugares donde más frecuentemente se requiere un tratamiento de excepciones es en los constructores de las clases, normalmente, esos constructores hacen peticiones de memoria, verifican condiciones, leen valores iniciales desde ficheros, etc.

Aunque no hay ningún problema en eso, no es así con los destructores, está desaconsejado que los destructores puedan producir excepciones. La razón es sencilla, los destructores pueden ser invocados automáticamente cuando se procesa una excepción, y si durante ese proceso se produce de nuevo una excepción, el programa terminará inmediatamente.

Sin embargo, si necesitamos generar una excepción desde un destructor, existe un mecanismo que nos permite comprobar si se está procesando una excepción, y en ese caso, no ejecutamos la sentencia `throw`. Se trata de la función estándar: *uncaught_exception*, que devuelve el valor `true` si se está procesando una excepción":

```
class CopiaEx {};  
  
Miclase::~Miclase() throw (CopiaEx) {  
    // Necesitamos copiar un fichero cuando se  
    // destruya un objeto de esta clase, pero  
    // CopiaFichero puede generar una excepción  
    // De modo que antes averiguamos si ya se  
    // está procesando una:  
    if(uncaught_exception()) return; // No hacemos nada  
  
    // En caso contrario, intentamos hacer la copia:  
    CopiaFichero("actual.log", "viejo.log");  
}
```

Pero, es mejor idea hacer el tratamiento de excepciones dentro del propio destructor:

```
class CopiaEx {};
```

```
MiClase::~~MiClase() throw () {  
    try {  
        CopiaFichero("actual.log", "viejo.log");  
    }  
    catch(CopiaEx&) {  
        cout << "No se pudo copiar el fichero 'actual.log'"  
            << endl;  
    }  
}
```

Excepciones estándar

Existen cuatro excepciones estándar, derivadas de la clase "exception", y asociadas a un operador o a un error de especificación:

```
std::bad_alloc        // Al operador new  
std::bad_cast         // Al operador dynamic_cast<>  
std::bad_typeid       // Al operador typeid  
std::bad_exception    // Cuando se viola una especificación
```

Cada vez que se usa uno de los operadores mencionados, puede producirse una excepción. Un programa bien hecho debe tener esto en cuenta, y hacer el tratamiento de excepciones cuando se usen esos operadores. Esto creará aplicaciones robustas y seguras.

Relanzar una excepción

Ya sabemos que los bloques try pueden estar anidados, y que si se produce una excepción en un nivel interior, y no se captura en ese nivel, se lanzará la excepción al siguiente nivel en el orden de anidamiento.

Pero también podemos lanzar una excepción a través de los siguientes niveles, aunque la hayamos capturado. A eso se le llama relanzarla, y para ello se usa throw;, sin argumentos:

```

#include <iostream>
using namespace std;

void Programa();

int main() {
    try {
        // Programa
        Programa();
    }
    catch(int x) {
        cout << "Excepción relanzada capturada." << endl;
        cout << "error: " << x << endl;
    }
    catch(...) {
        cout << "Excepción inesperada." << endl;
    }

    cin.get();
    return 0;
}

void Programa() {
    try {
        // Operaciones...
        throw 10;
    }
    catch(int x) {
        // Relanzar, no nos interesa manejar aquí
        throw;
    }
}

```

La función no hace nada con la excepción capturada, excepto reenviarla a nivel siguiente, donde podremos capturarla de nuevo.

Apéndice A: Codificación ASCII

El origen

Estamos acostumbrados a que los ordenadores manejen cualquier tipo de información: números, textos, gráficos, sonidos, fórmulas... Sin embargo, en realidad, los ordenadores sólo pueden manejar números, y más concretamente, sólo ceros y unos.

De hecho, si profundizamos más, incluso esto es una convención, y en lo más profundo, lo único que encontraremos en un ordenador son células básicas que pueden contener y manipular dos estados. A uno de esos estados se le asigna valor lógico cero, y al otro un uno.

Los técnicos electrónicos llaman a cada uno de estas células *biestable*, precisamente, porque pueden tener dos estados. A cada uno de esos estados se le llama bit. Un bit es la unidad mínima de información.

Desde los primeros ordenadores, acceder a bits como unidad ya resultaba poco práctico, así que los bits se agruparon formando unidades mayores.

En la práctica, nosotros hacemos lo mismo con los números, para manejar números mayores de nueve usamos dos dígitos, de modo que cada uno de ellos tiene un valor diferente según la posición que ocupe. Por ejemplo, en el número 23, el 2 tiene un *peso* 10 veces mayor que el 3, ya que ocupa una posición más a la izquierda. Si usamos tres dígitos, el tercero por la derecha tiene un *peso* 100 veces mayor que el primero, y así sucesivamente.

Los primeros ordenadores usaban unidades de cuatro bits, **nibbles**. El motivo era, sencillamente, simplificar el acceso a la información por parte de los humanos. Para almacenar un dígito en base 10, que son los que usamos nosotros, se necesitan al menos cuatro dígitos binarios.

Veamos esto. Análogamente a lo que pasa con nuestros números en base 10, cada posición contando desde la derecha será dos veces mayor que la anterior. El primer dígito, desde la derecha, tiene peso 1, el segundo 2, el tercero 4, el cuarto 8, etc.

Estas relaciones de peso son potencias de la base de numeración elegida. En base 10 son potencias de 10: 10^0 (=1), 10^1 (=10), 10^2 (=100), 10^3 (=1000), 10^4 (=10000), etc. En base 2 son potencias de dos: 2^0 (=1), 2^1 (=2), 2^2 (=4), 2^3 (=8), 2^4 (=16), etc.

Así, con tres bits tenemos que el número mayor que podemos codificar es "111", es decir:

$$\begin{aligned} 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 &= \\ &= 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 4 + 2 + 1 = 7 \end{aligned}$$

Esto es insuficiente para codificar números entre 0 y 9, nos faltan dos valores, por lo tanto, necesitamos otro bit. Con cuatro podemos llegar hasta:

$$1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 =$$

$$= 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 8 + 4 + 2 + 1 = 15$$

Con esto, en realidad, nos sobran seis valores, pero podemos ignorarlos (de momento).

Esta forma de codificación se denomina BCD, es decir, Decimal Codificado en Binario. La tabla de valores BCD es la siguiente:

Tabla BCD

Código binario	Dígito decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Durante un tiempo, los ordenadores trabajaban con *palabras* de cuatro bits, siempre se ha llamado palabras a las agrupaciones de bits, y según la tecnología ha ido avanzando, las palabras han ido teniendo más bits.

Para simplificar y sobre todo, para conseguir mejores resultados, el acceso a la memoria también se hace mediante palabras. El procesador no accede a bits de memoria, sino a palabras, y la memoria se organiza por palabras. A cada una de esas palabras se le asigna una dirección: una *dirección de memoria*.

De modo que en un procesador de cuatro bits, cada dirección de memoria contiene un nibble, y en cada nibble se pueden almacenar un dígito decimal en la forma de dígitos BCD o una instrucción del procesador.

Entre los avances más destacados de los procesadores están el de usar la misma memoria para almacenar los datos y los programas. El procesador puede leer una instrucción de programa desde la memoria, ejecutarla, pasar a la siguiente instrucción, y repetir el proceso.

Siguiendo la misma idea, se puede acceder a varias direcciones contiguas de memoria para almacenar números mayores o instrucciones más complejas (16 instrucciones son pocas incluso para un ordenador primitivo), con dos nibbles tenemos posibilidad de almacenar 256 instrucciones.

Las palabras de cuatro bits se quedaron pequeñas pronto, y rápidamente se pasó a palabras y a procesadores de 8 bits. La unidad de 8 bits se conoce como octeto o *byte*, y este tamaño tuvo tanto éxito que sigue siendo hoy en día la unidad básica para acceder a memoria.

Procesadores posteriores usan palabras que agrupaban bytes, los de 16 bits usan dos bytes, los de 32 usan cuatro bytes, y los de 64 ocho bytes. Sin embargo, se sigue usando una dirección de memoria para cada byte.

Los primeros procesadores eran en realidad máquinas calculadoras programables, es decir, sólo manejaban números y se usaban para realizar cálculos numéricos complejos o repetitivos. Pero según fue aumentando la complejidad de las máquinas, surgieron nuevas posibilidades, como proporcionar salidas más cómodas para las personas.

Pero los ordenadores siguen manipulado sólo números, por lo que había que inventar algo para que pudieran manejar letras y palabras. La solución es simple, y consiste en codificar cada letra mediante un valor numérico. De este modo nació el código ASCII.

El primer código ASCII almacenaba cada letra en un byte, pero usaba sólo 7 bits, dejando el octavo para añadir un bit de control de errores, que permite detectar fallos en las transmisiones de datos. Este bit se llama bit de paridad, y funciona del modo siguiente:

Para cada código ASCII se suman los bits con valor 1, si se usa un control de paridad par la suma de los bits en cada byte debe ser par (contando el bit de paridad), si se usa un control de paridad impar, la suma debe ser impar.

De todos modos, con siete bits se pueden codificar 128 caracteres, que según las personas que diseñaron el código, eran más que suficientes. Esto permite codificar los 10 dígitos numéricos, los 25 caracteres del alfabeto inglés en mayúsculas, otros 25 para las minúsculas, 32 para caracteres destinados a símbolos, paréntesis, y signos de puntuación, el espacio. El resto, hasta 127 (33 caracteres), se usaron para codificar caracteres no imprimibles, que permiten formatear texto: retornos de línea, avances y retrocesos de caracteres, caracteres destinados a protocolos de transmisión, etc.

Tabla ASCII

Tabla ASCII correspondiente a los primeros 32 caracteres y al último. Estos son los no imprimibles:

Tabla ASCII (Caracteres no imprimibles)

Código binario	Código decimal	Abreviatura	Nombre
00000000	0	NUL	Caracter Nulo (NULL)
00000001	1	SOH	Inicio de Encabezado (Start Of Header)
00000010	2	STX	Inicio de Texto (Start Text)
00000011	3	ETX	Fin de Texto (End Text)
00000100	4	EOT	Fin de Transmisión (End Of Transmision)
00000101	5	ENQ	Pregunta (Enquiry)
00000110	6	ACK	Reconocimiento (Acknowledgement)
00000111	7	BEL	Timbre (Bell)
00001000	8	BS	Retroceso (Back Space)
00001001	9	HT	Tabulación horizontal (Horizontal Tab)

00001010	10	LF	Avance de Línea (Line feed)
00001011	11	VT	Tabulación Vertical (Vertical Tab)
00001100	12	FF	Salto de página (Form feed)
00001101	13	CR	Retorno de carro (Carriage return)
00001110	14	SO	Salida de turno (Shift Out)
00001111	15	SI	Entrada de turno (Shift In)
00010000	16	DLE	Escape de enlace de datos (Data Link Escape)
00010001	17	DC1	Device Control 1 — oft. XON
00010010	18	DC2	Device Control 2
00010011	19	DC3	Device Control 3 — oft. XOFF
00010100	20	DC4	Device Control 4
00010101	21	NAK	Reconocimiento negativo (Negative Acknowledgement)
00010110	22	SYN	Sincronismo sin usar (Synchronous Idle)
00010111	23	ETB	Fin de transmisión de bloque (End of Trans. Block)
00011000	24	CAN	Cancelar (Cancel)
00011001	25	EM	Fin de soporte (End of Medium)
00011010	26	SUB	Sustituto (Substitute)
00011011	27	ESC	Escape
00011100	28	FS	Separador de fichero (File Separator)
00011101	29	GS	Separador de grupo (Group Separator)
00011110	30	RS	Separador de registro (Record Separator)
00011111	31	US	Separador de unidad (Unit Separator)
01111111	127	DEL	Borrar (Delete)

Tabla ASCII correspondiente a los caracteres imprimibles:

Tabla ASCII (Caracteres imprimibles)

Binario	Decimal	Carácter	Binario	Decimal	Carácter	Binario	Decimal	Carácter
00100000	32	espacio	01000000	64	@	01100000	96	`
00100001	33	!	01000001	65	A	01100001	97	a
00100010	34	"	01000010	66	B	01100010	98	b
00100011	35	#	01000011	67	C	01100011	99	c
00100100	36	\$	01000100	68	D	01100100	100	d
00100101	37	%	01000101	69	E	01100101	101	e
00100110	38	&	01000110	70	F	01100110	102	f
00100111	39	'	01000111	71	G	01100111	103	g

00101000	40	(01001000	72	H	01101000	104	h
00101001	41)	01001001	73	I	01101001	105	i
00101010	42	*	01001010	74	J	01101010	106	j
00101011	43	+	01001011	75	K	01101011	107	k
00101100	44	,	01001100	76	L	01101100	108	l
00101101	45	-	01001101	77	M	01101101	109	m
00101110	46	.	01001110	78	N	01101110	110	n
00101111	47	/	01001111	79	O	01101111	111	o
00110000	48	0	01010000	80	P	01110000	112	p
00110001	49	1	01010001	81	Q	01110001	113	q
00110010	50	2	01010010	82	R	01110010	114	r
00110011	51	3	01010011	83	S	01110011	115	s
00110100	52	4	01010100	84	T	01110100	116	t
00110101	53	5	01010101	85	U	01110101	117	u
00110110	54	6	01010110	86	V	01110110	118	v
00110111	55	7	01010111	87	W	01110111	119	w
00111000	56	8	01011000	88	X	01111000	120	x
00111001	57	9	01011001	89	Y	01111001	121	y
00111010	58	:	01011010	90	Z	01111010	122	z
00111011	59	;	01011011	91	[01111011	123	{
00111100	60	<	01011100	92	\	01111100	124	
00111101	61	=	01011101	93]	01111101	125	}
00111110	62	>	01011110	94	^	01111110	126	~
00111111	63	?	01011111	95	_			

Las letras son números

Bueno, creo que ahora queda más claro por qué se puede usar un valor entero como un carácter o como un número. Dentro de un ordenador no existe diferencia entre ambas cosas.

El valor 65 puede ser un número entero, o, si se interpreta como un carácter, puede ser la letra 'A'.

Manejar signos

Pero aún nos queda un detalle importante: el signo. Cuando hemos hablado de variables de tipo **char** hemos comentado que pueden ser con y sin signo. Veamos cómo se las arregla el ordenador con los signos.

Lo primero que podemos decir del signo es que hay dos posibilidades: puede ser positivo o negativo. Esto parece hecho a la medida de un bit, de modo que podemos

usar un único bit para indicar el signo, de modo que un valor 0 indica que se trata de un número positivo y un 1 de un número negativo.

Esta es la primera solución, podemos usar el bit más a la izquierda para codificar el signo. Con un número de ocho bits, eso nos deja siete para codificar el valor absoluto, es decir, 127 valores posibles positivos y otros tantos negativos.

Pero esta solución tiene dos inconvenientes:

1. Tenemos dos codificaciones diferentes para el cero. No es un inconveniente pequeño, ya que dificulta las comparaciones y crea muchos casos particulares.
2. La aritmética se complica en cuanto a sumas y restas. Veremos que esto es sólo en comparación con otros sistemas de codificación.

Existe una solución mejor que mantiene la codificación del bit de signo, pero que elimina estos dos inconvenientes. Se trata de la codificación conocida como "complemento a dos".

Pero antes de ver qué es el complemento a dos, veamos qué es el complemento a uno. El complemento a uno consiste, en tomar un número en binario y cambiar los ceros por unos y los unos por ceros.

El complemento a uno de 01001010 es 10110101.

El complemento a dos consiste en hacer el complemento a uno y sumar 1 al resultado.

Ya sabemos sumar, espero, con números en base diez. Con números en base dos es igual de fácil. La tabla de sumar en binario es:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 10$, es decir: 0 y nos llevamos 1
- $1 + 1 + 1 = 11$, es decir 1 y nos llevamos 1

Por ejemplo:

```
  111111
  01101011
+ 00110110
-----
 10100001
```

Sigamos con el complemento a dos. Para el ejemplo anterior, el complemento a dos de 01001010 sería 10110101+1:

```
      1
  10110101
+ 00000001
-----
 10110110
```

Lo primero que vemos es que cuando se hace el complemento a dos de cualquier número, el signo cambia. Otra cosa que podemos ver es que el complemento a dos del cero es cero.

Complemento a uno de 00000000 es 11111111, y sumando 1 tenemos:

```
11111111
 11111111
+ 00000001
-----
100000000
```

El uno de la izquierda no cuenta, ya que sólo tenemos ocho bits, el noveno se pierde, y el resultado es cero.

Pero lo mejor de todo es que la suma de cualquier número con su complemento a dos es siempre cero:

```
11111111
 01001010
+ 10110110
-----
100000000
```

Esto es una gran ventaja, ya que podemos usar esta propiedad para decir que el complemento a dos equivale a cambiar el signo de un número, ya que la suma de un número y su complemento es cero, y el complemento de cero es cero.

Esto además nos da otra pequeña ventaja. Al tener una sólo combinación para el cero, tenemos un valor extra, de modo que el valor positivo máximo es 127 (01111111), pero para los negativos podemos llegar hasta el -128 (10000000).

Así, si usamos una variable **char** sin signo para almacenar números, podremos manejar valores entre 0 y 255. Si usamos variables **char** con signo, los valores posibles estarán entre -128 y 127.

Apéndice B: Palabras reservadas

A continuación se muestra una tabla con las palabras reservadas de C++, y los capítulos en los que se explica su uso:

Palabras reservadas C++.

Palabra	Capítulos donde se habla de ellos
asm	
auto	25
bool	2
break	5
case	5
catch	43
char	2
class	28
const	25, 33, 42
const_cast	42
continue	5
default	5
delete	12, 13
do	5
double	2
dynamic_cast	42
else	5
enum	2

explicit	
extern	3, 25
false	2
float	2
for	5
friend	31, 40
goto	5
if	5
inline	20, 32
int	2
long	2
mutable	25
namespace	26
new	12
operator	22, 35, 40
private	28, 36
protected	28, 36
public	28, 36
register	25
reinterpret_cast	42
return	5
short	2
signed	2
sizeof	4, 10
static	3, 25, 33
static_cast	42
struct	11
switch	5
template	40
this	31
throw	43
true	2

try	43
typedef	19
typeid	42
typename	42
union	16
unsigned	2
using	26
virtual	37, 38
void	2
volatile	25, 42
while	5

Palabras reservadas C.

Palabra	Capítulos donde se habla de ellos
auto	25
break	5
case	5
char	2
const	32
continue	5
default	5
do	5
double	2
else	5
enum	2
extern	3, 25
float	2
for	5
goto	5
if	5
int	2

long	2
register	25
return	5
short	2
signed	2
sizeof	12
static	3, 25, 32
struct	10
switch	5
typedef	19
union	16
unsigned	2
void	2
volatile	25, 42
while	5

Apéndice C: Bibliotecas estándar

Todos los compiladores C y C++ disponen de ciertas bibliotecas 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 bibliotecas. Nos hacen la vida más fácil, y no tendría sentido pasarlas por alto.

Existen muchas de estas bibliotecas, algunas tienen sus características definidas según diferentes estándares, 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 bibliotecas ANSI C y C++.

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

Biblioteca de entrada y salida fluidas "iostream"

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. Sintaxis:

```
cout << <variable/constante> [<< <variable/constante>...];
```

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 biblioteca 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>
using namespace std;

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;
}
```

Biblioteca C de entrada y salida estándar "stdio.h"

En esta biblioteca 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 biblioteca 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 estará 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 se 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.

Función printf()

Sintaxis:

```
int fprintf(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, límtate 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 y tipo de los 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. Estas cadenas controlan 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, mínimo número de caracteres a imprimir, se completa con espacios o ceros.
[anchura]	Opcional	Precisión, máximo número de caracteres.
[precisión]	Opcional	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

Para los tipos d, i, o, u, x, precisión por defecto.

.0 ó .

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	Al menos n dígitos serán impresos.
X	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 <i>far</i>
N	Puntero p s n	Un puntero <i>near</i>
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 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 `fprintf` se muestra la salida prevista:

```
#include <cstdio>
using namespace std;

int main() {
    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.

Biblioteca de rutinas de conversión estándar `stdlib.h`

En esta biblioteca 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.

Función `system()`

Ejecuta un comando del sistema o un programa externo almacenado en disco. Esta función nos será muy útil para detener el programa antes de que termine.

Si compilas los ejemplos, ejercicios o tus propios programas usando un compilador de Windows para consola, como Dev-C++, habrás notado que la consola se cierra cuando el programa termina, antes de que puedas ver los resultados del programa, para evitar eso podemos añadir una llamada a la función `system` para ejecutar el comando del sistema "pause", que detiene la ejecución hasta que se pulse una tecla. Por ejemplo:

```
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
{
    cout << "Hola, mundo." << endl;
    system("pause");
}
```

```
    return 0;
}
```

De este modo el programa se detiene antes de devolver el control y de que se cierre la consola.

Nota: en este capítulo se menciona el concepto macro.

Una macro es una fórmula definida para el preprocesador. Es un concepto que se estudia en los capítulos 13 y 23. Consiste, básicamente, en una fórmula que se sustituye por su valor antes de compilar un programa, de eso se encarga el preprocesador.

La diferencia principal es que las funciones se invocan, y la ejecución del programa continúa en la dirección de memoria de la función, cuando ésta termina, retorna y la ejecución continúa en el programa que invocó a la función. Con las macros no es así, el código de la macro se inserta por el preprocesador cada vez que es invocada y se compila junto con el resto del programa, eso elimina los saltos, y aumenta el código.

De todos modos, no te preocupes demasiado por estos conceptos, el curso está diseñado para tener una progresión más o menos lineal y estos temas se tratan con detalle más adelante.

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 biblioteca "`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 rand()

Generador de números aleatorios.

Sintaxis:

```
int rand(void);
```

La función `rand` devuelve un número aleatorio entre 0 y `RAND_MAX`. La constante `RAND_MAX` está definida en `stdlib.h`.

Valor de retorno:

`rand` devuelve un número entre 0 y `RAND_MAX`.

Función srand()

Inicializa el generador de números aleatorios.

Sintaxis:

```
void srand(unsigned semilla);
```

La función `srand` sirve para cambiar el origen del generador de números aleatorios.

Valor de retorno:

`srand` no devuelve ningún valor.

Biblioteca de tratamiento de caracteres `ctype.h`

En esta biblioteca contiene funciones y macros de tratamiento y clasificación de caracteres.

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 minúscula.

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**.

Macros **is<conjunto>()**

Las siguientes macros 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 macros 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)

isctrl (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 macros devolverá un valor distinto de cero si el argumento `c` pertenece al conjunto.

Biblioteca de manipulación de cadenas `string.h`

En esta biblioteca 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>
#include <cstring>
using namespace std;

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>
#include <cstring>
using namespace std;

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>
#include <cstring>
using namespace std;

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>
#include <cstring>
using namespace std;

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>
#include <cstring>
using namespace std;

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>
#include <cstring>
using namespace std;

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>
#include <cstring>
using namespace std;

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

    strcpy(cadena3, cadena1);
    cout >> strncat(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 carácter 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 extraído, o **NULL** cuando no quedan símbolos.

Ejemplo:

```
#include <cstring>
#include <iostream>
using namespace std;

int main() {
    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;  
}
```

Apéndice D: Trigramos y símbolos alternativos

Trigramos

La primera tarea del compilador, antes de ninguna otra, consiste en buscar y sustituir secuencias de trigramos, conjuntos de tres caracteres que empiezan por `??`. Esta capacidad existe para solventar el problema con teclados que no dispongan de ciertos caracteres muy usados en C++, como la `~`, o en ciertos países `{`, `}`, `[` o `]`.

Los trigramos son en esos casos de gran utilidad. Pero aunque no los necesitemos, es bueno conocerlos, ya que podemos ver listados de programas C++ con extraños símbolos, y debemos ser capaces de interpretarlos.

Sólo existen nueve de esos trigramos, así que tampoco es tan complicado:

Trigrafo	Sustitución
<code>??=</code>	<code>#</code>
<code>??/</code>	<code>\</code>
<code>??'</code>	<code>^</code>
<code>??(</code>	<code>[</code>
<code>??)</code>	<code>]</code>
<code>??!</code>	<code> </code>
<code>??<</code>	<code>{</code>
<code>??></code>	<code>}</code>
<code>??-</code>	<code>~</code>

Por ejemplo:


```

??=include <iostream>
using namespace std;

class class ??<
    public:
        clase();
        ??-clase();
    private:
        char *s;
??>;

int main()
??<
    char cad??(10??) = "hola";

    if(cad??(0??) == 'h' ??!?! cad??(1??) == 'o')
        cout << cad << endl;
    return 0;
??>

```

Se convierte en:

```

#include <iostream>
using namespace std;

class class {
    public:
        clase();
        ~clase();
    private:
        char *s;
};

int main()
{
    char cad[10] = "hola";

    if(cad[0] == 'h' || cad[1] == 'o')
        cout << cad << endl;
    return 0;
}

```

No todos los compiladores disponen de este mecanismo, por ejemplo, el compilador GCC incluido en las últimas versiones de Dev-C++ no lo tiene.

Símbolos alternativos

La utilidad de estos símbolos es parecida a la de los trigrafos: proporcionar alternativas para incluir ciertos caracteres que pueden no estar presentes en ciertos teclados.

Existen más de estos símbolos que de los trigrafos, y en general son más manejables y legibles, veamos la lista:

Alternativo	Primario
<%	{
%>	}
<:	[
:>]
%:	#
%:%:	##
and	&&
bitor	
or	
xor	^
compl	~
bitand	&
and_eq	&=
or_eq	=
xor_eq	^=
not	!
not_eq	!=

Veamos el mismo ejemplo anterior:

```
%:include <iostream>
```

```

using namespace std;

class clase <%
public:
    clase();
    compl clase();
private:
    char *s;
%>;

int main()
<%
    char cad<:10:> = "hola";

    if(cad<:0:> == 'h' or cad<:1:> == 'o') cout << cad <<
endl;
    return 0;
%>

```

Se convierte en:

```

#include <iostream>
using namespace std;

class clase {
public:
    clase();
    ~clase();
private:
    char *s;
};

int main()
{
    char cad[10] = "hola";

    if(cad[0] == 'h' || cad[1] == 'o') cout << cad << endl;
    return 0;
}

```

En este caso, Dev-C++ sí dispone de estos símbolos, y algunos pueden añadir claridad al código, como usar **or** ó **and**, pero no

deberían usarse para entorpecer la lectura del programa, por muy tentador que esto sea. ;-)

Apéndice E Streams

Las operaciones de entrada y salida nunca formaron parte de C ni tampoco lo forman de C++. En ambos lenguajes, todas las operaciones de entrada y salida se hacen mediante bibliotecas externas.

En el caso de C, esa biblioteca es `stdio`, que agrupa todas las funciones de entrada y salida desde teclado, pantalla y ficheros de disco. En el caso de C++ se dispone de varias clases: *streambuf*, *ios*, *istream*, *ostream* y *fstream*.

Dejar fuera del lenguaje todas las operaciones de entrada y salida tiene varias ventajas:

1. Independencia de la plataforma: cada compilador dispone de diferentes versiones de cada biblioteca para cada plataforma. Tan sólo se cambia la definición de las clases y bibliotecas, pero la estructura, parámetros y valores de retorno son iguales. Los mismos programas, compilados para diferentes plataformas funcionan del mismo modo.
2. Encapsulación: para el programa todos los dispositivos son de entrada y salida se tratan del mismo modo, es indiferente usar la pantalla, el teclado o ficheros.
3. Buffering: el acceso a dispositivos físicos es lento, en comparación con el acceso a memoria. Las operaciones de lectura y escritura se agrupan, haciéndolas en memoria, y las operaciones físicas se hacen por grupos o bloques, lo cual ahorra mucho tiempo.

Clases predefinidas para streams

Un *stream* es una abstracción para referirse a cualquier flujo de datos entre una fuente y un destinatario. Los streams se encargan de convertir cualquier tipo de objeto a texto legible por el usuario, y viceversa. Pero no se limitan a eso, también pueden hacer manipulaciones binarias de los objetos y cambiar la apariencia y el formato en que se muestra la salida.

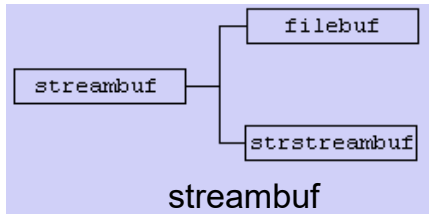
C++ declara varias clases estándar para el manejo de *streams*:

- *streambuf*: manipulación de buffers.
- *ios*: entradas y salidas, incluye en su definición un objeto de la clase *streambuf*.
- *istream*: derivada de *ios*, clase especializada en entradas.
- *ostream*: derivada de *ios*, clase especializada en salidas.

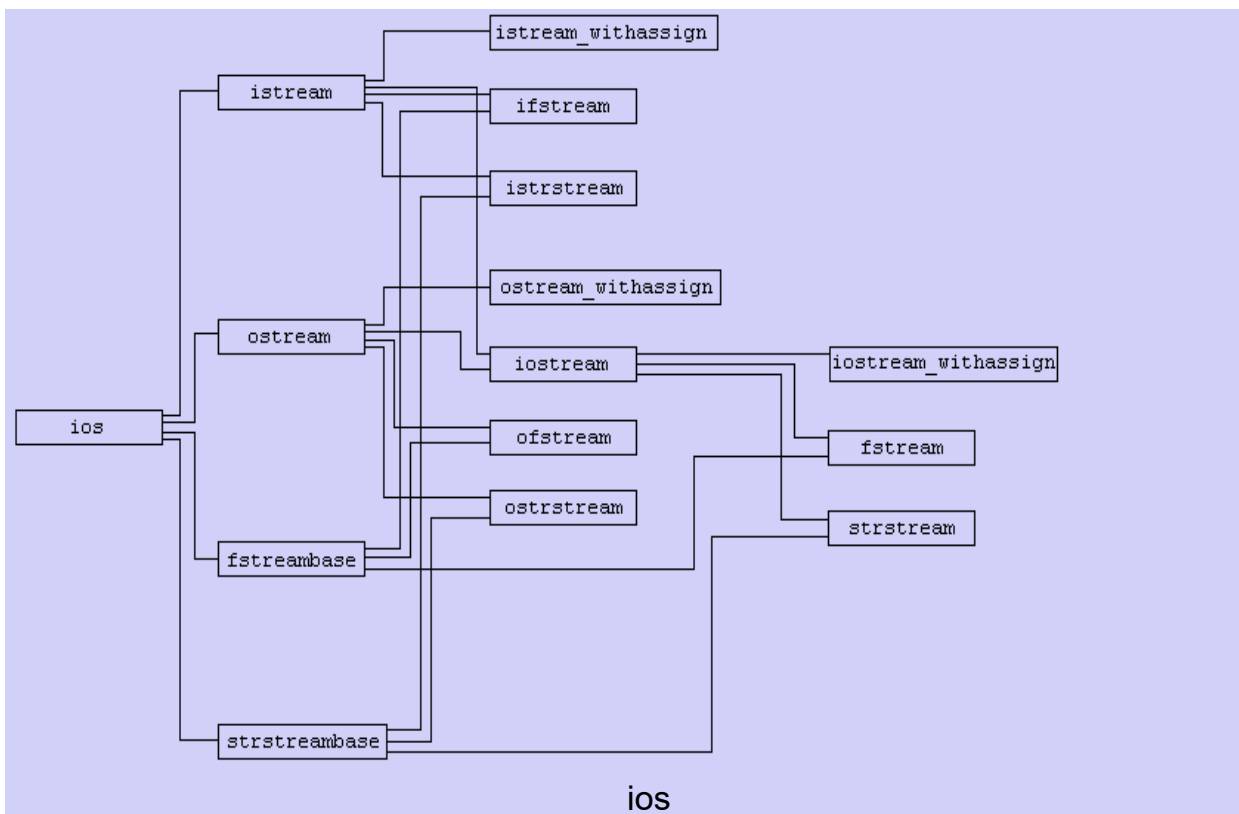
- *iostream*: derivada de *istream* y *ostream*, se encarga de encapsular las funciones de entrada y salida por teclado y pantalla.
- *fstream*: entrada y salida desde ficheros.

Las clases base son *streambuf* e *ios*, las demás se derivan de estas dos.

La clase *streambuf* proporciona un interfaz entre la memoria y los dispositivos físicos.



La clase *ios* contiene además un puntero a un objeto de la clase *streambuf*. Proporciona soporte para entradas y salidas con validación y formato usando un *streambuf*.



Veremos ahora algunas de las funciones que forman parte de las principales clases relacionadas con los *streams*.

No es necesario estudiar en profundidad estas clases, puede usarse este capítulo como consulta para el uso de *streams*. Con la práctica se aprende a usar las funciones necesarias en cada caso.

Clase *streambuf*

Es la clase base para todas las clases con buffer, proporciona el interfaz entre los datos y las áreas de almacenamiento como la memoria o los dispositivos físicos.

Si las aplicaciones necesitan acceder al buffer de un *stream*, lo hacen a través del puntero almacenado en la clase *ios*. Pero normalmente el acceso se hace a alto nivel, directamente desde funciones de *ios* y sus clases derivadas, y casi nunca directamente a través de *streambuf*.

Por eso veremos muy pocas funciones de la clase *streambuf*, ya que la mayoría tienen escasa utilidad en programación normal.

Por otra parte, he consultado bastante documentación al respecto de las estructuras de las clases, y varias implementaciones de distintos compiladores, y no parece existir gran unanimidad al respecto de las funciones que deben incluir ciertas clases. El caso de *streambuf* es de los más heterogéneos, de modo que sólo incluiré algunas de las funciones más frecuentes.

Funciones protegidas

Función *allocate* (no siempre disponible)

```
int allocate();
```

Prepara el área del buffer.

Función *base* (no siempre disponible)

```
char *base();
```

Devuelve la dirección de comienzo del área del buffer.

Función *blen* (no siempre disponible)

```
int blen();
```

Devuelve la longitud del área del buffer.

Función *unbuffered* (no siempre disponible)

```
void unbuffered(int);  
int unbuffered();
```

La primera forma modifica el estado del buffer, la segunda devuelve un valor no nulo si no está activado el buffer.

Funciones públicas

Función `in_avail`

```
int in_avail();
```

Devuelve el número de caracteres que permanecen en el buffer de entrada interno disponibles para su lectura.

Función `out_waiting`

```
int out_waiting();
```

Devuelve el número de caracteres que permanecen en el buffer interno de salida.

Función `seekoff`

```
virtual streampos seekoff(streamoff offset,  
ios::seek_dir, int mode);
```

Cambia la posición relativa del puntero del fichero desde el punto definido por `seek_dir`, el valor de `offset`.

Para `seek_dir` se usan los valores definidos en el enum de la clase `ios`:

Valor	Significado
<code>ios::beg</code>	Desplazamiento desde el principio del fichero
<code>ios::cur</code>	Desplazamiento desde la posición actual del puntero
<code>ios::end</code>	Desplazamiento desde el final del fichero

El valor de `offset` puede ser positivo o negativo, si es negativo, el desplazamiento es en la dirección del principio del fichero.

El parámetro *mode* especifica que el movimiento puede ser en el área de entrada, salida o ambos, especificado por *ios::in*, *ios::out* o los dos.

Se trata de una función virtual, cuando se redefine en clases derivadas, puede funcionar con respecto al *stream*, y no sobre el buffer interno de *streambuf*.

Función seekpos

```
virtual streampos seekpos(streampos,
    int = (ios::in | ios::out));
```

Cambia o lee la posición del puntero del buffer interno de *streambuf* a una posición absoluta *streampos*.

También es una función virtual, de modo que puede ser redefinida en clases derivadas para modificar la posición en un *stream* de entrada o salida.

Función setbuf

```
streambuf* setbuf(unsigned char*, int);
```

Especifica el *array* para ser usado como buffer interno.

Función sgetc

```
int sgetc();
```

Toma el siguiente carácter del buffer interno de entrada.

Función sgetn

```
int sgetn(char*, int n);
```

Toma los siguientes n caracteres del buffer interno de entrada.

Función snextc

```
int snextc();
```

Avanza y toma el siguiente carácter del buffer interno de entrada.

Función `sputbackc`

```
int sputbackc(char);
```

Devuelve un carácter al buffer de entrada interno.

Función `sputc`

```
int sputc(int);
```

Coloca un carácter en el buffer de salida interno.

Función `sputn`

```
int sputn(const char*, int n);
```

Coloca n caracteres en el buffer de salida interno.

Función `stossc`

```
void stossc();
```

Avanza al siguiente carácter en el buffer de entrada interno.

Clase `ios`

La clase `ios` está diseñada para ser la clase base de otras clases derivadas como *istream*, *ostream*, *iostream*, *fstreambase* y *strstreambase*. Proporciona operaciones comunes de entrada y salida

Enums

Dentro de la clase *ios* se definen varios tipos enumerados que son útiles para modificar flags y opciones o para el tratamiento de errores o estados de un *stream*.

Todos los miembros de *enums* definidos en la clase *ios* son accesibles mediante el operador de ámbito. Por ejemplo:

```
ios::eofbit  
ios::in  
ios::beg  
ios::uppercase
```

io_state

```
enum io_state { goodbit, eofbit, failbit, badbit };
```

open_mode

```
enum open_mode { in, out, ate, app, trunc, nocreate,  
               noreplace, binary };
```

seek_dir

```
enum seek_dir { beg, cur, end };
```

Flags de modificadores de formato

```
enum { skipws, left, right, internal,  
      dec, oct, hex, showbase, showpoint,  
      uppercase, showpos, scientific,  
      fixed, unitbuf, stdio };
```

Máscaras de modificadores

Permiten trabajar con grupos de modificadores afines.

```
enum {  
    basefield = dec+oct+hex,  
    floatfield = scientific+fixed,  
    adjustfield = left+right+internal  
};
```

Funciones

No nos interesan todas las funciones de las clases que vamos a estudiar, algunas de ellas raramente las usaremos, y en general son de poca o ninguna utilidad.

Función bad

```
int bad();
```

Devuelve un valor distinto de cero si ha ocurrido un error.

Sólo se comprueba el bit de estado *ios::badbit*, de modo que esta función no equivale a *!good()*.

Función clear

```
void clear(iostate state=0);
```

Sirve para modificar los bits de estado de un *stream*, normalmente para eliminar un estado de error. Se suelen usar constantes definidas en el enum *io_state* definido en *ios*, usando el operador de bits OR para modificar varios bits a la vez.

Función eof

```
int eof();
```

Devuelve un valor distinto de cero si se ha alcanzado el fin de fichero.

Esta función únicamente comprueba el bit de estado *ios::eofbit*.

Función fail

```
int fail();
```

Devuelve un valor distinto de cero si una operación sobre el *stream* ha fallado.

Comprueba los bits de estado *ios::badbit* y *ios::failbit*.

Función fill

Cambia el carácter de relleno que se usa cuando la salida es más ancha de la necesaria para el dato actual.

```
int fill();  
int fill(char);
```

La primera forma devuelve el valor actual del carácter de relleno, la segunda permite cambiar el carácter de relleno para las siguientes salidas, y también devuelve el valor actual.

Ejemplo:

```
int x = 23;  
cout << "|";  
cout.width(10);  
cout.fill('%');  
cout << x << "|" << x << "|" << endl;
```

Función flags

Permite cambiar o leer los flags de manipulación de formato.

```
long flags () const;  
long flags (long valor);
```

La primera forma devuelve el valor actual de los flags.

La segunda cambia el valor actual por valor, el valor de retorno es el valor previo de los flags.

Ejemplo:

```
int x = 235;  
long f;  
  
cout << "|";  
f = flags();  
f &= !(ios::adjustfield);  
f |= ios::left;  
cout.flags(f);  
cout.width(10);  
cout << x << "|" << endl;
```

Función good

```
int good();
```

Devuelve un valor distinto de cero si no ha ocurrido ningún error, es decir, si ninguno de los bits de estado está activo.

Aunque pudiera parecerlo, (*good* significa bueno y *bad* malo, en inglés) esta función no es exactamente equivalente a *!bad()*.

En realidad es equivalente a *rdstate() == 0*.

Función precision

Permite cambiar el número de caracteres significativos que se mostrarán cuando trabajemos con números en coma flotante: float o double.

```
int precision();  
int precision(char);
```

La primera forma devuelve el valor actual de la precisión, la segunda permite modificar la precisión para las siguientes salidas, y también devuelve el valor actual.

```
float x = 23.45684875;  
  
cout << "|";  
cout.precision(6);  
cout << x << "|" << x << "|" << endl;
```

Función rdbuf

```
streambuf* rdbuf();
```

Devuelve un puntero al *streambuf* asignado a este *stream*.

Función rdstate

```
int rdstate();
```

Devuelve el estado del *stream*. Este estado puede ser una combinación de cualquiera de los bits de estado definidos en el enum *ios::io_state*, es decir *ios::badbit*, *ios::eofbit*, *ios::failbit* e *ios::goodbit*. El *goodbit* no es en realidad un bit, sino la ausencia de todos los demás. De modo que para verificar que el valor obtenido por *rdstate* es *ios::goodbit* tan sólo hay que comparar. En cualquier caso es mejor usar la función *good()*.

En cuanto a los restantes bits de estado se puede usar el operador *&* para verificar la presencia de cada uno de los bits. Aunque de nuevo, es preferible usar las funciones *bad()*, *eof()* o *fail()*.

Función *setf*

Permite modificar los flags de manipulación de formato.

```
long setf(long);  
long setf(long valor, long mascara);
```

La primera forma activa los flags que estén activos tanto en el parámetro y deja sin cambios el resto.

La segunda forma activa los flags que estén activos tanto en valor como en máscara y desactiva los que estén activos en *mask*, pero no en valor. Podemos considerar que *mask* contiene activos los flags que queremos modificar y *valor* los flags que queremos activar.

Ambos devuelven el valor previo de los flags.

```
int x = 235;  
  
cout << "|";  
cout.setf(ios::left, ios::left |  
          ios::right | ios::internal);  
cout.width(10);  
cout << x << "|" << endl;
```

Función *tie*

Algunos *streams* de entrada están enlazados a otros. Cuando un *stream* de entrada tiene caracteres que deben ser leídos, o un *stream* de salida necesita más caracteres, el buffer del fichero enlazado se vacía automáticamente.

Por defecto, *cin*, *err* y *clog* están enlazados con *cout*. Cuando se usa *cin*, el buffer de *cout* se vacía automáticamente.

```
ostream* tie();  
ostream* tie(ostream* val);
```

La primera forma devuelve el *stream* (enlazado), o cero si no existe. La segunda enlaza otro *stream* al actual y devuelve el previo, si existía.

Función unsetf

Permite eliminar flags de manipulación de formato:

```
void unsetf(long mascara);
```

Desactiva los flags que estén activos en el parámetro.

Nota:

En algunos compiladores he comprobado que esta función tiene como valor de retorno el valor previo de los flags.

```
int x = 235;  
  
cout << "|";  
cout.unsetf(ios::left | ios::right | ios::internal);  
cout.setf(ios::left);  
cout.width(10);  
cout << x << "|" << endl;
```

Función width

Cambia la anchura en caracteres de la siguiente salida de *stream*:

```
int width();  
int width(int);
```

La primera forma devuelve el valor de la anchura actual, la segunda permite cambiar la anchura para las siguientes salidas, y también devuelve el valor actual de la anchura.


```
int x = 23;

cout << "#";
cout.width(10);
cout << x << "#" << x << "#" << endl;
```

Función xalloc

```
static int xalloc();
```

Devuelve un índice del array de las palabras no usadas que pueden ser utilizadas como flags de formatos definidos por el usuario.

Función init (protegida)

```
void init(streambuf *);
```

Asocia el objeto de la clase ios con el *streambuf* especificado.

Función setstate (protegida)

```
void setstate(int);
```

Activa los bits de estado seleccionados. El resto de los bits de estado no se ven afectados, si llamamos a `setstate(ios::eofbit)`, se añadirá ese bit, pero no se eliminarán los bits *ios::badbit* o *ios::failbit* si ya estaban activos.

Clase filebuf

La declaración de esta clase está en el fichero `fstream`.

Esta clase se base en la clase `streambuf`, y le proporciona las funciones necesarias para manipular entrada y salida de caracteres en ficheros.

Las funciones de entrada y salida de `istream` y `ostream` hacen llamadas a funciones de `filebuf`, mediante el puntero a `filebuf` que existe en la clase `ios`.

Constructores

```
filebuf::filebuf();  
filebuf::filebuf(int fd);  
filebuf::filebuf(int fd, char *, int n);
```

La primera forma crea un filebuf que no está asociado a ningún fichero.

La segunda forma crea un filebuf asociado a un fichero mediante el descriptor *fd*.

La tercera forma crea un filebuf asociado a un fichero especificado mediante el descriptor *fd* y asociado a un buffer *buf* de tamaño *n* bytes. Si *n* es cero o negativo, el filebuf es sin buffer.

Nota:

He comprobado que algunos compiladores sólo disponen de la primera versión del constructor.

Funciones

Función *attach* (no siempre disponible)

```
filebuf* attach(int fd);
```

El filebuf debe estar cerrado, esta función asocia el filebuf a otro fichero especificado mediante el descriptor *fd*.

Función *close*

```
filebuf* close();
```

Actualiza la información actualmente en el buffer y cierra el fichero. En caso de error, retorna el valor 0.

Función *fd* (no siempre disponible)

```
int fd();
```

Devuelve el descriptor de fichero o EOF.

Función `is_open`

```
int is_open();
```

Si el fichero está abierto devuelve un valor distinto de cero.

Función `open`

```
filebuf* open(const char *name, int mode,  
int prot = filebuf::openprot);
```

Abre un fichero para un objeto de una clase específica, con el nombre *name*. Para el parámetro *mode* se puede usar el enum `open_mode` definido en la clase `ios`.

Parámetro <code>mode</code>	Efecto
<code>ios::app</code>	(append) Se coloca al final del fichero antes de cada operación de escritura.
<code>ios::ate</code>	(at end) Se coloca al final del stream al abrir el fichero.
<code>ios::binary</code>	Trata el stream como binario, no como texto.
<code>ios::in</code>	Permite operaciones de entrada en un stream.
<code>ios::out</code>	Permite operaciones de salida en un stream.
<code>ios::trunc</code>	(truncate) Trunca el fichero a cero al abrirlo.

El parámetro *prot* se corresponde con el permiso de acceso DOS y es usado siempre que no se indique el modo `ios::nocreate`. Por defecto se usa el permiso para leer y escribir. En algunas versiones de las bibliotecas de streams no existe este parámetro, ya que está íntimamente asociado al sistema operativo.

Función `overflow`

```
virtual int overflow(int = EOF);
```

Vacía un buffer a su destino. Todas las clases derivadas deben definir las acciones necesarias a realizar.

Función seekoff

```
virtual streampos seekoff(streamoff offset,  
    ios::seek_dir, int mode);
```

Mueve el cursor del fichero a una posición relativa *offset* a la posición actual en la dirección indicada por *seek_dir*.

Para *seek_dir* se usan los valores definidos en el enum *seek_dir* de la clase *ios*.

Valor

Significado

ios::beg Desplazamiento desde el principio del fichero

ios::cur Desplazamiento desde la posición actual del puntero

ios::end Desplazamiento desde el final del fichero

Cuando se especifica un valor negativo como desplazamiento, éste se hace en la dirección del comienzo del fichero desde la posición actual o desde el final del fichero.

El parámetro *mode* indica el tipo de movimiento en el área de entrada o salida del buffer interno mediante los valores *ios::in*, *ios::out* o ambos.

Cuando esta función virtual se redefine en una clase derivada, debe desplazarse en el stream, y no en el buffer del miembro *streambuffer* interno.

Función setbuf

```
virtual streambuf* setbuf(char*, int);
```

Especifica un buffer del tamaño indicado para el objeto. Cuando se usa como un *strstreambuf* y la función se sobrecarga, el primer argumento no tiene sentido, y debe ponerse a cero.

Función sync

```
virtual int sync();
```

Sincroniza las estructuras de datos internas y externas del stream.

Función underflow

```
virtual int underflow();
```

Hace que la entrada esté disponible. Se llama a esta función cuando no hay más datos disponibles en el buffer de entrada. Todas las clases derivadas deben definir las acciones a realizar.

Clase istream

La declaración de esta clase está en el fichero `istream.h`.

Proporciona entrada con y sin formato desde una clase derivada de `streambuf` via `ios::bp`.

El operador `>>` está sobrecargado para todos los tipos fundamentales, y puede formatear los datos.

La clase `istream` proporciona el código genérico para formatear los datos después de que son extraídos desde el stream de entrada.

Constructor

```
istream(streambuf *);
```

Asocia una clase derivada dada de `streambuf` a la clase que proporciona un stream de entrada. Esto se hace asignando `ios::bp` al parámetro del constructor.

Función `rdbuf` (protegida)

```
void eatwhite();
```

Extrae espacios en blanco consecutivos.

Función `gcount`

```
int gcount();
```

Devuelve el número de caracteres sin formato de la última lectura. Las lecturas sin formato son las realizadas mediante las funciones `get`, `getline` y `read`.

Función get

```
int get();  
istream& get(char*, int len, char = '\n');  
istream& get(char&);  
istream& get(streambuf&, char = '\n');
```

La primera forma extrae el siguiente carácter o EOF si no hay disponible ninguno.

La segunda forma extrae caracteres en la dirección proporcionada en el parámetro `char*` hasta que se recibe el delimitador del tercer parámetro, el fin de fichero o hasta que se leen `len-1` bytes. Siempre se añade un carácter nulo de terminación en la cadena de salida. El delimitador no se extrae desde el stream de entrada. La función sólo falla si no se extraen caracteres.

La tercera forma extrae un único carácter en la referencia a `char` proporcionada.

La cuarta forma extrae caracteres en el `streambuf` especificado hasta que se encuentra el delimitador.

Función getline

```
istream& getline(char*, int, char = '\n');
```

Extrae caracteres hasta que se encuentra el delimitador y los coloca en el buffer, elimina el delimitador del stream de entrada y no lo añade al buffer.

Función ignore

```
istream& ignore(int n = 1, int delim = EOF);
```

Hace que los siguientes `n` caracteres en el stream de entrada sean ignorados; la extracción se detiene antes si se encuentra el delimitador `delim`.

El delimitador también es extraído del stream.

Función ipfx

```
istream& ipfx(int n = 0);
```

Esta función es previamente llamada por las funciones de entrada para leer desde un stream de entrada. Las funciones que realizan entradas con formato la llaman como `ipfx(0)`; las que realizan entradas sin formato la llaman como `ipfx(1)`.

Función peek

```
int peek();
```

Devuelve el siguiente carácter sin extraerlo del stream.

Función putback

```
istream& putback(char);
```

Devuelve un carácter al stream.

Función read

```
istream& read(char*, int);
```

Extrae el número indicado de caracteres en el array `char*`. Se puede usar la función `gcount()` para saber el número de caracteres extraídos si ocurre algún error.

Función seekg

```
istream& seekg(streampos pos);  
istream& seekg(streamoff offset, seek_dir dir);
```

La primera forma se mueve a posición absoluta, tal como la proporciona la función `tellg`.

La segunda forma se mueve un número *offset* de bytes la posición del cursor del stream relativa a `dir`. Este parámetro puede tomar los valores definidos en el enum `seek_dir`: `{beg, cur, end}`;

Para streams de salida usar `ostream::seekp`.

Usar `seekpos` o `seekoff` para moverse en un buffer de un stream.

Función tellg

```
long tellg();
```

Devuelve la posición actual del stream.

Clase ostream

La declaración de esta clase está en el fichero iostream.h.

Proporciona salida con y sin formato a un streambuf.

Un objeto de la clase ostream no producirá la salida actual, pero sus funciones miembro pueden llamar a las funciones miembro de la clase apuntada por bp para insertar caracteres en el stream de salida.

El operador << da formato a los datos antes de enviarlos a bp.

La clase ostream proporciona el código genérico para formatear los datos antes de que sean insertados en el stream de salida.

Constructor

```
ostream(streambuf *buf);
```

Asocia el streambuf dado a la clase, proporcionando un stream de salida. Esto se hace asignando el puntero ios::bp a *buf*.

Función flush

```
ostream& flush();
```

Vacía el buffer asociado al stream. Procesa todas las salidas pendientes.

Función opfx

```
int opfx();
```

Esta función es llamada por funciones de salida para antes de hacer una inserción en un stream de salida. Devuelve cero si el ostream tiene un estado de error distinto de cero. En caso contrario, opfx devuelve un valor distinto de cero.

Función osfx

```
void osfx();
```

Realiza operaciones de salida (post?). Si está activado `ios::unitbuf`, `osfx` vacía el buffer de `ostream`. En caso de error, `osfx` activa el flag `ios::failbit`.

Función put

```
ostream& put(char ch);
```

Inserta un carácter en el stream de salida.

Función seekp

```
ostream& seekp(streampos);  
ostream& seekp(streamoff, seek_dir);
```

La primera forma mueve el cursor del stream a una posición absoluta, tal como la devuelve la función `tellp`.

La segunda forma mueve el cursor a una posición relativa desde el punto indicado mediante el parámetro `seek_dir`, que puede tomar los valores del enum `seek_dir`: *beg*, *cur*, *end*.

Función tellp

```
streampos tellp();
```

Devuelve la posición absoluta del cursor del stream.

Función write

```
ostream& write(const char*, int n);
```

Inserta *n* caracteres (aunque sean nulos) en el stream de salida.

Clase ostream

La declaración de esta clase está en el fichero `iostream.h`.

Esta clase está derivada de `istream` y `ostream`, es una mezcla de sus clases base, y permite realizar tanto entradas como salidas en un stream. Además es la base para otras clases como `fstream` y `strstream`.

El stream se implementa mediante la clase `ios::bp` a la que apunta. Dependiendo del tipo de clase derivada a la que apunta `bp`, se determina si los streams de entrada y salida pueden ser el mismo.

Por ejemplo, si `iostream` usa un `filebuf` podrá hacer entradas y salidas en el mismo fichero. Si `iostream` usa un `strstreambuf` podrá hacer entradas y salidas en la misma o en diferentes zonas de memoria.

Constructor

```
iostream(streambuf *);
```

Asocia el `streambuf` dado a la clase.

Clase `fstreambase`

La declaración de esta clase está en el fichero `fstream`.

Esta clase proporciona acceso a funciones de `filebuf` inaccesibles a través de `ios::bp` tanto para `fstreambase` como para sus clases derivadas.

La una función miembro de `filebuf` no es un miembro virtual de la clase base `filebuf` (`streambuf`), ésta no será accesible. Por ejemplo: *attach*, *open* y *close* no lo son.

Los constructores de `fstreambase` inicializan el dato `ios::bp` para que apunte al `filebuf`.

Constructores

```
fstreambase();  
fstreambase(const char *name,  
             int mode, int = filebuf::openprot);  
fstreambase(int fd);  
fstreambase(int fd, char *buf, int len);
```

La primera forma crea un `fstreambase` que no está asociando a ningún fichero.

La segunda forma crea un `fstreambase`, abre el fichero especificado por *name* en el modo especificado por *mode* y lo conecta a ese fichero.

La tercera forma crea un `fstreambase` y lo conecta a un descriptor de fichero abierto y especificado por *fd*.

La cuarta forma crea un `fstreambase` y lo conecta a un descriptor de fichero abierto especificado por *fd* y usando un buffer especificado por *buf* con el tamaño indicado por *len*.

Función `attach`

```
void attach(int);
```

Conecta con un descriptor de fichero abierto.

Función `close`

```
void close();
```

Cierra el `filebuf` y el fichero asociados.

Función `open`

```
void open(const char *name, int mode,
          int prot=filebuf::openprot);
```

Abre un fichero para el objeto especificado.

Para el parámetro `mode` se pueden usar los valores del enum `open_mode` definidos en la clase `ios`.

Clase	Parámetro mode
<code>fstream</code>	<code>ios::in</code>
<code>ofstream</code>	<code>ios::out</code>

El parámetro *prot* se corresponde con el permiso de acceso DOS, y se usa salvo que se use el valor `ios::nocreate` para el parámetro *mode*. Por defecto se usa el valor de permiso de lectura y escritura.

Función `rdbuf`

```
filebuf* rdbuf();
```

Devuelve el buffer usado.

Función `setbuf`

```
void setbuf(char*, int);
```

Asigna un buffer especificado por el usuario al `filebuf`.

Clase `ifstream`

La declaración de esta clase está en el fichero `fstream.h`.

Proporciona un stream de entrada para leer desde un fichero usando un `filebuf`.

Constructores

```
ifstream();  
ifstream(const char *name, int mode = ios::in,  
         int = filebuf::openprot);  
ifstream(int fd);  
ifstream(int fd, char *buf, int buf_len);
```

La primera forma crea un `ifstream` que no está asociando a ningún fichero.

La segunda forma crea un `ifstream`, abre un fichero de entrada en modo protegido y se conecta a él. El contenido del fichero, si existe, se conserva; los nuevos datos escritos se añaden al final. Por defecto, el fichero no se crea si no existe.

La tercera forma crea un `ifstream`, y lo conecta a un descriptor de un fichero *fd* abierto previamente.

La cuarta forma crea un `ifstream` conectado a un fichero abierto especificado mediante su descriptor, *fd*. El `ifstream` usa el buffer especificado por *buf* de longitud *buf_len*.

Función `open`

```
void open(const char *name, int mode,  
         int prot=filebuf::openprot);
```

Abre un fichero para el objeto especificado.

Para el parámetro *mode* se pueden usar los valores del enum *open_mode* definidos en la clase *ios*.

Clase	Parámetro mode
<i>fstream</i>	<i>ios::in</i>
<i>ofstream</i>	<i>ios::out</i>

El parámetro *prot* se corresponde con el permiso de acceso DOS, y se usa salvo que se use el valor *ios::nocreate* para el parámetro *mode*. Por defecto se usa el valor de permiso de lectura y escritura.

Función *rdbuf*

```
filebuf* rdbuf();
```

Devuelve el buffer usado.

Clase *ofstream*

La declaración de esta clase está en el fichero *fstream*.

Proporciona un stream de salida para escribir a un fichero usando un *filebuf*.

Constructores

```
ofstream();  
ofstream(const char *name, int mode = ios::out,  
         int = filebuf::openprot);  
ofstream(int fd);  
ofstream(int fd, char *buf, int buf_len);
```

La primera forma crea un *ofstream* que no está asociando a ningún fichero.

La segunda forma crea un *ofstream*, abre un fichero de salida y se conecta a él.

La tercera forma crea un *ofstream*, y lo conecta a un descriptor de un fichero *fd* abierto previamente.

La cuarta forma crea un *ofstream* conectado a un fichero abierto especificado mediante su descriptor, *fd*. El *ofstream* usa el buffer especificado por *buf* de longitud *buf_len*.

Función *open*

```
void open(const char *name, int mode,
          int prot=filebuf::openprot);
```

Abre un fichero para el objeto especificado.

Para el parámetro *mode* se pueden usar los valores del enum *open_mode* definidos en la clase *ios*.

Clase	Parámetro mode
<i>fstream</i>	<i>ios::in</i>
<i>ofstream</i>	<i>ios::out</i>

El parámetro *prot* se corresponde con el permiso de acceso DOS, y se usa salvo que se use el valor *ios::nocreate* para el parámetro *mode*. Por defecto se usa el valor de permiso de lectura y escritura.

Función *rdbuf*

```
filebuf* rdbuf();
```

Devuelve el buffer usado.

Clase *fstream*

La declaración de esta clase está en el fichero *fstream.h*.

Proporciona un stream de salida y salida a un fichero usando un *filebuf*.

La entrada y la salida se inicializan usando las funciones de las clases base *istream* y *ostream*. Por ejemplo, *fstream* puede usar la función *istream::getline()* para extraer caracteres desde un fichero.

Constructores

```
fstream();
fstream(const char *name, int mode = ios::in,
         int = filebuf::openprot);
fstream(int fd);
fstream(int fd, char *buf, int buf_len);
```

La primera forma crea un *fstream* que no está asociando a ningún fichero.

La segunda forma crea un *fstream*, abre un fichero con el acceso especificado por *mode* y se conecta a él.

La tercera forma crea un `fstream`, y lo conecta a un descriptor de un fichero `fd` abierto previamente.

La cuarta forma crea un `fstream` conectado a un fichero abierto especificado mediante su descriptor, `fd`. El `fstream` usa el buffer especificado por `buf` de longitud `buf_len`. Si `buf` es `NULL` o `n` no es positivo, el `fstream` será sin buffer.

Función `open`

```
void open(const char *name, int mode,
          int prot=filebuf::openprot);
```

Abre un fichero para el objeto especificado.

Para el parámetro `mode` se pueden usar los valores del enum `open_mode` definidos en la clase `ios`.

Clase	Parámetro <code>mode</code>
<code>fstream</code>	<code>ios::in</code>
<code>ofstream</code>	<code>ios::out</code>

El parámetro `prot` se corresponde con el permiso de acceso DOS, y se usa salvo que se use el valor `ios::nocreate` para el parámetro `mode`. Por defecto se usa el valor de permiso de lectura y escritura.

Función `rdbuf`

```
filebuf* rdbuf();
```

Devuelve el buffer usado.

Clase `strstreambuf`

La declaración de esta clase está en el fichero `strstrea.h`.

Clase base para especializar la clase `ios` para el manejo de streams de cadenas, esto se consigue inicializando `ios::bp` de modo que apunte a un objeto `strstreambuf`. Esto proporciona las comprobaciones necesarias para cualquier operación de entrada y salida de cadenas en memoria. Por ese motivo, la clase `strstreambase` está protegida y sólo es accesible para clases derivadas que realicen entradas y salidas.

Constructores

```
strstreambase();  
strstreambase(const char*, int, char *start);
```

La primera forma crea un `strstreambase` con el buffer de su dato `streambuf` en memoria dinámica reservada la primera vez que se usa. Las zonas de lectura y escritura son la misma.

La segunda forma crea un `strstreambase` con el buffer y la posición de comienzo especificados.

Función `rdbuf`

```
strstreambuf * rdbuf();
```

Devuelve un puntero a `strstreambuf` asociado con este objeto.

Clase `strstreambase`

La declaración de esta clase está en el fichero `strstrea.h`.

Especialización de la clase `ios` para streams de cadena inicializando `ios::bp` para que apunte a un `strstreambuf`. Esto proporciona la condición necesaria para cualquier operación de entrada/salida en memoria. Por esa razón, `strstreambase` está diseñada completamente protegida y accesible sólo para clases derivadas que realicen entradas y salidas. Hace uso virtual de la clase `ios`.

Constructores

```
strstreambase();  
strstreambase(const char*, int, char *start);
```

La primera forma crea un `strstreambase` con el buffer de `streambuf` creado dinámicamente la primera vez que se usa. Las áreas de lectura y escritura son la misma.

La segunda forma crea un `strstreambase` con el buffer especificado y al posición de comienzo *start*.

Función `rdbuf`

```
strstreambuf* rdbuf();
```


Devuelve un puntero al `strstreambuf` asociado con este objeto.

Clase `istream`

La declaración de esta clase está en el fichero `strstream`.

Proporciona las operaciones de entrada en un `strstreambuf`.

El bloque formado por `ios`, `istream`, `ostream`, `iostream` y `streambuf`, proporciona una base para especializar clases que trabajen con memoria.

Constructores

```
istream(char *);  
istream(char *str, int n);
```

La primera forma crea un `istream` con la cadena especificada (el carácter nulo nunca se extrae).

La segunda forma crea un `istream` usando n bytes para *str*.

Clase `ostream`

La declaración de esta clase está en el fichero `strstream.h`.

Proporciona un stream de salida para inserción desde un array usando un `strstreambuf`.

Constructores

```
ostream();  
ostream(char *buf, int len, int mode = ios::out);
```

La primera forma crea un `ostream` con un array dinámico como stream de entrada.

La segunda forma crea un `ostream` con un buffer especificado por *buf* y un tamaño especificado por *len*. Si *mode* es `ios::app` o `ios::ate`, los punteros de lectura/escritura se colocan en la posición del carácter nulo de la cadena.

Función `pcount`

```
int pcount();
```

Devuelve el número de caracteres actualmente almacenados en el buffer.

Función `str`

```
char *str();
```

Devuelve y bloquea el buffer. El usuario debe liberar el buffer si es dinámico.

Clase `stringstream`

La declaración de esta clase está en el fichero `strstream.h`.

Proporciona entrada y salida simultanea en un array usando un `strstreambuf`. La entrada y la salida son realizadas usando las funciones de las clases base `istream` y `ostream`.

Por ejemplo, `stringstream` puede usar la función `istream::getline()` para extraer caracteres desde un buffer.

Constructores

```
stringstream();  
stringstream(char*, int sz, int mode);
```

La primera forma crea un `stringstream` con el buffer del dato miembro `strambuf` de la clase base `strstreambase` creado dinámicamente la primera vez que se usa. Las áreas de entrada y salida son la misma.

La segunda forma crea un `stringstream` con un buffer del tamaño especificado. Si el parámetro `mode` es `ios::app` o `ios::ate`, el puntero de entrada/salida se coloca en el carácter nulo que indica el final de la cadena.

Función `str`

```
char *str();
```

Devuelve y bloquea el buffer. El usuario debe liberar el buffer si es dinámico.

Objetos predefinidos

C++ declara y define cuatro objetos predefinidos, uno de la clase `istream`, y tres más de la clase `ostream_withassign` estos objetos están disponibles para cualquier programa C++:

- `cin`: entrada estándar: teclado.
- `cout`: salida estándar: pantalla.
- `cerr`: salida sin buffer a pantalla, la salida es inmediata, no es necesario vaciar el buffer.
- `clog`: igual que `cerr`, aunque suele redirigirse a un fichero log en disco.

Objeto `cout`

Se trata de un objeto global definido en `"iostream.h"`.

A lo largo de todo el curso hemos usado el objeto `cout` sin preocuparnos mucho de lo qué se trataba en realidad, ahora veremos más profundamente este objeto.

El operador `<<`

Ya conocemos el operador `<<`, lo hemos usado a menudo para mostrar cadenas de caracteres y variables.

```
ostream &operator<<(int)
```

El operador está sobrecargado para todos los tipos estándar: `char`, `char *`, `void *`, `int`, `long`, `short`, `bool`, `double` y `float`.

Además, el operador `<<` devuelve una referencia objeto `ostream`, de modo que puede asociarse. Estas asociaciones se evalúan de izquierda a derecha, y permiten expresiones como:

```
cout << "Texto: " << variable << "\n";
```

C++ reconoce el tipo de la variable y muestra la salida de la forma adecuada, siempre como una cadena de caracteres.

Por ejemplo:

```
int entero = 10;
char caracter = 'c';
char cadena[] = "Hola";
```

```
float pi = 3.1416;
void *puntero = cadena;

cout << "entero=" << entero << endl;
cout << "character=" << character << endl;
cout << "cadena=" << cadena << endl;
cout << "pi=" << pi << endl;
cout << "puntero=" << puntero << endl;
```

La salida tendrá este aspecto:

```
entero=10
character=c
cadena=Hola
pi=3.1416
puntero=0x254fdb8
```

Funciones interesantes de cout

Hay que tener en cuenta que cout es un objeto de la clase "ostream", que a su vez está derivada de la clase "ios", así que heredará todas las funciones y operadores de ambas clases. Se mostrarán todas esas funciones con más detalle en la documentación de las bibliotecas, pero veremos ahora las que se usan más frecuentemente.

Formatear la salida

El formato de las salidas de cout se puede modificar mediante flags. Estos flags pueden leerse o modificarse mediante las funciones flags, setf y unsetf.

Otro medio es usar manipuladores, que son funciones especiales que sirven para cambiar la apariencia de una operación de salida o entrada de un stream. Su efecto sólo es válido para una operación de entrada o salida. Además devuelven una referencia al stream, con lo que pueden ser insertados en una cadena entradas o salidas.

Por el contrario, modificar los flags tiene un efecto permanente, el formato de salida se modifica hasta que se restaure o se modifique el estado del flag.

Funciones manipuladoras con parámetros

Para usar estos manipuladores es necesario incluir el fichero de cabecera iomanip.

Existen seis de estas funciones manipuladoras: `setw`, `setbase`, `setfill`, `setprecision`, `setiosflags` y `resetiosflags`.

Todas trabajan del mismo modo, y afectan sólo a la siguiente entrada o salida.

Manipulador `setw`

Permite cambiar la anchura en caracteres de la siguiente salida de `cout`. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int x = 123, y = 432;

    cout << "#" << setw(6) << x << "#"
         << setw(12) << y << "#" << endl;
    return 0;
}
```

La salida tendrá este aspecto:

```
#    123#           432#
```

Manipulador `setbase`

Permite cambiar la base de numeración que se usará para la salida. Sólo se admiten tres valores: 8, 10 y 16, es decir, octal, decimal y hexadecimal. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int x = 123;

    cout << "#" << setbase(8) << x
         << "#" << setbase(10) << x
         << "#" << setbase(16) << x
         << "#" << endl;
}
```

```
    return 0;
}
```

La salida tendrá este aspecto:

```
#173#123#7b#
```

Manipulador `setfill`

Permite especificar el carácter de relleno cuando la anchura especificada sea mayor de la necesaria para mostrar la salida. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int x = 123;

    cout << "#" << setw(8) << setfill('0')
         << x << "#" << endl;
    cout << "#" << setw(8) << setfill('%')
         << x << "#" << endl;
    return 0;
}
```

La salida tendrá este aspecto:

```
#00000123#
#%%%%%%%%123#
```

Manipulador `setprecision`

Permite especificar el número de dígitos significativos que se muestran cuando se imprimen números en punto flotante: float o double. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    float x = 121.0/3;

    cout << "#" << setprecision(3)
```

```

        << x << "#" << endl;
    cout << "#" << setprecision(1)
        << x << "#" << endl;
    return 0;
}

```

La salida tendrá este aspecto:

```

#40.3#
#4e+01#

```

Manipuladores setiosflags y resetiosflags

Permiten activar o desactivar, respectivamente, los flags de formato de salida. Existen quince flags de formato a los que se puede acceder mediante un enum definido en la clase ios:

flag	Acción
skipws	ignora espacios en operaciones de lectura
left	ajusta la salida a la izquierda
right	ajusta la salida a la derecha
internal	deja hueco después del signo o el indicador de base
dec	conversión a decimal
oct	conversión a octal
hex	conversión a hexadecimal
showbase	muestra el indicador de base en la salida
showpoint	muestra el punto decimal en salidas en punto flotante
uppercase	muestra las salidas hexadecimales en mayúsculas
showpos	muestra el signo '+' en enteros positivos
scientific	muestra los números en punto flotante en notación exponencial
fixed	usa el punto decimal fijo para números en punto flotante
unitbuf	vacía todos los buffers después de una inserción
stdio	vacía los buffers stdout y stderr después de una inserción

Veamos un ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    float x = 121.0/3;
    int y = 123;

    cout << "#" << setiosflags(ios::left)
         << setw(12) << setprecision(4)
         << x << "#" << endl;
    cout << "#"
         << resetiosflags(ios::left | ios::dec)
         << setiosflags(ios::hex |
                        ios::showbase | ios::right)
         << setw(8) << y << "#"
         << endl;
    return 0;
}
```

La salida tendrá este aspecto:

```
#40.33      #
#          0x7b#
```

Manipuladores sin parámetros

Existe otro tipo de manipuladores que no requieren parámetros, y que ofrecen prácticamente la misma funcionalidad que los anteriores. La diferencia es que los cambios son permanentes, es decir, no sólo afectan a la siguiente salida, sino a todas las salidas hasta que se vuelva a modificar el formato afectado.

Manipuladores dec, hex y oct

```
inline ios& dec(ios& i)
inline ios& hex(ios& i)
inline ios& oct(ios& i)
```

Permite cambiar la base de numeración de las salidas de enteros, supongo que resulta evidente, pero de todos modos lo diré.

Función

Acción

dec Cambia la base de numeración a decimal

hex Cambia la base de numeración a hexadecimal

oct Cambia la base de numeración a octal

El cambio persiste hasta un nuevo cambio de base. Ejemplo:

```
#include <iostream>
using namespace std;

int main() {
    int a = 123, c = 432, b = 543;

    cout << "Decimal:      " << dec
         << a << ", " << b
         << ", " << c << endl;
    cout << "Hexadecimal: " << hex
         << a << ", " << b
         << ", " << c << endl;
    cout << "Octal:        " << oct
         << a << ", " << b
         << ", " << c << endl;

    return 0;
}
```

La salida tendrá éste aspecto:

```
Decimal:      123, 543, 432
Hexadecimal:  7b, 21f, 1b0
Octal:        173, 1037, 660
```

Funciones ws y ends

La función ws sólo es para streams de entrada.

La función ends no tiene sentido en cout, ya que sirve para añadir el carácter nulo de fin de cadena.

Función flush

```
ostream& flush(ostream& outs);
```

Vacía el buffer de salida. Puede ser invocada de dos modos:

```
cout.flush();
cout << flush;
```

Función endl

```
ostream& endl(ostream& outs);
```

Vacía el buffer de salida y además cambia de línea. Puede ser invocada de dos modos:

```
cout endl();  
cout << endl;
```

Función width

Cambia la anchura en caracteres de la siguiente salida de stream:

```
int width();  
int width(int);
```

La primera forma devuelve el valor de la anchura actual, la segunda permite cambiar la anchura para la siguiente salida, y también devuelve el valor actual de la anchura.

```
int x = 23;  
  
cout << "#";  
cout.width(10);  
cout << x << "#" << x << "#" << endl;
```

Función fill

Cambia el carácter de relleno que se usa cuando la salida es más ancha de la necesaria para el dato actual:

```
int fill();  
int fill(char);
```

La primera forma devuelve el valor actual del carácter de relleno, la segunda permite cambiar el carácter de relleno para la siguiente salida, y también devuelve el valor actual.

```
int x = 23;
cout << "|";
cout.width(10);
cout.fill('%');
cout << x << "|" << x << "|" << endl;
```

Función precision

Permite cambiar el número de caracteres significativos que se mostrarán cuando trabajemos con números en coma flotante: float o double:

```
int precision();
int precision(char);
```

La primera forma devuelve el valor actual de la precisión, la segunda permite modificar la precisión para la siguiente salida, y también devuelve el valor actual.

```
float x = 23.45684875;

cout << "|";
cout.precision(6);
cout << x << "|" << x << "|" << endl;
```

Función setf

Permite modificar los flags de manipulación de formato:

```
long setf(long);
long setf(long valor, long mascara);
```

La primera forma activa los flags que estén activos en el parámetro *valor* y deja sin cambios el resto.

La segunda forma activa los flags que estén activos tanto en *valor* como en *mascara* y desactiva los que estén activos en *mascara*, pero no en *valor*. Podemos considerar que *mascara* contiene activos los flags que queremos modificar y *valor* los flags que queremos activar.

Ambos devuelven el valor previo de los flags.

```
int x = 235;
```

```
cout << "|";  
cout.setf(ios::left, ios::left |  
    ios::right | ios::internal);  
cout.width(10);  
cout << x << "|" << endl;
```

Función unsetf

Permite eliminar flags de manipulación de formato:

```
void unsetf(long mascara);
```

Desactiva los flags que estén activos en el parámetro.

Nota:

En algunos compiladores he comprobado que esta función tiene como valor de retorno el valor previo de los flags.

```
int x = 235;  
  
cout << "|";  
cout.unsetf(ios::left | ios::right | ios::internal);  
cout.setf(ios::left);  
cout.width(10);  
cout << x << "|" << endl;
```

Función flags

Permite cambiar o leer los flags de manipulación de formato:

```
long flags () const;  
long flags (long valor);
```

La primera forma devuelve el valor actual de los flags.

La segunda cambia el valor actual por *valor*, el valor de retorno es el valor previo de los flags.

```
int x = 235;
```

```
long f;

cout << "|";
f = flags();
f &= !(ios::left | ios::right | ios::internal);
f |= ios::left;
cout.flags(f);
cout.width(10);
cout << x << "|" << endl;
```

Función put

Imprime un carácter:

```
ostream& put(char);
```

Ejemplo:

```
char l = 'l';
unsigned char a = 'a';

cout.put('H').put('o').put(l).put(a) << endl;
```

Función write

Imprime varios caracteres:

```
ostream& write(char* cad, int n);
```

Imprime n caracteres desde el principio de la cadena cad. Ejemplo:

```
char cadena[] = "Cadena de prueba";

cout.write(cadena, 12) << endl;
```

Función form

Imprime expresiones con formato, es análogo al printf de "stdio":

```
ostream& form(char* format, ...);
```

Nota:

Algunos compiladores no disponen de esta función.

Ejemplo:

```
char l = 'l';  
int i = 125;  
float f = 125.241;  
char cad[] = "Hola";  
  
cout.form("char: %c, int: %d, float %.2f, char*: %s",  
          l, i, f, cad);
```

Objeto cin

Se trata de un objeto global definido en "iostream.h".

En ejemplos anteriores ya hemos usado el operador >>.

El operador >>

Ya conocemos el operador >>, lo hemos usado para capturar variables.

```
istream &operator>>(int&)
```

Este operador está sobrecargado en cin para los tipos estándar: int&, short&, long&, double&, float&, char& y char*.

Además, el operador << devuelve una referencia objeto ostream, de modo que puede asociarse. Estas asociaciones se evalúan de izquierda a derecha, y permiten expresiones como:

```
cin >> var1 >> var2;  
cin >> variable;
```

Cuando se usa el operador >> para leer cadenas, la lectura se interrumpe al encontrar un carácter '\0', '\n' o '\n'.

Hay que tener cuidado, ya que existe un problema cuando se usa el operador >> para leer cadenas: cin no comprueba el desbordamiento del

espacio disponible para el almacenamiento de la cadena, del mismo modo que la función `gets` tampoco lo hace. De modo que resulta poco seguro usar el operador `>>` para leer cadenas.

Por ejemplo, declaramos:

```
char cadena[10];  
cin >> cadena;
```

Si el usuario introduce más de diez caracteres, los caracteres después de décimo se almacenarán en una zona de memoria reservada para otras variables o funciones.

Existe un mecanismo para evitar este problema, consiste en formatear la entrada para limitar el número de caracteres a leer:

```
char cadena[10];  
cin.width(sizeof(cadena));  
cin >> cadena;
```

De este modo, aunque el usuario introduzca una cadena de más de diez caracteres sólo se leerán diez.

Funciones interesantes de cin

Hay que tener en cuenta que `cin` es un objeto de la clase `"istream"`, que a su vez está derivada de la clase `"ios"`, así que heredarán todas las funciones y operadores de ambas clases. Se mostrarán todas esas funciones con más detalle en la documentación de las bibliotecas, pero veremos ahora las que se usan más frecuentemente.

Formatear la entrada

El formato de las entradas de `cin`, al igual que sucede con `cout`, se puede modificar mediante flags. Estos flags pueden leerse o modificarse mediante las funciones `flags`, `setf` y `unsetf`.

Otro medio es usar manipuladores, que son funciones especiales que sirven para cambiar la apariencia de una operación de salida o entrada de un stream. Su efecto sólo es válido para una operación de entrada o salida. Además devuelven una referencia al stream, con lo que pueden ser insertados en una cadena entradas o salidas.

Por el contrario, modificar los flags tiene un efecto permanente, el formato de salida se modifica hasta que se restaure o se modifique el estado del flag.

Funciones manipuladoras con parámetros

Para usar estos manipuladores es necesario incluir el fichero de cabecera `iomanip`.

Existen cuatro de estas funciones manipuladoras aplicables a `cin`: `setw`, `setbase`, `setiosflags` y `resetiosflags`.

Todas trabajan del mismo modo, y afectan sólo a la siguiente entrada o salida.

En el caso de `cin`, no todas las funciones manipuladoras tienen sentido, y algunas trabajan de un modo algo diferentes que con streams de salida.

Manipulador `setw`

Permite establecer el número de caracteres que se leerán en la siguiente entrada desde `cin`. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    char cad[10];

    cout << "Cadena:"
    cin >> setw(10) >> cad;

    cout << cad << endl
    return 0;
}
```

La salida tendrá este aspecto, por ejemplo:

```
Cadena: 1234567890123456
123456789
```

Hay que tener en cuenta que el resto de los caracteres no leídos por sobrepasar los diez caracteres, se quedan en el buffer de entrada de `cin`, y serán leídos en la siguiente operación de entrada que se haga. Ya veremos algo más abajo cómo evitar eso, cuando veamos la función `"ignore"`.

El manipulador `setw` no tiene efecto cuando se leen números, por ejemplo:


```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int x;

    cout << "Entero:"
    cin << setw(3) << x

    cout << x << endl
    return 0;
}
```

La salida tendrá este aspecto, por ejemplo:

```
Entero: 1234567
1234567
```

Manipulador setbase

Permite cambiar la base de numeración que se usará para la entrada de números enteros. Sólo se admiten tres valores: 8, 10 y 16, es decir, octal, decimal y hexadecimal. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int x;

    cout << "Entero: ";
    cin << setbase(16) << x;

    cout << "Decimal: " << x << endl;
    return 0;
}
```

La salida tendrá este aspecto:

```
Entero: fed4
Decimal: 65236
```

Manipuladores setiosflags y resetiosflags

Permiten activar o desactivar, respectivamente, los flags de formato de entrada. Existen quince flags de formato a los que se puede acceder mediante un enum definido en la clase ios:

flag	Acción
skipws	ignora espacios en operaciones de lectura
left	ajusta la salida a la izquierda
right	ajusta la salida a la derecha
internal	deja hueco después del signo o el indicador de base
dec	conversión a decimal
oct	conversión a octal
hex	conversión a hexadecimal
showbase	muestra el indicador de base en la salida
showpoint	muestra el punto decimal en salidas en punto flotante
uppercase	muestra las salidas hexadecimales en mayúsculas
showpos	muestra el signo '+' en enteros positivos
scientific	muestra los números en punto flotante en notación exponencial
fixed	usa el punto decimal fijo para números en punto flotante
unitbuf	vacía todos los buffers después de una inserción
stdio	vacía los buffers stdout y stderr después de una inserción

De los flags de formato listados, sólo tienen sentido en cin los siguientes: skipws, dec, oct y hex.

Veamos un ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    char cad[10];

    cout << "Cadena: ";
    cin >> setiosflags(ios::skipws) >> cad;
```

```

    cout << "Cadena: " << cad << endl;

    return 0;
}

```

La salida tendrá este aspecto:

```

Cadena:          prueba
Cadena: prueba

```

Manipuladores sin parámetros

Existen otro tipo de manipuladores que no requieren parámetros, y que ofrecen prácticamente la misma funcionalidad que los anteriores. La diferencia es que los cambios son permanentes, es decir, no sólo afectan a la siguiente entrada, sino a todas las entradas hasta que se vuelva a modificar el formato afectado.

Manipuladores dec, hex y oct

```

inline ios& dec(ios& i)
inline ios& hex(ios& i)
inline ios& oct(ios& i)

```

Permite cambiar la base de numeración de las entradas de enteros:

Función	Acción
dec	Cambia la base de numeración a decimal
hex	Cambia la base de numeración a hexadecimal
oct	Cambia la base de numeración a octal

El cambio persiste hasta un nuevo cambio de base. Ejemplo:

```

#include <iostream>
using namespace std;

int main() {
    int x, y, z;

    cout << "Entero decimal (x y z): ";
    cin >> dec >> x >> y >> z;
    cout << "Enteros: " << x << ", "
        << y << ", " << z << endl;
    cout << "Entero octal (x y z): ";
    cin >> oct >> x >> y >> z;
}

```

```

    cout << "Enteros: " << x << ", "
        << y << ", " << z << endl;
    cout << "Entero hexadecimal (x y z): ";
    cin >> hex >> x >> y >> z;
    cout << "Enteros: " << x << ", "
        << y << ", " << z << endl;

    return 0;
}

```

La salida tendrá éste aspecto:

```

Entero decimal (x y z): 10 45 25
Enteros: 10, 45, 25
Entero octal (x y z): 74 12 35
Enteros: 60, 10, 29
Entero hexadecimal (x y z): de f5 ff
Enteros: 222, 245, 255

```

Función ws

```

extern istream& ws(istream& ins);

```

Ignora los espacios iniciales en una entrada de cadena. Ejemplo:

```

#include <iostream>
using namespace std;

int main() {
    char cad[10];

    cout << "Cadena: ";
    cin >> ws >> cad;
    cout << "Cadena: " << cad << endl;

    return 0;
}

```

La salida tendrá éste aspecto:

```

Cadena:      hola
Cadena: hola

```

Función width()

Cambia la anchura en caracteres de la siguiente entrada de stream:

```
int width();  
int width(int);
```

La primera forma devuelve el valor de la anchura actual, la segunda permite cambiar la anchura para la siguiente entrada, y también devuelve el valor actual de la anchura. Esta función no tiene efecto con variables que no sean de tipo cadena.

```
char cadena[10];  
  
cin.width(sizeof(cadena));  
cin >> cadena;
```

Función setf()

Permite modificar los flags de manipulación de formato:

```
long setf(long);  
long setf(long valor, long mascara);
```

La primera forma activa los flags que estén activos tanto en el parámetro y deja sin cambios el resto.

La segunda forma activa los flags que estén activos tanto en valor como en máscara y desactiva los que estén activos en mask, pero no en valor. Podemos considerar que mask contiene activos los flags que queremos modificar y valor los flags que queremos activar.

Ambos devuelven el valor previo de los flags.

```
int x;  
  
cin.setf(ios::oct, ios::dec | ios::oct | ios::hex);  
cin >> x;
```

Función unsetf()

Permite eliminar flags de manipulación de formato:

```
void unsetf(long mascara);
```

Desactiva los flags que estén activos en el parámetro.

Nota:

En algunos compiladores he comprobado que esta función tiene como valor de retorno el valor previo de los flags.

```
int x;

cin.unsetf(ios::dec | ios::oct | ios::hex);
cin.setf(ios::hex);
cin >> x;
```

Función flags()

Permite cambiar o leer los flags de manipulación de formato:

```
long flags () const;
long flags (long valor);
```

La primera forma devuelve el valor actual de los flags.

La segunda cambia el valor actual por valor, el valor de retorno es el valor previo de los flags.

```
int x;
long f;

f = flags();
f &= !(ios::hex | ios::oct | ios::dec);
f |= ios::dec;
cin.flags(f);
cin >> x;
```

Función get

La función get() tiene tres formatos:

```
int get();
istream& get(char& c);
```

```
istream& get(char* ptr, int len, char delim = '\\n');
```

Sin parámetros, lee un carácter, y lo devuelve como valor de retorno:

Nota:

Esta forma de la función get() se considera obsoleta.

Con un parámetro, lee un carácter:

En este formato, la función puede asociarse, ya que el valor de retorno es una referencia a un stream. Por ejemplo:

```
char a, b, c;  
cin.get(a).get(b).get(c);
```

Con tres parámetros: lee una cadena de caracteres:

En este formato la función get lee caracteres hasta un máximo de 'len' caracteres o hasta que se encuentre el carácter delimitador.

```
char cadena[20];  
cin.get(cadena, 20, '#');
```

Función getline

Funciona exactamente igual que la versión con tres parámetros de la función get(), salvo que el carácter delimitador también se lee, en la función get() no.

```
istream& getline(char* ptr, int len, char delim = '\\n');
```

Función read

Lee n caracteres desde el cin y los almacena a partir de la dirección ptr.

```
istream& read(char* ptr, int n);
```

Función ignore

Ignora los caracteres que aún están pendientes de ser leídos:

```
istream& ignore(int n=1, int delim = EOF);
```

Esta función es útil para eliminar los caracteres sobrantes después de hacer una lectura con el operador >>, get o getline; cuando leemos con una anchura determinada y no nos interesa el resto de los caracteres introducidos. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    char cad[10];
    int i;

    cout << "Cadena: ";
    cin >> setw(10) >> cad;
    cout << "Entero: ";
    cin.ignore(100, '\n') >> i;
    cout << "Cadena: " << cad << endl;
    cout << "Entero: " << i << endl;

    cin.get();
    return 0;
}
```

La salida podría tener este aspecto:

```
Cadena: cadenademasiadolarga
Entero: 123
Cadena: cadenadem
Entero: 123
```

Función peek

Esta función obtiene el siguiente carácter del buffer de entrada, pero no lo retira, lo deja donde está.

```
int peek();
```

Función putback

Coloca un carácter en el buffer de entrada:

```
istream& putback(char);
```

Función scan

Lee variables con formato, es análogo al scanf de "stdio":

```
istream& scan(char* format, ...);
```

Nota:

Algunos compiladores no disponen de esta función.

Ejemplo:

```
char l;  
int i;  
float f;  
char cad[15];  
  
cin.scan("%c%d%f%s", &l, &i, &f, cad);
```

Tabla de contenido

- **Introducción**
 - Proceso para la obtención de un programa ejecutable
 - Fichero fuente y programa o código fuente
 - Interpretes y compiladores
 - Ficheros objeto, código objeto y compiladores
 - Librerías o bibliotecas
 - Ficheros ejecutables y enlazadores
 - Errores
 - Propósito de C y C++
- **1 Toma de contacto**
- **2 Variables I**
 - Sobre la sintaxis
 - Tipos fundamentales
 - Tipo "char" o carácter
 - Tipo "int" o entero
 - Tipo "long long"
 - Tipo "float" o coma flotante
 - Tipo "bool" o Booleano
 - Tipo "double" o coma flotante de doble precisión
 - Tipo "void" o sin tipo
 - Tipo "enum" o enumerado
 - Palabras reservadas usadas en este capítulo
- **3 Funciones I: Declaración y definición**
 - Prototipos de funciones
 - Definición de funciones
 - Estructura de un programa C++
 - Estructuras más complejas
 - Palabras reservadas usadas en este capítulo
- **4 Operadores I**
 - Operadores aritméticos
 - Operadores de asignación
 - Operador coma
 - Operadores de comparación

- Expresiones con operadores de igualdad
- Operadores lógicos
 - Cortocircuito
 - Tablas de verdad
 - Expresiones lógicas frecuentes
- Operador sizeof
- Asociación de operadores binarios
 - Generalización de cortocircuitos
- Palabras reservadas usadas en este capítulo
- 5 Sentencias
 - Bloques
 - Expresiones
 - Llamadas a función
 - Asignación
 - Nula
 - Bucles
 - Bucle "mientras"
 - Bucle "hacer...mientras"
 - Bucle "para"
 - Etiquetas
 - Etiquetas de identificación
 - Etiquetas case y default
 - Sentencias de selección
 - Sentencia if...else
 - Sentencia switch
 - Sentencias de salto
 - Sentencia de ruptura
 - Sentencia continue
 - Sentencia de salto
 - Sentencia de retorno
 - Uso de las sentencias de salto y la programación estructurada
 - Comentarios
 - Palabras reservadas del capítulo 5
- 6 Declaración de variables
 - Cómo se declaran las variables
 - Ámbitos

- **Ámbito de las variables**
- **Enmascaramiento de variables**
- **Operador de ámbito**
- **Problemas resueltos de capítulos 1 a 6**
 - **Ejemplo 6.1**
 - **Ejemplo 6.2**
 - **Ejemplo 6.3**
 - **Ejemplo 6.4**
 - **Ejemplo 6.5**
- **7 Normas para la notación**
 - **Constantes int**
 - **Constantes long**
 - **Constantes long long**
 - **Constantes unsigned**
 - **Constantes unsigned long**
 - **Constantes unsigned long long**
 - **Constantes float**
 - **Constantes double**
 - **Constantes long double**
 - **Constantes enteras**
 - **Constantes en punto flotante**
 - **Constantes char**
 - **Secuencias de escape**
 - **¿Por qué es necesaria la notación?**
- **8 Cadenas de caracteres**
- **9 Conversión de tipos**
 - **Conversiones a **bool****
 - **Casting: conversiones explícitas de tipo**
 - **Ejemplos capítulos 8 y 9**
 - **Ejemplo 9.1**
 - **Ejemplo 9.2**
 - **Ejemplo 9.3**
 - **Ejemplo 9.4**
- **10 Tipos de variables II: Arrays**
 - **Inicialización de arrays**
 - **Operadores con arrays**
 - **Algoritmos de ordenación, método de la burbuja**

- Problemas
- 11 Tipos de objetos III: Estructuras
 - Funciones en el interior de estructuras
 - Inicialización de estructuras
 - Asignación de estructuras
 - Arrays de estructuras
 - Estructuras anidadas
 - Estructuras anónimas
 - Operador sizeof con estructuras
 - Campos de bits
 - Palabras reservadas usadas en este capítulo
 - Problemas
 - Ejemplos capítulos 10 y 11
 - Ejemplo 11.1
 - Ejemplo 11.2
 - Ejemplo 11.3
 - Ejemplo 11.4
 - Ejemplo 11.5
- 12 Tipos de objetos IV: Punteros 1
 - Declaración de punteros
 - Obtener punteros a objetos
 - Objeto apuntado por un puntero
 - Diferencia entre punteros y otros objetos
 - Correspondencia entre *arrays* y punteros
 - Operaciones con punteros
 - Asignación
 - Operaciones aritméticas
 - Comparación entre punteros
 - Punteros genéricos
 - Punteros a estructuras
 - Ejemplos
 - Objetos dinámicos
 - Problemas
 - Ejemplos capítulo 12
 - Ejemplo 12.1
 - Ejemplo 12.2
 - Ejemplo 12.3

- 13 Operadores II: Más operadores
 - Operadores de Referencia (&) e Indirección (*)
 - Operadores . y ->
 - Operador de preprocesador
 - Directiva define
 - Directiva include
 - Operadores de manejo de memoria new y delete
 - Operador new
 - Operador delete
 - Palabras reservadas usadas en este capítulo
- 14 Operadores III: Precedencia
- 15 Funciones II: Parámetros por valor y por referencia
 - Referencias a variables
 - Pasando parámetros por referencia
 - Punteros como parámetros de funciones
 - Arrays como parámetros de funciones
 - Estructuras como parámetros de funciones
 - Funciones que devuelven referencias
- 16 Tipos de variables V: Uniones
 - Estructuras anónimas
 - Inicialización de uniones
 - Discriminadores
 - Funciones dentro de uniones
 - Palabras reservadas usadas en este capítulo
 - Ejemplos capítulo 16
 - Ejemplo 16.1
 - Ejemplo 16.2
 - Ejemplo 16.3
 - Ejemplo 16.4
- 17 Tipos de variables VI: Punteros 2
 - Problemas
- 18 Operadores IV: Más operadores
 - Operadores de bits
 - Ejemplos
 - Operador condicional
- 19 Definición de tipos, tipos derivados
 - Ejemplos

- Palabras reservadas usadas en este capítulo
- Ejemplos capítulo 18 y 19
 - Ejemplo 19.1
 - Ejemplo 19.2
 - Ejemplo 19.3
- 20 Funciones III: más cosas
 - Parámetros con valores por defecto
 - Funciones con número de argumentos variable
 - Tipos
 - Macros
 - Leer la lista de parámetros
 - Argumentos en main
 - Funciones inline
 - Punteros a funciones
 - Utilidad de los punteros a funciones
 - Asignación de punteros a funciones
 - Llamadas a través de un puntero a función
 - Palabras reservadas usadas en este capítulo
- 21 Funciones IV: Sobrecarga
 - Resolución de sobrecarga
 - Problema
- 22 Operadores V: Operadores sobrecargados
 - Operadores binarios
 - Operadores unitarios
 - Forma prefija
 - Forma sufija
 - Operador de asignación
 - Notación funcional de los operadores
 - Palabras reservadas usadas en este capítulo
 - Problemas
 - Ejemplos capítulo 22
 - Ejemplo 22.1
 - Ejemplo 22.2
- 23 El preprocesador
 - Directiva #define
 - Directiva #undef
 - Directivas #if, #elif, #else y #endif

- Directivas #ifdef e #ifndef
- Directiva #error
- Directiva #include
- Directiva #line
- Directiva #pragma
 - Directiva #pragma pack()
 - Atributos
- Directiva #warning
- 24 Funciones V: Recursividad
 - Otras formas de recursividad
 - Ejemplos capítulo 24
 - Ejemplo 24.1
 - Ejemplo 24.2
 - Ejemplo 24.3
 - Ejemplo 24.4
 - Ejemplo 24.5
- 25 Variables VII: Tipos de almacenamiento
 - Almacenamiento automático
 - Almacenamiento estático
 - Almacenamiento externo
 - Almacenamiento en registro
 - Modificador de almacenamiento constante
 - Modificador de almacenamiento volatile
 - Modificador de almacenamiento mutable
 - Palabras reservadas usadas en este capítulo
- 26 Espacios con nombre
 - Declaraciones y definiciones
 - Utilidad
 - Espacios anónimos
 - Espacio global
 - Espacios anidados
 - Palabras reservadas usadas en este capítulo
- 27 Clases I: definiciones
 - Definiciones
 - POO
 - Objeto
 - Mensaje

- Método
 - Clase
 - Interfaz
 - Herencia
 - Jerarquía
 - Polimorfismo
- 28 Declaración de una clase
 - Especificadores de acceso
 - Acceso privado, private
 - Acceso público, public
 - Acceso protegido, protected
 - Palabras reservadas usadas en este capítulo
- 29 Constructores
 - Constructor por defecto
 - Inicialización de objetos
 - Sobrecarga de constructores
 - Constructores con argumentos por defecto
 - Asignación de objetos
 - Constructor copia
 - Problemas
- 30 Destruyores
- 31 El puntero this
 - Palabras reservadas usadas en este capítulo
 - Ejemplos capítulos 27 a 31
 - Ejemplo 31.1
 - Ejemplo 31.2
 - Ejemplo 31.3
- 32 Sistema de protección
 - Declaraciones friend
 - Funciones amigas externas
 - Funciones amigas en otras clases
 - Clases amigas
 - Palabras reservadas usadas en este capítulo
- 33 Modificadores para miembros
 - Funciones en línea (inline)
 - Funciones miembro constantes
 - Valores de retorno constantes

- Miembros estáticos de una clase (Static)
 - Palabras reservadas usadas en este capítulo
- 34 Más sobre las funciones
 - Funciones sobrecargadas
 - Funciones con argumentos con valores por defecto
- 35 Operadores sobrecargados
 - Sobrecarga de operadores binarios
 - Sobrecargar el operador de asignación: ¿por qué?
 - Operadores binarios que pueden sobrecargarse
 - Forma funcional de los operadores
 - Sobrecarga de operadores para clases con punteros
 - Notas sobre este tema
 - Sobrecarga de operadores unitarios
 - Operadores unitarios sufijos
 - Operadores unitarios que pueden sobrecargarse
 - Operadores de conversión de tipo
 - Sobrecarga del operador de indexación []
 - Sobrecarga del operador de llamada ()
- 36 Herencia
 - Jerarquía, clases base y clases derivadas
 - Derivar clases, sintaxis
 - Constructores de clases derivadas
 - Inicialización de clases base en constructores
 - Inicialización de objetos miembros de clases
 - Sobrecarga de constructores de clases derivadas
 - Destructores de clases derivadas
- 37 Funciones virtuales
 - Redefinición de funciones en clases derivadas
 - Superposición y sobrecarga
 - Polimorfismo
 - Funciones virtuales
 - Destructores virtuales
 - Constructores virtuales
 - Palabras reservadas usadas en este capítulo
- 38 Derivación múltiple
 - Constructores de clases con herencia múltiple
 - Herencia virtual

- Funciones virtuales puras
 - Clases abstractas
 - Uso de derivación múltiple
- 39 Trabajar con ficheros
 - Crear un fichero de salida, abrir un fichero de entrada
 - Ficheros binarios
 - Ficheros de acceso aleatorio
 - Ficheros de entrada y salida
 - Sobrecarga de operadores << y >>
 - Comprobar estado de un stream
- 40 Plantillas
 - Sintaxis
 - Plantillas de funciones
 - Plantilla para tabla
 - Ficheros de cabecera
 - Ejemplo de uso de plantilla Tabla
 - Posibles problemas
 - Tablas de cadenas
 - Funciones que usan plantillas como parámetros
 - Pasar una instancia de una plantilla
 - Pasar una plantilla genérica
 - Amigos de plantillas
 - Clase o función amiga de una plantilla
 - Clase o función amiga de una instancia de una plantilla
 - Miembros estáticos: datos y funciones
 - Ejemplo de implementación de una plantilla para una pila
 - Bibliotecas de plantillas
 - Palabra typename
 - Palabras reservadas usadas en este capítulo
- 41 Punteros a miembros de clases o estructuras
 - Asignación de valores a punteros a miembro
 - Operadores .* y ->*
- 42 Castings en C++
 - Operador static_cast<>
 - Operador const_cast<>
 - Operador reinterpret_cast<>
 - Operador typeid

- Operador `dynamic_cast<>`
 - Castings cruzados
- 43 Manejo de excepciones
 - La clase "exception"
 - Orden en la captura de excepciones
 - Especificaciones de excepciones
 - Excepciones en constructores y destructores
 - Excepciones estándar
 - Relanzar una excepción
- A Codificación ASCII
 - El origen
 - Tabla ASCII
 - Las letras son números
 - Manejar signos
- B Palabras reservadas
 - Palabras reservadas C++
 - Palabras reservadas C
- C Bibliotecas estándar
 - Biblioteca de entrada y salida fluidas
 - Biblioteca C de entrada y salida estándar
 - Función `getchar()`
 - Función `putchar()`
 - Función `gets()`
 - Función `puts()`
 - Función `printf()`
 - Biblioteca de rutinas de conversión estándar `stdlib.h`
 - Función `atoi()`
 - Función `system()`
 - Función `abs()`
 - Función `rand()`
 - Función `srand()`
 - Biblioteca de tratamiento de caracteres `ctype.h`
 - Función `toupper()`
 - Función `tolower()`
 - Macros `is<conjunto>()`
 - Biblioteca de manipulación de cadenas `string.h`
 - Función `strlen()`

- Función strcpy()
 - Función strcmp()
 - Función strcat()
 - Función strncpy()
 - Función strncmp()
 - Función strncat()
 - Función strtok()
- D Trigramos y símbolos alternativos
 - Trigramos
 - Símbolos alternativos
- E Streams
 - Clases predefinidas para streams
 - Clase streambuf
 - Funciones protegidas
 - Funciones públicas
 - Clase ios
 - Enums
 - Funciones
 - Clase filebuf
 - Constructores
 - Funciones
 - Clase istream
 - Constructor
 - Clase ostream
 - Constructor
 - Clase iostream
 - Constructor
 - Clase fstreambase
 - Constructores
 - Clase ifstream
 - Constructores
 - Clase ofstream
 - Constructores
 - Clase fstream
 - Constructores
 - Clase stringstream
 - Constructores

- Clase stringstream
 - Constructores
- Clase ifstream
 - Constructores
- Clase ofstream
 - Constructores
- Clase ostream
 - Constructores
- Objetos predefinidos
- Objeto cout
 - El operador <<
 - Funciones interesantes de cout
- Objeto cin
 - El operador >>
 - Funciones interesantes de cin