



Introducción a la programación C



Los propietarios del © tienen reservados todos los derechos. Cualquier reproducción, total o parcial de este texto, por cualquier medio, o soporte sonoro, visual o informático, así como su utilización fuera del ámbito estricto de la información del alumno comprador, sin la conformidad expresa por escrito de los propietarios de los derechos, será perseguida con todo el rigor que prevé la ley y se exigirán las responsabilidades civiles y penales, así como las reparaciones procedentes.

Introducción a la programación C

Autor: SEAS, Estudios Superiores Abiertos

Imprime: El depositario, con autorización expresa de SEAS

D.L.: Z-1896-2015

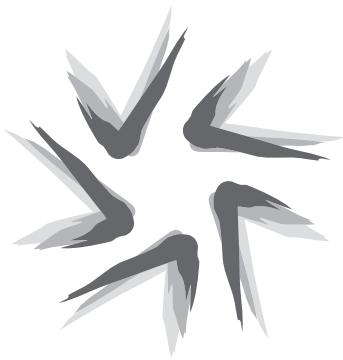
ISBN: 978-84-16442-63-8



ÍNDICE GENERAL

1. Introducción a la programación C	5
ÍNDICE	7
OBJETIVOS	9
INTRODUCCIÓN	10
1.1. Principios básicos.....	11
1.2. El lenguaje C.....	19
1.3. Estructura de un programa en C	22
1.4. Definición de variables.....	29
1.5. Constantes	35
1.6. Operadores, expresiones y sentencias	39
1.7. Entrada y salida de información.....	48
RESUMEN	57
2. Programación estructurada en C	59
ÍNDICE	61
OBJETIVOS	63
INTRODUCCIÓN	64
2.1. Programación estructurada en C	65
2.2. Estructuras de selección o decisión.....	66
2.3. Estructuras iterativas	79
RESUMEN	89
3. Tipos de datos estructurados en C	91
ÍNDICE	93
OBJETIVOS	95
INTRODUCCIÓN	96
3.1. Arrays unidimensionales o vectores	97
3.2. Declaración de cadenas.....	107
RESUMEN	113

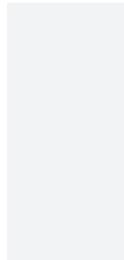
4. Diseño modular en C.....	115
ÍNDICE	117
OBJETIVOS	119
INTRODUCCIÓN	120
4.1. Tipos de datos derivados	121
4.2. Funciones en C	128
4.3. Paso de parámetros a las funciones	133
4.4. Alcance de las variables: globales y locales	138
RESUMEN	147



estudios abiertos

SEAS

GRUPO SANVALERO



1
UNIDAD
DIDÁCTICA

Introducción a la programación C

1. Introducción a la programación C

ÍNDICE

ÍNDICE	7
OBJETIVOS	9
INTRODUCCIÓN	10
1.1. Principios básicos	11
1.1.1. ¿Qué es programa?.....	11
1.1.2. Fase de compilación y linkado.....	12
1.1.3. Tipos de errores al programar	14
1.1.4. ¿Qué es una función?	15
1.2. El lenguaje C	19
1.3. Estructura de un programa en C	22
1.3.1. Componentes sintácticos.....	24
1.4. Definición de variables	29
1.4.1. Inicialización de variables	30
1.4.2. Tipos de variables	31
1.5. Constantes	35
1.6. Operadores, expresiones y sentencias	39
1.6.1. Operadores	39
1.6.2. Expresiones.....	44
1.6.3. Reglas de precedencia y asociatividad.....	45
1.6.4. Sentencias	47
1.7. Entrada y salida de información	48
1.7.1. Salida de información: escritura	48
1.7.2. Entrada de información: lectura	53
1.7.3. Macros getchar() y putchar()	55
RESUMEN	57



OBJETIVOS

- Antes de crear y de ejecutar programas en C hay que entender la estructura de los programas en C. Se verá la estructura de los programas, donde se definen los datos y donde se pueden escribir las instrucciones.
- Relacionar la expresión de algoritmos mediante las herramientas vistas en la unidad didáctica anterior y la transformación al lenguaje de programación C.
- Aprender a declarar constantes y variables en C.
- En esta unidad didáctica se aprenderá a evaluar expresiones en C mediante la sintaxis correcta. Se evaluarán expresiones que se han visto en algunas de las herramientas para expresar algoritmos, tales como diagramas de flujo o pseudocódigo principalmente.
- Se aprenderá el manejo de las instrucciones básicas de lectura y de escritura. Varemos que resultado producen cada una de las dos sentencias que hay, tanto de escritura como de lectura.

INTRODUCCIÓN



En esta unidad didáctica se comenzarán a ejecutar programas sencillos en los que las instrucciones que se van a utilizar principalmente serán las instrucciones de asignación e instrucciones de lectura y escritura, que es lo que hemos visto hasta ahora a nivel algorítmico,

Previamente se ha de conocer cual es la sintaxis de todos los programas en C. Hay una sintaxis general para todos los programas en las que hay distintas secciones que se usan para definir constantes utilizadas, variables utilizadas etc. No es obligatorio utilizar todas las secciones, sólo hay que declarar aquellas que se vayan a utilizar.

En este módulo se describe de forma abreviada la sintaxis del lenguaje C. se trata de presentar los recursos o las posibilidades que el C pone a disposición de los programadores.

Conocer un vocabulario y una gramática no equivale a saber un idioma. Conocer un idioma implica además el hábito de combinar sus elementos de forma semiautomática para producir frases que expresen lo que uno quiere decir. Conocer las palabras, las sentencias y la sintaxis del C no equivalen a saber programar, pero son condición necesaria para estar en condiciones de empezar a hacerlo, o de entender cómo funcionan programas ya hechos. El proporcionar la base necesaria para aprender a programar en C es el objetivo de estas páginas.

C++ puede ser considerado como una extensión de C. En principio, casi cualquier programa escrito en ANSI C puede ser compilado con un compilador de C++. El mismo programa, en un fichero con extensión `*.c` puede ser convertido en un programa en C++ cambiando la extensión a `*.cpp`. C++ permite muchas más posibilidades que C, pero casi cualquier programa en C, con algunas restricciones, es aceptado por un compilador de C++.

1.1. Principios básicos

Un pc es un sistema capaz de *almacenar* y *procesar* con gran rapidez una *gran cantidad* de información. Los ordenadores de ahora pueden realizar conexiones entre ellos a través de cable o wifi etc. de esta manera son capaces de comunicarse entre sí, y poder transmitir datos a través de este medio.

Los ordenadores usan sistemas digitales, es decir que trabajan y procesan impulsos eléctricos que los convertimos a ceros y unos, por ello pueden trabajar a gran velocidad. La memoria de un ordenador contiene millones de transistores, y al no tener partes móviles de forma mecánica puedes cambiar de estado muchas veces por segundo, de una manera rápida.



En la actualidad los PC, están presentes en todas las partes y forma parte de nuestra vida, de forma que algunas veces es indispensable para realizar algunas tareas.

Un PC puede realizar una multitud de tareas, desde el sistema y gestión financiera de una empresa hasta las cuentas de una casa, procesamiento de textos, diseño digital, juegos etc...

Existen muchas *aplicaciones* capaces de resolver los más variados problemas, pero además, cuando lo que uno busca no está disponible en el mercado, el usuario puede realizar por sí mismo los programas que necesite.

Este es el objetivo de los *lenguajes de programación*, de los cuales el C es probablemente el más utilizado en la actualidad. Este es el lenguaje que será presentado a continuación.

1.1.1. ¿Qué es programa?

Un *programa* está constituido por un conjunto de *instrucciones* que se ejecutan secuencialmente, una detrás de otra. Actualmente, para decremento el tiempo de ejecución de los programas debido a su complejidad, se está desarrollando programas paralelos, de tal forma que pueden ejecutar *simultáneamente* en varios procesadores.

Los datos e instrucciones formadas por unos y ceros que un procesador es capaz de interpretar, se le llama lenguaje maquina o binario, y es uno de los más complejos de programar, después de este tipo de lenguaje se pasó a desarrollar los lenguajes de alto nivel, por ejemplo, Cobol, Fortran...etc. Que es mucho más cercano al lenguaje natural.

```
-u 100 la
OCFD:0100 8A0B01
OCFD:0103 B409
OCFD:0105 CD21
OCFD:0107 B400
OCFD:0109 CD21
-d 10b 13f
OCFD:0100 48 6F 6C 61 2C
OCFD:0101 20 65 73 74 65 20 65 73-20 75 6E 20 70 72 6F 67
OCFD:0102 72 61 6D 61 20 68 65 63-68 6F 20 65 6E 20 61 73
OCFD:0103 73 65 6D 62 6C 65 72 20-70 61 72 61 20 6C 61 20
OCFD:0104 57 69 68 69 70 65 64 69-61 24
OCFD:0105 Hola,
OCFD:0106 este es un progr
OCFD:0107 ama hecho en as
OCFD:0108 sembler para la
OCFD:0109 wikipedia$
```

Figura 1.1. Lenguaje máquina.

Lenguajes de alto nivel está basado en identificadores, pero algunos lenguajes se les llama rutinas o procedimientos y en C se pasan a denominar FUNCIONES. Cada lenguaje plantea su propia *sintaxis* o conjunto de reglas con las que se indica las operaciones que se quiere realizar.

Para la elaboración de un programa hay que distinguir entre las siguientes dos fases:

- Fase de compilación y linkado (link, montado o enlace).
- Fase de ejecución de un programa.

Los *lenguajes de alto nivel* son más claros para el programador, pero no para el procesador, por lo que para que pueda ejecutarse el programa es necesario traducirlo a su propio lenguaje máquina, esta es una tarea que realiza un programa especial llamado **compilador**.

1.1.2. Fase de compilación y linkado

Un programa escrito en un lenguaje de alto nivel, no puede ser ejecutado directamente por un ordenador, sino que debe ser traducido a lenguaje máquina.

Definiciones:

- **Programa fuente:** programa escrito en un lenguaje de alto nivel o lenguaje ordinario, en bruto sin traducir.
- **Compilador:** programa encargado de traducir los archivos o programas fuentes escritos en un lenguaje de alto nivel a lenguaje máquina, comprobar que la sintaxis escrita es la correcta y de comprobar que las llamadas a las funciones de librería se realizan correctamente.
- **Programa (o código) objeto:** es el programa fuente traducido a código máquina. Aún no es directamente ejecutable.
- **Programa Ejecutable:** el archivo o programa ejecutable realizado por el enlazador, del programa fuente.
- **Linker (enlazador):** es el programa encargado de insertar al programa objeto el código máquina de las funciones de las librerías usadas en el programa y realizar el proceso de montaje. Este programa genera un archivo .exe.

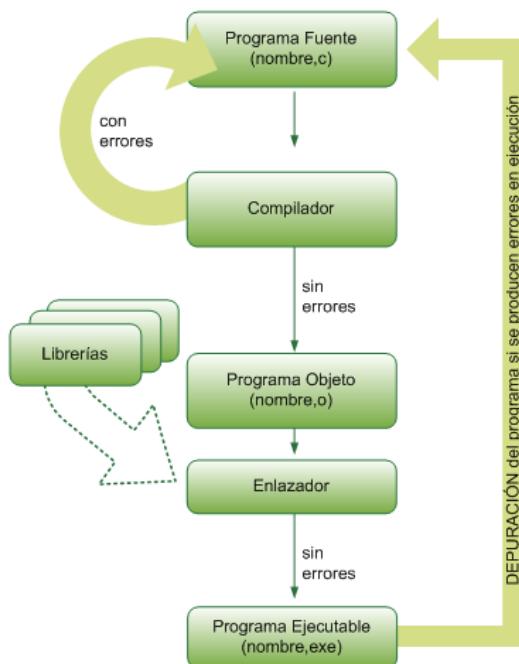


Figura 1.2. Proceso de transformación de un programa fuente a un programa ejecutable.

Vamos a explicar las fases de lenguaje C, desde que generamos o escribimos el código fuente hasta que tenemos el programa ejecutable.

Tal y como muestra la figura anterior primero se traduce el código o programa fuente, que los archivos son con extensión .c, a lenguaje máquina. El siguiente punto podemos tener dos vías si todo es correcto, es decir no existe ningún tipo de error, este programa compilador crea un programa objeto con el mismo nombre que el fuente pero con extensión .obj. Pero en el caso de que exista errores en el código fuente, nos informa de cuales son y debemos depurarlos, en el siguiente punto pasaremos a estudiar los diferentes tipos de errores.

En una segunda etapa se realiza el proceso de *montaje* que consiste en producir un **programa ejecutable .exe** en lenguaje de máquina, en el que están ya incorporados todas la librerías, funciones, recursos del S.O., etc., que aportan al sistema sin que intervenga el programador. Para un sistema operativo Windows el programa ejecutable se guarda con extensión .exe como ya hemos mencionado. Este fichero es cargado por el sistema en la memoria RAM cuando el programa va a ejecutarse.



Una de las ventajas más importantes de los lenguajes de alto nivel es la *portabilidad* de los ficheros resultantes. El lenguaje C, originalmente desarrollado por D. Ritchie en los laboratorios Bell de la AT&T, fue posteriormente estandarizado por un comité del ANSI (American National Standard Institute) con objeto de garantizar su portabilidad entre distintos ordenadores, dando lugar al **ANSI C**, que es la variante que actualmente se utiliza casi universalmente.

1.1.3. Tipos de errores al programar

Cada lenguaje de programación tiene sus propias reglas, es decir cada uno tiene su sintaxis, por ello debe existir un compilador específico para cada lenguaje de programación.

Como ya hemos definido en los pasos anteriormente, no se crea ningún archivo o programa objeto si el compilador ha encontrado errores en el código fuente.

Para proceder a su depuración, es decir a subsanar los errores encontrados, el compilador nos mostrará una lista de los mismos o de advertencia.

Estos errores pueden producirse en varias fases:

- Fase de compilación.
- Fase de enlace.
- Fase de ejecución de un programa.

Errores en Fase compilación

Los **errores en tiempo de compilación** son los que se producen antes de la ejecución del programa, durante el proceso de compilación del programa.

Los errores que se pueden producir en la **fase de compilación** son:

- **Errores fatales:** son extraños de ver y lo que indican es que ha existido un error interno de compilación, estos errores provocan una detención de la compilación inmediatamente, la solución es volver a compilar y si persiste el error, volver a instalar el compilador.
- **Errores de sintaxis:** son los más usuales, no detienen la compilación sino que al finalizar ésta se mostrará la lista con todos los errores encontrados. Algunos errores suelen ser consecuencia de otros cometidos con anterioridad. Por ejemplo, realizar la llamada a una variable sin haberla declarado previamente, con este tipo de errores no se puede obtener un programa objeto y por lo tanto tampoco el ejecutable.
- **Advertencias o avisos (warnings):** indican que hay error de código pero que no vulnera ninguna regla de sintaxis, pero advierte de que puede provocar algún error. Cuando se detecta un warning la compilación no se detiene, solo avisa y crea el programa objeto para luego ser ejecutado.

Errores en Fase de ejecución

Los **errores en tiempo de ejecución** son los que se producen durante la ejecución del programa. Son los más difíciles de encontrar, no son detectados por el compilador, ya que son errores de lógica, no de sintaxis.

Aunque después de compilar no nos muestre ningún error, el programa puede funcionar incorrectamente y dar errores durante su ejecución. Por ejemplo:

- Un programa puede producir resultados erróneos, que sume en vez de restar.
- Un programa puede crear un bucle infinito y quedarse bloqueado, porque hemos implementado mal el contador.
- Un programa puede interrumpirse bruscamente, por ejemplo si tenemos que hacer una división y el divisor es cero, etc.

1.1.4. ¿Qué es una función?

Las aplicaciones informáticas suelen contener muchísimas líneas de código fuente, a medida que los programas se van desarrollando estas aumentan de tamaño y por lo tanto se hacen poco manejables, por eso existe la modulación.



Definición: **Modulación** es el proceso de dividir un programa muy grande en una serie de módulo más pequeños y manejables. A estos módulos se les ha solido denominar de distintas formas: subprogramas, subrutinas, procedimientos, funciones, etc.

El lenguaje C hace uso del concepto de función.

Independientemente del nombre que se le dé el concepto para todos los lenguajes es el mismo: dividir un programa grande en un conjunto de subprogramas o funciones más pequeñas que son llamadas por el programa principal.

La división de un programa en partes más pequeñas o funciones presenta –entre otras- las ventajas siguientes:

- **Modularización.** Cada subprograma o función tiene una tarea exacta, de modo que no tendrá un número de líneas excesivo y será mucho más manejable. Además, una misma función puede ser llamada muchas veces en un mismo programa, e incluso puede ser reutilizada por otros programas. *Cada función puede ser desarrollada y comprobada por separado.* Esto también hace que se ahore memoria y por supuesto tiempo de desarrollo por parte del programador, ya que si está programado lo volvemos a reutilizar y evitamos también muchos errores.
- **Independencia de datos y ocultamiento de información.** Uno de los errores más frecuentes en los programas que al hacer cualquier modificación toquemos algo o afecte a otras líneas de código, y creemos errores colaterales. Una función es capaz de mantener una independencia con el resto del programa, manteniendo sus propios datos y definiendo muy claramente la interfaz o comunicación con la función que la ha llamado y con las funciones a las que llama, y no teniendo ninguna posibilidad de acceso a la información que no le atañe.



NOTA

Él en lenguaje C, las funciones de C es uno de los aspectos más potentes del lenguaje. Es muy importante entender bien su funcionamiento y sus posibilidades.

1.1.4.1. Partes de una función

Aunque este punto de desarrollará en próximas unidades didácticas, conviene realizar una asimilación del concepto.



DEFINICIÓN

Una función de C

Es una porción de código o programa que realiza una determinada tarea.

Una función está asociada con un nombre, que la identifica del resto de funciones y es la forma de referenciarla dentro del lenguaje.

En toda función utilizada en C hay que distinguir los siguientes tres conceptos:

Definición, declaración, llamada, valor retorno y argumentos

Las funciones en C se **llaman** incluyendo su nombre, seguido de los **argumentos**, en una sentencia del programa principal o de otra función.



DEFINICIÓN

Argumentos

Son datos o variables que se envían a la función incluyéndolos entre paréntesis a continuación del nombre, separados por comas.

Veamos todo con un ejemplo: supongamos una función llamada suma que calcula la suma de a+b. Una forma de llamar a esta función es escribir la siguiente sentencia suma (a,b);

- El nombre de la función es suma.
- Los argumentos son a e b, que en este caso constituyen los datos necesarios para calcular el resultado deseado, la suma de dos valores enteros.

En el ejemplo el resultado es el **valor de retorno** de la función, que está disponible pero no se utiliza , pero si se incluye una variable o una escritura, a esto se le llama **valor de retorno**. Otra forma de llamar a esta función utilizando el resultado podría ser la siguiente:

```
Resultado=suma(a,b);
Printf("la suma es: %d ",suma(a,b));
```

Para el primer caso, guarda en la variable resultado la suma de a+b, y en el segundo caso imprime en pantalla el resultado de la suma de los valores a, b.

Para poder **llamar** a una función es necesario en el mismo código fuente o en algún otro fichero fuente, este desarrollada esta función.



Función

Conjunto de sentencias necesarias para que la función pueda realizar su tarea cuando sea llamada, en la función cuando se desarrolla debe también definirse el valor de retorno y de cada uno de los argumentos.

Vamos a desarrollar el ejemplo anterior, la función suma:

```
int suma(int a, int b)
{
    int resultado;
    resultado = a+b ;
    return resultado;
}
```

La primera línea es importante ya que indica el tipo del valor de retorno, que en este caso será entero, no obstante los tipos de datos se verán más adelante.

Después, el nombre de la función, seguido de paréntesis, donde se definen los argumentos junto al tipo de los mismos, que en este caso también van a ser de tipo entero.

Se abren las llaves y se cierran, en medio contiene el código necesario para que la función funcione correctamente, la primera instrucción del código, es una declaración de la variable resultado, donde se va a guardar la suma de los dos números. y luego la sentencia aritmética de sumar y guarda el resultado.

Finalmente, con la sentencia return se devuelve resultado al programa o función que ha llamado a suma.



NOTA

Conviene anotar que las variables a y b han sido declaradas en la cabecera o definición de la función, por ese motivo no hace falta declararlas, sin embargo si debemos declarar resultado, que debe ser del mismo valor que el retorno.

Una función tiene que estar declarada antes de ser llamada, además de la llamada, la declaración de la función.

No obstante todo lo relacionado con las funciones se profundizara más adelante.

1.2. El lenguaje C

El lenguaje C está constituido por tres elementos: el compilador, el preprocesador y la librería estándar. A continuación se explica brevemente en qué consiste cada uno de estos elementos.

Compilador

El compilador es el elemento más característico del lenguaje C. Como ya se ha dicho anteriormente, su objetivo consiste en traducir a lenguaje de máquina el programa C contenido en uno o más ficheros fuente.



El compilador

Es el programa que traduce a lenguaje de máquina el programa escrito por el usuario.

Preprocesador

El preprocesador es un componente característico del lenguaje C que no existe en otros lenguajes de programación. El preprocesador actúa y realiza ciertas operaciones sobre el programa fuente antes de que empiece la compilación del código.

Librería estándar

Una librería es un conjunto de algoritmos prefabricados, que pueden ser utilizados por el programador para realizar determinadas operaciones, es decir son funciones ya realizadas y que podemos reutilizar.

Estos ficheros pueden llamarse “de cabecera”, porque se definen o se llaman en las primeras líneas del programa para poner las directivas `#include` que los incluirá en el fuente durante la fase de preprocesado

Ficheros

El código de los programas escritos se almacena en un fichero o varios si es necesario. Dependiendo de la extensión del programa escrito se almacenara en uno o más ficheros, en el disco del ordenador. Cuando los programas son pequeños, es decir menos de 150 líneas, en un fichero suele bastar.

Sin embargo cuando un programa es muy extenso suele utilizar varios archivos para mantener la independencia y ser más manejables.



Recordar que cada vez que se introduce una modificación en el código hay que volver a compilarlo. La compilación se realiza a nivel del fichero o código, por lo que sólo los ficheros donde está modificado el código hay que volver a compilar de nuevo.

Lectura y escritura de datos

La lectura y escritura de datos se realiza por medio de llamadas a funciones de una librería que tiene el nombre de *stdio.h* que significa Estándar input output.

Las declaraciones de las funciones de esta librería están en un fichero llamado *stdio.h* y habrá que realizar la llamada desde el fichero principal con la siguiente directiva en la primera linea de programa:

```
#include stdio.h
```

Se utilizan funciones diferentes para leer datos desde teclado (*scanf*), y lo mismo para escribir resultados o texto en la pantalla (*Printf*).



Las funciones de entrada y salida de datos son funciones, con todas sus componentes: nombre, valor de retorno y argumentos.

Interfaz con el sistema operativo

Hace años el compilador utilizado era bajo sistema operativo *MS-DOS*, y no desde *Windows*. Ahora los entornos de trabajo basados en *Windows* se han estandarizado.

En cualquier caso, el formato de llamada a un compilador varía de un entorno a otro por lo que es conveniente, estudiar o tener los manuales de los compiladores que se utilicen.

Existen también muchos entornos de programación más manejables que están bajo *Windows*, como por ejemplo el *DevC++*, o el *C++* de *Borland*, por ejemplo disponen de editores propios con ayuda a la hora de escribir código, como introducción de colores para distinguir de palabras claves.

Para este curso de programación en C, el alumno puede elegir el entorno con el que esté más familiarizado, pero en los apuntes y ejercicios realizados están desarrollados en *DEV-C++*:

- **Dev-C++** es uno de estos programas, desarrollado por Bloodshed Software, un entorno cómodo para la realización de nuestros proyectos tanto en lenguaje C como en C++. Permite la incorporación de librerías que añaden más funciones del programa, y utiliza un compilador basado en gcc. El programa es muy sencillo y fácil de manejar para los principiantes en la programación.



Otros entornos de programación del mercado:

- **Code: Blocks**: es un programa similar a Dev-C++, un poco más complicado cuanto a la configuración, pero admite añadir numerosos compiladores como Digital Mars, Microsoft Visual C++, Borland C++ o Watcom, también permite incorporar herramientas para crear gráficos, que el devC-C++ no permite.
- **Microsoft Visual C++:** está orientado al desarrollo de aplicaciones para Windows, ofrece herramientas para programación utilizando librerías de .Net Framework. Es de pago y tiene muchos más lenguajes, también existe una versión gratuita llamada Express.



Para este módulo se ha utilizado el Dev-C++, y en la plataforma docente puedes encontrar el instalable además de un manual del compilador y manejo del entorno.

1.3. Estructura de un programa en C

Como ya hemos comentado anteriormente, todo programa en **C** consta de una o más funciones, una de las cuales se llama **main (principal)**. El programa comienza en la función main, desde la cual es posible llamar a otras funciones. Veamos un ejemplo del esquema general.



EJEMPLO

```
declaraciones variables globales
main( ) {
    declaracion variables locales
    instrucciones o sentencias
}
funcion1( ) {
    variables locales
    intrucciones de la funcion
}
```

Vamos a realizar nuestro primer programa,

Importante; hay que leer pero no memorizar, y según vayamos avanzando en la materia iremos comprendiendo más cosas que al principio nos parece incomprensibles.

Lo primero que veremos será el código que aparece a continuación y posteriormente aparece una explicación de cada línea, no intentes realizarlo ya en el ordenador, es importante que obtengas la base entendiendo que es cada línea.



EJEMPLO

```
Veamos un programa muy simple en C.
#include <stdio.h>
int main(void)
{
    printf("primer programa en c \n");
    return(0);
}
```

Encabezamiento

La primera línea del programa está compuesta por una directiva:

#include incluye el código del archivo que aparece posteriormente, en este caso **<stdio.h>**. En este archivo están incluidas declaraciones de las funciones luego llamadas por el programa principal como la entrada y salida de datos.



A

AVISO

De momento no hace falta más indicar sobre las directivas, en posteriores unidades se detallaran cuales vamos a utilizar y profundizar más en cada una de ellas.

Hay dos formas de llamar los archivos en las directivas:

- Si el nombre del archivo está delimitado entre comillas dobles (" ") : el compilador lo buscará en el directorio donde hemos guardado el archivo fuente del programa principal,
- Si el nombre del archivo está delimitado con los signos de mayor y menor, (<>) lo buscara en algún otro directorio.

No obstante, podemos desde el entorno, configurar el camino o directorio donde van a vincularse las librerías (EJ: C:\.\include..) Por lo general estos archivos son guardados en un directorio llamado INCLUDE y el nombre de los mismos está terminado con la extensión .h.

N

NOTA

La directiva "#include" no es una sentencia de programa sino una orden que lo que hace es copiar literalmente el archivo llamado de texto en el lugar donde la hemos llamado, por este motivo no es necesario incluir el punto y coma (;) al finalizar la instrucción.

Función main ()

Todo programa realizado en lenguaje C, tiene un programa principal que es con el que se comienza la ejecución del programa, este programa principal es también una función, es decir **función principal** por lo que prevalece por encima del resto de funciones.

La función main() indica donde empieza el programa, cuyo cuerpo principal es un conjunto de sentencias, que inician con una apertura de llaves y finaliza con las llaves de cierre " { ... } ". Todos los programas C arrancan del mismo punto después de la apertura de llaves.

Como podemos observar en el ejemplo citado anteriormente, el programa principal está compuesto por las dos instrucciones, la primera escribir en pantalla con el Printf la frase: "primer programa en C", la segunda, return, que finaliza el programa devolviendo el control al Sistema Operativo.



Que el lenguaje C no tiene instrucciones de entrada-salida por lo que para escribir en pantalla es necesario llamar a una función que está incluida en el archivo stdio.h.

- **La primera instrucción** `Printf("primer programa en c \n")`; escribe en pantalla el mensaje delimitado por las dobles comillas y la sentencia `\n`, realiza un salto de línea, más adelante analizaremos más exhaustivamente la instrucción y sus modificadores.
- **La segunda sentencia (return 0)** termina el programa y devuelve un valor al sistema operativo, por lo general cero si la ejecución fue correcta.

Cada sentencia debe acabar con `";` ya que es la forma de decirle al compilador que es el final de la instrucción. Puede haber sentencias que ocupen más de dos reglones, por ello hay que avisar al compilador donde termina.



Por ejemplo podría ser:

```
printf("Bienvenido a la Programación"  
"en lenguaje C \n");
```

El uso de mayúscula ó minúscula, utilizado para escribir el código es muy importante, puesto que **C sí hace distinción entre mayúsculas y minúsculas**, lo cual quiere decir que no es lo mismo escribir Variable, VARIABLE, VARIABLE.

1.3.1. Componentes sintácticos

Existen seis clases de *componentes sintácticos* o *tokens* en el vocabulario del lenguaje C:

- Palabras clave.
- Identificadores.
- Constantes.
- Cadenas de caracteres.
- Operadores.
- Separadores.

Palabras clave del C

Existen una lista de palabras clave, llamadas también keywords, que el usuario no puede utilizar como identificadores (nombres de variables y/o de funciones).

Estas palabras están reservadas ya que son instrucciones del propio compilador o del lenguaje, y realizar una tarea exacta.

A continuación se presenta la lista de las 30 palabras clave, para que se tengan en cuenta:

auto	break	case	Char
continue	default	do	double
Else	Enum	Extern	Flota
for	goto	if	int
long	register	return	Short
sizeof	static	struct	switch
ttypedef	union	unsigned	Void
while			



Algunos compiladores añaden otras palabras clave, propias de cada uno de ellos. Es importante evitarlas como identificadores.

Identificadores

Ya se ha explicado lo que es un *identificador* un nombre con el que se hace referencia a una función o al contenido de una zona de la memoria, es decir una variable. Cada lenguaje tiene sus propias reglas respecto a la elección de los nombre o identificadores.

Para C las reglas son las siguientes:

- Un identificador se forma con una secuencia de letras donde incluye minúsculas de la a a la z; mayúsculas de la A a la Z; y dígitos del 0 al 9.
- El carácter *subrayado* () se considera como una letra más.
- Un identificador no puede contener espacios en blanco,
- Un identificador no puede contener carácter distintos a los ya citados por ejemplo (-+* ; . : etc.).
- El primer carácter de un identificador debe ser siempre una letra o un (), es decir, no puede ser un dígito.

- Se hace distinción entre letras mayúsculas y minúsculas por lo que Suma es un identificador y SUMA se considera otro.

En general es muy aconsejable elegir los nombres de las funciones y las variables de forma que permitan conocer a simple vista qué representan.

EJEMPLO

Hallar la media de notas, podríamos identificar a la variable de la siguiente manera. `media_notas`

Constantes

Las variables pueden cambiar de valor a lo largo de la ejecución del programa, además de variables, un programa utiliza también constantes, es decir, valores que siempre son los mismos. El ejemplo más usual es el número π , que vale 3.1416. En C existen distintos tipos de constantes:

- **Constantes numéricas.** Son valores numéricos, enteros o de punto flotante.
- **Constantes carácter.** Cualquier carácter individual encerrado entre comilla simple ejemplo: ‘p’
- Existe un código, llamado **código ASCII**, que establece una equivalencia entre cada carácter y un valor numérico correspondiente.
- **Cadenas de caracteres.** Un conjunto de caracteres alfanuméricos encerrados entre comillas dobles por ejemplo: “esto es una cadena”.

Operadores

Los **operadores** son signos que indican determinadas operaciones a realizar con las variables o constantes sobre las que actúan en el programa. El lenguaje C podemos encontrar los siguientes tipos de operadores:

- *aritméticos* (+, -, *, /, %),
- *de asignación* (=, +=, -=, *=, /=),
- *relacionales* (==, <, >, <=, >=, !=),
- *lógicos* (&&, ||, !) y otros.



Por ejemplo, en la sentencia:

`Velocidad = Espacio/tiempo`

Aparece un operador de asignación (=) y el operador aritmético (/).

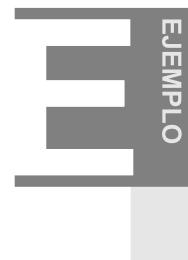
Separadores

Los separadores, es decir los espacios en blanco, tabuladores, etc... y los comentarios escritos por el programador. Su función es facilitar al compilador a descomponer el programa fuente en cada uno de sus tokens. Es conveniente introducir espacios en blanco o tabuladores para mejorar la legibilidad de los programas.

Comentarios

El lenguaje C permite que al programador introducir comentarios, para dar una explicaciones o aclaración sobre cómo se está realizando el programa. Esto sirve de ayuda para el que coja el código que no ha realizado el mismo.

Los caracteres /* se emplean para iniciar un comentario y finaliza con los caracteres */ . No se puede introducir un comentario dentro de otro. Todo texto introducido entre los símbolos de comienzo /* y final */ de comentario son siempre ignorados por el compilador.



Por ejemplo:

`Variable_1 = variable_2;`

`/* En esta línea se asigna a variable_1 el valor
contenido en variable_2 */`



Una fuente frecuente de errores, es el olvidarse de cerrar un comentario que se ha abierto previamente.

También se puede realizar un comentario en una sola línea, pero debe iniciarse con la doble barra //, y no hace falta cerrar, y el compilador considera que es un comentario hasta el final de línea ya que esta actúa como cierre. Esto es utilizado para comentarios corto, si se desea añadir otro comentario deberás inicializar otra vez con la doble barra. Véase el ejemplo:

EJEMPLO

Por ejemplo:

```
variable_1 = variable_2; // En esta línea se asigna a  
// variable_1 el valor  
// contenido en variable_2
```

Sangrado del código fuente

En el editor y en los programas escritos se utilizará en sangrado o también llamado identación. Más adelante se verá como se identan las instrucciones. Recordar que la identación no afecta para nada al programa, sirve para que este sea más legible.

1.4. Definición de variables

En los lenguajes de programación tenemos la posibilidad de trabajar con datos de distinta forma: texto, números enteros, números reales, etc. Además, algunos de estos tipos de datos admiten distintos números de cifras, rango y precisión, también pueden ser positivos o negativos etc.

Si queremos imprimir, en pantalla, los resultados de multiplicar un número fijo por otro que adopta valores entre 0 y 9, es decir la tabla de multiplicar, el modo más sencillo es crear una constante y un par de variables para el segundo (0.9) y para el resultado del producto.



Una variable, es una posición de memoria donde el programa guarda los distintos valores que va adoptando las variables en el transcurso del programa.

En un programa debemos definir a todas las variables que utilizarán, antes de usarlas, y también debemos indicar de qué tipo son y como se van a llamar. De esta manera el compilador reservara el espacio suficiente y necesario para guardar estos datos.

Veamos el siguiente ejemplo.

```
#include <stdio.h>
int main(void)
{
    int multiplicador; /* multiplicador como un entero
*/
    int multiplicando; /* multiplicando como un entero
*/
    int resultado; /* defino resultado como un entero */

    multiplicador = 10; /* les asigno valores */

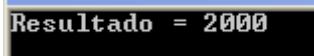
    multiplicando = 2 ;
    resultado = multiplicando * multiplicador ;

    printf(«Resultado = %d\n», resultado);

    /* muestro el resultado */
    return 0;
}
```

Después de las líneas de la declaración de las variables como asigna los valores a estas de 10 y 2 y luego realiza el cálculo de la variable resultado, donde guarda la multiplicación.

La función **printf()**, nos muestra el valor de una variable, insertada en el texto a mostrar, aparece una secuencia de control de impresión “%d” que indica, que en el lugar que ella ocupa, deberá ponerse el contenido de la variable, expresado como un número entero decimal. Así, si compilamos y corremos el programa, obtendremos:



Resultado = 2000

Figura 1.3. Salida de pantalla.

1.4.1. Inicialización de variables

Las variables del mismo tipo pueden definirse en una misma línea, de forma múltiple, separando cada identificador con una coma. veamos el ejemplo de antes pero en una misma línea en vez de tres.

```
int multiplicador, multiplicando, resultado;
```

También las variables pueden ser inicializadas en el momento de definirse:

```
int multiplicador = 1000, multiplicando = 2, resultado;
```

De esta manera el anterior ejemplo podría escribirse:

```
#include <stdio.h>
int main(void)
{
    int multiplicador=1000 , multiplicando=2 ;
    printf("Resultado = %d\n", multiplicando * multiplicador);
    return 0;
}
```



Hemos quitado la variable “resultado” ya que es innecesaria, ya que si vemos la función **printf()** vemos que se ha reemplazado “resultado” por la operación aritmética directamente. Esto es una de las ventajas de C, que podemos pasar operaciones como parámetros dentro de las funciones.

1.4.2. Tipos de variables

El tipo de un dato determina el rango de valores que puede tomar el dato y su ocupación en memoria durante la ejecución del programa, existen dos grupos de datos para C, los tipos fundamentales y tipos derivados.

	char	signed char	unsigned char
Datos enteros	signed short int	signed int	signed long int
	unsigned int	unsigned int	unsigned long int
Datos reales	float	double	long double

Figura 1.4. Tipos de datos fundamentales (notación completa).

La palabra `char` hace referencia a que se trata de un carácter.

La palabra `int` indica que se trata de un número entero,

La palabra `float` se refiere a un número real (también llamado de punto o coma flotante). Los números enteros pueden ser positivos o negativos (`signed`), o bien esencialmente no negativos (`unsigned`); los caracteres tienen un tratamiento muy similar a los enteros y admiten estos mismos cualificadores. En los datos enteros, las palabras `short` y `long` hacen referencia al número de cifras o rango de dichos números. En los datos reales las palabras `double` y `long` apuntan en esta misma dirección, aunque con un significado ligeramente diferente, como más adelante se verá.

	char	signed char	unsigned char
Datos enteros	short	int	Long
	unsigned short	unsigned	unsigned long
Datos reales	float	double	long double

Figura 1.5. Notación abreviada.

Esta nomenclatura puede simplificarse: las palabras `signed` e `int` son las opciones por defecto para los números enteros y pueden omitirse, resultando la Tabla siguiente, que indica la nomenclatura más habitual para los tipos fundamentales del C.



Recuerda que en C es necesario declarar todas las variables que se vayan a utilizar.

Una variable no declarada produce un mensaje de error en la compilación. Cuando una variable es declarada se le reserva memoria de acuerdo con el tipo incluido en la declaración. Es posible inicializar –dar un valor inicial– las variables en el momento de la declaración; ya se verá que en ciertas ocasiones el compilador da un valor inicial por defecto, mientras que en otros casos no se realiza esta inicialización y la memoria asociada con la variable correspondiente contiene basura informática (combinaciones sin sentido de unos y ceros, resultado de operaciones anteriores con esa zona de la memoria, para otros fines).

Únicamente vamos a ver los tipos de datos fundamentales que se describen a continuación uno a uno.

1.4.2.1. Variables del tipo entero

De acuerdo a la cantidad de bytes que reserve el compilador para este tipo de variable, queda determinado el “alcance” ó máximo valor que puede adoptar la misma. Debido a que el tipo int ocupa 2 bytes su alcance queda restringido al rango entre -32.768 y +32.767 (incluyendo 0).

En caso de necesitar un rango más amplio, puede definirse la variable como “*long int nombre_de_variable*” ó en forma abreviada “*long nombre_de_variable*”. Declarada de esta manera, *nombre_de_variable* puede alcanzar valores entre: -2.347.483.648 y +2.347.483.647.

A la inversa, si se quisiera un alcance menor al de int, podría definirse “*short int*” ó simplemente “*short*”, aunque por lo general, los compiladores modernos asignan a este tipo el mismo alcance que “int”.



Debido a que la norma ANSI C no establece de forma tajante la cantidad de bytes que ocupa cada tipo de variable, sino tan sólo que un “*long*” no ocupe menos memoria que un “*int*” y este no ocupe menos que un “*short*”, los alcances de los mismos pueden variar de compilador en compilador.

Todos los tipos citados hasta ahora pueden alojar valores positivos ó negativos y, aunque es redundante, esto puede explicitarse agregando el calificador “*signed*” delante; por ejemplo:

```
signed int  
signed long  
signed long int  
signed short
```

```
signed short int
signed char
```

Si en cambio, tenemos una variable que sólo puede adoptar valores positivos (como por ejemplo la edad de una persona) podemos aumentar el alcance de cualquiera de los tipos, restringiéndolos a que sólo representen valores sin signo por medio del calificador “`unsigned`”. En la tabla siguiente se resume los alcances de distintos tipos de variables enteras.

Tipo	Bytes	Valor mínimo	Valor máximo
<code>signed char</code>	1	-128	127
<code>unsigned char</code>	1	0	255
<code>unsigned short</code>	2	-32.768	+32.767
<code>unsigned short</code>	2	0	+65.535
<code>signed int</code>	2	-32.768	+32.767
<code>unsigned int</code>	2	0	+65.535
<code>signed long</code>	4	-2.147.483.648	+2.147.483.647
<code>unsigned long</code>	4	0	+4.294.967.295

Figura 1.6. Variables del tipo número entero.



Si se omite el calificador delante del tipo de la variable entera, éste se adopta por omisión (default) como “`signed`”.

1.4.2.2. Variables de número real o punto flotante

Un número real ó de punto flotante es aquel que además de una parte entera, posee fracciones de la unidad. En nuestra convención numérica solemos escribirlos de la siguiente manera: 2,3456. Los compiladores usan la convención del punto decimal (en vez de la coma) .

Así el número pi se escribirá: 3.14159 Otro formato de escritura, normalmente aceptado, es la notación científica. Por ejemplo podrá escribirse 2.345E+02, equivalente a 2.345 * 100 ó 234.5

De acuerdo a su alcance hay tres tipos de variables de punto flotante, las mismas están descriptas en la siguiente tabla.

Tipo	Bytes	Valor mínimo	Valor máximo
float	4	3.4E-38	3.4E+38
double	8	1.7E-308	1.7E+308
long double	10	3.4E-4932	3.4E+4932

Figura 1.7. Tipos de variables de punto flotante.

Las variables de punto flotante siempre son con signo, y en el caso que el exponente sea positivo puede obviarse el signo del mismo.

1.4.2.3. Variables de tipo carácter

Las variables carácter (**char**) contienen un único carácter y se almacenan en un **byte** de memoria (8 bits). En un bit se pueden almacenar dos valores (0 y 1); con dos bits se pueden almacenar $2^2 = 4$ valores (00, 01, 10, 11 en binario; 0, 1, 2, 3 en decimal). Con 8 bits se podrán almacenar $2^8 = 256$ valores diferentes (normalmente entre 0 y 255; con ciertos compiladores entre -128 y 127).

La declaración de variables tipo carácter puede tener la forma:

```
char nombre;
char nombre1, nombre2, nombre3;
```

Se puede declarar más de una variable de un tipo determinado en una sola sentencia. Se puede también inicializar la variable en la declaración. Por ejemplo, para definir la variable carácter **letra** y asignarle el valor **a**, se puede escribir:

```
char letra = 'a';
```

A partir de ese momento queda definida la variable **letra** con el valor correspondiente a la letra **a**. Recuérdese que el valor '**a**' utilizado para inicializar la variable **letra** es una constante carácter. En realidad, **letra** se guarda en un solo byte como un número entero, el correspondiente a la letra **a** en el código ASCII, para los caracteres estándar (existe un código ASCII extendido que utiliza los 256 valores y que contiene caracteres especiales y caracteres específicos de los alfabetos de diversos países, como por ejemplo las vocales acentuadas y la letra **ñ** para el castellano).

1.5. Constantes

Se entiende por constantes aquel tipo de información que no puede cambiar más que con una nueva compilación del programa.

Constantes enteras

Un constante entero decimal está formado por una secuencia de dígitos del 0 al 9, constituyendo un número entero.



Los constantes enteros decimales solo pueden ser variables tipo *int* y *long*, pudiendo también ser *unsigned*.

Constantes de punto flotante

Constantes de punto flotante, pueden ser de tipo *float*, *double* y *long double*.

Una constante de punto flotante se almacena de la misma forma que la variable correspondiente del mismo tipo. Notaciones de punto flotante a tener en cuenta:

- Por defecto las constantes de punto flotante son de tipo **double**.
- Para indicar que una constante es de tipo **float** se le añade una f.
- Para indicar que es de tipo **long double**, se le añade una l o una L.
- El punto decimal siempre debe ponerse si expresamos un número real.



Ejemplos correctos e incorrectos:

- 1.2 constante tipo double (opción por defecto) ✓
- 2.963f constante tipo float ✓
- .0874 constante tipo double ✓
- 22e2 constante tipo double (igual a 2300.0) ✓
- .84e-2 tipo double en notación científica (=0.00874) ✓
- 1,23 error: la coma no está permitida ✗
- 23963f error: no hay punto decimal ni carácter e ó E ✗
- .e4 error: no hay ni parte entera ni fraccionaria ✗
- -3.14 error: sólo el exponente puede llevar ✗

Constantes carácter

Una constante de tipo carácter: es un letra o carácter de cualquier tipo cualquiera encerrado entre comilla simple o apostrofo(' '). El valor de una constante carácter es el valor numérico asignado a ese carácter según el código ASCII ya que en lenguaje C no existen constantes tipo *char*; ya que son constantes entera.

Una secuencia de escape está constituida por la barra invertida (\) seguida de otro carácter.

D
DEFINICIÓN

Secuencias de escape

Las secuencias de caracteres en las que el primero es la barra invertida, se denominaron secuencias de escape y aunque originariamente se utilizaron para la representación de los caracteres de control, por extensión pueden representarse de este modo todos los códigos ASCII. Además se dispone de algunos símbolos predefinidos para los caracteres más frecuentes. Por ejemplo, \n se utiliza para representar el carácter nueva línea (decimal 10).

La finalidad de la secuencia de escape es cambiar el significado habitual del carácter que sigue a la barra invertida. Los símbolos utilizados se muestran en la tabla adjunta.

Nombre completo	Constante	En C	ASCII
Sonido de alerta	BEL	\a	7
Nueva línea	NL	\n	20
Tabulador horizontal	HT	\t	9
Retroceso	BS	\b	8
Retorno de carro	CR	\r	13
Salto de página	FF	\f	12
Barra invertida	\	\\	92
Apóstrofo	'	\'	39
Comillas	"	\"	34
Carácter nulo	NULL	\0	0

Figura 1.8. Secuencia de escape.

Cadenas de caracteres

Una cadena de caracteres es una secuencia de caracteres delimitada por comillas (""), como por ejemplo: “**Esto es una cadena de caracteres**”. Dentro de la cadena, puede contener caracteres en blanco y se pueden emplear las mismas secuencias de escape válidas para las constantes carácter.



Ejemplos de cadenas de caracteres:

- “Informática I”
- “A”
- “cadena con espacios en blanco “
- “Esto es una \”cadena de caracteres\”.\\n”

Constantes de tipo enumeración

En C existen una clase diferentes a las anteriores de constantes, que son las constantes *enumeración*.

Se caracterizan por poder aceptar valores entre una selección de constantes enteras denominadas enumeradores, cuyos valores son establecidos en el momento de la declaración del tipo.

Por ejemplo, se puede pensar en una variable llamada **dia_de_la_semana** que toma 7 valores: **lunes, martes, miércoles, jueves, viernes, sábado y domingo**.

```
enum dia {lunes, martes, miercoles, jueves, viernes, sa-
bado, domingo};
```



Este tipo de constantes hace que el código sea más legible y claro, también disminuye la posibilidad de cometer errores.

Podemos asociar a cada elemento un valor entero, de esta manera puede estar más controlado por el programador.

```
enum dia {lunes=1, martes, miercoles, jueves, viernes, sa-
bado, domingo};
```

Asocia un valor 1 a **lunes**, 2 a **martes**, 3 a **miércoles**, etc.,

Ejemplo

Declarar en un programa las siguientes constantes y variables.

- PI que toma el valor 3.1416,
- limitesup que toma el valor 10.
- limiteinf que toma el valor 0.

Además declarar las variables número, mínimo, máximo de tipo entero, y media de tipo real.

```
#include <stdio.h>
int main(void)
{
    const pi = 3.14159;
    const max = 100;
    const min = 0;
    int numero, maximo, minimo;
    float media;
    return(0);
}
```



NOTA

- Cada una de las constantes va separada de la anterior por punto y coma, puesto que se están definiendo constante que, lógicamente toman distinto valor.
- Las variables que son del mismo tipo pueden declararse juntas separadas por una coma. Si las variables son de distinto tipo no se pueden declarar juntas, deben ir separadas por el signo de punto y coma.



1.6. Operadores, expresiones y sentencias

Si un lenguaje solo permitiese definir valores y dar valores a las variables sería un lenguaje pobre. Los programadores no podrían calcular valores nuevos a partir de combinaciones de valores anteriores. Dado que los tipos primitivos de datos son el punto de comienzo para construir todas las abstracciones, las operaciones primitivas sobre estos tipos de datos se necesitan para comenzar el proceso de construcción de expresiones.

Las expresiones en cualquier lenguaje son parecidas a las matemáticas, ya que son combinaciones de valores, operadores y expresiones, que dan un resultado.

Vamos a describir cada uno de las partes que componen la expresión.

1.6.1. Operadores

Un operador es un carácter que actúa sobre las variables para realizar una determinada operación para obtener un determinado **resultado**. Ejemplos más conocidos son los operadores aritméticos de suma (+), resta (-), producto o multiplicación (*), etc.

Los operadores pueden ser **unarios**, **binarios** y **ternarios**, según actúen sobre uno, dos o tres operandos, respectivamente.

En C existen muchos operadores de diversos tipos, que se verán a continuación.

Operadores aritméticos

Los **operadores aritméticos** son los más sencillos de utilizar. Todos ellos son operadores binarios. En C se utilizan los cinco operadores siguientes:

- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- Resto, modulo: %

Todos estos operadores se pueden aplicar a constantes, variables y expresiones. El resultado es el que se obtiene de aplicar la operación correspondiente entre los dos operandos.



El único operador que requiere una explicación adicional es el operador resto %. En realidad su nombre completo es resto de la división entera. Este operador se aplica solamente a constantes, variables o expresiones de tipo int.

Ejemplo: $25\%2$ es 1, puesto que el resto de dividir 25 por 2 es 1.

Como veremos en próximo punto de esta unidad, una expresión es un conjunto de variables y constantes también de otras expresiones más sencillas, que están relacionadas con los operadores. Es aconsejable utilizar **paréntesis (...)** para agrupar a algunos términos y que sea más legible la expresión. La introducción de los de los paréntesis es común a las expresiones matemáticas.

Operadores de asignación

Los operadores de asignación depositan en la zona de memoria correspondiente a dicha variable, el resultado de una expresión o el valor de otra variable. El operador de asignación más usado es el operador de igualdad (=), su sintaxis es la siguiente:

```
nombre_de_variable = expresion;
```

Primero se evalúa expresión, es decir si es suma de dos valores realiza la suma y el resultado se asigna en nombre_de_variable, sustituyendo cualquier otro valor que hubiese en esa posición de memoria anteriormente. Veamos un ejemplo:

```
variable = variable + 1;
```

En lenguaje C, al igual que otros lenguajes de programación el operador de asignación (=), realizar una sustitución, en el ejemplo anterior, toma el valor de la variable que está contenido en memoria, le suma 1, y posteriormente introduce este valor en la zona de memoria correspondiente al identificador de la variable, es decir el resultado ha sido incrementar el valor de variable en una unidad.



A la izquierda del operador de asignación (=) no puede haber nunca una expresión: tiene que ser necesariamente el nombre de una variable. ejemplo de asignación incorrecto. $a + b = c;$

Existen otros cuatro operadores de asignación, que simplifican algunas operaciones, que son recurrentes a la misma variable:

- $+=$, $-=$, $*=$ y $/=$ formados por los 4 operadores aritméticos seguidos por el carácter de igualdad.



A continuación se presentan algunos ejemplos con estos operadores de asignación:

- distancia $+= 1$; equivale a: distancia = distancia + 1;
- rango $/= 2.0$; equivale a: rango = rango /2.0;
- $x^*= 3.0 * y - 1.0$ equivale a: $x = x^* (3.0 * y - 1.0)$

Operadores incrementales (++) y (--)

Los operadores incrementales son unarios, que incrementan o restan la unidad el valor de la variable. Si preceden a la variable, ésta es incrementada antes de que el valor de dicha variable sea utilizado en la expresión en la que aparece. Si es la variable la que precede al operador, la variable es incrementada después de ser utilizada en la expresión. Vamos a ver los ejemplos de los siguientes operadores:

```
i = 2;
j = 2;
m = i++; //después de ejecutarse esta sentencia m=2 e i=3
n = ++j; //después de ejecutarse esta sentencia n=3 y j=3
```

Estos operadores son muy utilizados. Es importante entender muy bien por qué los resultados m y n del ejemplo anterior son diferentes.

Operadores relacionales

Una característica imprescindible de cualquier lenguaje de programación es la de considerar opciones y alternativas, esto es, el código deberá ir o proceder según las necesidades del programador.

Los operadores relacionales permiten aplicar si se cumplen o no las condiciones. De esta manera, los operadores dan un resultado u otro según se cumplan o no algunas condiciones.



Los operadores lógicos, también llamados booleanos, son no, y y o. El funcionamiento de estos operadores es equivalente al funcionamiento de las puertas lógicas not, and y or de electrónica, y la correspondencia es no-not, y-and, o-or.

Los operadores relacionales permiten hacer comparaciones entre los valores de tipo de datos simples. En los algoritmos sirven para expresar condiciones.

Los operadores o e y actúan sobre dos operando u expresiones lógicas, mientras que el operador de negación no actúa sobre un único operando u expresión lógica.

Cuando se aplica el operador lógico `y`, el resultado produce el valor lógico verdadero si y sólo si los dos operandos o expresiones tienen el valor lógico verdadero.

En C un 0(cero) representa la condición de *false*, y cualquier número distinto de 0(cero) equivale la condición *true*. Cuando el resultado de una expresión es *true* y hay que asignar un valor concreto distinto de cero, por defecto se toma un valor unidad.

Los *operadores relacionales* de C son los siguientes:

- Igual que: `==`
- Menor que: `<`
- Mayor que: `>`
- Menor o igual que: `<=`
- Mayor o igual que: `>=`
- Distinto que: `!=`

El funcionamiento es sencillo, se evalúan `expresion1` y `expresion2`, y se comparan los valores resultantes, si la condición se cumple el resultado es 1 y por el contrario si no cumple el resultado es 0. Analizar los siguientes ejemplos.

EJEMPLO

- `(2==1)//resultado=0` porque la condición no se cumple.
- `(3<=3)//resultado=1` porque la condición se cumple.
- `(3<3)//resultado=0` porque la condición no se cumple.
- `(1!=1)//resultado=0` porque la condición no se cumple.

Operadores lógicos

Los operadores lógicos son operadores binarios que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen simultáneamente varias condiciones, que se cumple una u otra, etc.

Los operadores lógicos, también llamados booleanos, el operador Y (`&&`) y el operador O (`||`). El funcionamiento de estos operadores es equivalente al funcionamiento de las puertas lógicas and y or de electrónica.

ATENCIÓN

El lenguaje C tiene dos operadores lógicos: el operador Y (`&&`) y el operador O (`||`).

El operador `&&` devuelve un 1 si tanto `expresion1` como `expresion2` son verdaderas, y 0 en caso contrario, es decir si una de las dos expresiones o las dos son falsas (iguales a 0); por otra parte, el operador `||` devuelve 1 si al menos una de las expresiones es cierta.

Cuando se aplica el operador lógico `y`, el resultado produce el valor lógico verdadero si y sólo si los dos operandos o expresiones tienen el valor lógico verdadero.

A	B	A && B
1	1	1
1	0	0
0	1	0
0	0	0

Figura 1.9. Resultado del operador lógico Y.

Cuando se aplica el operador lógico `o`, el resultado produce el valor lógico verdadero si y sólo si al menos uno de los dos operandos o expresiones tienen el valor lógico verdadero.

A	B	A B
1	1	1
1	0	1
0	1	1
0	0	0

Figura 1.10. Resultado del operador lógico O.



Es importante tener en cuenta que los compiladores de C tratan de optimizar la ejecución de estas expresiones, lo cual puede tener a veces efectos no deseados. Por ejemplo:

Para que el resultado del operador `&&` sea verdadero, ambas expresiones tienen que ser verdaderas; si se evalúa `expresion1` y es falsa, ya no hace falta evaluar `expresion2`, y de hecho no se evalúa. Algo parecido pasa con el operador `||`: si `expresion1` es verdadera, ya no hace falta evaluar `expresion2`.

Los operadores `&&` y `||` se pueden combinar entre sí –quizás agrupados entre paréntesis–, dando a veces un código de más difícil interpretación. Por ejemplo:

```
(2==1) || (-1==-1) // el resultado es 1
```

```
(2==2) && (3==-1) // el resultado es 0
```

```
((2==2) && (3==3)) || (4==0) // el resultado es 1
```

```
(( 6==6 ) || ( 8==0 ) ) && ( ( 5==5 ) && ( 3==2 ) ) // el resultado es 0
```

1.6.2. Expresiones

Una expresión es una combinación de variables, constantes, y operadores. La expresión es equivalente al resultado que proporciona al aplicar sus operadores a sus operandos.

EJEMPLO

Por ejemplo, $2+1$ es una expresión formada por dos *operando* (2,1) y un *operador* (+); esta expresión es equivalente al valor 3.

En el lenguaje C existen distintos tipos de expresiones.

Expresiones aritméticas

Están formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales (+, -, *, /, %, ++, --), también se pueden emplear paréntesis de tantos niveles como se desee, y su interpretación sigue las normas aritméticas normales. Por ejemplo, en la ecuación de segundo grado:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

En C se escribe de la siguiente forma:

```
x=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
```

Donde lo que está en negrita o lo que es lo mismo todo lo que está a la derecha del operador de asignación (=) es una expresión aritmética.

En la expresión anterior tiene la llamada a la *función sqrt()*, que tiene como *valor de retorno* la raíz cuadrada de su único *argumento*.

DEFINICIÓN

Todo el conjunto: expresión aritmética, la variable, el signo de asignación y el carácter (;), forma lo que se llama **sentencia**.

Expresiones LÓGICAS

Como su palabra indica son expresiones que contienen valores lógicos: *verdaderos* (distintos a 0) y *falsos* (iguales a 0), y los *operadores lógicos* `|| (or)`, `&& (and)` y `! (negación)`. También se pueden emplear los *operadores relacionales* (`<`, `>`, `<=`, `>=`, `==`, `!=`) para producir estos valores lógicos a partir de valores numéricos.

EJEMPLO

```
a = ((b>c)&&(c>d))||(c==e)|(e==b);
```

Expresiones generales

Una de las características importantes del lenguaje C es su manejo para combinar expresiones y operadores de distintos tipos en una expresión. El resultado de una expresión lógica es siempre un valor numérico ya que puede ser 1 o 0, con esto podemos jugar con expresiones lógicas para poder realizar sub-expresiones en una expresión aritmética. De igual manera, cualquier valor numérico puede ser tratado como un valor lógico.

EJEMPLO

```
(a - b*2.0) && (c != d)
```

Otra expresión que se puede realizar es la asignación de variables con un mismo valor. Por ejemplo, el siguiente código que inicializa a 5 las tres variables **a**, **b** y **c**:

```
a = b = c = 5;
```

1.6.3. Reglas de precedencia y asociatividad

El resultado de una expresión va a depender del orden en que se ejecutan los comandos de la propia expresión, tomamos como ejemplo `2+6*5`.

Veamos como el compilador procesa la expresión:

- Primero se realiza el producto (`6*5`).
- Despues se realiza la suma (`2+30`).
- El resultado es 32.

Si realizásemos primero la suma y posteriormente la multiplicación el resultado sería otro muy diferente.

Por este motivo, para que quede claro es necesario explicar las reglas de prioridades y precedencia.



Hay que tener en cuenta que lo primero que siempre se hace son las operaciones entre paréntesis.

En la siguiente tabla se muestra la precedencia, disminuyendo de arriba abajo, y la asociatividad de los operadores de C.

Precedencia	Asociatividad
() [] -> .	izda a dcha
++ -- ! sizeof (tipo)	dcha a izda
+ (unario)- (unario)*(indir.)&(dirección)	
* / %	izda a dcha
+	izda a dcha
< <= > >=	izda a dcha
== !=	izda a dcha
&&	izda a dcha
	izda a dcha
? :	dcha a izda
= += -= *= /=	dcha a izda
, (operador coma)	dcha a izda

Figura 1.11. Tabla de prioridades.



El operador (*) tiene precedencia sobre el operador (+). Los operadores binarios (+) y (-) tienen igual precedencia, y asociatividad de izquierda a derecha, esto es el orden es el indicado por los paréntesis de la parte derecha de la expresión. Eso quiere decir que en la expresión, el orden de evaluación es el indicado por los paréntesis en la parte derecha de la línea.

$a - b + d * 5.0 + u / 2.0$
 $// (((a - b) + (d * 5.0)) + (u / 2.0))$

1.6.4. Sentencias

Las sentencias son partes completas, ejecutables en sí mismas. Muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias. Existen varios tipos de sentencias:

- Simples.
- Nula.
- Bloques.

Sentencias simples

Una sentencia simple es una expresión de algún tipo terminada con un carácter (;), el ejemplo más común es la declaración de variables o una expresión aritmética.

```
float real;  
espacio = espacio_inicial + velocidad * tiempo;
```

Sentencia nula

La sentencia nula, está vacía solo está compuesta por el (;). En algunas ocasiones hay que introducir al código este tipo de sentencias para que no realice ninguna tarea.

Sentencias bloques

Una sentencia compuesta es un conjunto de declaraciones y de sentencias agrupadas dentro de llaves { ... }. Una sentencia compuesta puede incluir otras sentencias anteriormente explicadas. Este tipo de sentencias se utiliza con mucha frecuencia, para modificar el flujo del programa. Un ejemplo es el siguiente:

```
{  
int i = 1, j = 3, k;  
double masa;  
masa = 3.0;  
k = y + j;  
}
```

1.7. Entrada y salida de información

Antes de pasar a ver ejemplos concretos en C, vamos a ver las instrucciones de **lectura y de escritura**. En la gran mayoría de los programas tienen en común el cálculo de valores y posteriormente mostrarlo en la salida. Otras veces es necesario que el usuario introduzca un valor, en ese caso es necesario leer los datos que introduzca el usuario, ya sea por teclado.

La lectura y escritura, es decir la entrada y salida de valores se efectuaran a través de llamadas a las funciones que están en la librería stdio, que es *standard input/output*, por lo que tendremos que declarar esta funciones en directivas, llamando a fichero stdio.h.

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

Estas funciones son:

- printf() Salida de datos por pantalla.
- scanf() Entrada/lectura de datos.
- getchar() Entrada de un carácter.
- putchar() Salida de un carácter.
- fflush() Borrado del buffer del teclado.



Se utilizarán diferentes funciones para leer los datos de teclado o de disco, lo mismo pasa con los de escritura, utilizaremos diferentes funciones si lo queremos dar salida en disco o pantalla.

Estas funciones de entrada y salida de datos tienen que tratarse como funciones con todas sus partes: nombre, valor retorno y argumentos.

1.7.1. Salida de información: escritura

Por defecto, la salida de información se va a hacer hacia la pantalla. En todos los programas que se desarrollen, y el programa escriba algún tipo de dato, o mensaje se escribirá en periférico de salida que por defecto toma el ordenador, la pantalla.

```
int printf("cadena_de_control", tipo arg1, tipo arg2, ...)
```



La función printf() imprime el texto contenido entre las comillas dobles, llamada **cadena_de_control**, con el valor de los otros argumentos, y con el formato incluido en la cadena de control.

puede haber un número indeterminado de argumentos y cada formato comienza con el carácter (%) y termina con un carácter de conversión.

Veamos el siguiente ejemplo

Deseamos imprimir varias variables: int, double y float, para ver como funciona la cadena de control con cada tipo.

Para ello lo primero debemos declararlas con el tipo, y les asignamos ya desde el código un valor y posteriormente llamamos a la función printf.

```
#include <stdio.h>
int main(void)
{
    int          a=2;
    double       b=123.2222;
    float        c =2.5;
    printf("\n variable entera, i: %d. \n la variable
tiempo: %lf \n la masa vale %f\n", i, tiempo, masa);
    return 0;
}
```

- En la primera línea escribimos la directiva que incluye stdio para realizar el printf.
- Segunda línea: función de programa principal.
- Tercera y última línea: abrimos y cerramos llaves, para indicar cual es el bloque de sentencias principales.
- Línea cuarta hasta sexta: declaramos la variables con su identificadores y del tipo que son, en este caso *int*, *double* y *float*.
- Línea séptima: Se imprimen 3 variables (**a**, **b** y **c**) con los formatos (%d, %lf y %f), correspondientes a los tipos (*int*, *double* y *float*), respectivamente. La cadena de control se imprime con el valor de cada variable intercalado en el lugar del formato correspondiente, teniendo en cuenta que realiza un salto de línea en cada \n.

Veamos el siguiente ejemplo

Veamos otro ejemplo, donde dentro del printf podemos ver la realización de la operación de una suma:

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

```
main()
{
    int a=20,b=10;
    printf("El valor de a es %d\n",a);
    printf("El valor de b es %d\n",b);
    printf("Por tanto %d+%d=%d",a,b,a+b);
}
```

En la línea 5, imprime el valor de la variable a, para ello en la cadena de control, ponemos una descripción de lo que estamos escribiendo ("el valor de a es..."), por otro lado dentro de la cadena de control debemos indicar el tipo que vamos a imprimir, en este caso como a es un entero por lo que debemos incluir el %d, a esto se le llama modificador que más adelante analizaremos los existentes. En la misma cadena de control indicaremos un salto de línea. Fuera de la cadena de control indicamos que variable va a ir en ese modificador o tipo, en este caso es la variable a.

En la línea 6, ídem a la anterior, solo cambia la variable a mostrar, que en este caso es la variable b.

Por último, en la línea séptima, podemos observar que en vez de mostrar solo una variable, podemos realizar varias variables, y también expresiones, como en el ejemplo de la suma. Esto es habitual, ya que nos reduce una línea de código, en este caso la sentencia del resultado, y la declaración de otra variable para guardar el mismo.

Veamos el siguiente ejemplo

En este programa se declaran 4 variables de tipo entero, uno, dos, diez y veinte. Iniciado el programa a estas variables se les da el valor 1, 2, 10 y veinte y posteriormente vamos a imprimirlas en pantalla, con la función anteriormente explicada printf.

```
#include <stdio.h>
int main(void)
{
    int uno, dos, diez, veinte;
    uno=1;
    dos=2;
    diez=10;
    veinte=20;
    printf("Este programa emplea el comando printf\n");
    printf("uno=%i \n", uno);
    printf("dos=%i\n", dos);
    printf("diez=%i\n", diez);
    printf("veinte=%i\n", veinte);
    return (0);
}
```

La primera instrucción, después de la asignación de valores a las variables, es escribir un mensaje al usuario. Con este mensaje lo que se pretende es informar al usuario acerca del objetivo del programa, es muy recomendable, ya que el usuario o programador en pruebas, nos informa de que va a hacer el programa antes de empezar. Se utiliza el procedimiento printf, que en este caso tiene un único parámetro que es una cadena.

La siguiente instrucción la función printf hay dos parámetros, que como se puede ver van separados por una coma, uno es una cadena de texto y otro es una variable, por lo que escribirá la cadena de caracteres correspondiente para mostrar el número. Lo que se verá en pantalla al ejecutarse esta sentencia es:

Uno= 1

Puesto que la función de escritura es printf, el cursor se queda a continuación del número 1. El modificador \n introduce un salto de línea de forma que los datos no aparezcan todos juntos en la misma línea de pantalla.

Los **modificadores** más utilizados son:

- %c Un único carácter.
- %d Un entero con signo, en base decimal.
- %u Un entero sin signo, en base decimal.
- %o Un entero en base octal.
- %x Un entero en base hexadecimal.
- %e Un número real en coma flotante, con exponente.
- %f Un número real en coma flotante, sin exponente.
- %s Una cadena de caracteres.
- %p Un puntero o dirección de memoria.

EJEMPLO

Ejemplo de uso de modificadores.

```
/* Uso de la sentencia printf() 2. */
#include <stdio.h>
main() /* Modificadores 1 */
{
    char cad[]="El valor de";
    int a=-15;
    unsigned int b=3;
    float c=932.5;
    printf("%s a es %d\n",cad,a);
    printf("%s b es %u\n",cad,b);
    printf("%s c es %e o %f",cad,c,c);
}
```

El resultado del código lo podemos ver en la siguiente pantalla.

```
El valor de a es -15
El valor de b es 3
El valor de c es 9.325000e+002 o 932.500000
```

Figura 1.12. Salida del programa.

El formato completo de los modificadores es el siguiente:

% [signo] [longitud] [.precisión] [l/L] conversión

- **Signo:** indicamos si el valor se ajustará a la izquierda, en cuyo caso utilizaremos el signo menos, o a la derecha que esto se asigna por defecto.
- **Longitud:** especifica la longitud máxima del valor que aparece por pantalla. Si la longitud es menor que el número de dígitos del valor, éste aparecerá ajustado a la izquierda.
- **Precisión:** indicamos el número máximo de decimales que tendrá el valor.
- **I/L:** utilizamos l (minúsculas) cuando se trata de una variable de tipo long y L (mayúsculas) cuando es de tipo double.

1.7.2. Entrada de información: lectura

La función **scanf()** es análoga en muchos aspectos a **printf()**, y se utiliza para leer datos de la entrada estándar, que por defecto es el teclado, la sintaxis de la función es la siguiente:

```
int scanf(<%x1%x2...>, &arg1, &arg2, ...);
```

Donde:

- x1, x2, son los caracteres de conversión o modificadores, mostrados en la siguiente tabla, que representan los formatos con los que se espera encontrar los datos.
- arg1,arg2, son las variables donde se van a almacenar los datos.

La siguiente tabla representa los caracteres de conversión para la función **scanf**:

Carácter	Caracteres leídos	Argumento
c	cualquier carácter	char *
d, i	entero decimal con signo	int *
u	entero decimal sin signo unsigned	int
o	entero octal unsigned	int
x, X	entero hexadecimal	unsigned int
e, E, f, g, G	número de punto flotante	float
s	cadena de caracteres sin ‘ ’	char
h, l	para short, long y double	ninguno
L	modificador para long double	ninguno

Figura 1.13. Caracteres de conversión para *scanf*.

Por ejemplo, para leer los valores de dos variables int y double y de una cadena de caracteres, se utilizarían la sentencia:

```
int n;
double distancia;
char nombre[20];
scanf("%d%lf%s", &n, &distancia, nombre);
```

Debemos realizar una correlación entre la variable o argumento de la función, es decir si el primer modificado es %d, el primer argumento debe ser una variable de ese tipo.



El nombre no va precedido por el operador (&) ya que, para leer un valor cadena de caracteres se detiene en cuanto lea un espacio en blanco por lo que: para leer una línea completa con varias palabras hay que usar otras técnicas.

En los formatos de la cadena de control de **scanf()** pueden introducirse corchetes [...], que lee sólo los caracteres que aparecen en el corchete. Cuando se encuentra un carácter distinto de éstos se acaba de leer. Si los corchetes contienen un carácter (^), se leen todos los caracteres distintos de los caracteres que se encuentran dentro de los corchetes a continuación del (^). Veamos un ejemplo:

```
scanf(" %[^\n]", s);
```

Lee todos los caracteres que encuentra hasta que llega al carácter **nueva línea** '\n'. Esta sentencia puede utilizarse por tanto para leer líneas completas, con blancos incluidos.



Que con el modificador o formato %s la lectura se detiene al llegar al primer delimitador, carácter blanco, tabulador o nueva línea.

Veamos el siguiente ejemplo

En el siguiente programa, guardamos los datos de las variables: largo alto y ancho. Si el usuario introduce una única línea de datos con los valores 2 3 4 en cada variable asociada al scanf y muestra el dato con el printf.

```
#include <stdio.h>
int main(void)
{
    int largo, ancho, alto, volumen;
    printf ("Este programa calcula el volumen de un paralelepipedo: \n");
    printf ("\nCuanto mide de largo: ");
    scanf ("%d" , &largo);
    printf ("\nCuanto mide de ancho: ");
    scanf ("%d" , &ancho);
    printf ("\nCuanto mide de alto: ");
    scanf ("%d" , &alto);
    volumen=largo*ancho*alto;
    printf ("\nEl volumen es: %d", volumen);
```

```

    return 0;
}

```

Al ejecutar este programa lo que se ve en la pantalla de ejecución son los siguientes mensajes:

```

Este programa calcula el volumen de un paralelepípedo:
Cuanto mide de largo: 2
Cuanto mide de ancho: 3
Cuanto mide de alto: 4
El volumen es: 24

```

Figura 1.14. Salida del programa.

1.7.3. Macros getchar() y putchar()

Son marcos y no funciones, pero casi todos los efectos de comportan como funciones. Estas macros, `getchar()` y `putchar()`, permiten leer e imprimir *un sólo carácter* cada vez respectivamente, en la entrada o en la salida estándar.



Ambas funciones devuelven un valor Char, es decir un carácter, que se encuentra en el índice especificado de la cadena suministrada.

- La función `getchar()` recoge un carácter introducido por teclado y lo deja disponible como valor de retorno.
- La función `putchar()` escribe en la pantalla el carácter que se le pasa como argumento.

Veamos varios ejemplos para aclarar cada una de las funciones:

Escribe en pantalla el carácter a, véase que va con apostrofe, ya que no es una cadena como podemos observar en la siguiente línea, que es como se trata las cadenas de control con `printf`, equivalente a `putchar`:

```

putchar('a');
printf("a");

```

Para `getchar`, debemos guardar en la variable, el carácter que escribamos por teclado, en la segunda instrucción podemos ver la equivalencia con el `scanf`.

```
c = getchar();  
scanf("%c", &c);
```

Estas macros están definidas en el fichero **stdio.h**, y su código es sustituido en el programa por el *preprocesador* antes de la compilación.



RESUMEN

- Al utilizar el procedimiento de salida, printf, se escribe, por defecto, en el monitor del ordenador. Lo que se escribe es el flujo de caracteres correspondiente a una constante una variables ó una expresión.
- Los procedimientos de salida o escritura transmiten a la pantalla las representaciones como caracteres de los valores de sus parámetros en el orden que aparecen en la lista. En la lista de parámetros a representar, los parámetros deben ir separados por comas.
- Se puede controlar el aspecto de los caracteres que representan a los datos en la pantalla.
- La inclusión de comentarios en un programa es una práctica muy recomendable, como lo reconocerá cualquiera que haya tratado de leer un listado hecho por otro programador ó por sí mismo, varios meses atrás. Para el compilador, los comentarios son inexistentes, por lo que no generan líneas de código, permitiendo abundar en ellos tanto como se deseé.
- En C, las instrucciones básicas de salida y entrada son printf y scanf, las cuales permiten respectivamente enviar (print, imprimir) o recibir (scan, leer) información.
- Todo programa en C se compone de una secuencia de declaraciones y de un conjunto de instrucciones ejecutables, cada una de las cuales va en un sitio determinado.
- todos los ficheros que contienen código fuente en C deben terminar con la extensión (.c), como por ejemplo: producto.c, solucion.c, etc.
- Las llaves {...} constituyen el modo utilizado por el lenguaje C para agrupar varias sentencias de modo que se comporten como una sentencia única (*sentencia compuesta o bloque*). Todo el cuerpo de la función debe ir comprendido entre las llaves de apertura y cierre.



estudios abiertos

SEAS

GRUPO SANVALERO

2
UNIDAD
DIDÁCTICA

Introducción a la programación C

2. Programación estructurada en C

ÍNDICE

ÍNDICE	61
OBJETIVOS	63
INTRODUCCIÓN	64
2.1. Programación estructurada en C	65
2.2. Estructuras de selección o decisión.....	66
2.2.1. Estructura de decisión simple.....	66
2.2.2. Estructura de decisión doble	70
2.2.3. Estructura de decisión anidada	73
2.2.4. Estructura de decisión múltiple	75
2.3. Estructuras iterativas	79
2.3.1. Estructura <i>while</i>	79
2.3.2. Estructura <i>for</i>	82
2.3.3. Sentencia do ... While	86
2.3.4. Sentencias break, continue, goto	87
RESUMEN	89

OBJETIVOS

- Conocer y saber utilizar correctamente las estructuras de selección, también llamadas de decisión. Se aprenderá a interpretar y a manejar la estructura de decisión simple, la doble, estructuras anidadas y por último la estructura de selección múltiple. Veremos cuando es conveniente utilizar cada una de ellas.
- Conocer y saber utilizar correctamente las estructuras iterativas. Se aprenderá a manejar las tres estructuras iterativas de que dispone el lenguaje C, la estructura iterativa WHILE, la estructura iterativa FOR y por último la estructura iterativa DO...WHILE.
- Una vez vistas las estructuras básicas de las que dispone el lenguaje de programación C, se desarrollarán ejercicios que impliquen el manejo de varias de estas estructuras en el mismo programa.

INTRODUCCIÓN



Antes de pasar a escribir directamente los programas en C, hay que conocer cuáles son las herramientas de que dispone este programa para poder expresar los algoritmos vistos en la unidad didáctica anterior.

Hemos de conocer la sintaxis de las diferentes estructuras de que dispone C, es decir, hemos de saber cómo se escriben las instrucciones correctamente para luego poder escribir los programas.

Así mismo veremos las herramientas de que dispone el compilador de Turbo C, que nos facilitará la comprensión del funcionamiento de las diferentes estructuras, así como la depuración de los mismos en caso de que existan errores.

Por último, comentar que la comprensión de esta unidad didáctica, al igual que la anterior, es clave puesto que el resto de la programación se apoya en la utilización de estas estructuras básicas.

En principio, las sentencias de un programa en C se ejecutan secuencialmente, esto es, cada una a continuación de la anterior empezando por la primera y acabando por la última. El lenguaje C dispone de varias sentencias para modificar este flujo secuencial de la ejecución.

Las más utilizadas se agrupan en dos familias: las bifurcaciones, que permiten elegir entre dos o más opciones según ciertas condiciones, y los bucles, que permiten ejecutar repetidamente un conjunto de instrucciones tantas veces como se desee, cambiando o actualizando ciertos valores.



2.1. Programación estructurada en C

Existen dos técnicas principales a la hora de programar que son la programación estructurada y la programación modular.

La programación estructurada se basa en una serie de estructuras básicas para construir todos los programas. Los problemas que se pueden resolver (eficientemente) utilizando esta técnica de programación son problemas de dificultad media o baja.

Por el contrario, existe la programación modular, que se basa en dividir en pequeños problemas o subproblemas el problema principal. Cada uno de estos subproblemas se puede implementar o programar independientemente o a su vez pueden ser divididos en otros subproblemas. Cada uno de los subproblemas, que se puede resolver de manera independiente, utiliza la programación estructurada. Por lo tanto, la técnica de programación modular se basa en la técnica de programación estructurada para resolver los problemas.

La programación estructurada se basa en las siguientes estructuras para resolver los problemas:

- Estructuras de decisión:
 - Estructura de decisión simple.
 - Estructura de decisión doble.
 - Estructura de decisión anidada.
 - Estructura de decisión múltiple.
- Estructuras iterativas:
 - Estructura iterativa WHILE.
 - Estructura iterativa FOR.
 - Estructura iterativa anidada.

Cada una de las estructuras las veremos en detalles en las siguientes secciones. Antes de pasar a ver su funcionamiento, hay que saber que estas estructuras son muy similares a las que se han visto en la unidad didáctica 4, por lo que no se encontrará mayor dificultad en la comprensión de cada una de las estructuras. La sintaxis varía puesto que vamos a ver cada una de las estructuras en C. Es importante prestar atención a la sintaxis de cada una de las estructuras en función de si se ejecutan una instrucción o más de una instrucción puesto que la sintaxis es diferente.

2.2. Estructuras de selección o decisión

Las estructuras de selección ó decisión sirven, para decidir si se ejecutarán o no otros enunciados en función de una ó más condiciones.



Se evalúa expresión. Si el resultado es true (#0), se ejecuta sentencia; si el resultado es false (=0), se salta sentencia y se prosigue en la línea siguiente. Hay que recordar que sentencia puede ser una sentencia simple o compuesta (bloque { ... }).

2.2.1. Estructura de decisión simple

Esta estructura sirve para ejecutar una sentencia o grupo de sentencias en función del valor de una expresión booleana. Para que se ejecute la sentencia el resultado de la expresión booleana debe ser el valor lógico verdadero, conocido en C como valor lógico *true*.

La sintaxis de la estructura de selección simple, para la cual únicamente hay que ejecutar una única sentencia es la siguiente.

```
If expresión  
    Sentencia ;
```

Tenemos que tener en cuenta lo siguiente:

- Esta sentencia es equivalente a la sentencia *si..entonces* que se puede ver en pseudocódigo y su correspondiente en diagrama de flujo.
- Las palabras *if* es una palabra reservadas del lenguaje.
- Después de la sentencia **no** debe ir signo de punto y coma (;), que como es sabido, finaliza las sentencias en C.
- La sangría o tabulación no es obligatoria en C, pero se utiliza para hacer más comprensible la codificación.

La sintaxis de la estructura de selección simple, para la cual se ejecutan una o más sentencias es la siguiente.

```
If expresiónbooleana
{
    Sentencia 1;
    Sentencia 2;
    .
    .
    Sentencia n;
}
```

Puesto que las sentencias deben acabar con un punto y coma, si hay que ejecutar más de una sentencia en la estructura selectiva simple, hay que poner las palabras las llaves {...} para que el compilador sepa que conjunto de sentencias debe ejecutar en caso de que la expresión booleana sea verdadera (true).



El tema de tabulación o sangría, no es obligatoria la sangría, aunque sí es MUY recomendable.

Para ilustrar el funcionamiento de esta estructura vamos a ver algunos ejemplos.

Ejemplo 1

Hacer un programa en C, para calcular el área y el volumen de un cilindro.

Para poder calcular el volumen y el área de cilindro se necesitan conocer los valores del radio y de la altura. **Como es sabido, sólo se puede calcular el área y el volumen del cilindro si los valores del radio y altura del cilindro son positivos.**

- Datos de entrada: radio y altura.
- Datos de salida: área y volumen.

El programa resuelto en C es el siguiente.

```

1 #include <stdio.h>
2 #define pi 3.1416
3 int main(void)
4 {
5     //declaracion de variables
6     double radio, altura, area, volumen;
7     //solicitud de datos de entrada
8     printf("Introduce el valor del radio: ");
9     scanf("%lf",&radio);
10    printf("Introduce el valor de la altura: ");
11    scanf("%lf", &altura);
12    area=0;
13    volumen=0;
14    // si los valores introducidos son >0 puede operar
15    if (radio>0 && altura>0)
16    {
17        area = (2*pi*(radio *
18 radio))+(2*pi*radio*altura);
19        volumen = pi * (radio * radio) * altura;
20    }
21    printf("\n El valor del area es: %f",area);
22    printf("\n El valor del volumen es: %f", volumen)
23    fflush(stdin);
24    getchar();
25    return 0;
26 }
```

Notas:

- En la línea 2 se crea una constante (PI) y se le asigna su valor.
- En la linea 6 se declaran las variables de entrada y salida.
- En la linea 9 y 11 se hace lectura de los valores de entrada.
- Sólo calculamos el área y el volumen si el valor del radio y del círculo son positivos (línea 15).
- La instrucción de escritura de las líneas 21 y 22 muestran los resultados de los cálculos.
- En la linea 24 getchar(): va a pausar la ejecucion del programa para esperar el ingreso de un caracter por la entrada estandar del sistema, en este caso, el teclado. Esto es utilizado ya que en muchas ocasiones no deja ver el resultado y devuelve el retorno al sistema, de esta manera hasta que no pulsamos una tecla, lo podemos estar visualizando el resultado.



- Por otro lado, en la linea 23, utilizamos el fflush(stdin): Limpia el buffer utilizado por la entrada estandar, en este caso, el teclado, esto es debido que el buffer del teclado se queda con datos residuales de salto de lineas, etc... por lo que vaciamos para que a la lectura del getchar() y asi hace una espera de teclado.

Ejemplo 2

¿Cómo haríamos para saber si alguien es mayor de edad? Los pasos serian los siguientes:

1. Preguntarle su edad.
2. Si tiene 18 o más entonces es mayor de edad.
3. Si no, es menor de edad.

```

1 #include<stdio.h>
2 int main(void)
3 {
4     int edad=0;
5     printf("Cual es tu edad?: ");
6     scanf("%d", &edad);
7     if(edad>=18)
8         printf("Eres mayor de edad");
9     printf("pulse una tecla para finalizar....!");
10    getch();
11    return 0;
12 }
```

Notas:

Línea 4, declaramos la variable edad, que en la misma linea la podemos inicializar a 0.

- en la linea 6 hacemos lectura de la edad.
- en la linea 7, realizamos el if **edad>=18**, que es una comparación. Si la variable 'edad' es mayor o igual a 18, se hacen la instrucción que va seguida (linea 8), en este caso como solo es una instrucción, la que esta dentro del if, no hace falta la insercción de llaves.
- Si en la condicion no cumple es decir, es menor a 18, no entra y pasa directamente a realizar la linea 9.

2.2.2. Estructura de decisión doble

En la estructura anterior se ejecutaban una acción o un conjunto de acciones si la condición es evaluada como verdadera. En ocasiones, es frecuente que un programa tenga que decidir entre dos alternativas en función del valor que tome una determinado expresión.

Por ejemplo, en el ejemplo anterior, sería conveniente que, en caso de que alguna de las variables sea negativa, se muestre un mensaje de error. Bien, en este caso el programa debe decidir entre dos alternativas que son:

- Calcular el área, volumen y mostrar el resultado.
- Mostrar un mensaje de error.

En tales casos se utiliza la estructura de decisión doble que, como hemos comentado antes, decide entre dos alternativas cuál de ellas se va a ejecutar, siempre en función de una condición. La estructura de decisión doble tiene la siguiente sintaxis.

```
if ExpresionBooleana
    Sentencias 1;
else
    Sentencias 2;
```

Notas importantes:

- La sintaxis anterior es válida en el caso de que **únicamente haya que ejecutar una única sentencia**, tanto si la expresión es verdadera como si es falsa.
- Si el resultado de evaluar expresión devuelve un valor verdadero (representado por un valor numérico distinto de cero), se ejecutará la sentencia de instrucciones situada a continuación del if (sentencias 1).
- Si la expresión devuelve un valor falso (representado por un valor numérico igual a cero), se ejecutará la secuencia de instrucciones situada a continuación de else (sentencias 2).
- La expresión debe ir encerrada entre parentesis para distinguirla del resto del código.

La sintaxis de la estructura de decisión doble cambia si en lugar de ejecutar una única sentencia hay que ejecutar más de una (también es válido si se ejecuta una única sentencia). Como en todas las instrucciones que veremos, si hay más de una sentencia a ejecutar, para cualquier estructura.

Por lo tanto, si hay que ejecutar más de una sentencia la sintaxis es la siguiente.

```

1  If ExpresionBooleana
2  {
3      Sentencia 1V;
4      Sentencia 2V;
5      .
6      .
7  }
8  Else
9  {
10     Sentencia 1F;
11     Sentencia 2F;
12 }
```



Un If no tiene que tener un else a la fuerza, es opcional. Es muy importante que se fijen muy bien en las llaves, cada If tiene su llave de apertura y de cierre, al igual que el else.

Para comprender el funcionamiento de esta instrucción vamos a ver un ejemplo.

Ejemplo

Hacer un programa en C para calcular el área y el volumen de un cilindro. En caso de que no se pueda calcular estos datos (valor de radio o altura negativo) se mostrará un mensaje de error.

Este ejercicio es muy similar al anterior, la única diferencia es que en este caso el programa ha de elegir entre dos alternativas: calcular el valor de área, volumen y mostrar el resultado ó mostrar un mensaje de error.

- Datos de entrada: radio y altura.
- Datos de salida: área y volumen o mensaje de error.

Los pasos que hay que seguir para resolver este algoritmo son los siguientes:

- Pedir los datos asuario, radio y altura.
- Si los datos anteriores son positivos:
 - Calcular el área.
 - Calcular el volumen.
 - Escribir los resultados, para que el ususraio los pueda ver.

- Si alguno de los datos son negativos se muestra el mensaje de error. Esto equivale a que la condición anterior es evaluada como falsa, puesto que será evaluada de esta forma en el caso de que alguna de las condiciones sea falsa, es decir el radio ó la altura son menores que cero.

El programa, en C, es el siguiente:

```

1 #include <stdio.h>
2 #define pi 3.1416
3 int main(void)
4 {
5     //declaracion de variables
6     double radio, altura, area, volumen;
7     //solicitud de datos de entrada
8     printf("Introduce el valor del radio: ");
9     scanf("%lf",&radio);
10    printf("Introduce el valor de la altura: ");
11    scanf("%lf", &altura);
12    area=0;
13    volumen=0;
14    //filtramos valores positivos o negativos
15    if (radio>0 && altura>0)
16    {
17        area = (2 * pi * (radio * radio )) + (2 * pi *
18 radio * altura);
19        volumen = pi * (radio * radio) * altura;
20        printf("\n El valor del area es:   %f",area);
21        printf("\n El valor del volumen es:  %f",
22 volumen);
23    }
24    else
25        printf("Error en los valores de entrada, son
26 negativos! \n");
27    flush(stdin);
28    printf("pulse una tecla para finalizar....!");
29    getchar();
30    return 0;
31 }
```

Como se puede ver, puesto que hay que decidir entre dos alternativas, se utiliza la sentencia *if..else..*

- Si la condición es evaluada como verdadera se ejecuta el bloque de código comprendido entre las líneas 16 y 23.



- Si la condición es evaluada como falsa, es decir uno de los dos valores es negativo, se ejecuta la sentencia que viene a continuación de else. En esta ocasión no hemos puesto llaves, pero se podría incorporar, no emitirá error ya que solo es una instrucción y podemos prescindir de las llaves.



Observe como se deben sangrar las llaves para una mejor comprensión del código.

2.2.3. Estructura de decisión anidada

Hemos visto que la estructura de decisión doble nos permite elegir entre dos alternativas, pero *¿que ocurre si tenemos que elegir entre más de dos alternativas?*, C nos permite “anidar”.

Esta sentencia permite realizar una ramificación múltiple, ejecutando una entre varias partes del programa según se cumpla una entre n condiciones. La forma general es la siguiente:

```
if (expresion_1)
    sentencia_1;
else if (expresion_2)
    sentencia_2;
else if (expresion_3)
    sentencia_3;
else if (...)

...
[else
sentencia_n; ]
```

- Se evalúa expresion_1. Si el resultado es true, se ejecuta sentencia_1.
- Si el resultado es false, se salta sentencia_1 y se evalúa expresion_2.
- Si el resultado es true se ejecuta sentencia_2, mientras que si es false se evalúa expresion_3 y así sucesivamente.
- Si ninguna de las expresiones o condiciones es true se ejecuta expresion_n que es la opción por defecto. Esta última puede ser vacía, y en ese caso se puede eliminar junto con la palabra else.
- Todas las sentencias pueden ser simples o compuestas.

Ejemplo

Hacer un programa para que, dada una nota muestre la calificación obtenida para dicha nota. Las calificaciones son:

1. 10: Matrícula de Honor.
2. Entre 9 y 10: Sobresaliente.
3. Entre 7 y 9 (menor estricto): Notables.
4. Entre 6 y 7 (menor estricto):: Bien.
5. Entre 5 y 6 (menor estricto):: Suficiente.
6. Menor que 5 (estricto): Insuficiente.
7. En cualquier otro caso se muestra un mensaje de error.

Como podemos ver en el enunciado, hemos de elegir entre 7 alternativas. Cuando se tienen más de dos alternativas es posible enlazar las estructuras *if..else..*. Tantas veces como sea necesario, para de esta forma poder decidir entre el número adecuado de alternativas.

Vamos a resolver el problema, hacer un programa para que, dada una nota (real) muestre la calificación obtenida para dicha nota, el programa, en C, es el siguiente.

```

1 #include <stdio.h>
2 int main(void)
3 {
4     float nota;
5     printf("programa de calificacion de notas:\n ");
6     printf("Introduce una nota: ");
7     scanf("%f", &nota);
8     if(nota == 10)
9         printf("La calificacion es Matricula de Honor");
10    else if(nota >= 9 && nota <10)
11        printf("La calificacion es Sobresaliente");
12    else if(nota >= 7 && nota < 9)
13        printf("La calificacion es Notable");
14    else if(nota>=6 && nota < 7)
15        printf("La calificacion es Bien");
16    else if(nota >= 5 && nota < 6)
17        printf("La calificacion es Suficiente");
18    else if(nota>=0 && nota < 5)
19        printf("Estas suspendido chavalote");
20    else
21        printf ("El numero es erroneo");
22     fflush(stdin);
23     getchar();
24     return 0;
25 }
```

Notas:

- Las estructura de decisión anidada va desde la línea 8 a la 21 ambas incluídas.

2.2.4. Estructura de decisión múltiple

Las estructuras de decisión anidadas son un poco farragosas y difíciles de entender y escribir si hay muchas. Para solucionar esta dificultad, se puede utilizar la estructura de decisión múltiple.

En situaciones en las que se tienen diferentes posibilidades y la variable o expresión es de tipo ordinal (se excluyen los reales y subtipos derivados) se puede aprovechar el uso de la estructura de decisión múltiple.

La sentencia switch que se detalla a continuación desarrolla una función similar a la de la sentencia *if... else* con múltiples ramificaciones, aunque como se puede ver presenta también importantes diferencias. La forma general de la sentencia switch es la siguiente:

```
switch (expresion)
{
    case expresion_cte_1:
        sentencia_1;
    case expresion_cte_2:
        sentencia_2;
        ...
    case expresion_cte_n:
        sentencia_n;
    default:
        sentencia;
}
```

Explicación:

- Se evalúa **expresión** y se considera el resultado de dicha evaluación. Si dicho resultado coincide con el valor constante **expresion_cte_1**, se ejecuta **sentencia_1** seguida de **sentencia_2**, **sentencia_3**, ..., **sentencia**.
- Si el resultado coincide con el valor constante **expresion_cte_2**, se ejecuta **sentencia_2** seguida de **sentencia_3**,..., **sentencia**.
- En general, se ejecutan todas aquellas sentencias que están a continuación de la **expresion_cte** cuyo valor coincide con el resultado calculado al principio.
- Si ninguna **expresion_cte** coincide se ejecuta la **sentencia** que está a continuación de **default**.
- Si se desea ejecutar únicamente una **sentencia_i** (y no todo un conjunto de ellas), basta poner una sentencia **break** a continuación (en algunos casos puede utilizarse la sentencia **return** o la función **exit()**).
- El efecto de la sentencia **break** es dar por terminada la ejecución de la sentencia **switch**. Existe también la posibilidad de ejecutar la misma **sentencia_i** para varios valores del resultado de **expresion**, poniendo varios **case expresion_cte** seguidos.

Ejemplo 1

Resolver el ejercicio planteado anterior de las calificaciones de las notas utilizando la estructura de decisión múltiple.

El programa es el siguiente.

```

1 #include <stdio.h>
2 int main(void)
3 {
4     int nota;
5     printf("Introduce la nota: ");
6     scanf("%d", &nota);
7     switch(nota) {
8         case 10:
9             printf("La nota es Matricula de Honor");
10            break;
11         case 9:
12             printf("La nota es sobresaliente");
13             break;
14         case 8:
15             printf("La nota es notable");
16             break;
17         case 7:
18             printf("La nota es notable");
19             break;
20         case 6:
21             printf("La nota es bien");
22             break;
23         case 5:
24             printf("La nota es suficiente");
25             break;
26         case 4:printf("Estas suspendido
27 chavalote");break;
28         case 3:printf("Estas suspendido
29 chavalote");break;
30         case 2:printf("Estas suspendido
31 chavalote");break;
32         case 1:printf("Estas suspendido
33 chavalote");break;
34     }
35     getchar();
36     return 0;

```

Notas:

- Vemos que esta estructura es más fácil identificar que sentencia se ejecuta en cada caso.
- Para los valores 7 y 8 se ejecuta la misma instrucción líneas 14-19.

- Para los valores 0, 1, 2, 3, 4 se ejecuta la misma instrucción, pero a continuacion se muestra como poder hacer el switch de forma mas comprimida:

```
...
switch(nota) {
    case 10:
        printf("La nota es Matricula de Honor");
        break;
    case 9:
        printf("La nota es sobresaliente");
        break;
    case 8:
    case 7:
        printf("La nota es notable");
        break;
    case 6:
        printf("La nota es bien");
        break;
    case 5:
        printf("La nota es suficiente");
        break;
    case 4:
    case 3:
    case 2:
    case 1:
    case 0:
        printf("Estas suspendido chavalote");break;
...
}
```

- Si la variable nota toma un valor distinto de las constantes que se han enumerado se puede incluir la opcion de Default que se implementaria de la siguiente manera:

```
default:
    printf("el numero es incorrecto");break;
```



2.3. Estructuras iterativas

Las estructuras iterativas, también conocidas como estructuras repetitivas o bucles, sirven para ejecutar una sentencia o grupos de sentencias que se repiten en función de una determinada condición.

Además de bifurcaciones, en el lenguaje C existen también varias sentencias que permiten repetir una serie de veces la ejecución de unas líneas de código. Esta repetición se realiza, bien un número determinado de veces, bien hasta que se cumpla una determinada condición de tipo lógico o aritmético. De modo genérico, a estas sentencias se les denomina bucles.

Para ello, antes hemos de conocer las estructuras iterativas que dispone C y la sintaxis de cada una de ellas. Las estructuras iterativas que dispone C son las siguientes:

- La estructura repetitiva WHILE.
- La estructura repetitiva DO ...WHILE.
- La estructura repetitiva FOR.

En las siguientes secciones se presentan la sintaxis y ejecución de estas tres estructuras repetitivas, comentaremos las ventajas del uso de cada una de ellas, y veremos ejemplo que nos ayuden a entender el funcionamiento de estas estructuras.

2.3.1. Estructura *while*

En esta estructura se ejecuta la sentencia o grupo de sentencias asociadas si la condición o Expresión booleana que controla la ejecución del bucle es evaluada como true, por el contrario deja de ejecutarse cuando esa expresión booleana es evaluada como false.



La estructura repetitiva WHILE se corresponde con la estructura mientras en pseudocódigo.

Esta estructura, debido a que se evalúa la condición al principio del bucle, se puede ejecutar entre 0 y N veces.

La sintaxis general de la estructura iterativa WHILE es la siguiente.

```
while (expresion_de_control)
{
    Sentencia 1;
    Sentencia 2;
    .
    .
    sentencia n;
}
```

Notas:

- Si expresión booleana es falsa al principio, la primera vez que se evalúa, nunca se ejecutarán las sentencias que se contienen dentro del cuerpo del bucle.
- Si sólo hay una sentencia dentro del cuerpo del bucle, se pueden eliminar los identificadores de llaves de inicio y fin.
- Al llegar al final del bucle (cierre de llave }), se vuelve al comienzo del bucle a evaluar nuevamente la condición que controla la ejecución del bucle.

Esta estructura resulta apropiada cuando no es conocido de antemano si las condiciones requerirán la ejecución del cuerpo del bucle.



Por otro lado, al diseñar este tipo de estructura iterativa hay que asegurar que la expresión booleana que controla ejecución del cuerpo del bucle de cómo resultado el valor false, para que finalice la ejecución del mismo puesto que si no el bucle se ejecutará de forma indefinida y el ordenador se quedará "colgado". A este tipo de bucles se les conoce como bucles infinitos.

Para ilustrar el funcionamiento de esta estructura iterativa vamos a ver un ejemplo.



Ejemplo 1

Hacer un programa que sirva para hacer la media de 5 valores que se pedirán al usuario.

```

1 #include <stdio.h>
2 int main(void)
3 {
4     int valor, suma, cantidad;
5     printf("Calculo de la media de 5 valores\n");
6     cantidad = 0;
7     valor = 0;
8     suma = 0;
9     while (cantidad < 5) {
10         printf("Introduce un valor: ");
11         scanf("%d", &valor);
12         suma = suma + valor;
13         cantidad = cantidad + 1;
14     }
15     printf("La media es: %d", suma/5);
16     fflush(stdin);
17     getchar();
18     return 0;
19 }
```

Se evalúa expresión_de_control que en este caso es `cantidad < 5`

- Si el resultado es false se salta sentencia y se prosigue la ejecución en la línea 15.
- Si el resultado es true se ejecuta las sentencias (10..13) y se vuelve a evaluar `expresión_de_control`.

Evidentemente la variable cantidad se suma 1 para que intervenga en la expresión de control, pues si no el bucle continuaría indefinidamente.

La ejecución de sentencia prosigue hasta que expresión_de_control se hace false, en cuyo caso la ejecución continúa en la línea siguiente a sentencia, línea 15.

En otras palabras, sentencia se ejecuta repetidamente mientras cantidad < 5 sea true, y se deja de ejecutar cuando se hace false , es decir mayor a 5.



Obsérvese que en este caso el control para decidir si se sale o no del bucle está antes de sentencia, por lo que es posible que sentencia no se llegue a ejecutar ni una sola vez

2.3.2. Estructura *for*

La estructura repetitiva FOR se corresponde con la estructura *desde* en pseudocódigo. Muchas de las estructuras iterativas son controladas por contadores, en las cuales es conocido el valor inicial de la variable contadora que controla la ejecución del bucle y el valor final de la misma. En tales casos es más sencillo utilizar una estructura iterativa *FOR*, puesto que incrementa automáticamente en uno el valor del contador en cada iteración.

Se controla el ascenso o descenso de la estructura mediante la sentencia de incremento o decremento.

A continuación, vamos a ver en detalle cada una de las estructuras iterativas.

La estructura FOR, tiene las siguientes características:

- Preiniciación de la variable contadora (que controla la ejecución del bucle).
- El cuerpo del bucle se ejecuta mientras el valor de la variable contadora sea menor o igual que el valor final de la misma.
- Al llegar al final del bucle la variable contadora se incrementa automáticamente en uno (por lo tanto no es necesario escribir tal instrucción como ocurre en otras estructuras iterativas).

La sintaxis de esta estructura es la siguiente.

```
for (inicializacion; expresion_de_control; actualizacion)
{
    Sentencia 1;
    Sentencia 2;
    .
    .
    sentencia n;
}
```

Explicación

Possiblemente la forma más sencilla de explicar la sentencia *for* sea utilizando la construcción *while* que sería equivalente. Dicha construcción es la siguiente:

```
inicializacion;
while (expresion_de_control)
{
    Sentencia;
    Actualización;
}
```

Donde sentencia puede ser una única sentencia terminada con (;), otra sentencia de control ocupando varias líneas (if, while, for, ...), o una sentencia compuesta o un bloque encerrado entre llaves {...}. Antes de iniciarse el bucle se ejecuta inicializacion, que es una o más sentencias que asignan valores iniciales a ciertas variables o contadores. A continuación se evalúa expresion_de_control y si es false se prosigue en la sentencia siguiente a la construcción for; si es true se ejecutan sentencia y actualizacion, y se vuelve a evaluar expresion_de_control.

El proceso prosigue hasta que expresion_de_control sea false. La parte de actualización sirve para actualizar variables o incrementar contadores.

Un ejemplo típico puede ser el producto escalar de dos vectores a y b de dimensión n:

```
for (pe = 0.0, i=1; i<=n; i++)
    pe += a[i]*b[i];
```

Primeramente se inicializa la variable pe a cero y la variable i a 1; el ciclo se repetirá mientras que i sea menor o igual que n, y al final de cada ciclo el valor de i se incrementará en una unidad. En total, el bucle se repetirá n veces. La ventaja de la construcción for sobre la construcción while equivalente está en que en la cabecera de la construcción for se tiene toda la información sobre cómo se inicializan, controlan y actualizan las variables del bucle.



Obsérvese que la inicialización consta de dos sentencias separadas por el operador (,).

Es aconsejable utilizar un bucle FOR ascendente, cuando en cada iteración se asocia un valor distinto a la variable contadora y además, se conocen los límites del contador, siendo el valor inicial menor que el valor final.

Para ilustrar el funcionamiento de esta estructura iterativa vamos a ver algunos ejemplos.

Ejemplo

Hacer un programa en C que sirva para hacer la suma de 10 primeros números naturales.

En este caso se dan las condiciones necesarias para que se pueda utilizar un bucle desde ascendente. En cada iteración tendremos una variable que tomará los valores 1, 2, 3., .., 10, es decir toma un valor distinto y además los límites de esta variable son 1 y 10. Por último, el valor inicial es menor que el valor final.

El programa es el siguiente.

```

1 #include <stdio.h>
2 int main(void)
3 {
4     int numero, suma;
5     numero = 0;
6     suma = 0;
7     printf("Suma de los 10 primeros naturales\n");
8     for(numero=1;numero<=10;numero++)
9         suma = suma + numero;
10    printf("El resultado de la suma es: %d", suma);
11    getchar();
12    return 0;
13 }
```

Nota:

- En la línea 8 podemos ver que se inicializa la variable (entera, por lo tanto es un ordinal) al valor 1, y que el valor final es 10, la tercera parte de la sentencia es el incrementador (numero ++), si quisieramos decrementarla emplearíamos el operador -- (numero --).
- Al ejecutar paso a paso el programa anterior los valores de las variables, así como el resultado de la expresión es la siguiente.

Iteración	Expresión (numero<=10)	Suma	Numero
No iteración	True	0	1
1	True	1	2
2	True	3	3
3	True	6	4
4	True	10	5
5	True	15	6
6	True	21	7
7	True	28	8
8	True	36	9
9	True	45	10
10	True	55	11
11	false		

Lo que veremos en pantalla será:

```
C:\Dev-Cpp\ud2_6.exe
Suma de los 10 primeros naturales
El resultado de la suma es: 55.
```

Ejemplo 2

Hacer un programa en C que escriba la tabla de multiplicar de un número leído por teclado.

El programa es el siguiente.

```

1 #include <stdio.h>
2 int main(void)
3 {
4     int numero, i;
5     numero = 0;
6     i = 0;
7     printf("Tabla de multiplicar\n");
8     printf("Introduce el numero: ");
9     scanf("%d", &numero);
10    for(i = 1 ; i<11; i++)
11        printf("%d X %d = %d \n", numero,i,numero*i);
12    fflush(stdin);
13    getchar();
14    return 0;
15 }
```

Notas:

- La tabla de multiplicar de un numero empieza en 1 y acaba en 10, luego el valor inicial y el valor final del contador que controla la ejecución del bucle deben ser estos (línea 10), o bien tambien se puede poner `i<=10`.
- En la línea 11 se utiliza el especificador de formato para presentar en pantalla cada una de las multiplicaciones.

```
printf("%d X %d = %d \n", numero,i,numero*i);
```

Donde:

- El primer %d se refiere al numero que hemos introducido por teclado.
- El segundo %d corresponde a i, incrementador o contador que va sumando uno a uno y finaliza con 10.
- Por último, la multiplicación en si.

2.3.3. Sentencia do ... While

Esta sentencia funciona de modo análogo a **while**, con la diferencia de que la evaluación de **expresion_de_control** se realiza al final del bucle, después de haber ejecutado al menos una vez las sentencias entre llaves; éstas se vuelven a ejecutar mientras **expresion_de_control** sea **true**. La forma general de esta sentencia es:

```
do  
    sentencia;  
    while(expresion_de_control);
```

Donde **sentencia** puede ser una única sentencia o un bloque, y en la que debe observarse que *hay que poner ;) a continuación del paréntesis* que encierra a **expresion_de_control**, entre otros motivos para que esa línea se distinga de una sentencia **while** ordinaria.

Ejemplo

Se quiere escribir un programa que:

1. Pida por teclado un número (dato entero).
4. Pregunte al usuario si desea introducir otro o no.
5. Repita los pasos 1º y 2º, mientras que, el usuario no responda 'n' de (no).
6. Muestre por pantalla la suma de los números introducidos por el usuario.

En pantalla:

```
Introduzca un numero entero: 7  
¿Desea introducir otro (s/n)?: s  
Introduzca un numero entero: 16  
¿Desea introducir otro (s/n)?: s  
Introduzca un numero entero: -3  
¿Desea introducir otro (s/n)?: n  
La suma de los números introducidos es: 20
```

El código propuesto es:

```

1 #include <stdio.h>
2 int main()
3 {
4     char seguir;
5     int acumulador, numero;
6     /* En acumulador se va a guardar la suma de
7        los números introducidos por el usuario. */
8     acumulador = 0;
9     do
10    {
11        printf( "\n    Introduzca un numero: " );
12        scanf( "%d", &numero );
13        acumulador += numero;
14        printf( "\nDesea introducir otro numero (s/n)?:
15    " );
16        fflush( stdin );
17        scanf( "%c", &seguir );
18    } while ( seguir != 'n' );
19
20    /* Mientras que el usuario desee introducir
21 más números, el bucle iterará. */
22
23    printf( "\n    La suma de los numeros introducidos
24 es: %d", acumulador );
25    fflush( stdin );
26    getchar();
27
28    return 0;
29 }
```

2.3.4. Sentencias break, continue, goto

La instrucción ***break*** interrumpe la ejecución del bucle donde se ha incluido, haciendo al programa salir de él aunque la ***expresion_de_control*** correspondiente a ese bucle sea verdadera.

La sentencia ***continue*** hace que el programa comience el siguiente ciclo del bucle donde se halla, aunque no haya llegado al final de las sentencias compuesta o bloque.

La sentencia ***goto*** *etiqueta* hace saltar al programa a la sentencia donde se haya escrito la *etiqueta* correspondiente.

Por ejemplo:

```
sentencias ...
...
if (condicion)
    goto otro_lugar; // salto al lugar indicado por
la etiqueta
    sentencia_1;
    sentencia_2;
    ...
otro_lugar: // esta es la sentencia a la que se salta
    sentencia_3;
...
...
```

Obsérvese que la etiqueta termina con el carácter (:). La sentencia goto no es una sentencia muy prestigiada en el mundo de los programadores de C, pues disminuye la claridad y legibilidad del código. Fue introducida en el lenguaje por motivos de compatibilidad con antiguos hábitos de programación, y siempre puede ser sustituida por otras construcciones más claras y estructuradas.



RESUMEN

- En este capítulo se han visto las estructuras de decisión. La estructura de decisión simple (IF..) puede o no ejecutar, únicamente, una sentencia o un grupo de sentencias, la estructura de decisión doble (IF..ELSE) ejecuta uno de los dos grupos de sentencias y con la estructura de decisión múltiple (CASE) se pueden tomar decisiones entre una serie de opciones (normalmente más de dos) determinada por el valor de una variable ordinal.
- Hemos visto las estructuras de decisión anidadas, y a través de un ejemplo hemos llegado a la conclusión de que es más fácil el uso de la estructura de decisión múltiple puesto que facilita la codificación y la lectura del programa.
- En esta unidad didáctica también se han visto las estructuras iterativas: WHILE, y FOR.
- La estructura iterativa WHILE es un bucle de aplicación general con evaluación de la condición de control al principio.
- La estructura iterativa FOR, es un bucle de aplicación general con evaluación de la condición de control al principio. El bucle está controlado por un contador que se inicia en la primera iteración y luego se incrementa o se decrementa.
- Para todas las estructuras iterativas, se ejecuta el cuerpo del bucle si la condición de control del mismo es evaluada como verdadera, en caso contrario se finaliza la ejecución del mismo.
- Las estructuras iterativas hay que diseñarlas con precaución puesto que es posible que nunca terminen, en tal caso, el ordenador se quedará bloqueado ejecutando de forma continua las sentencias contenidas dentro del cuerpo del bucle.



estudios abiertos

SEAS

GRUPO SANVALERO



Introducción a la programación C

3. Tipos de datos estructurados en C

ÍNDICE

ÍNDICE	93
OBJETIVOS	95
INTRODUCCIÓN	96
3.1. Arrays unidimensionales o vectores	97
3.1.1. Definición de arrays	99
3.1.2. El número de elementos en un array	102
3.1.3. array multidimensionales	103
3.2. Declaración de cadenas	107
3.2.1. Función Strcpy()	108
3.2.2. Función Strlen()	109
3.2.3. Función Strcat	110
3.2.4. Funciones Strcmp y Strcomp	111
RESUMEN	113



OBJETIVOS

- Desarrollar programas que manejen el tipo de datos estructurados en los que el alumno aplicará todos los conocimientos adquiridos hasta ahora.
- Conocer qué son los *arrays* (unidimensionales y multidimensionales) y cuáles son las operaciones que pueden realizarse con ellos.
- Profundizar en un tipo de dato estructurado concreto, la *cadena*. Una cadena es una agrupación de caracteres, cada uno de los cuales ocupa una posición determinada. Se aprenderá a acceder a cada elemento de la cadena, tanto directamente como secuencialmente.
- Conocer las funciones y procedimientos estándar que C proporciona para el manejo de cadenas.
- Aprender a realizar ordenación de array búsquedas de elementos de un array.

INTRODUCCIÓN



Además de los tipos de datos fundamentales vistos anteriormente, en C existen algunos otros tipos de datos muy utilizados y que se pueden considerar derivados de los anteriores. En esta sección se van presentar las matrices.

Prestaremos especial interés a los recursos de los que dispone este lenguaje de programación en cuanto al manejo de cadenas.

Hay que tener presente que, hoy en día, hay una gran cantidad de programas que tienen que tratar con este tipo de datos, desde un procesador de textos, hasta el compilador de cualquier lenguaje de programación que tiene que pasar de un fichero de tipo texto a un archivo ejecutable.

Para ello lo primero que hemos de conocer es cómo almacena las matrices, cadenas en este lenguaje de programación así como la forma de definirlas.

3.1. Arrays unidimensionales o vectores



En programación, vectores o matrices (llamados en inglés arrays) es una zona de almacenamiento contiguo, que contiene una serie de elementos del **mismo tipo**. Desde el punto de vista lógico un vector se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos dimensiones).

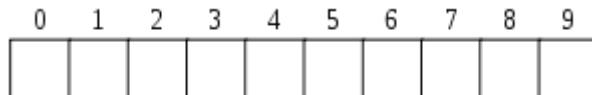


Figura 3.1. Matriz unidimensional con 10 elementos.

Un array unidimensional, es una colección ordenada de elementos de un mismo tipo de datos, agrupados de forma consecutiva. Cada elemento del array tiene asociado un índice, un número natural, que lo identifica inequívocamente y nos permite acceder a él.

Ejemplo de un array, que contiene las notas de 50 alumnos:

Calificaciones	
Calificaciones [1]	7.50
Calificaciones [2]	4.75
Calificaciones [2]	5.25
...	...
...	...
...	...
Calificaciones [50]	6

Nombre del vector: Calificaciones
 Subíndice: [1], [2], [3], ..., [50]
 Contenido: Calificaciones [1] = 7.50
 Calificaciones [2] = 4.75
 Calificaciones [3] = 5.25
 ...
 ...
 ...
 Calificaciones [50] = 6

Figura 3.2. Ejemplo calificaciones array de 50 posiciones.



Podemos encontrarnos los siguientes términos para definir lo mismo, estos son: vector, array, arreglo, matriz. Nosotros en este texto lo llamaremos array.

Se puede acceder a cada uno de los componentes de un array de forma individual a través del índice correspondiente.

Los arreglos se clasifican de acuerdo con el número de dimensiones que tienen. Así se tienen los:

- **Unidimensionales (vectores):** cómo podemos observar en la siguiente imagen podemos tener una sola columna o fila.

Elemento 1
Elemento 2
Elemento 3
.....
Elemento n

Figura 3.3. unidimensional.

- **Bidimensionales (tablas o matrices),** cuando tenemos unos datos ordenados en dos dimensiones, o filas y columnas.

Elemento 1,1	Elemento 1,n
Elemento 2,1	Elemento 2,n
Elemento 3,1	Elemento 3,n
.....
Elemento m,1	Elemento m,n

Figura 3.4. Array bidimensional.

- **Multidimensionales (tres o más dimensiones):** son pocos usados, pero tenemos la disposición de realizarlos.

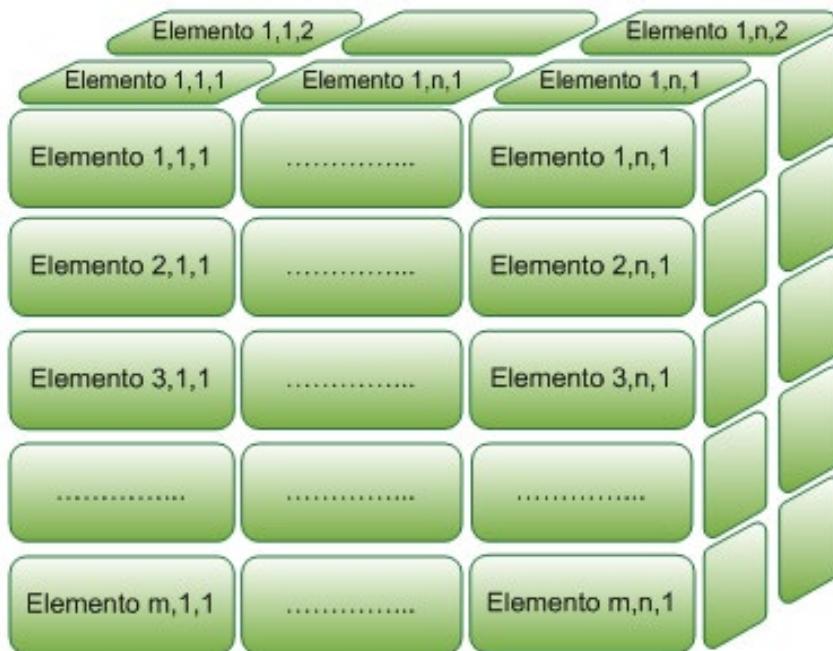


Figura 3.5. Array multidimensional.

Índice

Todo vector se compone de un determinado número de elementos. Cada elemento es referenciado por la posición que ocupa dentro del vector. Dichas posiciones son llamadas **índice** y siempre son correlativos. Existen tres formas de indexar los elementos de un array:

- **Indexación base-cero (0):** En este modo el primer elemento del vector será la componente cero ('0') del mismo, es decir, tendrá el índice '0'. En consecuencia, si el vector tiene 'n' componentes la última tendrá como índice el valor 'n-1'. El lenguaje C, java es un ejemplo típico que utiliza este modo de indexación.
- **Indexación base-uno (1):** En esta forma de indexación, el primer elemento de la matriz tiene el índice '1' y el último tiene el índice 'n'.
- **Indexación base-n (n):** Este es un modo versátil de indexación en la que el índice del primer elemento puede ser elegido libremente, en algunos lenguajes de programación se permite que los índices puedan ser negativos e incluso de cualquier tipo escalar (también cadenas de caracteres).

3.1.1. Definición de arrays

Como ya hemos comentado anteriormente, un array representa una estructura de datos, que permite almacenar una secuencia finita de valores del mismo tipo a los que se da un nombre común y a los que se accede a través de un índice. Este índice es la posición que ocupan en la secuencia.

Veamos como se definiría un array en lenguaje C:

```
int primeros_primos[10];
```

En este array se pueden distinguir tres partes:

- Int: Aquí se define el tipo de datos de los elementos del array.
- Primeros_primos: es el nombre de la variable de tipo de array de enteros.
- [10]: en esta parte se almacena el máximo número de elementos.

Es posible agregar los valores del array en el momento de definirlo. En este caso el array anterior que define el conjunto de los diez primeros números primos quedaría de la siguiente forma:

```
int primero_primos[10] = {1,2,3,5,7,9,11,13,17,19};
```

Si al definir un array no se indica el número de elementos que forman parte de él, éste toma como número de elementos los que se definan. Por ejemplo:

```
int mi_array[ ] = {1,3,5,7};
```

En este caso se define un array de 4 elementos.

También es posible inicializar el array de forma parcial, es decir, dando valores a una parte de los elementos. Pero siempre desde la posición 0 en adelante. No se puede dar valor a un elemento si no se han definido los anteriores. En el siguiente ejemplo se asignan valores a los tres primeros elementos de un array.

```
int mi_array[5] = {1,3,5};
```

Operar con todo el array solo es posible en la inicialización del mismo. En C no es posible operar con todo el array para:

- Asignar una tupla de valores directamente.
- Asignar un array a otro array.
- Comparar arrays directamente.

Veamos un ejemplo de programación empleando arrays.

El siguiente programa lee la temperatura al mediodía de cada día del mes y después informa de la temperatura media mensual, así como de los días más calurosos y más fríos.

```

#include "stdio.h"
int main(void)
{
    /*declaracion variables.*/
    int temp[31], i, min, max, avg, dias;

    printf("Cuántos días tiene 1 mes? ");
    scanf("%d", &dias);

    for(i=0;i<dias; i++)
    {
        printf("introduzca temperatura de cada dia %d: ", i+1);
        scanf("%d", &temp[i]);
    }

    /* Encontrar media */
    avg = 0;
    for(i=0; i<dias;i++)
        avg = avg + temp[i];
    printf("Temperatura media: %d\n", avg/dias);

    /*Encontrar min y max */
    min = 200; /* inicializar min y max*/
    max = 0;
    for(i=0; i<dias; i++)
    {
        if(min>temp[i]) min = temp[i];
        if(max<temp[i]) max = temp[i];
    }
    printf("temperatura minima; %d\n", min);
    printf("temperatura máxima; %d\n", max);
    /*función que realiza una pausa en el programa para
    ver el resultado*/
    fflush( stdin );
    getchar();
    return 0;
}

```



Recuerde siempre un array comienza por 0, por lo que el índice de referencia 1 siempre se aplica al segundo elemento.

C no realiza ninguna comprobación de los límites de un array, de forma que es posible sobrepasar de forma inadvertida (o premeditada) la longitud máxima de un array. Por supuesto intentar acceder a un elemento inexistente de un array tiene efectos desastrosos para la ejecución de un programa. Es responsabilidad del programador, asegurarse de estar siempre dentro de los límites de un array.

Si se desean copiar los elementos de un array en otro debe hacerse una copia individual de cada elemento. El siguiente programa crea un array a1 con los números del 1 al 10 y después lo copia en el array a2.

```
#include "stdio.h"
int main(void)
{
    int a1[10], a2[10];
    int i;

    for(i=1; i<11; i++) a1[i-1] = i;

    for(i=1;i<11; i++) a2[i-1] = a1[i-1];

    printf("Cadena 1: ");
    for(i=1;i<11;i++) printf("%d", a1[i-1]);
    printf("\nCadena 2: ");
    for(i=1; i<11; i++) printf("%d", a2[i-1]);
    getchar();
    return 0;
}
```

3.1.2. El número de elementos en un array

Cuando definimos un array solo indicamos el máximo numero de elementos que puede contener, pero en muchas ocasiones necesitaremos conocer el número actual de elementos que conforman el array. Como C carece de operadores específicos para trabajar con todo un array habrá que recurrir a trabajar elemento a elemento. Para conocer el número de elementos de un array se puede recurrir a dos opciones:

- Definir, a la vez que el array, una variable entera que almacene el número de elementos.



- Utilizar un valor que nunca se almacenaría en el array como indicador del último elemento.

La segunda opción es un poco peor, ya que hay que estar muy seguro de que el valor escogido no forma parte nunca de los valores posibles de array.

Por último citar que en todo lo explicado nos hemos basado en arrays unidimensionales. Para multidimensionales seguiríamos la misma pauta pero con varios índices.

3.1.3. array multidimensionales

Las estructuras de datos del tipo array pueden tener más de una dimensión, es bastante común el uso de arrays “planos” ó matriciales de dos dimensiones, por ejemplo:

```
int matriz[ número total de filas ] [ número
total de columnas ] ;
```

Si declaramos:

```
int matriz[3][4] ;
```

Esquemáticamente la disposición “espacial” de los elementos sería:

Columna	0	1	2	3
Fila				
0	matriz[0][0]	matriz[1][0]	matriz[2][0]	matriz[3][0]
1	matriz[1][0]	matriz[1][1]	matriz[1][2]	matriz[1][3]
2	matriz[2][0]	matriz[2][1]	matriz[2][2]	matriz[2][3]

Para inicializar arrays multidimensionales, se aplica una técnica muy similar a la ya vista, por ejemplo para dar valores iniciales a un array de caracteres de dos dimensiones, se escribirá:

```
char dia_de_la_semana[7][8] = { "lunes" ,
"martes" , "miercoles" , "jueves" , "viernes"
, "sábado" , "domingo" } ;
```

Donde el elemento [0][0] será la “l” de lunes , el [2][3] la “r” de miercoles , el [5][2] la “b” de sábado, etc.

Declaramos un array bidimensional de 3 filas y 5 columnas

```
int anArray [3] [5];
```

En este caso, ya que tienen 2 subíndices, esta es una matriz de dos dimensiones. En una matriz de dos dimensiones, es conveniente pensar en el subíndice primero como la fila, y el subíndice segunda como la columna.

Para acceder a los elementos de una matriz de dos dimensiones, sólo tiene que utilizar dos subíndices:

```
anArray [2] [3] = 7;
```

Para inicializar una matriz de dos dimensiones, es más fácil de utilizar llaves anidadas, con cada conjunto de números que representan una fila:

```
int anArray [3] [5] =
{
{1, 2, 3, 4, 5,} // fila 0
{6, 7, 8, 9, 10,} // fila 1
{11, 12, 13, 14, 15} // fila 2
};
```



Cuando el compilador C procesa esta lista, de hecho, desconoce por completo las llaves internas. Sin embargo, es muy recomendable usarlos todos modos para los propósitos de legibilidad.

Matrices bidimensionales con las listas de inicializadores puede omitir (sólo) la especificación de tamaño en primer lugar:

```
int anArray [] [5] =
{
{1, 2, 3, 4, 5,}
{6, 7, 8, 9, 10,}
{11, 12, 13, 14, 15}
};
```

El compilador puede hacer los cálculos para averiguar cuál es el tamaño de la matriz es. Sin embargo, el siguiente **no** está permitido:

```
int anArray [] []
{
{1, 2, 3, 4},
{5, 6, 7, 8}
};
```



Debido a que el paréntesis interno se ignora, el compilador no puede saber si tiene la intención de declarar una $\times 1$ 8, 2×4 , 4×2 , y 8×1 matriz en este caso.



Al igual que las matrices normales, siendo las matrices multidimensionales se pueden inicializar a 0 de la siguiente manera:

```
int anArray [3] [5] = {0};
```

Hay que tener en cuenta que esto sólo funciona si se declara explícitamente el tamaño de la matriz. De lo contrario, obtendrá una matriz de dos dimensiones con una fila.

Acceso a todos los elementos de una matriz de dos dimensiones requiere dos bucles: uno para la fila, y una para la columna. Desde matrices bidimensionales son típicamente visitada fila por fila, generalmente el índice de fila se utiliza como el bucle exterior.

Las matrices multidimensionales pueden ser mayores que dos dimensiones. Esta es una declaración de una matriz de tres dimensiones:

```
int anArray [5] [4] [3];
```

Tres arreglos bidimensionales son difíciles de iniciar en cualquier tipo de forma intuitiva el uso de listas de inicializador, por lo que es típicamente mejor para inicializar el array a 0 y se especifican los valores mediante bucles anidados.

Ejemplo práctico array multidimensional

Cálculo del determinante de una matriz 3x3.

Vamos a realizar un programa que calcula el determinante de un array bidimensional de 3x3, el programa deberá advertirlo con un mensaje. Recuerda que la fórmula para el cálculo del determinante es la siguiente:

$$\det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{11}a_{32}a_{23} - a_{21}a_{12}a_{33}$$

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 3

int main(void) {
    int matriz[SIZE][SIZE]={1,5,6,9,7,8,15,2,4};
    int determinante;
    int i, j;

    printf("\n");
    determinante = matriz[0][0]*matriz[1][1]*matriz[2][2];
    determinante += matriz[0][1]*matriz[1][2]*matriz[2][0];
    determinante += matriz[1][0]*matriz[2][1]*matriz[0][2];
    determinante -= matriz[0][2]*matriz[1][1]*matriz[2][0];
    determinante -= matriz[0][1]*matriz[1][0]*matriz[2][2];
    determinante -= matriz[0][0]*matriz[2][1]*matriz[1][2];
    printf("La matriz introducida es:\n");
    for (i=0; i<SIZE; i++) {
        for (j=0; j<SIZE; j++)
            printf(" %d ", matriz[i][j]);
        printf("\n");
    }
    printf("\nY su determinante es: %d \n", determinante);

    system("pause");
    return 0;
}
```

Comentario

En principio este programa no existe ninguna dificultad respecto al manejo de las matrices (ya se ha visto en programas anteriores). Lo único que es importante comentar es el algoritmo: tenemos siempre una matriz 3x3 y, por ello, podemos realizar las operaciones “a mano” (sin más complejidades en el algoritmo, es decir, sin introducir bucles o bifurcaciones).

Si quisieramos calcular el determinante de una matriz cualquiera, tendríamos que triangularizar la matriz y, una vez triangularizada, el determinante sería el producto de los elementos de la diagonal.

Es interesante observar como se ha dividido una expresión muy larga en seis expresiones más cortas, utilizando los operadores de suma y diferencia acumulativas ($+=$ y $-=$).

Luego se recorre para mostrar la matriz, con los datos introducidos en la declaración y los muestra por pantalla.

Y por último muestra el valor del determinante.



3.2. Declaración de cadenas

El uso mas común de un array unidimensional en C es la cadena. A diferencia de la mayoría de los lenguajes, C no tiene un tipo de datos de cadena. Pero C si soporta cadenas utilizando arrays unidimensionales de caracteres.

En C se define una cadena como un array de caracteres con un carácter de terminación nulo. **El carácter nulo es 0.** Esto implica que al definir el array tenemos que contar con el carácter nulo que finaliza la cadena, por lo que la longitud del array tendrá que tener un carácter más que la cadena más larga que deba contener.



Una cadena de caracteres no es sino un vector de tipo char, con alguna particularidad que conviene resaltar. Las cadenas suelen contener texto (nombres, frases, etc.), y éste se almacena en la parte inicial de la cadena (a partir de la posición cero del vector).

Para separar la parte que contiene texto de la parte no utilizada, se utiliza un *carácter fin de texto* que es el carácter nulo ('\0') según el código ASCII. Este carácter se introduce automáticamente al leer o inicializar las cadenas de caracteres, como en el siguiente ejemplo:

```
char ciudad[20] = "San Sebastián";
```

Donde a los 13 caracteres del nombre de esta ciudad se añade un decimocuarto: el '\0'. El resto del espacio reservado –hasta la posición `ciudad[19]`– no se utiliza. De modo análogo, una cadena constante tal como “`mar`” ocupa 4 bytes (para las 3 letras y el '\0').

La función `gets()` contenida en la librería `stdio.h` es la que se encarga en C de leer los caracteres de una cadena introducida por el teclado. Esta función lee caracteres desde el teclado hasta que se pulsa la tecla ENTER. La pulsación de esta tecla no se almacena sino que es reemplazada por un carácter nulo 0 para marcar el final de la cadena. El siguiente programa lee y escribe una cadena introducida por el teclado.

```
#include "stdio.h"
int main(void)
{
    char str[20];
    int i;
    printf("Introduzca una cadena (20 caracteres) :\n");
    gets(str);
    for(i=0; str[i]; i++) printf("%c", str[i]);
    getchar();
    return 0;
}
```

Observe que se llama a `gets()` pasando el nombre del array sin ningún índice. Esta función no realiza ningún control sobre los límites del array.

Al ejecutar este programa se muestra la cadena carácter a carácter. Veamos ahora como hacer que la cadena se muestre de una sola vez. Para ello emplearemos la función `printf` pasándole como parámetro el nombre del array que contiene la cadena.

```
#include "stdio.h"
int main(void)
{
    char str[20];
    int i;
    printf("Introduzca una cadena (20 caracteres) :\n");
    gets(str);
    printf(str);
    getchar();
    return 0;
}
```

Si se quisiera mostrar en una nueva línea se podría mostrar así.

```
printf("%s\n", str);
```

La biblioteca de C ofrece muchas funciones para manejar y modificar cadenas. Todas ellas se encuentran en el archivo de cabecera `string.h`. Veamos algunas de ellas.

3.2.1. Función Strcpy()

Esta función permite copiar una cadena o parte de ella en otra. El formato de la función es el siguiente:

`Strcpy(destino, origen)`

Copia los contenidos de origen en destino. Evidentemente los contenidos de desde no se modifican. Veamos el siguiente ejemplo:

```
char str[80];
strcpy(str, "prueba");
printf("%s", str);
```

Esta secuencia de comandos copia la cadena “prueba” en str y después la muestra en pantalla.



Esta función tampoco comprueba los límites de la cadena ni del array, por lo que hay que ser cuidadoso para que el array sea capaz de recibir la longitud de la cadena.

3.2.2. Función Strlen()

El nombre de esta función proviene de string length, y su misión es contar el número de caracteres de una cadena, sin incluir el ‘\0’ final. El paso del argumento se realiza por referencia, pues como argumento se emplea un puntero a la cadena (tal que el valor al que apunta es constante para la función; es decir, ésta no lo puede modificar), y devuelve un entero sin signo que es el número de caracteres de la cadena.

El prototipo de esta función es como sigue:

```
unsigned strlen(const char *s);
```

La palabra const impide que dentro de la función la cadena de caracteres que se pasa como argumento sea modificada.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
    system("CLS");
    char frase[100];
    int n;
    printf("Introduce una frase de menos de 100
letras.\n");
    printf("Para finalizar pulsa Intro:\n");
    scanf(" %[^\n]", frase);
    printf("\n");
    for (n=strlen(frase)-1; n>=0; n--)
        putchar(frase[n]);
    printf("\n");
    system("pause");
    return 0;
}
```



NOTA

En este código hemos añadido dos comandos que no han sido utilizados hasta ahora, de la función `system`:

(“`pause`”): Ejecuta un comando del sistema o un programa externo almacenado en disco. Esta función nos será muy útil para detener el programa antes de que termine.

(“`CLS`”): Para limpiar la pantalla sólo en sistemas de Windows.

Estas dos funciones funcionan con la librería `#include <stdlib.h>`

3.2.3. Función Strcat

El nombre de esta función proviene de *string concatenation*, y se emplea para unir dos cadenas de caracteres poniendo **s2** a continuación de **s1**. El valor de retorno es un puntero a **s1**. Los argumentos son los punteros a las dos cadenas que se desea unir. La función almacena la cadena completa en la primera de las cadenas.

Esta función no prevé si tiene sitio suficiente para almacenar las dos cadenas juntas en el espacio reservado para la primera. Esto es responsabilidad del programador.

El prototipo de esta función es como sigue:

```
char *strcat(char *s1, const char *s2);
```



3.2.4. Funciones Strcmp y Strncmp

El nombre de esta función proviene de *string comparison*. Sirve para comparar dos cadenas de caracteres. Como argumentos utiliza punteros a las cadenas que se van a comparar. La función devuelve cero si las cadenas son iguales, un valor menor que cero si **s1** es menor –en orden alfabético– que **s2**, y un valor mayor que cero si **s1** es mayor que **s2**. La función **strcmp()** es completamente análoga, con la diferencia de que no hace distinción entre letras mayúsculas y minúsculas).

El prototipo de la función **strcmp()** es como sigue:

```
int strcmp(const char *s1, const char *s2)
```

El siguiente programa...

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
    system("CLS");
    char **cadena, *aux;
    int i, j, n=3;
    char temp[20];
    printf("%s%s\n", "Ordena 3 nombres ",
"introducidos por teclado.");
    printf("Introduce los 3 nombres:\n");
    cadena = (char**)malloc(10*sizeof(char*));
    for (i=0; i<n; i++)
    {
        printf("Nombre %d: ", i+1);
        scanf(" %[^\n]", temp);
        cadena[i]=(char*)malloc((strlen(temp)+1)*sizeof
(char));
        strcpy(cadena[i], temp);
    }
    for (i=0; i<n-1; i++)
        for (j=i+1; j<n; j++)
            if ((strcmp(cadena[i], cadena[j]))>0)
            {
                aux=cadena[i];
                cadena[i]=cadena[j];
                cadena[j]=aux;
            }
    printf("La lista ordenada es:\n");
    for (i=0; i<n ; i++)
        printf("%s\n", cadena[i]);
    /* se libera la memoria reservada */
    for (i=0; i<n; i++)
        free(cadena[i]);
    free(cadena);
    system("pause");
}
```

RESUMEN

- Los array y estructuras nos permiten trabajar con tipos de datos estructurados.
- En particular, las cadenas son un tipo de datos que nos permiten tratar un conjunto de caracteres como grupo o individualmente.
- C proporciona funciones y procedimientos para operar con cadenas que facilitan el trabajo con este tipo de datos.



estudios abiertos

SEAS

GRUPO SANVALERO

4
UNIDAD
DIDÁCTICA

Introducción a la programación C

4. Diseño modular en C

ÍNDICE

ÍNDICE	117
OBJETIVOS	119
INTRODUCCIÓN	120
4.1. Tipos de datos derivados	121
4.1.1. Punteros	121
4.1.2. Operadores dirección e indirección (&, *)	122
4.1.3. Aritmética de punteros	123
4.1.4. Relación entre vectores y punteros	126
4.1.5. Relación entre matrices y punteros	126
4.2. Funciones en C	128
4.2.1. Declaración de una función	130
4.3. Paso de parámetros a las funciones	133
4.3.1. Paso de parametros por valor	133
4.3.2. Paso de parametros por referencia	134
4.3.3. Recursividad	135
4.4. Alcance de las variables: globales y locales	138
4.4.1. Variables locales	139
4.4.2. Variables globales	140
4.4.3. Variables estáticas	143
RESUMEN	147

OBJETIVOS

- Aprender características particulares de este lenguaje a la hora de definir subprogramas.
- Veremos los tipos de subprogramas que hay en C, los procedimientos y las funciones y cuál es objetivo de cada uno de ellos y cuando se usan.
- Se aprenderá a comunicar valores de parámetros a los subprogramas que se construyan.
- Entender cómo se pasan los parámetros a los subprogramas. Hay dos formas de pasar parámetros por valor y por referencia. Aprenderemos cuando hemos de usar cada uno de estos pasos de parámetro.
- Entender que, en general, el propósito de un procedimiento es modificar el estado de un programa.
- Aprender la diferencia entre alcance global y alcance local.

INTRODUCCIÓN



Un programa en C consiste de uno o más módulos, los cuales a su vez también pueden estar constituidos por otros módulos, y así sucesivamente. Estos módulos serán llamados, en términos generales, **SUBPROGRAMAS**, y en particular **PROCEDIMIENTOS** y **FUNCIONES**. Un subprograma es un bloque de programa que realiza una tarea determinada, y que al llamarlo o invocarlo puede necesitar que se le pasen **PARAMETROS**.

Los parámetros son identificadores que proveen un mecanismo para pasar información hacia los subprogramas invocados.

Los parámetros que se utilizan en la declaración de subprogramas se llaman **PARAMETROS FORMALES**.

Un subprograma puede ser invocado desde cualquier punto del programa, por medio de una instrucción **INVOCADORA**, la cual puede contener una lista de parámetros llamados **PARAMETROS ACTUALES** o **REALES**.

La modulación, es una técnica usada por los programadores para hacer sus códigos más legibles y cortos, ya que consiste en reducir un problema grande y complejo, en pequeños problemitas más sencillos, concentrándose en la solución por separado, de cada uno de ellos.



En C, se conocen como funciones, aquellos trozos de códigos utilizado para dividir un programa con el objetivos que cada bloque realice una tarea determinada.

4.1. Tipos de datos derivados

En lenguaje C no sólo existen los tipos de datos ya estudiados, también existen otros tipos de datos que se pueden considerar como derivados de los anteriores. El ejemplo más típico y más comúnmente utilizado son los punteros.

Los punteros son muy utilizados para la programación modular, por eso, llegados a este punto es bueno que conozcamos qué son y cómo se utilizan.

4.1.1. Punteros

Un puntero es una variable que representa la posición que ocupa un dato en memoria.

Para entender este concepto de puntero hemos de tener claro que cada dato almacenado en memoria ocupa una o varias celdas de memoria consecutivas. El tamaño normal de cada celda de memoria es un byte (8 bits). El número de celdas dependerá del tipo de dato. Así, un carácter suele ocupar una única celda de memoria; un dato de tipo entero ocupará dos celdas consecutivas; un número en punto flotante ocupará cuatro celdas consecutivas y una cantidad en doble precisión puede ocupar hasta ocho celdas.

El ordenador conserva una tabla de direcciones que relaciona cada variable definida en un programa con su dirección en memoria. De tal manera que podremos acceder a un dato bien por medio del nombre de la variable o bien por medio de la dirección que ocupa en memoria.

La mejor manera de entender cómo funcionan los punteros es mediante un ejemplo, por tanto vamos a suponer que var es una variable que representa un dato específico. En cuanto la variable se declara en el programa el compilador reserva espacio de memoria para almacenarla. De tal manera que:

var	1170
-----	------

Figura 4.1. La variable var ocupa la dirección de memoria 1170.

Por tanto tenemos que la variable var ocupa la dirección de memoria 1170. Al conocer la posición de memoria que ocupa la variable var, somos capaces de acceder directamente al dato mediante la expresión &var, donde & es un operador monario, llamado **operador dirección**, que proporciona la dirección de memoria que ocupa la variable a la que acompaña.

Llegados a este punto podemos asignar la dirección de memoria que ocupa la variable var a otra variable. Así:

```
pvar = &var
```

Ahora tenemos la variable pvar que se trata de un puntero a var, puesto que almacena la dirección de memoria que ocupa la variable var, pero no su valor.

D

DEFINICIÓN

Un puntero es una variable cuyo valor corresponde con la dirección de memoria que ocupa otra variable.

Ciertamente, si una variable ocupa una posición de memoria, se puede determinar que un puntero, que no es más que una variable que contiene la dirección de memoria que ocupa otra variable, ocupará a su vez su propia dirección. De esta afirmación podemos deducir que pueden existir punteros de punteros los cuales veremos más adelante.

El tamaño de los punteros ordinarios suele ser 4 bytes de memoria, y para poder declararlos debemos atender al tipo de dato al que apunta. Así pues la declaración de un puntero a una variable de tipo entero será:

```
int *var;
```

Por medio de esta declaración, la variable podrá contener la dirección que ocupa en memoria cualquier variable de tipo entero. La declaración de punteros a tipos de datos *long*, *char*, *float* y *doublé* es similar a la de los punteros a *int*.

4.1.2. Operadores dirección e indirección (&, *)

Como ya hemos visto, en lenguaje C se utiliza el operador dirección (*&*), para obtener la dirección que ocupa en memoria una variable a la que acompaña.

Volviendo a la figura 4.1, teníamos que var es una variable que ocupa la dirección de memoria 1170. Mediante el operador dirección podíamos decir que pvar (pvar =*&var*) tiene el valor de la dirección de memoria que ocupa la variable var.

Llegados a este punto hemos de incorporar el **operador indirección ***. Este operador nos ayudará a acceder al contenido de la variable var por medio de la dirección que ocupa en memoria. De tal manera que *pvar y var representan el mismo dato.

Así pues, si tenemos que pvar=&var y var2=*pvar, podemos decir que var y var2 representan el mismo valor, es decir el valor de var, siempre y cuando var y var2 sean del mismo tipo de dato.

EJEMPLO

Por ejemplo, supóngase las siguientes declaraciones y sentencias,

```
int i, j, *p; // p es un puntero a int
p = &i; // p contiene la dirección de i
*p = 10; // i toma el valor 10
p = &j; // p contiene ahora la dirección de j
*p = -2; // j toma el valor -2
```

Con respecto a los punteros hay que tener en cuenta que:

- Las constantes no se almacenan en una memoria accesible, por lo que el operador & no se puede utilizar con constantes.
- Las variables ocupan posiciones de memoria que no se pueden sustituir.
- Los valores posibles de un puntero son todas las direcciones de memoria posibles que una variable puede ocupar.
- Un puntero puede tener valor 0, ya que puede ser que apunte a NULL.
- El operador & sólo puede actuar sobre operandos con dirección única, es decir, sobre variables y elementos individuales de un array.



Las siguientes sentencias son ilegales:

```
p = &3.14; // las constantes no tienen dirección  
&i = p; // las direcciones no se pueden cambiar
```

Para visualizar por pantalla los datos de los punteros mediante la función printf() debemos de utilizar los modificadores %u y %p. Es inviable asignar un puntero a otro puntero con diferentes tipos de datos. Para realizar esa asignación deberemos realizar un casting. También existe la posibilidad de definir un puntero de tipo void que puede asignarse a cualquier puntero de cualquier tipo de dato.



Por ejemplo:

```
int *a;  
double *b;  
void *c;  
a = b; // ilegal  
a = (int *)b; // legal  
a = c = b; // legal
```

4.1.3. Aritmética de punteros

Ya sabemos que un puntero es una variable que almacena la dirección de memoria que ocupa una variable. También sabemos que un puntero debe de tener el mismo tipo de datos que la variable a la que apunta. Ahora vamos a descubrir las operaciones aritméticas que podemos implementar con punteros.

Cuando hemos estudiado los arrays, hemos visto que se puede sumar un valor entero a un nombre de array para acceder a una posición específica del array. En ese caso el valor se interpreta como el índice del array y representa la posición relativa del

elemento deseado con respecto a la primera posición del array. Esta misma operativa la podemos llevar a cabo con los punteros, de este modo a un puntero se le puede sumar o restar un valor entero de tal manera que por medio de esa operación accederemos a las posiciones contiguas a la posición de memoria que almacena la variable puntero. Así pues en lenguaje C podemos encontrar perfectamente las siguientes expresiones:

```
p = p + 1;  
p = p + i;  
p += 1;  
p++;
```

En el siguiente ejemplo se explica la aritmética de punteros.

```
void main(void) {  
    int r, s, t, *puntero1, *puntero2;  
    void *punterovoid;  
    puntero1 = &r; // Paso 1. La dirección de r es asignada a  
    puntero1  
    *puntero1 = 1; // Paso 2. puntero1 (r) es igual a 1.  
    Equivale a r = 1;  
    puntero2 = &s; // Paso 3. La dirección de s es asignada a  
    puntero2  
    *puntero2 = 2; // Paso 4. puntero2 (s) es igual a 2.  
    Equivale a s = 2;  
    puntero1 = puntero2; // Paso 5. El valor del puntero1 =  
    puntero2  
    *puntero1 = 0; // Paso 6. s = 0  
    puntero2 = &t; // Paso 7. La dirección de t es asignada a  
    puntero2  
    *puntero2 = 3; // Paso 8. t = 3  
    printf("%d %d %d\n", r, s, t); // Paso 9. ¿Qué se imprime?  
    punterovoid = &puntero1; // Paso 10. punterovoid contiene  
    la dirección de puntero1  
    *punterovoid = puntero2; // Paso 11. puntero1= puntero2;  
    *puntero1 = 1; // Paso 12. t = 1  
    printf("%d %d %d\n", r, s, t); // Paso 13. ¿Qué se  
    imprime?  
}
```

Vamos a suponer que las direcciones de memoria asignadas para las variables definidas en el ejemplo anterior son las siguientes:

Variable	Direccion de memoria
r	1170
s	1172
t	1174
puntero1	1176
puntero2	117A
punterovoid	117E

Figura 4.2. Tabla direcciones.

En la siguiente tabla se muestra los valores que van tomando las variables en cada paso de la ejecución del programa:

Paso	r 1170	s 1172	t 1174	puntero1 1176	puntero2 117A	punterovoid 117E
1				1170		
2	1			01170		
3	1			01170	01172	
4	1	2		01170	01172	
5	1	2		01172	01172	
6	1	0		01172	01172	
7	1	0		01172	01174	
8	1	0	3	01172	01174	
9	1	0	3	01172	01174	
10	1	0	3	01172	01174	01176
11	1	0	3	01174	01174	01176
12	1	0	1	01174	01174	01176
13	1	0	1	01174	01174	01176

Figura 4.3. Tabla de valores.

4.1.4. Relación entre vectores y punteros

Teniendo en cuenta que el nombre de un array es realmente un puntero al primer elemento de ese array, podemos deducir que existe una fuerte relación entre ambos elementos. Así, si `x` es un array, entonces la dirección al primer elemento del array se puede expresar tanto por `&x[0]` o simplemente por `x`. Si queremos acceder a la segunda posición del array podemos utilizar `&x[1]` o bien `(x+1)`, y así sucesivamente. Por tanto tenemos dos métodos diferentes para acceder a los diferentes elementos de un array, bien mediante `&array[i]` o bien mediante `array+i`. Ejemplo:

```
int vect[10]; // vect es un puntero a vect [0]
int *p;
...
p = &vect[0]; // p = vect;
...
```

`vect` es el nombre de un array de 10 posiciones de datos enteros. Atendiendo a la explicación anterior, podemos decir que `vect` es un puntero a la primera posición del array `vect[]`. Es decir, que `vect=&vect[0]`; `vect[1]=(vect+1)` y como consecuencia `vect[i]=(vect+i)`.

Por tanto hemos llegado a la conclusión de que un puntero apunta a la primera posición de un array, así pues si un array admite posiciones, un puntero también los debe de admitir. Así pues suponiendo el siguiente ejemplo:

```
int vect[4]={3,4,5,6}; // vect es un puntero a vect [0]
int *p;
p = vect;
```

Podemos decir que `p[0]=3; p[1]=4; p[2]=5` y `p[3]=6`.

Y siguiendo con el mismo ejemplo podemos determinar que:

```
*p equivale a vect[0], a *vect y a p[0]
*(p+1) equivale a vect[1], a *(vect+1) y a p[1]
*(p+2) equivale a vect[2], a *(vect+2) y a p[2]
```

4.1.5. Relación entre matrices y punteros

Al igual que una array puede ser representado como un puntero, es razonable pensar que los arrays multidimensionales o matriz puedan ser representados con una notación equivalente de punteros. Efectivamente, este es el caso. Así pues, una matriz es en realidad un conjunto de array unidimensionales, por tanto, podemos definir una matriz como un puntero a un grupo de arrays contiguos. De este modo podemos declarar una matriz como:

```
double (*x) [];
```

Que sería lo mismo que declararla como:

```
double x[] [];
```

Lo más fácil para establecer la relación entre punteros y matrices es fijarse en el siguiente ejemplo:

```
double x[5][3]; // declaración de una matriz de 5 x 3
double **a;      //declaración de un puntero a un puntero
double *b;       //declaración de un puntero
```

x es un puntero a la primera posición de un array de punteros **x[]**, así pues, podemos determinar que existe un array de punteros llamado igual que la matriz **x** y que contiene las direcciones de las primeras posiciones de cada fila de la matriz. También podemos determinar que **x** es un puntero a un puntero. Así pues, **x=&x[0]** y **x[0]=&x[0][0]**; **x[1]=&x[1][0]**; **x[2]=&x[2][0]**; y así sucesivamente.

Así pues siguiendo con el ejemplo anterior y teniendo en cuenta que ****a=x**, podemos determinar que:

```
*a es el valor de x[0] **a es x[0][0]
*(a+1) es el valor de x[1] **(a+1) es x[1][0]
*(*(a+1)+1) es x[1][1]
```

La siguiente tabla resume gráficamente la relación entre matrices y vectores de punteros.

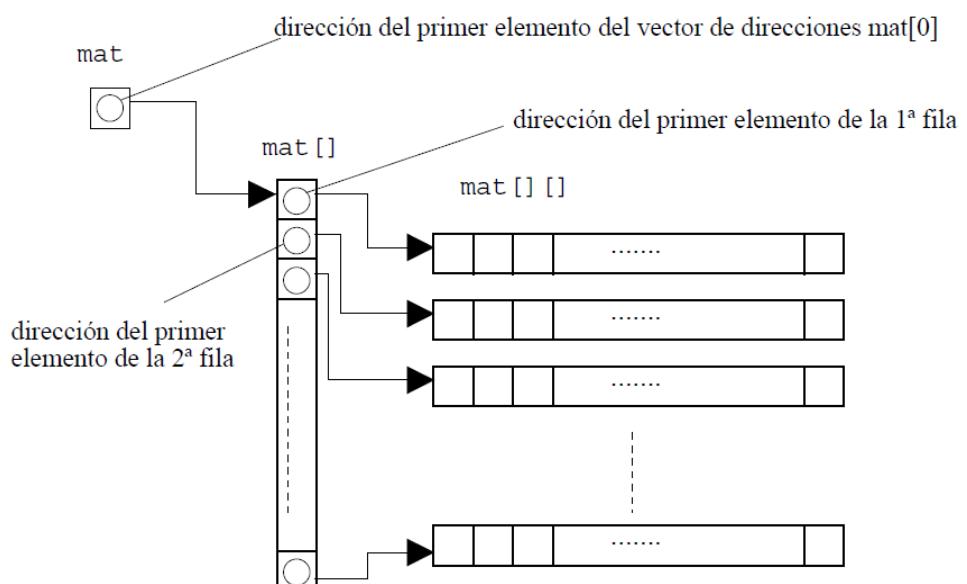


Figura 4.4. Esquema gráfico de la relación entre matrices y vectores de punteros.

4.2. Funciones en C

Las funciones en C son las piezas básicas que componen los programas.

Las funciones son fragmentos de código independiente con nombre y entidad propia, que se agrupan en archivos de programación que se compilan por separado y que luego se enlazan entre si para formar un programa completo.

A la hora de definir funciones en C se han de tener en cuenta los siguientes puntos:

- C se basa en el uso de funciones. No se puede escribir ninguna línea de código ejecutable que no pertenezca a una función (excluyendo las declaraciones y las definiciones).
- El lenguaje C no dispone de procedimientos, **solo permite el uso de funciones**. Para emular el uso de procedimientos se utilizan funciones que devuelven como valor de retorno un dato de tipo **void**.
- No se puede definir una función dentro de otra función. Todas las funciones deben estar en el mismo nivel.
- Siempre debe existir una función denominada main dentro del código del programa. Esta función será la que se ejecutará cuando se arranque el programa.

La sintaxis de una función en lenguaje C es:

```
tipo_de_retorno nombre_de_la_función (lista_de_parámetros)
{
    cuerpo_de_la_función
    return expresión
}
```

Donde:

- **tipo_de_retorno**: es el tipo del valor devuelto por la función, o, en caso de que la función no devuelva valor alguno, la palabra reservada void.
- **nombre_de_la_función**: es el nombre o identificador asignado a la función.
- **lista_de_parámetros**: es la lista de declaración de los parámetros que son pasados a la función. Éstos se separan por comas. Debemos tener en cuenta que pueden existir funciones que no utilicen parámetros.
- **Cuerpo_de_la_función**: está compuesto por un conjunto de sentencias que llevan a cabo la tarea específica para la cual ha sido creada la función.
- **return expresión**: mediante la palabra reservada return, se devuelve el valor de la función, en este caso representado por expresión.



Una función puede no tener ningún parámetro o tener tantos como se desee o se requiera. Pero el tipo de retorno es imprescindible, incluso si no se desea que devuelva nada es necesario emplear el tipo void.

Una función termina cuando se ejecutan todas sus sentencias o bien cuando se ejecuta una sentencia return. La sintaxis de esta sentencia se muestra a continuación.

```
return (expresión);
```

La expresión debe mostrar siempre un valor en el mismo tipo de datos que devuelven la función.



El valor de retorno es un valor único: no puede ser un vector o una matriz, aunque sí un puntero a un vector o a una matriz. Sin embargo, el valor de retorno sí puede ser una estructura, que a su vez puede contener vectores y matrices como elementos miembros.

Para ejecutar una función es necesario realizar una llamada a dicha función desde el código de otra función (excepto la función main que es llamada por el sistema cada vez que se ejecuta el programa). La sintaxis para llamar a una función es:

```
NombreFuncion (ExpresionParam1....ExpresionParamN)
```

El identificador debe coincidir con el nombre de la función que se quiere llamar. Las expresiones deben ser cada una de ellas del mismo tipo que el parámetro formal que ocupa su misma posición en la definición de la función.

En el momento en el que se evalúe la llamada a dicha función se procederá a ejecutar su código. El primer paso consistirá en evaluar cada una de las expresiones que forman los parámetros de las llamadas. Dichos parámetros se conocerán como parámetros reales, ya que son el valor real de dichos parámetros durante la ejecución de la función. El valor resultante de estas funciones será copiado en las variables de la función que actúan como parámetros formales.

A continuación se ejecuta el cuerpo de la función. Este código se ejecutara hasta que la función termine o se ejecute una sentencia return,

Después el control volverá a la función que realizó dicha llamada. Cuando esto ocurre, la llamada a la función será sustituida por el valor expresado en la sentencia return (si no se ha ejecutado una sentencia return el valor sustituido será aleatorio) y se continuara ejecutando la función actual.

Utilidad de las funciones

Parte esencial del correcto diseño de un programa de ordenador es su modularidad, esto es su división en partes más pequeñas de finalidad muy concreta, como ya hemos comentado anteriormente, es decir las funciones.

Las funciones facilitan el desarrollo y mantenimiento de los programas, evitan errores, y ahorran memoria y trabajo innecesario. Una misma función puede ser utilizada por diferentes programas, y por tanto no es necesario reescribirla. Además, una función es una parte de código independiente del programa principal y de otras funciones, manteniendo una gran independencia entre las variables respectivas, y evitando errores y otros efectos colaterales de las modificaciones que se introduzcan.

Mediante el uso de funciones se consigue un código limpio, claro y elegante. La adecuada división de un programa en funciones constituye un aspecto fundamental en el desarrollo de programas de cualquier tipo. Las funciones, ya compiladas, pueden guardarse en librerías. Las librerías son conjuntos de funciones compiladas, normalmente con una finalidad análoga o relacionada, que se guardan bajo un determinado nombre listas para ser utilizadas por cualquier usuario.

4.2.1. Declaración de una función

Para emplear una función en lenguaje C es suficiente con definirla para que pueda ser invocada desde cualquier punto del programa. Para poder emplear una función el compilador necesita conocer el tipo que devuelve la función y los parámetros que acepta, de forma que pueda comprobar que los tipos de parámetros reales coinciden con los tipos de los parámetros formales. Cuándo una función llama a otra definida anteriormente, el compilador ya lo sabe, pero ¿qué ocurre cuando se llama a una función que no ha sido definida previamente? El compilador asumirá que la función devuelve int y que todos los parámetros son de tipo int, lo cual puede que no se corresponda con la definición de la llamada.

Para que el compilador pueda conocer el tipo y los parámetros de las funciones antes de su uso se utilizan los prototipos de las funciones. El formato de un prototipo de función es el siguiente:

```
TipoRetorno NombreFuncion (TipoParam1 NombreParam1);
```

En un prototipo se indica el nombre de la función, su tipo y los parámetros que acepta, finalizando con un punto y coma. Un prototipo lo que hace es declarar una función, es decir indicar al compilador que existe una función con un nombre, tipo y parámetro concretos. La definición lo que incluye es el código de la función.

Normalmente cuando se codifica un módulo se incluyen al principio del módulo todos los prototipos de las funciones que tiene. De esta forma se podrá llamar a todas las funciones del módulo sin preocuparse de donde están situadas.



Escribir la declaración de una función que calcule la potencia de a^b
 $\text{int potencia (int } a, \text{ int } b);$

La función indica que la función deseada se llama `potencia`. Recoge dos parámetros de entrada (`a` y `b`) de tipo entero y devuelve un valor de retorno de tipo entero.

Vamos a suponer que queremos crear un programa para calcular el precio de un producto basándose en el precio base del mismo y el impuesto aplicable.

A continuación mostramos el código fuente de dicho programa:

```
#include <stdio.h>

float precio(float base, float impuesto); /* declaración */

main()
{
    float importe = 2.5;
    float tasa = 0.07;

    printf("El precio a pagar es: %.2f\n", precio(importe, tasa));
    return 0;
}

float precio(float base, float impuesto) /* definición */
{
    float calculo;
    calculo = base + (base * impuesto);
    return calculo;
}
```

Figura 4.5. Ejemplo función.

El ejemplo anterior se compone de dos funciones, la función requerida `main` y la función creada por el usuario `precio`, que calcula el precio de un producto tomando como parámetros su precio base y el impuesto aplicable.

La función `precio` calcula el *precio* de un producto sumándole el impuesto correspondiente al *precio base* y devuelve el valor calculado mediante la sentencia `return`.

Por otra parte, en la función *main* declaramos dos variables de tipo *float* que contienen el precio base del producto y el impuesto aplicable. La siguiente sentencia dentro de la función *main* es la llamada a la función de biblioteca *printf*, que recibe como parámetro una llamada a la función *precio*, que devuelve un valor de tipo *float*. De esta manera, la función *printf* imprime por la salida estándar el valor devuelto por la función *precio*.

Es importante tener en cuenta que las variables importe y tasa (argumentos) dentro de la función *main* tienen una correspondencia con las variables base e impuesto (parámetros) dentro de la función *precio* respectivamente.

En el ejemplo anterior, justo antes de la función *main*, hemos declarado la función *precio*. La intención es que la función *main* sea capaz de reconocerla. Sin embargo, la definición de dicha función aparece después de la función *main*. Las definiciones de función pueden aparecer en cualquier orden dentro de uno o más ficheros fuentes. Más adelante, en esta unidad, veremos en detalle la declaración y definición de funciones. Por otra parte, hemos añadido la sentencia *return 0* al final de la función *main*, puesto que se trata de una función como otra cualquiera y puede devolver un valor a quien le ha llamado, en este caso el entorno en el que se ejecuta el programa. Generalmente, el valor 0 implica un fin de ejecución normal, mientras que otro valor diferente implica un final de ejecución inusual o erróneo.

4.3. Paso de parámetros a las funciones

Para que una función sea útil debe ser capaz de recoger los datos de entrada y obtener de ellos los resultados pertinentes, que serán devueltos como datos de salida. En una función en C la única forma de comunicar una función con el código que la ejecuta es mediante los parámetros que recibe y el valor de retorno.

Mientras que el valor de retorno siempre será considerado como un resultado o dato de salida, los parámetros de las funciones pueden ser utilizados a la vez como datos de entrada o de salida. Existen dos métodos para realizar el paso de parámetros a una función: por valor o por referencia.

4.3.1. Paso de parámetros por valor

Es el método utilizado por defecto en C y el único que esta soportado directamente por el lenguaje. Consiste en copiar el valor del parámetro real en el parámetro formal, es decir, utilizar el valor de la expresión de la llamada como valor de inicio de la variable local que actúa de parámetro formal. La función puede trabajar con el parámetro formal y modificarlo sin que estos cambios se vean reflejados en el código que realizó la llamada. De esta forma, los parámetros reales no sufrirán cambios cuando la función finalice.

El paso de parámetros por valor solo permite utilizar dichos parámetros como datos de entrada, siendo imposible utilizarlos para devolver resultados de salida.

Lo veremos mucho mejor con un ejemplo:

```
#include "stdio.h"
int funcion(int n, int m) {
    n = n + 2;
    m = m - 5;
    return n+m;
}
int main(void)
{
    int a, b;
    a = 10;
    b = 20;
    printf ("\n a,b -> %d, %d",a,b);
    printf ("\n funcion(a,b) -> %d",funcion(a,b));
    printf ("\n a,b: %d, %d",a,b);
    printf ("\n funcion(a,b) -> %d",funcion(10,20));
    getchar();
    return 0;
}
```

Bien, ¿qué es lo que pasa en este ejemplo?

Empezamos haciendo `a = 10` y `b = 20`, después llamamos a la función “`funcion`” con las objetos `a` y `b` como parámetros. Dentro de “`funcion`” esos parámetros se llaman `n` y `m`, y sus valores son modificados. Sin embargo al retornar a `main`, `a` y `b` conservan sus valores originales. ¿Por qué?

La respuesta es que lo que pasamos no son los objetos `a` y `b`, sino que copiamos sus valores a los objetos `n` y `m`.

Piensa, por ejemplo, en lo que pasa cuando llamamos a la función con parámetros constantes, es lo que pasa en la segunda llamada a “`funcion`”. Los valores de los parámetros no pueden cambiar al retornar de “`funcion`”, ya que esos valores son constantes.

Si los parámetros por valor no funcionasen así, no sería posible llamar a una función con valores constantes o literales.

4.3.2. Paso de parámetros por referencia

El paso de datos por referencia se caracteriza por no transportar el dato correspondiente de forma directa: lo que pasa en la función es una referencia a la dirección de memoria donde se almacena dicho dato. De esta forma, la función utiliza dicha referencia para modificar el dato real, en lugar de trabajar sobre una copia de dato.

En C no existe esta característica por lo que es necesario pasar la referencia de forma explícita mediante punteros.

Ejemplo: Escribir una función que incremente una variable que se le pase como parámetro.

```
#include <stdio.h>
void incrementar (int *a);
int main(void)
{
    int var = 1;
    printf("valor de var antes: %d\n", var);
    incrementar (&var);
    printf("valor de var despues: %d\n", var);
    getchar();
    return 0;
}
void incrementar (int *a)
{
    (*a) = (*a) + 1;
}
```

Observemos este ejemplo:

- Declaramos la variable var de tipo int en la función main.
- El parámetro real de la llamada es la dirección de var (`&var`).
- El parámetro formal declarado en la función es un puntero a int (`int *a`)
- Dentro de la función se accede al dato usando el operador de indirección (`*a`).

4.3.3. Recursividad

Una vez que se conoce la forma de realiza llamadas a funciones una cuestión que queda en el aire es que sucede si una función realiza una llamada a si misma. En C lo que sucede es que la función se vuelve a ejecutar. Dado su especial relevancia y sus características, este tipo de llamada se denomina llamadas recursivas y las funciones que realizan este tipo de llamadas se denominan funciones recursivas.

Todas las funciones recursivas deben distinguir entre al menos dos ramas de ejecución.

- **La rama de salida:** en ella se encuentran el final de la función recursiva, ocurre cuando los valores de los parámetros alcanzan la condición de salida.
- **La rama recursiva:** esta rama es la que realiza la llamada recursiva. Cada vez que se realice una llamada recursiva habrá que modificar los parámetros que se le pasen de forma que se aproximen cada vez más a la condición de salida.

Si los parámetros modificados no se aproximan hacia la condición de salida o la condición de salida o la condición de salida impide que la progresión de los parámetros se ajuste a ella, en esos casos las llamadas recursivas no tendrán fin. Por tanto, es muy importante asegurarse de que para cualquier valor inicial la progresión de parámetros que se genera pueda ajustarse en algún momento a la condición de salida.



No todas las funciones pueden llamarse a sí mismas, sino que deben estar diseñadas especialmente para que sean recursivas, de otro modo podrían conducir a bucles infinitos, o a que el programa termine inadecuadamente. Tampoco todos los lenguajes de programación permiten usar recursividad.

Ejemplo factorial

Podríamos crear una función recursiva para calcular el factorial de un número entero.

El factorial se simboliza como $n!$, se lee como “n factorial”, y la definición es:

$$n! = n * (n-1) * (n-2) * \dots * 1$$

Hay algunas limitaciones:

- No es posible calcular el factorial de números negativos, no está definido.
- El factorial de cero es 1.

De modo que una función bien hecha para cálculo de factoriales debería incluir un control para esos casos:

```
/* Función recursiva para cálculo de factoriales */
int factorial(int n) {
    if(n < 0) return 0;
    else if(n > 1) return n*factorial(n-1);
    return 1; /* Condición de terminación, n == 1 */
}
```

Veamos paso a paso, lo que pasa cuando se ejecuta esta función, por ejemplo: factorial (4):

1^a Instancia

- n=4
- n > 1
- salida \leftarrow 4 * factorial(3) (Guarda el valor de n = 4)

2^a Instancia

- n > 1
- salida \leftarrow 3*factorial(2) (Guarda el valor de n = 3)

3^a Instancia

- n > 1
- salida \leftarrow 2*factorial(1) (Guarda el valor de n = 2)

4^a Instancia

- n == 1 \rightarrow retorna 1

3^a Instancia

- (recupera n=2 de la pila) retorna 1*2=2

2^a instancia

- (recupera n=3 de la pila) retorna 2*3=6

1^a instancia

- (recupera n=4 de la pila) retorna $6*4=24$
- Valor de retorno → 24



Aunque la función factorial es un buen ejemplo para demostrar cómo funciona una función recursiva, la recursividad no es un buen modo de resolver esta función, que sería más sencilla y rápida con un simple bucle `for`.

La recursividad consume muchos recursos de memoria y tiempo de ejecución, y se debe aplicar a funciones que realmente le saquen partido.

4.4. Alcance de las variables: globales y locales

Como ya se comentó en la anterior unidad didáctica, dentro de los subprogramas hay que evitar, siempre que sea posible el empleo de variables globales. La comunicación entre subprogramas se realizará mediante el paso de parámetros, y si dentro del subprograma se necesitan otras variables, se declararán dentro del subprograma y estas serán locales al subprograma, es decir que no podrán ser utilizadas en otros módulos.

Además, si se usan variables globales dentro de un módulo esto hará que no se pueda reutilizar código puesto que habría que declarar las mismas variables globales en el subprograma donde se va a reutilizar el código.

Las variables globales se declaran al principio del programa. Por el contrario, las variables locales se declaran dentro del procedimiento o función donde se van a utilizar. Los parámetros formales también son locales a los procedimientos o funciones en las que se declaran. La declaración y utilización de variables puede verse en la siguiente figura.

Antes de pasar a ver algún ejemplo en el que intervienen variables globales y variables locales hay que hacerse la siguiente pregunta ¿Qué pasa si hay variables locales con el mismo nombre que unas variables globales?

En general, las variables locales, son celdas de memoria distintas de las variables globales, así que en el subprograma en el cual se hayan definido variables locales con el mismo nombre que las variables globales, dentro del subprograma nos estaremos refiriendo a la variables local y no podemos tener acceso a la variable global puesto que hay una con el mismo nombre.

En C se definen dos tipos de almacenamiento:

- **Automático:** se corresponden a las variables locales que se definen dentro de una función.
- **Estático:** variables locales o globales cuyo valor persiste entre las distintas de un objeto. Si son variables locales se distinguen de las automáticas por utilizar la palabra reservada static.

Las variables automáticas se crean cuando se llama a la función u desaparecen cuando finaliza su ejecución, por lo tanto no conservan su valor de una llamada a otra. Las variables automáticas pueden ser inicializadas con un valor constante cuando son declaradas. Esta inicialización tendrá lugar cada vez que se llame a la función que contenga dicha variable.

Las variables estáticas proporcionan almacenamiento permanente para la variable, de forma que no se destruye cuando finaliza la ejecución de la función en la que se define. Además, su valor perdura de una ejecución a otra. Las variables estáticas pueden ser inicializadas con un valor constante cuando son declaradas, pero esa inicialización solo tiene lugar la primera vez que se llame al objeto y se usa la variable.

Vamos con varios ejemplos a aclarar todo esto:

4.4.1. Variables locales

Como ya hemos comentado, cuando declaramos variables dentro de la función principal del programa, es decir, dentro de la función *main*, están únicamente asociadas a esta función, al igual que sucede con las variables declaradas dentro de la función *main*, cualquier variable que declaremos dentro de una función, es local a esa función, es decir, su ámbito está confinado a dicha función.



Esta situación permite que existan variables con el mismo nombre en diferentes funciones y que no mantengan ninguna relación entre sí.

Debemos tener en cuenta que cualquier variable declarada dentro de una función se considera como una **variable automática** (*auto*) a menos que utilicemos algún modificador de tipo.

Una variable se considera **automática** porque cuando se accede a la función se le asigna espacio en la memoria automáticamente y se libera dicho espacio tan pronto se sale de la función.



En otras palabras, una variable automática no conserva un valor entre dos llamadas sucesivas a la misma función. Con el propósito de garantizar el contenido de las variables automáticas, éstas deben inicializarse al entrar a la función para evitar que su valor sea indeterminado.

Todas las variables que hemos utilizado en los ejemplos vistos hasta ahora son variables automáticas. La utilización de la palabra reservada *auto* es opcional, aunque normalmente no se utiliza, por ejemplo:

`auto int contador;`

equivale a

`int contador;`

Utilización del mismo identificador de variable en diferentes funciones mostrando su localidad.

```
#include <stdio.h>

void imprimeValor();
main()
{
    int contador = 0;
    contador++;
    printf("El valor de contador es: %d\n", contador);
    imprimeValor();
    printf("Ahora el valor de contador es: %d\n", contador);
    return 0;
}

void imprimeValor()
{
    int contador = 5;
    printf("El valor de contador es: %d\n", contador);
}
```

La salida es:

```
El valor de contador es: 1
El valor de contador es: 5
Ahora el valor de contador es: 1
```

4.4.2. Variables globales

A diferencia de las variables locales cuyo ámbito estaba confinado a la función donde estaban declaradas, el ámbito de las variables globales se extiende desde el punto en el que se definen hasta el final del programa.

En otras palabras, si definimos una variable al principio del programa, cualquier función que forme parte de éste podrá utilizarla simplemente haciendo uso de su nombre.

La utilización de variables globales proporciona un mecanismo de intercambio de información entre funciones sin necesidad de utilizar argumentos.

Por otra parte, las variables globales mantienen el valor que se les ha asignado dentro de su ámbito, incluso después de finalizar las funciones que modifican dicho valor.



Debemos tener en cuenta que el uso de variables globales para el intercambio de informaciones entre funciones puede resultar útil en algunas situaciones (como cuando se desea transferir más de un valor desde una función), pero su utilización podría llevarnos a programas de difícil interpretación y complejos de depurar.

Cuando trabajamos con variables globales debemos distinguir entre la definición de una variable global y su declaración. Cuando definimos una variable global, lo hacemos de la misma forma en que se declara una variable ordinaria.

La definición de una variable global se realiza fuera de cualquier función. Además, las definiciones de variables globales suelen aparecer antes de cualquier función que desee utilizar dicha variable. La razón es que una variable global se identifica por la localización de su definición dentro del programa. Cuando se define una variable global automáticamente, se reserva memoria para el almacenamiento de ésta. Además, podemos asignarle un valor inicial a la misma.

Si una función desea utilizar una variable global previamente definida, basta con utilizar su nombre sin realizar ningún tipo de declaración especial dentro de la función. Sin embargo, si la definición de la función aparece antes de la definición de la variable global, se requiere incluir una declaración de la variable global dentro de la función.

Para declarar una variable global se utiliza la palabra reservada *extern*. Al utilizar *extern*, le estamos diciendo al compilador que el espacio de memoria de esa variable está definido en otro lugar. Es más, en la declaración de una variable externa (*extern*) no se puede incluir la asignación de un valor a dicha variable. Por otro lado, el nombre y el tipo de dato utilizados en la declaración de una variable global debe coincidir con el nombre y el tipo de dato de la variable global definida fuera de la función.

Debemos recordar que sólo se puede inicializar una variable global en su definición. El valor inicial que se le asigne a la variable global debe expresarse como una constante y no como una expresión. Es importante señalar que si no se asigna un valor inicial a la variable global, automáticamente se le asigna el valor cero (0). De esta manera, las variables globales siempre cuentan con un valor inicial.

Cabe señalar que la declaración de una variable global puede hacer referencia a una variable que se encuentra definida en otro fichero. Por esta razón, podemos decir que el especificador de tipo *extern* hace referencia a una variable que ha sido definida en un lugar distinto al punto en el que se está declarando y donde se va a utilizar.

En aplicaciones grandes compuestas de varios ficheros, es común que las definiciones de variables globales estén agrupadas y separadas del resto de ficheros fuente. Cuando se desea utilizar cualquiera de las variables globales en un fichero fuente, se debe incluir el fichero en el que están definidas las variables mediante la directiva de precompilación *#include*.

Utilización de variables globales como mecanismo de intercambio de información entre funciones.

```
#include <stdio.h>

void unaFuncion();
void otraFuncion();
int variable;

main()
{
    variable = 9;
    printf("El valor de variable es: %d\n", variable);
    unaFuncion();
    otraFuncion();
    printf("Ahora el valor de variable es: %d\n", variable);
    return 0;
}

void unaFuncion()
{
    printf("En la función unaFuncion, variable es: %d\n", variable);
}

void otraFuncion()
{
    variable++;
    printf("En la función otraFuncion, variable es: %d\n", variable);
}
```

La salida es:

```
El valor de variable es: 9
En la función unaFuncion, variable es: 9
En la función otraFuncion, variable es: 10
Ahora el valor de variable es: 10
```

```

#include <stdio.h>

void unaFuncion();
void otraFuncion();


main()
{
    extern variable;
    variable = 9;
    printf("El valor de variable es: %d\n", variable);
    unaFuncion();
    printf("Ahora el valor de variable es: %d\n", variable);
    return 0;

}

void unaFuncion()
{
    extern variable;
    printf("En la función unaFunción, variable es: %d\n", variable);
}

int variable;

```

Su salida es:

```

El valor de variable es: 9
En la función unaFunción, variable es: 9
Ahora el valor de variable es: 9

```

4.4.3. Variables estáticas

Otro tipo de almacenamiento son las variables estáticas identificadas por la palabra reservada *static*. Las variables estáticas pueden ser tanto locales como globales. Una variable estática local, al igual que una variable automática, está únicamente asociada a la función en la que se declara con la salvedad de que su existencia es permanente.

En otras palabras, su contenido no se borra al finalizar la función, sino que mantiene su valor hasta el final del programa. Por ejemplo, en el siguiente programa declaramos la variable contador como estática dentro de la función *imprimeValor* y desde la función *main* llamamos a esta función varias veces:

```
#include <stdio.h>
void imprimeValor();
main()
{
    imprimeValor();
    imprimeValor();
    imprimeValor();
    imprimeValor();
    return 0;
}
void imprimeValor()
{
    static int contador = 0;
    printf("El valor de contador es: %d\n", contador);
    contador++;
}
```

La primera vez que se llama a la función `imprimeValor` se inicializa su valor a cero, y tras imprimir su valor se incrementa éste. En las sucesivas llamadas, el valor de la variable `contador` se mantiene y el resultado es el siguiente:

- El valor de contador es: 0
- El valor de contador es: 1
- El valor de contador es: 2
- El valor de contador es: 3

Como hemos visto, el valor de la variable `contador` se mantiene de una llamada a otra de la función `imprimeValor`. Esto quiere decir que las variables locales estáticas proporcionan un medio privado de almacenamiento permanente en una función.

Por otro lado, la aplicación del calificador `static` a variables globales hace que a éstas sólo se pueda acceder desde el fichero fuente en el que se definieron y no desde ningún otro fichero fuente.

Por ejemplo, si definimos las variables globales `tiempo` y `reloj` en el siguiente fichero fuente:

```

static int tiempo;
static int reloj;

main()
{
    ...
}

void funcion1()
{
    ...
}

```

Cualquier otra función que forme parte de la aplicación y que no forme parte de este fichero fuente no podrá disponer de acceso a tiempo ni a reloj. Es más, se pueden utilizar los mismos nombres para definir variables en funciones en otros ficheros sin ningún tipo de problema.

Al igual que las variables globales, las funciones son objetos externos cuyos nombres, generalmente, se desea que se conozcan de manera global. Sin embargo, en algunas situaciones resulta deseable limitar dicho acceso al fichero en el que se declara la función. En estos casos, se utiliza la palabra reservada static y su mecanismo de aplicación es similar al caso de las variables, como podemos apreciar en el siguiente ejemplo:

```

static int cuadrado (int numero)
{
    int calculo=0;
    calculo = numero * numero;
    return calculo;
}

```

Limitar el acceso tanto a variables globales como a funciones, mediante su declaración como estáticas, resulta útil en algunas situaciones en las que se quiere evitar que entren en conflicto con otras variables o funciones, incluso inadvertidamente.



RESUMEN

- La modulación, es una técnica usada por los programadores para hacer sus códigos más legibles y cortos, ya que consiste en reducir un problema grande y complejo, en pequeños problemitas más sencillos, concentrándose en la solución por separado, de cada uno de ellos.
- En c, se conocen como funciones, aquellos trozos de códigos utilizado para dividir un programa con el objetivos que cada bloque realice una tarea determinada.
- Las funciones se utilizan cuando el propósito del programa es calcular un único valor, en otro caso es preferible utilizar un procedimiento, es decir, en las siguientes situaciones:
 - Cuando no se devuelve ningún valor.
 - Cuando se devuelve más de un valor.
 - Cuando se efectúa una E/S además de devolver un único valor.
- El paso de parámetros por valor se efectúa copiando en los parámetros formales asociados a los parámetros reales el valor del parámetro real. El paso de parametro por referencia se efectua pasando la dirección de la variable, con lo que el parámetro real queda modificado al acabar la ejecución de la función.
- Cuando declaramos variables dentro de la función principal del programa, es decir, dentro de la función main, están únicamente asociadas a esta función, en otras palabras, son variables locales de la función main y no se puede acceder a ellas a través de ninguna otra función.
- Al igual que sucede con las variables declaradas dentro de la función main, cualquier variable que declaremos dentro de una función, es local a esa función, es decir, su ámbito esta confinado a dicha función.

