CURSO DE C++

CON DEV-C++

Introducción

Bien, aquellos que hayáis seguido el curso desde sus comienzos, en septiembre de 2000, conocéis la trayectoria y la evolución que ha tenido. El curso está ya muy avanzado, parecía imposible al principio, pero ya están tratados la mayor parte de los temas sobre C++.

Lo que queda de comentar sobre C++ se reduce a un único tema: asm, y algo sobre el modificador explicit. Actualmente estoy haciendo un repaso a fondo y añadiendo más ejercicios y ejemplos.

Sigo esperando 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 y sigo manteniéndolo.

No he pretendido ser original, (al menos no demasiado), como dije que haría, he consultado libros, tutoriales, revistas, listas de correo, news, páginas web... En fin, cualquier fuente de datos que he podido, con el fin de conseguir un buen nivel. Espero haber conseguido mi objetivo, y seguiré completando explicaciones sobre todo aquello que lo requiera. Espero 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, propondré problemas, cuya resolución pasará a ser parte del curso. Además se añadirán las preguntas que vaya recibiendo, así como sus respuestas. Y en la lista de correo podremos discutir sobre los temas del curso entre todos aquellos que lo sigan.

He intentado que los ejemplos que ilustran cada capítulo corran en cualquier versión de compilador, sin embargo, he de decir que yo he usado el compilador Dev-C++ de Bloodshed en modo consola. Este compilador, está pensado para hacer programas en Windows. 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 librerías externas no estándar.

- El otro modo es el GUI, Interfaz Gráfico de Usuario. Es el modo tradicional de los programas de Windows, con ventanas, menús, iconos, etc. La creación de este tipo de programas se explica en otro curso de este mismo sitio, y requiere el conocimiento de la librería de funciones Win API32.

Para aquellos de vosotros que programéis en otros entornos como Linux, Unix o Mac, he de decir que no os servirá el compilador Dev-C++, ya que está diseñado especialmente para Windows. Pero esto no es un problema serio, todos los sistemas operativos disponen de compiladores de C++ que soportan la norma ANSI, sólo menciono Dev-C++ y Windows porque es el entorno en el que yo, me muevo actualmente.

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

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

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

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

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

Cuando se exponga la sintaxis de cada sentencia se adoptarán ciertas reglas, que por lo que sé son de uso general en todas 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 [1] [1]

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

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

Fichero fuente y programa o código fuente:

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

Ficheros objeto, código objeto y compiladores:

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

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

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

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

Librerías:

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

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

Ficheros ejecutables y enlazadores:

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

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

Muy a menudo, nuestros programas estarán compuestos por varios ficheros fuente, y de cada uno de ellos se obtendrá un fichero objeto. Es necesario unir todos los ficheros objeto, más las librerías en un único fichero ejecutable.

Hay que dar ciertas instrucciones al ordenador para que cargue en memoria el programa y los datos, y para que organice la memoria de modo que se disponga de una pila de tamaño adecuado, etc. La pila es una zona de memoria que se usa para que el programa intercambie datos con otros programas o con otras partes del propio programa. Veremos esto con más detalle durante el curso.

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

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

Errores:

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

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

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

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

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

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

Propósito de C y C++ [1] [1]

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

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

ello, se limitan a usarlas.

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

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

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

l l " src="imagen/pagsig.gif" width=24 border=noneindex.php?cap=001 index.php?cap=001" src="imagen/capsig.gif" width=24 border=noneindex.php?cap=001

1 Toma de contacto

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

```
int main()
{
   int numero;

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

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

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

- Primera línea: "`int main()`"

Se trata de una línea muy especial, y la encontrarás en todos los programas C y C++. Es el

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

C++ se basa en gran parte en C, y C fue creado en la época de los lenguajes procedimentales y orientado a 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 sería 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 los valores que devuelven 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, en éstos sólo existen funciones y para crear un procedimiento se hace 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 se ejecute nuestro programa. Otra pista por la que sabemos que se trata de una función son los paréntesis, todas las definiciones de funciones incluyen una lista de argumentos de entrada entre paréntesis, en nuestro ejemplo es una lista vacía, es decir, nuestra función no admite parámetros.

- Segunda línea: "{"

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

- Tercera línea: "`int numero;`"

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

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

- Cuarta línea: ""

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

- Quinta línea: "`numero = 2 + 2;`"

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

- Sexta línea: "`return 0;`"

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

- Séptima línea: "`}`"

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

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

variables, funciones y operadores. Después estaremos en disposición de empezar a trabajar con ejemplos más interactivos.

2 Tipos de variables I

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

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

Tipos fundamentales [1] [1]

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.

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

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

Para crear un identificador hay que tener en cuenta algunas reglas, 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 y C++ distinguen entre mayúsculas y minúsculas, de modo que los identificadores numero y Numero son diferentes.

Serán válidos estos ejemplos:

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

Tipo "char" o carácter:

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

Es el tipo básico alfanumérico, es decir que puede contener un carácter, un dígito numérico o un signo de puntuación. Desde el punto de vista del ordenador, todos esos valores son caracteres. En C y C++ 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 puede ser declarado con y sin signo. Y si no se especifica el modificador de signo, se asume que es con signo. Este tipo de variables es apto para almacenar números pequeños, como los dedos que tiene una persona, o letras, como la inicial de mi nombre de pila.

Tipo "int" o entero:

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

Las variables enteras almacenan números enteros dentro de los límites de su tamaño, a su vez, ese tamaño depende de la plataforma del compilador, y del número de bits que use por palabra de memoria: 8, 16, 32... No hay reglas fijas para saber el mayor número que podemos almacenar en cada tipo: int, long int o short int; depende en gran medida del compilador y del ordenador. Sólo podemos estar seguros de que ese número en short int es menor o igual que en int, y éste a su vez es menor o igual que en long int y que long long int es mayor o igual a long int. Veremos cómo averiguar estos valores cuando estudiemos los operadores.

A cierto nivel, podemos considerar los tipos "char", "short", "int", "long" y "long long" como tipos diferentes, todos enteros. Pero sólo se diferencian en el tamaño del valor máximo que pueden contener.

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

Tipo "float" o coma flotante:

```
float <identificador>
```

Las variables de este tipo almacenan números en formato de coma flotante, mantisa y exponente, para entendernos, son números con decimales. Son aptos para variables de tipo real, como por ejemplo el cambio entre euros y pesetas. O para números muy grandes, como la producción mundial de trigo, contada en granos. El fuerte de estos números no es

la precisión, sino el orden de magnitud, es decir lo grande o pequeño que es el número que contiene. Por ejemplo, la siguiente cadena de operaciones no dará el resultado correcto:

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

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

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

Tipo "bool" o Booleana:

```
bool <identificador>
```

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

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

Nota: En algunos compiladores antiguos de C++ 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>
```

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

Bueno, también me han preguntado por qué no usar siempre double o long double y olvidarnos de float. La respuesta es que C siempre ha estado orientado a la economía de recursos, tanto en cuanto al uso de memoria como al uso de procesador. Si tu problema no requiere la precisión de un double o long double, ¿por 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 más los recursos de nuestros clientes, para los que haremos nuestros programas. C y C++ nos dan un gran control sobre estas características, es nuestra responsabilidad aprender a usarlos como es debido.

Tipo "void" o sin tipo:

```
void <identificador>
```

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

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

Tipo "enum" o enumerado:

```
enum [<identificador_de_enum>] {
   <nombre> [= <valor>], ...} [lista_de_variables];
```

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

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

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

```
[enum] <identificador_de_enum> <lista_de_identificadores>;
```

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

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

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

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

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

Palabras reservadas usadas en este capítulo

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

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

Ejercicios del capítulo 2 Tipos de variables I

1)¿Cuáles de los siguientes son tipos válidos?

a) `unsigned char`

Sí No

b) `long char`

Sí No

c) `unsigned float`

Sí No

d) `double char`

Sí No

e) `signed long`

Sí No

f) `unsigned short`

Sí No

g) `signed long int`

Sí No

h) `long double`

Sí No

i) `enum dia {lunes, martes, miercoles, jueves, viernes, sabado, domingo};`

Sí No

j) `enum color {verde, naranja, rojo};`
`enum fruta {manzana, fresa, naranja, platano};`

Sí No

k) `long bool`

Sí No

Ejercicios del capítulo 2 Tipos de variables I

1)¿Cuáles de los siguientes son tipos válidos?

a) `unsigned char` (X)

Mal: char es un tipo entero, por lo tanto admite el modificador de signo.

b) `long char` (X)

Mal: char sólo admite los modificadores signed y unsigned.

c) `unsigned float` (X)

Mal: los tipos float y double siempre tienen signo.

d) `double char` (X)

Mal: char sólo admite los modificadores signed y unsigned.

e) `signed long` (X)

Mal: los enteros admiten siempre los modificadores de signo.

f) `unsigned short` (X)

Mal: los enteros admiten siempre los modificadores de signo.

g) `signed long int` (X)

Mal: los enteros admiten siempre los modificadores de signo.

h) `long double` (X)

Mal: hay tres tipos en coma flotante: float, double y long double.

i) `enum dia {lunes, martes, miercoles, jueves, viernes, sabado, domingo};` (X)

Mal: los identificadores son únicos, es correcto.

j) `enum color {verde, naranja, rojo};`
`enum fruta {manzana, fresa, naranja, platano};` (X)

Mal: no se puede usar el mismo identificador en varios enum, como el caso de 'naranja'.

k) `long bool` (X)

Mal: el tipo bool no admite modificadores.

3 Funciones I: Declaración y definición

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

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

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

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

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

- Opcionalmente, una palabra que especifique el tipo de almacenamiento, puede ser "extern" o "static". Si no se especifica es "extern". No te preocupes de esto todavía, de momento sólo usaremos funciones externas, sólo lo menciono porque es parte de la declaración. Una pista: las funciones declaradas como extern están disponibles para todo el programa, las funciones static pueden no estarlo.

- El tipo del valor de retorno, que puede ser "void", si no necesitamos valor de retorno. En C, si no se establece, por defecto será "int", aunque en general se considera de mal gusto omitir el tipo de valor de retorno. 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 nombre de la función. Es costumbre, muy útil y muy recomendable, poner nombres que indiquen, lo más claramente posible, qué es lo que hace la función, y que permitan interpretar qué hace el programa con sólo leerlo. Cuando se precisen varias palabras para conseguir este efecto existen varias reglas aplicables de uso común. Una consiste en separar cada palabra con un "_", la otra, que yo prefiero, consiste en escribir la primera letra de cada palabra en mayúscula y el resto en minúsculas. Por ejemplo, si hacemos una función que busque el número de teléfono de una persona en una base de datos, podríamos llamarla "busca_telefono" o "BuscaTelefono".

- Una lista de declaraciones de parámetros entre paréntesis. Los parámetros de una función son los valores de entrada (y en ocasiones también de salida). Para la función se comportan exactamente igual que variables, y de hecho cada parámetro se declara igual que una variable. Una lista de parámetros es un conjunto de declaraciones de parámetros separados con comas. Puede tratarse de una lista vacía. En C es preferible usar la forma "func(void)" para listas de parámetros vacías. En C++ este procedimiento se considera obsoleto, se usa simplemente "func()"..

- Un cuerpo de función que representa el código que será ejecutado cuando se llame a la función. El cuerpo de la función se encierra entre llaves "{}"

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

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

Prototipos de funciones [1] [1]

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

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

Por ejemplo:

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

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

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

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

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

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

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

Estructura de un programa C/C++: [1] [1]

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]
```

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

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

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

La definición de las funciones puede hacerse dentro de los ficheros fuente o enlazarse desde

librerías precompiladas. La diferencia entre una declaración y una definición es que la definición posee un cuerpo de función.

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

Palabras reservadas usadas en este capítulo

extern y static.

Ejercicios del capítulo 3 Declaración y definición de funciones

1)¿Cuáles de los siguientes prototipos son válidos?

a) `Calcular(int, int, char r);`

Sí No

b) `void Invertir(int, unsigned char)`

Sí No

c) `void Aumentar(float valor);`

Sí No

d) `float Negativo(float int);`

Sí No

e) `int Menor(int, int, int);`

Sí No

f) `char Menu(int opciones);`

Sí No

2) Preguntas sobre la estructura de un programa.

a) ¿Entre qué zonas harías las declaraciones de variables globales?

Antes de la zona de las directivas del preprocesador

Entre la zona de las directivas del preprocesador y las declaraciones de prototipos

Después de la definición de la función "main"

b) ¿Qué aparecería normalmente justo después de la definición de la función "main"?

Las directivas del preprocesador

Los prototipos de funciones

Las definiciones de funciones

Ejercicios del capítulo 3 Declaración y definición de funciones

1)¿Cuáles de los siguientes prototipos son válidos?

a) `Calcular(int, int, char r);` (X)

Mal: C++ no permite declarar funciones sin indicar el tipo del valor de retorno.

b) `void Invertir(int, unsigned char)` (X)

Mal: Los prototipos terminan con un punto y coma.

c) `void Aumentar(float valor);` (X)

Mal: Aunque no es necesario, nada nos impide usar un nombre para cada parámetro.

d) `float Negativo(float int);` (X)

Mal: Podemos usar nombres para los parámetros, pero nunca una palabra reservada para ese fin.

e) `int Menor(int, int, int);` (X)

Mal: Es correcto.

f) `char Menu(int opciones);` (X)

Mal: Es correcto.

2) Preguntas sobre la estructura de un programa.

a) ¿Entre qué zonas harías las declaraciones de variables globales? (X)

Mal: Aunque se puede hacer antes de la zona de directivas, no es recomendable. Y

si lo hacemos después de la función "main" no estaría accesibles para el programa.

b) ¿Qué aparecería normalmente justo después de la definición de la función "main"? (X)

Mal: Aunque algunas directivas pueden aparecer en cualquier parte de un programa, las include y define suelen aparecer al principio y lo normal es que los prototipos se declaren antes de definir "main", de modo que estén disponibles para usarse en cualquier parte del programa.

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 un valor que almacena nuestro programa que puede cambiar a lo largo de su ejecución.

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

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

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

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

Operadores aritméticos [1] [1]

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 enteros y de coma flotante. Sintaxis:

```
<expresión> + <expresión>
<expresión> - <expresión>
```

```
<expresión> * <expresión>
<expresión> / <expresión>
```

Evidentemente se trata de las conocidísimas operaciones aritméticas de suma, resta, multiplicación y división, que espero que ya domines a su nivel tradicional, es decir, sobre el papel.

El operador de módulo '%', devuelve el resto de la división entera del primer operando entre el segundo. Por esta razón no puede ser aplicado a operandos en coma flotante.

```
<expresión> % <expresión>
```

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*7+3 = 17.

En C y 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*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 avalúa la expresión "a+b", que dará como resultado 110, y se asignará este valor a c, que valdrá 110. Finalmente se aplica en post-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.

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 mejorado o incrementado. Y ya que estamos, el lenguaje C se llama así porque las personas que lo desarrollaron crearon dos prototipos de lenguajes de programación con anterioridad a los que llamaron B y BCPL.

Operadores de asignación [1] [1]

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

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

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

En general, para todos los operadores mixtos la expresión

E1 op= E2

Tiene el mismo efecto que la expresión

E1 = E1 op E2

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

Operador coma [1] [1]

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

```
E1, E2, ..., En
```

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

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

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

Operadores de igualdad [1] [1]

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

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

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

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 de 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 [1] [1]

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

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

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

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

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

Expresión 1	Expresión 2	Expresión1 && Expresión2
false	ignorada	false
true	false	false
true	true	true

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

Expresión 1	Expresión 2	Expresión1 || Expresión2
false	false	false
false	true	true
true	ignorada	true

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

La expresión "!E" es equivalente a (0 == E).

Expresión	!Expresión
false	true
true	false

Operadores relacionales [1] [1]

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

```
<expresión1> > <expresión2>
<expresión1> < <expresión2>
<expresión1> <= <expresión2>
<expresión1> >= <expresión2>
```

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

> mayor que

< menor que

>= mayor o igual que

<= menor o igual que

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

Operador "sizeof" [1] [1]

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 [1] [1]

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 y C++ para asignar el mismo valor a varias variables, en este caso, todas las variables: A, B, C y D recibirán el valor 0.

Ejercicios del capítulo 4 Operadores I

1) Suponiendo los siguientes valores iniciales para las variables:
`x = 2; y = 6; z = 9; r = 100; s = 10; a = 15; b = 3;`
¿Cuáles son los valores correctos en cada expresión?

a) `x += 10;`

12

10

11

b) `s *= b;`

9

13

30

c) `r /= 0;`

infinito

1

error

d) `y += x + 10;`

8

12

18

e) `z -= a*b;`

-36

-18

36

2) Usar expresiones equivalentes para las siguientes, usando operadores mixtos.

a) `x = 10 + x - y;`

x += 10-y

x -= y+10

x += 10+y

b) `r = 100*r;`

r *= 100*r

r *= 100

r += 100

c) `y = y/(10+x);`

y /= 10*x

y /= 10 + y/x

y /= 10+x

d) `z = 3 * x + 6;`

z += 6

z *= 3

no es posible

3) Evaluar las siguientes expresiones. Siendo:

```
x = 10; y = 20; z = 30;
```

a) `z = x - y, t = z - y;`

z=-10, t=-30

t=10

z=30, t=-30

b) `(x < 10) && (y > 15)`

true

false

c) `(x <= z) || (z <= y)`

true

false

d) `!(x+y < z)`

true

false

e) `(x+y != z) && (1/(z-x-y) != 1)`

true

false

error

Ejercicios del capítulo 4 Operadores I

1) Suponiendo los siguientes valores iniciales para las variables:
`x = 2; y = 6; z = 9; r = 100; s = 10; a = 15; b = 3;`
¿Cuáles son los valores correctos en cada expresión?

a) `x += 10;` (X)

Mal: Equivale a x = x + 10 = 2 + 10 = 12.

b) `s *= b;` (X)

Mal: Equivale a s = s * b = 10 * 3 = 30.

c) `r /= 0;` (X)

Mal: No es posible dividir por cero, se obtiene un error.

d) `y += x + 10;` (X)

Mal: Equivale a y = y + x + 10 = 6 + 2 + 10 = 18.

e) `z -= a*b;` (X)

Mal: Equivale a z = z - a*b = 9 - 15*3 = -36.

2) Usar expresiones equivalentes para las siguientes, usando operadores mixtos.

a) `x = 10 + x - y;` (X)

Mal: Hay que sumar 10 y restar y a x.

b) `r = 100*r;` (X)

Mal: Hay que multiplicar por 100.

c) `y = y/(10+x);` (X)

Mal: Hay que dividir y entre x+10.

d) `z = 3 * x + 6;` (X)

Mal: La variable z debe aparecer en ambos lados de la sentencia de asignación.

3) Evaluar las siguientes expresiones. Siendo:
`x = 10; y = 20; z = 30;`

a) `z = x - y, t = z - y;` (X)

Mal: Las expresiones de coma se evalúan de izquierda a derecha, y los resultados de cada una se deben tener en cuenta para las siguientes.

b) `(x < 10) && (y > 15)` (X)

Mal: x no es menor de 10, luego x<10 es false, no necesitamos proseguir, ya que en una expresión AND, si uno de los operandos es false, el resultado es false.

c) `(x <= z) || (z <= y)` (X)

Mal: x es menor que z, luego x<=z es true, no necesitamos proseguir, ya que en una expresión OR, si uno de los operandos es true, el resultado es true.

d) `!(x+y < z)` (X)

Mal: x+y es igual a z, luego x+y<z es false, y la negación, por lo tanto, true.

e) `(x+y != z) && (1/(z-x-y) != 1)` (X)

Mal: x+y es igual a z, luego x+y != z es false, el compilador no evalúa el resto de la expresión, ya que el resultado de una expresión AND es false si uno de los operandos es false. Esto evita que se evalúe el segundo operador, lo que provocaría una excepción al intentar dividir por cero.

5 Sentencias

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

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

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

- Bloques
- Expresiones
    - Llamada a función
    - Asignación
    - Nula
- Bucles
    - while

    - do while
    - for
- Etiquetas
    - Etiquetas de identificación
    - case
    - default
- Saltos
    - break
    - continue
    - goto
    - return
- Selección
    - if...else
    - switch

Bloques [1] [1]

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 [1] [1]

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

```
<expresión>;
```

Llamadas a función

Esta es la manera de ejecutar las funciones que se definen en otras partes del programa o en

el exterior de éste, ya sea una librería estándar o particular. Consiste en el nombre de la función, una lista de parámetros entre paréntesis y un ";".

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

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

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

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

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

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

Asignación

Las sentencias de asignación responden al siguiente esquema:

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

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

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

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

Nula

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

Bucles [1] [1]

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

Bucles "while"

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

es la siguiente:

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

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

Por ejemplo:

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

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

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

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

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

Bucle "do while"

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

```
do <sentencia> while(<condicion>);
```

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

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

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

Bucle "for"

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

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

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

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

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

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

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

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

O bucles infinitos:

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

Etiquetas [1] [1]

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

Etiquetas de identificación

Corresponden con la siguiente sintaxis:

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

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

Etiquetas "case" y "default"

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

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

Selección [1] [1]

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

Sentencia "if...else"

Implementa la ejecución condicional de una sentencia. Sintaxis:

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

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

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

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

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

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

Sentencia "switch"

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

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

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

Sintaxis:

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

Por ejemplo:

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

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

Otro ejemplo:

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

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

Sentencias de salto [1] [1]

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

Sentencia de ruptura "break"

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

```
break
```

Ejemplo:

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

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

Sentencia de "continue"

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

```
continue
```

Ejemplo:

```
y = 0;
x = 0;
```

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

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

Sentencia de salto "goto"

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

```
goto <identificador>
```

Ejemplo:

```
x = 0;
Bucle:
x++;
if(x < 1000) goto Bucle;
```

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

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

Sentencia de retorno "return"

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

```
return [<expresión>]
```

Ejemplo:

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

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

Sobre las sentencias de salto y la programación estructurada

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

Comentarios

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

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

está permitido, sin embargo:

```
ma/*función*/in(void)
```

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

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

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

Las llaves {} no forman parte del comentario.

Palabras reservadas usadas en este capítulo

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

Ejercicios del capítulo 5 Sentencias

1) Mostrar los sucesivos valores de la variable x en los siguientes bucles:

a)

```
int x=0;
while(x < 5) x += 2;
```

0,2,4,6

0,2,4

0,2,4,6,8

b)

```
int x=10;
do x++; while(x < 10);
```

10

10,11

11

c)

```
bool salir = false;
int x = 13;
while(!salir) {
x++;
salir = x%7;
}
```

13,14

13,14,15

13

d)

```
int x = 6;
do {
switch(x%3) {
case 0: x=10; break;
case 1: x=17; break;
case 2: x=5; break;
}
} while(x != 5);
```

6,10,17

6,10,17,5

6,10,17,10,5

e)

```
int x=0, y=0;
do {
if(x>4) { x %= 4; y++; }
else x++;
} while(y < 2);
```

0,1,2,3,4,5,1,2,3,4,5,1

0,1,2,3,4,5,1,2,3,4,5

0,1,2,3,4,5,1,2,3,4,5,1,2

f)

```
int x=0, y=1;
while(y != 3) {
x++;
if(x<3) continue;
x=y; y++;
}
```

0,1,2,3,1,2,3,2,3

0,1,2,3,1,2,3,2

0,1,2,3,1,2,3,2,3,2

Ejercicios del capítulo 5 Sentencias

1) Mostrar los sucesivos valores de la variable x en los siguientes bucles:

a)

```
int x=0;
while(x < 5) x += 2;
```

0,2,4,6

0,2,4

0,2,4,6,8

b)

```
int x=10;
do x++; while(x < 10);
```

10

10,11

11

c)

```
bool salir = false;
int x = 13;
while(!salir) {
x++;
salir = x%7;
}
```

13,14

13,14,15

13

d)

```
int x = 6;
do {
switch(x%3) {
case 0: x=10; break;
case 1: x=17; break;
case 2: x=5; break;
}
} while(x != 5);
```

6,10,17

6,10,17,5

6,10,17,10,5

e)

```
int x=0, y=0;
do {
if(x>4) { x %= 4; y++; }
else x++;
} while(y < 2);
```

0,1,2,3,4,5,1,2,3,4,5,1

0,1,2,3,4,5,1,2,3,4,5

0,1,2,3,4,5,1,2,3,4,5,1,2

f)

```
int x=0, y=1;
while(y != 3) {
x++;
if(x<3) continue;
x=y; y++;
}
```

0,1,2,3,1,2,3,2,3

0,1,2,3,1,2,3,2

0,1,2,3,1,2,3,2,3,2

6 Declaración de variables

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

Cómo se declaran las variables

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

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

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

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

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

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

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

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

Ámbito de las variables 

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

Las variables declaradas dentro de un bucle, serán accesibles sólo desde el propio bucle, serán de ámbito local del bucle.

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

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

Ejemplo:

```
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 valiable 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;
   else
      return 0;
}
```

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

```
<tipo> funcion(parámetros) // (1)
{
   <tipo> var;             // (2)
   for(<tipo> var;...)     // (3)
   ...
   return var;
}
```

(1) los parámetros tienen ámbito local a la función.

(2) las variables declaradas aquí, también.

(3) las declaradas en bucles, son locales al bucle.

Es una buena costumbre inicializar las variables locales. Cuando se trate de variables estáticas se inicializan automáticamente a cero

Ejercicios del capítulo 6 Declaración de variables

1) En el siguiente ejemplo, ¿qué ámbito tiene cada una de las

variables?:

```
float                                                    s,i;

int                                                   main()
{
   int                                                   x;

   x=10;
   for(int                 i=0;                 i<x;                 i++)
      Mostrar(i);
   i                                =                     0.0;
   while(x>0)                                                {
      i                              *=                  10.3;
      x--;
   }
   return                                                   0;
}
```

a) La variable de tipo float s tiene ámbito

- global
- local en main
- local en bucle

b) La variable de tipo int i tiene ámbito

- global
- local en main
- local en bucle

c) La variable de tipo float i tiene ámbito

- global
- local en main
- local en bucle

d) La variable de tipo int x tiene ámbito

- global
- local en main
- local en bucle

Ejercicios del capítulo 6
Declaración de variables

1) En el siguiente ejemplo, ¿qué ámbito tiene cada una de las variables?:

```
float                                                    s,i;

int                                                    main()
{
   int                                                    x;

   x=10;
   for(int                 i=0;                 i<x;                 i++)
      Mostrar(i);
   i                                  =                    0.0;
   while(x>0)                                                 {
      i                                *=                 10.3;
      x--;
   }
   return                                                   0;
}
```

a) La variable de tipo float s tiene ámbito (X)

Mal: La variable s está declarada fuera de cualquier función, es por lo tanto, global.

b) La variable de tipo int i tiene ámbito (X)

Mal: La declaración de la variable i dentro del bucle "for" enmascara a la variable global i, de tipo float.

c) La variable de tipo float i tiene ámbito (X)

Mal: La variable i está declarada fuera de cualquier función, es por lo tanto, global, aunque sea inaccesible desde parte del programa.

d) La variable de tipo int x tiene ámbito (X)

Mal: La variable x está declarada dentro de la función main, por lo tanto es local en sa función.

7 Normas para la notación

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

Constantes "int"

En C se usan 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.

En notación octal se usan sólo los dígitos del '0' al '7', en hexadecimal, se usan 16 símbolos, los dígitos del '0' al '9' tienen el mismo valor que en decimal, para los otros seis símbolos se usan las letras de la 'A' a la 'F', indistintamente en mayúsculas o minúsculas. Sus valores son 10 para la 'A', 11 para la 'B', y sucesivamente, hasta 15 para la 'F'.

Según el ejemplo el número 0x7f, donde "0x" es el prefijo que indica que se trata de un número en notación hexadecimal, sería el número 7F, es decir, $7*16+15=127$. Del mismo modo que el número 127 en notación decimal sería, $1*10^2+2*10+7=127$. En octal se usa como prefijo el dígito 0. El número 0177 equivale a $1*8^2+7*8+7=127$.

Hay que tener mucho cuidado con las constantes numéricas, en C y 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 se puede tomar valores hexadecimales entre 0x00 y 0xff, dos bytes entre 0x0000 y 0xffff, etc. Además, la conversión a binario es casi directa, cada dígito hexadecimal se puede sustituir por cuatro bits, el '0x0' por '0000', el '0x1' por '0001', hasta el '0xf', que equivale a '1111'. En el ejemplo el número 127, o 0x7f, sería en binario '01111111'.

Con la numeración octal es análogo, salvo que cada dígito agrupa tres bits. Así un byte se puede tomar valores octales entre 0000 y 0377, dos bytes entre 0000000 y 0177777, etc. Además, la conversión a binario es casi directa, 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, o 0177, sería en binario '01111111'.

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

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

Que:

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

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

Constantes "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(x == 0L) cout << "Valor nulo" << endl;
```

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

```
if(x == 1343890883) cout << "Número long int" << endl;
```

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"

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

```
long long x = 16575476522787LL;
if(x == 1LL) cout << "Valor nulo" << endl;
```

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

```
if(x == 16575476522787) cout << "Número long long" << endl;
```

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(x == 3124232U) cout << "Valor encontrado" << endl;
```

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(x == 3124232UL) cout << "Valor encontrado" << endl;
```

Constantes "unsigned long long"

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

```
unsigned long long x = 123456534543ULL;
if(x == 3124232ULL) cout << "Valor encontrado" << endl;
```

Constantes "float"

También 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.

Si se usa el sufijo "f" 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"

Si no se usa el sufijo, se tratará de 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.

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, este entero, que es el exponente de una potencia de base 10.

Los valores anteriores son:

```
x = 10 x 10^4 = 100000
y = 4,12 x 10^2 = 412
pi = 3.141592 x 10^0 = 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.

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

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

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

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

Escape	Hexad	ANSI	Nombre o resultado
	0x00	NULL	Carácter nulo
\a	0x07	BELL	Sonido de campanilla
\b	0x08	BS	Retroceso

\f	0x0C	FF	Avance de página
\n	0x0A	LF	Avance de línea
\r	0x0D	CR	Retorno de línea
\t	0x09	HT	Tabulador horizontal
\v	0x0B	VT	Tabulador vertical
\\	0x5c	\	Barra descendente
\'	0x27	'	Comilla sencilla
\"	0x22	"	Comillas
\?	0x3F	?	Interrogación
\O		cualquiera	O=tres dígitos en octal
\xH		cualquiera	H=número hexadecimal
\XH		cualquiera	H=número hexadecimal

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

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

En este ejemplo todas las asignaciones son equivalentes y válidas.

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?

En todos estos casos, 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.

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. Lo mismo se aplica a las constantes long, unsigned y char.

Ejercicios del capítulo 7 Normas para la notación

1) ¿Qué tipo de constante es cada un de las siguientes?:

a) `'\x37'`

- char
- long
- int
- float

b) `123UL`

- unsigned
- int
- long
- unsigned long

c) `34.0`

- int
- double
- float

- long

d) `6L`

- int
- long
- double
- char

e) `67`

- char
- unsigned
- int
- float

f) `0x139`

- char
- unsigned
- int
- float

g) `0x134763df23LL`

- long
- unsigned
- int
- long long

Ejercicios del capítulo 7 Normas para la notación

1) ¿Qué tipo de constante es cada un de las siguientes?:

a) `'\x37'` (X)

Mal: Las comillas simples son la clave en este caso, se trata de un carácter expresado en hexadecimal.

b) `123UL` (X)

Mal: Evidentemente, el sufijo UL indica unsigned long.

c) `34.0` (X)

Mal: El punto implica que se trata de una constante en coma flotante, pero se usa el formato double en lugar de float.

d) `6L` (X)

Mal: El sufijo L indica que se trata de una constante long.

e) `67` (X)

Mal: Si no se especifica otra cosa, se trata de una constante entera (int).

f) `0x139` (X)

Mal: Sólo se especifica que se usa la numeración hexadecimal, pero se trata de una constante entera (int), ya que no añadimos sufijo alguno.

g) `0x134763df23LL` (X)

Mal: En este caso, si no se especifica el sufijo LL, para indicar que se trata de una constante long long, el compilador dará error por sobrepasarse es rango posible para un int.

8 Cadenas de caracteres

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

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

La manera de definir una cadena es la siguiente:

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

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

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

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

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

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

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

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

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

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.

La manera correcta de asignar una cadena es:

```
char Saludo[5];
Saludo[0] = 'H';
Saludo[1] = 'O';
Saludo[2] = 'L';
Saludo[3] = 'A';
Saludo[4] = '\000';
```

O bien:

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

Si te parece un sistema engorroso, no te preocupes, en próximos capítulos veremos funciones que facilitarán la asignación de cadenas. Existen muchas funciones para el tratamiento de cadenas, como veremos, 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.

Ejercicios del capítulo 8 Cadenas de caracteres

1) Teniendo en cuenta la asignación que hemos hecho para la cadena Saludo, hemos escrito varias versiones de una función que calcule la longitud de una cadena, ¿cuáles de ellas funcionan y cuáles no?:

a)

```
int LongitudCadena(char cad[])
{
int l = 0;
while(cad[l]) l++;
return l;
}
```

Sí No

b)

```
int LongitudCadena(char cad[])
{
int l;
for(l = 0; cad[l] != 0; l++);
return l;
}
```

Sí No

c)

```
int LongitudCadena(char cad[])
{
int l = 0;
do {
l++;
} while(cad[l] != 0);
return l;
}
```

Sí No

Ejercicios del capítulo 8 Cadenas de

caracteres

1) Teniendo en cuenta la asignación que hemos hecho para la cadena Saludo, hemos escrito varias versiones de una función que calcule la longitud de una cadena, ¿cuáles de ellas funcionan y cuáles no?:

a)

```
int LongitudCadena(char cad[])
{
int l = 0;
while(cad[l]) l++;
return l;
}
```

(X)

Mal: Bueno, no hacemos más que contar hasta encontrar el nulo.

b)

```
int LongitudCadena(char cad[])
{
int l;
for(l = 0; cad[l] != 0; l++);
return l;
}
```

(X)

Mal: Bueno, no hacemos más que contar hasta encontrar el nulo, pero con un bucle "for".

c)

```
int LongitudCadena(char cad[])
{
int l = 0;
do {
l++;
} while(cad[l] != 0);
return l;
}
```

(X)

Mal: Hay dos errores en esta función: el primero es que siempre dará como resultado uno más de la longitud de la cadena, ya que hemos usado un bucle "do...while". El segundo es que no funcionará con cadenas nulas, de cero caracteres.

9 Conversión de tipos

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

```
char n;
```

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

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

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.

Si algún operando es de tipo long double, el otro se convertirá a long double.

Si algún operando es de tipo double, el otro se convertirá a double.

Si algún operando es de tipo float, el otro se convertirá a float.

Si algún operando es de tipo unsigned long long, el otro se convertirá a unsigned long long.

Si algún operando es de tipo long long, el otro se convertirá a long long.

Si algún operando es de tipo unsigned long, el otro se convertirá a unsigned long.

Si algún operando es de tipo long, el otro se convertirá a long.

Si algún operando es de tipo unsigned int, el otro se convertirá a unsigned int.

En este caso ambos operandos son int.

Veamos ahora el ejemplo:

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

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

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

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

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

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

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

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

"Casting", conversiones explícitas de tipo:

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

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

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 formas:

```
(<nombre de tipo>)<expresión>
```

ó

```
<nombre de tipo>(<expresión>)
```

Esta última es conocida como notación funcional.

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

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

ó:

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

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

Ejercicios del capítulo 9 Conversión de tipos

1) Supongamos que tenemos estas variables:

```
int a=10;
float b=19.3;
double d=64.8;
char c=64;
```

Indicar el tipo resultante para las expresiones siguientes:

a) `a+b`

- char
- int
- float
- double

b) `c+d`

- char
- int
- float
- double

c)

```
(int)d+a
```

- char
- int
- float
- double

d)

```
d+b
```

- char
- int
- float
- double

e)

```
(float)c+d
```

char

int

float

double

10 Tipos de variables II: Arrays

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

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

Sintaxis:

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

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

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

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

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

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

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

Como subíndice podremos usar cualquier expresión entera.

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

Ejemplo:

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

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

Inicialización de arrays: 1 1

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};
```

En estos casos 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.

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: 1 1

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 dividiendo ese valor entre el tamaño de uno de los elementos.

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

int main()
{
   int array[231];

   cout << "Número de elementos: "
        << sizeof(array)/sizeof(int) << endl;
   cout << "Número de elementos: "
        << sizeof(array)/sizeof(array[0]) << endl;
```

```
    cin.get();
    return 0;
}
```

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

Algoritmos de ordenación, método de la burbuja: 1 1

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

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

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

Lo veremos mejor con un ejemplo:

Ordenar la siguiente lista de menor a mayor:

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

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

3, 15, 8, 6, 18, 1

Tomamos el siguiente par de valores: 15 y 8, y volvemos a intercambiarlos, y 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: Tenemos una sección sobre algoritmos de ordenación en la página: http://c.conclase.net/orden/ realizada por Julián Hidalgo.

Problemas (creo que ya podemos empezar :-) : 1 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.

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.

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.

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.

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 variables III: Estructuras

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

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

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

Sintaxis:

```
struct [<identificador>] {
   [<tipo> <nombre_variable>[,<nombre_variable>,...]];
   .
} [<variable_estructura>[,<variable_estructura>,...];
```

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

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

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

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

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

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

En C++ la palabra "struct" es opcional en la declaración de variables. En C es obligatorio usarla.

Ejemplo:

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

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

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

Funciones en el interior de estructuras: 1 1

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

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

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

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

Nota: para aquellos que usen un teclado español, el símbolo "~" se obtiene pulsando las teclas del teclado numérico 1, 2, 6, mientras se mantiene pulsada la tecla ALT, ([ALT]+126). También mediante la combinación [Atl Gr]+[4] (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.

Mencionar aquí, sólo a título de información, que el constructor no tiene por qué ser único. Se pueden incluir 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, salvo que tienen restringido su ámbito al interior de la estructura.

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;

   cin.get();
   return 0;
}
```

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

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

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

Inicialización de estructuras: 1 1

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: [1] [1]

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

Veamos un ejemplo:

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

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

La línea:

```
Punto2 = Punto1;
```

equivale a:

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

Arrays de estructuras: [1] [1]

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: 1 1

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

Ejemplo:

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

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

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

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

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

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

```
stDireccion DireccionAuxiliar;
```

Estructuras anónimas: 1 1

Una estructura anónima es la que carece de identificador de tipo de estructura y de declaración de variables 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?

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: [1] [1]

Cuando se aplica el operador sizeof a una estructura, el tamaño obtenido no siempre coincide con el tamaño de la suma de sus campos. 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;

   cin.get();
   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-aling). 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 4, y el compilador intenta alinear las variables con esas posiciones.

En el caso de variables "int" es fácil, ya que ocupan 4 bytes, pero con las variables "char" no, ya que sólo ocupan 1.

Cuando se accede a datos de menos de 4 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 4, dejando 3 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: [1] [1]

Existe otro tipo de estructuras que consiste en empaquetar los campos de la estructura en el interior de enteros, usando bloques o conjuntos de bits para cada campo.

Por ejemplo, una variable 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

una variable int de 16 bits podremos almacenar 16 bits, etc.

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_variables>];
```

Hay algunas limitaciones, por ejemplo, un campo de bits no puede ocupar dos variables distintas, todos sus bits tienen que estar en el mismo valor entero.

Veamos algunos ejemplos:

```
struct mapaBits {
   unsigned char bit0:1;
   unsigned char bit1:1;
   unsigned char bit2:1;
   unsigned char bit3:1;
   unsigned char bit4:1;
   unsigned char bit5:1;
   unsigned char bit6:1;
   unsigned char bit7:1;
   };

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;
};

struct mapaBits3 {
   unsigned char campo1:5;
   unsigned char campo2:5;
};
```

En el primer caso se divide un valor char sin signo en ocho campos de un bit cada uno:

7	6	5	4	3	2	1	0
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0

En el segundo caso dividimos un valor entero sin signo de dieciséis bits en cinco campos de distintas longitudes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
campo5						campo4	campo3		campo2				campo1		

Los valores del campo5 estarán limitados entre 0 y 63, que son los números que se pueden codificar con seis bits. Del mismo modo, el campo4 sólo puede valer 0 ó 1, etc.

unsigned char								unsigned char							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
			campo2								campo1				

En este ejemplo vemos que como no es posible empaquetar el campo2 dentro del mismo char que el campo1, se añade un segundo valor char, y se dejan sin usar 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 en general igual que cualquier otro de los campos de una estructura. Se les puede asignar valores (dentro del rango que admitan), pueden usarse en condicionales, imprimirse, etc.

```
#include <iostream>
#include <cstdlib>
using namesoace 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;
```

```
    cin.get();
    return 0;
}
```

No es normal 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 los bits de datos, 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 1

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

a) Buscar por nombre
b) Buscar por número de teléfono
c) Salir

Pulsa una opción:

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

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 de 1 a 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 año. 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 año.
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: dia, mes y anno.

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.

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.

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.

Ejercicios del capítulo 11 Estructuras

1) Supongamos la declaración de la siguiente estructura:

```
struct ejemplo1 {
unsigned char c1:7;
unsigned char c2:6;
unsigned char c3:3;
unsigned char c4:4;
};:
```

a) ¿Cuántos bytes ocupa esta estructura?

1

2

3

4

b) ¿Y si en lugar de un "unsigned char" usamos un "unsigned short" de 16 bits?

1

2

3

4

c) ¿Y si en lugar de un "unsigned short" usamos un "unsigned int" de 32 bits?

1

2

3

4

2) Tenemos la siguiente estructura:

```
struct A {
struct B {
int x,y;
float r;
} campoB;
float s;
struct {
int z;
float t;
} campoC;
} E;
```

Si tenemos la variable "E", indicar la forma correcta de acceder a las siguientes variables:

a) `x`

A.B.x

E.campoB.x

E.x

E.b.x

b) `s`

A.s

E.s

E.a.s

c) `t`

A.t

E.t

A.campoC.t

E.campoC.t

12 Tipos de variables IV: Punteros 1

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

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

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

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

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

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

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

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

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

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

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]. Prescindiendo de la notación de los arrays, el índice se comporta exactamente igual que un puntero.

grafico

El puntero índice podría tener por ejemplo, el valor 3, en ese caso, *indice tendría el valor 'valor3'.

Las celdas de memoria existirán independientemente del valor de indice, o incluso de la existencia de indice, por lo tanto, la existencia del puntero no implica nada más que eso, pero no que el valor de la dirección que contiene sea un valor válido de memoria.

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.

El propio puntero, como variable que es, ocupará ciertas direcciones de memoria.

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. El sistema operativo suele controlar la memoria, y no tiene por costumbre permitir el acceso al resto de la memoria.

Declaración de punteros: [1] [1]

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

Sintaxis:

```
<tipo> *<identificador>;
```

Ejemplos:

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

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

La forma:

```
<tipo>* <identificador>;
```

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

Veamos algunos matices. Tomemos el primer ejemplo:

```
int *entero;
```

equivale a:

```
int* entero;
```

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

Como pasa con todas las variables en C++, cuando se declaran sólo se reserva espacio para almacenarlas, pero no se asigna ningún valor inicial, el contenido de la variable permanecerá sin cambios, de modo que el valor inicial del puntero será aleatorio e indeterminado. Debemos suponer que contiene una dirección no válida.

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

Obtener punteros a variables: l l

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

Por supuesto, los tipos tienen que ser "compatibles", no podemos almacenar la dirección de una variable 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.

Diferencia entre punteros y variables: l l

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

```
int A, B;
int *entero;
...
B = 213; /* B vale 213 */
entero = &A;   /* entero apunta a la
```

```
                  dirección de la variable A */
*entero = 103; /* equivale a la línea A = 103; */
B = *entero;   /* equivale a B = A; */
...
```

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

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

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

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

Correspondencia entre arrays y punteros: 1 1

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

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

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

Ejemplo:

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

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

¿Qué hace cada una de estas instrucciones?:

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

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

Análogamente la operación:

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

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

Otro ejemplo:

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

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

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

Operaciones con punteros: 1 1

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

Asignación.

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

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

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

Operaciones aritméticas.

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

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

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

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

```
cout << q-p << endl;
```

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

La suma de punteros no está permitida.

Comparación entre punteros.

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

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

```
if(NULL != p)
```

o simplemente

```
if(p)
```

Y también:

```
if(NULL == p)
```

O simplemente

```
if(!p)
```

Punteros genéricos. [1] [1]

Es posible declarar punteros sin tipo concreto:

```
void *<identificador>;
```

Estos punteros pueden apuntar a objetos de cualquier tipo.

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

```
(<tipo> *)<variable puntero>
```

Por ejemplo:

```
#include <iostream>
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;
```

```
   cin.get();
   return 0;
}
```

El resultado será:

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

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

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

Punteros a estructuras: [1] [1]

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

   cin.get();
   return 0;
}
```

Ejemplos: [1] [1]

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;

   cin.get();
   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;

   cin.get();
   return 0;
}
```

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

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

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

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

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

Variables dinámicas: [1] [1]

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

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

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

Hay una regla de oro cuando se usa memoria dinámica, toda la memoria que se reserve durante el programa hay que liberarla antes de salir del programa. No seguir esta regla es una actitud muy irresponsable, y en la mayor parte de los casos tiene consecuencias desastrosas. No os 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;

   cin.get();
   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. Si no la necesitábamos habría que liberarla antes de reservarla otra vez, y si la necesitamos, hay que guardar su dirección, por ejemplo con otro puntero.

Problemas: 1 1

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

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

Ejercicios del capítulo 12 Punteros

1) ¿De qué tipo es cada una de las siguientes variables?:

a) `int* a,b;`

a puntero, b puntero

a puntero, b entero

a entero, b puntero

a entero, b entero

b) `int *a,b;`

a puntero, b puntero

a puntero, b entero

a entero, b puntero

a entero, b entero

c) `int *a,*b;`

a puntero, b puntero

a puntero, b entero

a entero, b puntero

a entero, b entero

d) `int* a,*b;`

a puntero, b puntero

a puntero, b puntero doble

a entero, b puntero

a entero, b puntero doble

2) Considerando las siguientes declaraciones y sentencias:

```
int array[]={1,2,3,4,5,6};
int *puntero;
puntero = array;
puntero++;
*puntero=*puntero+6;
puntero=puntero+3;
puntero=puntero-puntero[-2];
int x=puntero-array;
:
```

a) ¿Cuál es el valor de x?

1

2

3

4

b) ¿Cual es el valor de array[1]?

2

4

6

8

3) Considerando la siguiente declaración:

```
struct A {
struct {
int x;
int y;
} campoB;
} *estructuraA;
```

a) ¿Cómo se referenciaría el campo x de la estructuraA?

estructuraA.x

estructuraA.campoB.x

x" name=o3_aestructuraA.campoB->x

campoB.x" name=o3_aestructuraA->campoB.x

Ejercicios del capítulo 12 Punteros

1) ¿De qué tipo es cada una de las siguientes variables?:

a) `int* a,b;`
(X)

Mal: El asterisco se asocia al nombre de la variable, no al tipo.

b) `int *a,b;`
(X)

Mal: El asterisco se asocia al nombre de la variable, no al tipo. En este caso es más evidente.

c) `int *a,*b;`
(X)

Mal: El asterisco se asocia al nombre de la variable. Este caso es evidente.

d) `int* a,*b;`
(X)

Mal: El asterisco se asocia al nombre de la variable, no al tipo. En el caso de a es claro, el caso de b puede que no tanto, pero el primer asterisco pertenece a la variable 'a'.

2) Considerando las siguientes declaraciones y sentencias:

```
int array[]={1,2,3,4,5,6};
int *puntero;
puntero = array;
puntero++;
*puntero=*puntero+6;
puntero=puntero+3;
puntero=puntero-puntero[-2];
int x=puntero-array;
:
```

a) ¿Cuál es el valor de x? (X)

Mal: Inicialmente puntero apunta a array, después se incrementa en una unidad, y después en tres más, es decir, apunta a &array[4].puntero[-2] es por lo tanto array[2], que tiene valor 3. De modo que volvemos a decrementar puntero e tres unidades, y apunta a &array[1]. Por lo tanto, x=puntero-array=&array[1]-&array[0]=1.

b) ¿Cual es el valor de array[1]? (X)

Mal: Después de incrementar puntero por primera vez, apunta a &array[1]. En ese momento es cuando sumamos 6 a *puntero, que previamente vale 2, el resultado es 8, que se almacena en *puntero, que equivale a &array[1].

3) Considerando la siguiente declaración:

```
struct A {
struct {
int x;
int y;
} campoB;
} *estructuraA;
```

a) ¿Cómo se referenciaría el campo x de la estructuraA? (X)

Mal: estructuraA es un puntero, por lo tanto, para hacer referencia a uno de sus campos necesitamos usar el operador "->", no pasa lo mismo con campoB, que es un miembro estructura, no puntero.

13 Operadores II: Más operadores

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

Operadores de Referencia (&) e Indirección (*) [1] [1]

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

Sintaxis:

```
&<expresión simple>
```

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

Sintaxis:

```
*<puntero>
```

Operadores . y -> [1] [1]

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

Sintaxis:

```
<variable_estructura>.<nombre_de_variable>
```

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

Sintaxis:

```
<puntero_a_estructura>-><nombre_de_variable>
```

Operador de preprocesador [1] [1]

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

Veremos, sin embargo dos de las más usadas.

Directiva define:

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

Sintaxis:

```
#define <identificador_de_macro> <secuencia>
```

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

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

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

Existen otras restricciones a la expansión de macros:

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

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

Ejemplo:

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

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

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

#define mult1(a,b) a*b
#define mult2(a,b) ((a)*(b))
```

```
int main() {
   // En este caso ambas macros funcionan bien:
   cout << mult1(4,5) << endl;
   cout << mult2(4,5) << endl;
   // En este caso la primera macro no funciona, ¿por qué?:
   cout << mult1(2+2,2+3) << endl;
   cout << mult2(2+2,2+3) << endl;

   cin.get();
   return 0;
}
```

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

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

   cin.get();
   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 capítulo 25 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" [1] [1]

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 dinámicas.

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.

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

Veamos algunos ejemplos:

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

   // 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 los propios 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];
```

Palabras reservadas usadas en este capítulo

delete, new, sizeof.

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.

Ejercicios del capítulo 14 Precedencia

1) Dadas las siguientes variables:

```
int a = 10, b = 100, c = 30, d = 1, e = 54;
int m[10] = {10,20,30,40,50,60,70,80,90,100};
int *p = &m[3], *q = &m[6];
```

Evaluar, sin usar un compilador, las siguientes expresiones.
Considerar que los resultados de cada una de las expresiones no influyen en las siguientes:

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

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

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

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

e) `(b/a) * (m[3] * ++e);`

f) `++*p+++*q;`

g) `++*p + ++*q;`

h) `m[c/a]-*p;`

i) `q[-3] + q[2];`

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 "variables" locales de la función, estas "variables" 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;

   cin.get();
   return 0;
}

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

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

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

Piensa, 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 son valores constantes. Si no fuese así, no sería posible llamar a la función con estos valores.

Referencias a variables: [1] [1]

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

Sintaxis:

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

La primera forma es la que se usa para declarar variables de referencia, la asignación es

obligatoria ya que no pueden definirse referencias indeterminadas.

La segunda forma es la que se usa para definir parámetros por referencia en funciones, en las que las asignaciones son implícitas.

Ejemplo:

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

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

   a = 10;
   cout << r << endl;

   cin.get();
   return 0;
}
```

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

Pasando parámetros por referencia: [1] [1]

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

```
#include <iostream>
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 */

   cin.get();
   return 0;
}

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

```
   return n+m;
}
```

En este caso, las variables "a" y "b" tendrán valores distintos después de llamar a la función. Cualquier cambio que realicemos en los parámetros dentro de la función, se hará también en las variables referenciadas. Esto quiere decir que no podremos llamar a la función con parámetros constantes, como se indica en (1), ya que no se puede definir una referencia a una constante.

Punteros como parámetros de funciones: 1 1

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

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);
   cout << "Variable a: " << a << endl;
   cout << "Variable *p: " << *p << endl;
   // Llamada a funcion con la dirección de "a" (constante)
   funcion(&a);
   cout << "Variable a: " << a << endl;
   cout << "Variable *p: " << *p << endl;

   cin.get();
   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.

Con este tipo de parámetro para función pasamos el puntero por valor. ¿Y cómo haríamos

para pasar un puntero por referencia?:

```
void funcion(int* &q);
```

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

Arrays como parámetros de funciones: 1 1

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

Sin embargo, si sólo pasamos el nombre del array de más de una dimensión no podremos acceder a los elementos del array mediante 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;
...
}
```

Estructuras como parámetros de funciones: 1 1

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.

Funciones que devuelven referencias: [1] [1]

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 un objeto, y podremos hacer cosas como asignar valores a una llamada a función. 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;

   cin.get();
   return 0;
}

int &Acceso(int* vector, int indice) {
   return vector[indice];
}
```

Esta es una potente herramienta de la que disponemos, aunque ahora no se nos ocurra ninguna aplicación interesante.

Veremos en el capítulo sobre sobrecarga que este mecanismo es imprescindible.