Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Algoritmos y Programación

PROGRAMACIÓN BÁSICA EN C

Caracas, Noviembre de 2002

Contenido

1. Introducción

Este material ha sido preparado para servir como apoyo al curso de Algoritmos y Programación de la Licenciatura en Computación y como una presentación de los recursos y las posibilidades que el lenguaje 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.

El material está dividido en 15 secciones. Cada sección presenta un aspecto diferente del lenguaje, desde sus características principales (sección 3) hasta temas como el manejo de archivos (sección 13). Por último se incluye un Apéndice con las librerías más usadas y algunos de sus funciones principales.

Cada sección viene acompañada de ejemplos que sirven de guía al estudiante para conocer la sintaxis de las instrucciones a medida que las va conociendo, y no enfrentarse a un código de un programa complejo sin conocer todas las instrucciones que se utilizaron.

Debe tomarse en cuenta el nivel básico de este material el cual esta orientado a estudiantes de primer semestre con poco o ningún conocimiento previo de programación. No es la intención que sirva como referencia sobre aspectos teóricos del lenguaje C.

Cualquier comentario, sugerencia o críticas sobre este material lo pueden hacer a la dirección electrónica: **jjaimes@tyto.ciens.ucv.ve**

2. Esquema general de un computador

Un ordenador es un sistema capaz de almacenar y procesar con gran rapidez una gran cantidad de información. Además, un ordenador tiene capacidad para comunicarse con el exterior, recibiendo datos, órdenes y programas como entrada (por medio del teclado, del ratón, de un disquete, etc.), y proporcionando resultados de distinto tipo como salida (en la pantalla, por la impresora, mediante un archivo en un disquete, etc.).

Los computadores modernos tienen también una gran capacidad de conectarse en red para comunicarse entre sí, intercambiando mensajes y archivos, o compartiendo recursos tales como tiempo de CPU, impresoras, lectores de CD-ROM, escáner, etc. En la actualidad, estas redes de ordenadores tienen cobertura realmente mundial, y pasan por encima de fronteras, de continentes, e incluso de marcas y modelos de ordenador.

Los computadores que se utilizan actualmente tienen la característica común de ser sistemas digitales. Quiere esto decir que lo que hacen básicamente es trabajar a gran velocidad con una gran cantidad de unos y ceros. La memoria de un computador contiene millones de minúsculos interruptores electrónicos (transistores) y al no tener partes mecánicas móviles, son capaces de cambiar de estado muchos millones de veces por segundo. La tecnología moderna ha permitido miniaturizar estos sistemas y producirlos en grandes cantidades por un precio verdaderamente mínimo.

Actualmente, los ordenadores están presentes en casi todas partes: cualquier automóvil y gran número de electrodomésticos incorporan uno o –probablemente– varios procesadores digitales. La diferencia principal entre estos sistemas y los computadores personales que se utilizan en las prácticas de esta asignatura, está sobre todo en el carácter especializado o de propósito general que tienen, respectivamente, ambos tipos de ordenadores. El procesador que chequea el sistema eléctrico de un automóvil está diseñado para eso y probablemente no es capaz de hacer otra cosa; por eso no necesita de muchos elementos auxiliares. Por el contrario, un PC con una configuración estándar puede dedicarse a multitud de tareas, desde contabilidad doméstica o profesional, procesamiento de textos, dibujo artístico y técnico, cálculos científicos, etc.

Existen cientos de miles de aplicaciones gratuitas o comerciales capaces de resolver los más variados problemas. Pero además, cuando lo que uno busca no está disponible en el mercado (o es excesivamente caro, o presenta cualquier otro tipo de dificultad), 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 uno de los más utilizados en la actualidad. Este es el lenguaje que será presentado a continuación.

2.1 Partes o Elementos de un Computador

Un computador en general, o un PC en particular, constan de distintas partes interconectadas entre sí y que trabajan conjunta y coordinadamente. No es éste el momento de entrar en la descripción detallada de estos elementos, aunque se van a enumerar de modo muy breve.

- Procesador o CPU (Central Process Unit, o unidad central de proceso). Es el corazón del ordenador, que se encarga de realizar las operaciones aritméticas y lógicas, así como de coordinar el funcionamiento de todos los demás componentes.
- Memoria principal o memoria RAM (Random Access Memory). Es el componente del computador donde se guardan los datos y los programas que la CPU está utilizando. Se

llama también a veces memoria volátil, porque su contenido se borra cuando se apaga el ordenador, o simplemente cuando se reinicia.

- Disco duro. Es uno de los elementos esenciales del computador. El disco duro es capaz de mantener la información –datos y programas– de modo estable, también con el computador apagado. El computador no puede trabajar directamente con los datos del disco, sino que antes tiene que transferirlos a la memoria principal. Cada disco duro está fijo en un determinado computador.

- Disquetes. Tienen unas características y propiedades similares a las de los discos duros, con la diferencia de que los discos duros son mucho más rápidos y tienen mucha más capacidad. Los disquetes por su parte son muy baratos, son extraíbles y sirven para pasar información de un PC a otro con gran facilidad.

- Pantalla o monitor. Es el elemento "visual" del sistema. A través de él el computador nos pide datos y nos muestra los resultados. Puede ser de forma gráfica o simplemente alfanumérica.

- Ratón. Es el dispositivo más utilizado para introducir información no alfanumérica, como por ejemplo, seleccionar una entre varias opciones en un menú o caja de diálogo. Su principal utilidad consiste en mover con facilidad el cursor por la pantalla.

- Teclado. Es el elemento más utilizado para introducir información alfanumérica en el ordenador. Puede también sustituir al ratón por medio de las teclas de desplazamiento.

Otros elementos. Los PC modernos admiten un gran número de periféricos para entrada/salida y para almacenamiento de datos. Se pueden citar las impresoras, plotters, escáner, CD-ROMs, cintas, DVD, ZIP, webcam, cornetas, micrófonos, etc.

2.2 La Memoria: Bits, Bytes y Palabras

La memoria de un computador está constituida por un gran número de unidades elementales, llamadas bits, que contienen unos ó ceros. Un bit aislado tiene muy escasa utilidad; un conjunto adecuado de bits puede almacenar casi cualquier tipo de información. Para facilitar el acceso y la programación, casi todos los ordenadores agrupan los bits en conjuntos de 8, que se llaman bytes u octetos.

La memoria se suele medir en Kilobytes KB (1024 bytes), Megabytes MB (1024 KB) y Gigabytes (1024 MB). Un disquete de 3½" almacena 1,4 MB. El que la CPU pudiera acceder por separado a cada uno de los bytes de la memoria resultaría poco ventajosa. Normalmente se accede a una unidad de memoria superior llamada palabra (word), constituida por varios bytes. En los PC antiguos la palabra tenía 2 bytes (16 bits); a partir del procesador 386 la palabra tiene 4 bytes (32 bits). Algunos procesadores más avanzados tienen palabras de 8 bytes o más.

Hay que señalar que la memoria de un ordenador se utiliza siempre para almacenar diversos tipos de información. Quizás la distinción más importante que ha de hacerse es entre datos y programas. A su vez, los programas pueden corresponder a aplicaciones (programas de usuario, destinados a una tarea concreta), o al propio sistema operativo del ordenador, que tiene como misión el arrancar, coordinar y cerrar las aplicaciones, así como mantener activos y accesibles todos los recursos del ordenador.

2.3 Concepto de "programa"

Un programa –en sentido informático– está constituido por un conjunto de instrucciones que se ejecutan –ordinariamente– de modo secuencial, es decir, cada una a continuación de la anterior. Recientemente, con objeto de disminuir los tiempos de ejecución de programas críticos por su tamaño o complejidad, se está haciendo un gran esfuerzo en desarrollar programas paralelos, esto es, programas que se pueden ejecutar simultáneamente en varios procesadores. La programación paralela es mucho más complicada que la secuencial y no se hará referencia a ella en este curso.

Análogamente a los datos que maneja, las instrucciones que un procesador digital es capaz de entender están constituidas por conjuntos de unos y ceros. A esto se llama lenguaje de máquina o binario, y es muy difícil de manejar. Por ello, desde casi los primeros años de los ordenadores, se comenzaron a desarrollar los llamados lenguajes de alto nivel (tales como Fortran, Cobol, C, Java, etc.), que están mucho más cerca del lenguaje natural. Estos lenguajes están basados en el uso de identificadores, tanto para los datos como para las componentes elementales del programa, que en algunos lenguajes se llaman rutinas o procedimientos, y que en C se denominan funciones. Además, cada lenguaje dispone de una sintaxis o conjunto de reglas con las que se indica de modo inequívoco las operaciones que se quiere realizar.

Los lenguajes de alto nivel son más o menos comprensibles para el usuario, pero no para el procesador. Para que éste pueda ejecutarlos es necesario traducirlos a su propio lenguaje de máquina. Esta es una tarea que realiza un programa especial llamado compilador, que traduce el programa a lenguaje de máquina. Esta tarea se suele descomponer en dos etapas, que se pueden realizar juntas o por separado. El programa de alto nivel se suele almacenar en uno o más archivos llamados fuente, que en casi todos los sistemas operativos se caracterizan por una terminación –también llamada extensión– especial. Así, todos los archivos fuente de C deben terminar por (.cpp); ejemplos de nombres de estos archivos son calculos.cpp, derivada.cpp, etc.

La primera tarea del compilador es realizar una traducción directa del programa a un lenguaje más próximo al del computador (llamado ensamblador), produciendo un archivo objeto con el mismo nombre que el archivo original, pero con la extensión (.obj). En una segunda etapa se realiza el proceso de montaje (linkage) del programa, consistente en producir un programa ejecutable en lenguaje de máquina, en el que están ya incorporados todos los otros módulos que aporta el sistema sin intervención explícita del programador (funciones de librería, recursos del sistema operativo, etc.). En un PC con sistema operativo Windows el programa ejecutable se guarda en un archivo con extensión (*.exe). Este archivo es cargado por el sistema operativo en la memoria RAM cuando el programa va a ser ejecutado.

Una de las ventajas más importantes de los lenguajes de alto nivel es la portabilidad de los archivos fuente resultantes. Quiere esto decir que un programa desarrollado en un PC podrá ser ejecutado en un Macintosh o en una máquina UNIX, con mínimas modificaciones y una simple recompilación. 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 computadores, dando lugar al ANSI C, que es la variante que actualmente se utiliza casi universalmente.

2.4 Concepto de "función"

Las aplicaciones informáticas que habitualmente se utilizan, incluso a nivel de informática personal, suelen contener decenas y aún cientos de miles de líneas de código fuente. A medida que los programas se van desarrollando y aumentan de tamaño, se convertirían rápidamente en

sistemas poco manejables si no fuera por la modularización, que es el proceso consistente en dividir un programa muy grande en una serie de módulos mucho más pequeños y manejables.

A estos módulos se les suele denominar de distintas formas (subprogramas, subrutinas, procedimientos, funciones, etc.) según los distintos lenguajes. El lenguaje C hace uso del concepto de función (function). Sea cual sea la nomenclatura, la idea es sin embargo siempre la misma: dividir un programa grande en un conjunto de subprogramas o funciones más pequeñas que son llamadas por el programa principal; éstas a su vez llaman a otras funciones más específicas y así sucesivamente.

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

1. Modularización. Cada función tiene una misión muy concreta, de modo que nunca tiene un número de líneas excesivo y siempre se mantiene dentro de un tamaño manejable. Además, una misma función (por ejemplo, un producto de matrices, una resolución de un sistema de ecuaciones lineales,...) 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.

2. Ahorro de memoria y tiempo de desarrollo. En la medida en que una misma función es utilizada muchas veces, el número total de líneas de código del programa disminuye, y también lo hace la probabilidad de introducir errores en el programa.

3. Independencia de datos y ocultamiento de información. Una de las fuentes más comunes de errores en los programas de computador son los efectos colaterales o perturbaciones que se pueden producir entre distintas partes del programa. Es muy frecuente que al hacer una modificación para añadir una funcionalidad o corregir un error, se introduzcan nuevos errores en partes del programa que antes funcionaban correctamente. Una función es capaz de mantener una gran 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 compete.

Las funciones de C están implementadas con un particular cuidado y riqueza, constituyendo uno de los aspectos más potentes del lenguaje. Es muy importante entender bien su funcionamiento y sus posibilidades.

3. Características del Lenguaje C

El lenguaje C se conoce como un lenguaje compilado. Existen dos tipos de lenguaje: interpretados y compilados. Los interpretados son aquellos que necesitan del código fuente para

funcionar (Por ejemplo: Java). Los compilados convierten el código fuente en un archivo objeto y éste en un archivo ejecutable. Este es el caso del lenguaje C.

Podemos decir que el lenguaje C es un lenguaje de nivel medio, ya que combina elementos de lenguaje de alto nivel con la funcionalidad del lenguaje ensamblador. Es un lenguaje estructurado, ya que permite crear procedimientos en bloques dentro de otros procedimientos. Hay que destacar que el C es un lenguaje portable, ya que permite utilizar el mismo código en diferentes equipos y sistemas informáticos: el lenguaje es independiente de la arquitectura de cualquier máquina en particular.

Por último solo queda decir que el C es un lenguaje relativamente pequeño; se puede describir en poco espacio y aprender rápidamente. Este es sin duda el objetivo de éste curso. No pretende ser un completo manual de la programación, sino una base útil para que cualquiera pueda introducirse en este apasionante mundo.

Aunque en principio cualquier compilador de C es válido, para seguir este curso se recomienda utilizar el compilador **Turbo C/C++ 3.1** el cual se encuentra disponible en la página web de la materia: http://strix.ciens.ucv.ve/~algopu.

El lenguaje C es uno de los más rápidos y potentes que hay hoy en día. No hay más que decir que el sistema operativo Linux está desarrollado en C en su práctica totalidad. El conocerlo nos servirá como base para aprender C++ e introducirnos en el mundo de la programación Windows. Si optamos por Linux existe una biblioteca llamada gtk que permite desarrollar aplicaciones estilo Windows con C.

No debemos confundir C con C++, no son lo mismo. Se podría decir que C++ es una extensión de C. Para aprender C++ conviene tener una sólida base de C.

Una de las cosas importantes de C que se debe recordar es que es Case Sensitive (sensible a las mayúsculas). Es decir que para C no es lo mismo escribir Printf que printf. Conviene indicar también que las instrucciones se separan por ";".

4. Estructura de un programa en C

4.1 Estructura

Todo programa en C consta de una o más funciones, una de las cuales se llama **main**. El programa comienza en la función main, desde la cual es posible llamar a otras funciones. Cada

función estará formada por la cabecera de la función, compuesta por el nombre de la misma y la lista de argumentos (si los hubiese), la declaración de las variables a utilizar y la secuencia de sentencias a ejecutar.

Ejemplo:

```
Declaraciones globales

main ( ) {
       variables locales
       bloque de instrucciones
}

funcion1 ( ) {
       variables locales
       bloque de instrucciones
}
```

4.2 Comentarios

A la hora de programar es conveniente añadir comentarios (cuantos más mejor) para poder saber que función tiene cada parte del código, en caso de que no lo utilicemos durante algún tiempo. Además facilitaremos el trabajo a otros programadores que puedan utilizar nuestro archivo fuente.

Para poner comentarios en un programa escrito en C usamos los símbolos **/*** y ***/**:

```
/* Este es un ejemplo de comentario */

/* Un comentario también puede
estar escrito en varias líneas */
```

El símbolo **/*** se coloca al principio del comentario y el símbolo ***/** al final. El comentario contenido entre estos dos símbolos, no será tenido en cuenta por el compilador.

4.3 Palabras clave

Existen una serie de indicadores reservados, con una finalidad determinada, que no podemos utilizar como identificadores. A continuación vemos algunas de estas palabras clave:

char	**int**	**float**	**double**	**if**
else	**do**	**while**	**for**	**switch**
short	**long**	**extern**	**static**	**default**
continue	**break**	**register**	**sizeof**	**typedef**

4.4 Identificadores

Un identificador es el nombre que damos a las variables y funciones. Está formado por una secuencia de letras y números, aunque también acepta el caracter de subrayado **_**. Por el contrario, no acepta los acentos, símbolos de admiración, interrogación, ni la letra ñ o Ñ.

El primer caracter de un identificador no puede ser un número, es decir que debe ser una

letra o el símbolo **_**. Se diferencian las mayúsculas de las minúsculas, así **num**, **Num** y **nuM** son distintos identificadores.

A continuación vemos algunos ejemplos de identificadores válidos y no válidos:

Válidos	**No válidos**
_num	1num
var1	número2
fecha_nac	año_nac

5. Tipos de datos

5.1 Tipos

En C existen básicamente cuatro tipos de datos, aunque como se verá después, podremos definir nuestros propios tipos de datos a partir de estos cuatro. A continuación se detalla su nombre, el tamaño que ocupa en memoria y el rango de sus posibles valores.

Tipo	Tamaño	Rango de valores
char	1 byte	-128 a 127
int	2 bytes	-32768 a 32767
float	4 bytes	3'4 E-38 a 3'4 E+38
double	8 bytes	1'7 E-308 a 1'7 E+308

5.2 Calificadores de tipo

Los calificadores de tipo tienen la misión de modificar el rango de valores de un determinado tipo de variable. Estos calificadores son cuatro:

- **Signed**: Le indica a la variable que va a llevar signo. Es el utilizado por defecto.

	Tamaño	Rango de valores
signed char	1 byte	-128 a 127
signed int	2 bytes	-32768 a 32767

- **Unsigned**: Le indica a la variable que no va a llevar signo (valor absoluto).

	Tamaño	Rango de valores
unsigned char	1 byte	0 a 255
unsigned int	2 bytes	0 a 65535

- **Short**: Rango de valores en formato corto (limitado). Es el utilizado por defecto.

	Tamaño	Rango de valores
short char	1 byte	-128 a 127
short int	2 bytes	-32768 a 32767

- **Long**: Rango de valores en formato largo (ampliado).

	Tamaño	Rango de valores
long char	4 bytes	-2.147.483.648 a 2.147.483.647
long int	10 bytes	-3'36 E-4932 a 1'18 E+4932

También es posible combinar calificadores entre sí:

- signed long int = long int = long
- unsigned long int = unsigned long 4 bytes 0 a 4.294.967.295 (El mayor entero de C)

5.3 Las variables

Una variable es un tipo de dato, referenciado mediante un identificador (que es el nombre de la variable). Su contenido podrá ser modificado a lo largo del programa. Una variable sólo puede pertenecer a un tipo de dato. Para poder utilizar una variable, primero tiene que ser declarada:

<tipo> <nombre>

Es posible inicializar y declarar más de una variable del mismo tipo en la misma sentencia:

<tipo> <nombre1>,<nombre2>=<valor>,<nombre3>=<valor>,<nombre4>

Ejemplo:

```
/* Uso de las variables para la suma de dos valores */

main()
{
   int num1 = 4, num2, num3 = 6;
   num2 = num1 + num3;
}
```

5.4 ¿Dónde se declaran?

Las variables pueden ser de dos tipos según el lugar en que las declaremos: *globales* o *locales*.

La variable global se declara antes del **main( )**. Puede ser utilizada en cualquier parte del programa y se destruye al finalizar éste.

La variable local se declara después del **main( )**, en la función en que vaya a ser utilizada. Sólo existe dentro de la función en que se declara y se destruye al finalizar dicha función.

Ejemplo:

```
/* Declaración de variables globales y locales*/

int a; /*esta es una variable global*/

main() /* Muestra dos valores */
{
   int b = 4; /* esto es una variable local */

}
```

5.5 Constantes

Al contrario que las variables, las constantes mantienen su valor a lo largo de todo el programa. Para indicar al compilador que se trata de una constante, usaremos la directiva **#define**:

#define <identificador> <valor>

Observa que no se indica el punto y coma de final de sentencia ni tampoco el tipo de dato. La directiva **#define** no sólo nos permite sustituir un nombre por un valor numérico, sino también

por una cadena de caracteres. El valor de una constante no puede ser modificado de ninguna manera.

```
/* Uso de las constantes para el cálculo de un perímetro*/

#define pi 3.1416

main()
{
   int r = 10;
   float = 2*pi*r;
}
```

5.6 Secuencias de escape

Ciertos caracteres no representados gráficamente se pueden representar mediante lo que se conoce como secuencia de escape. A continuación vemos una tabla de las más significativas:

\n	salto de línea
\b	retroceso
\t	tabulación horizontal
\v	tabulación vertical
****	contrabarra
\f	salto de página
\'	apóstrofe
\"	comillas dobles
\0	fin de una cadena de caracteres

5.7 Inclusión de archivos

En la programación en C es posible utilizar funciones que no estén incluidas en el propio programa. Para ello utilizamos la directiva **#include**, que nos permite añadir librerías o funciones que se encuentran en otros archivos a nuestro programa. Para indicar al compilador que vamos a incluir archivos externos podemos hacerlo de dos maneras (siempre antes de las declaraciones).

1. Indicándole al compilador la ruta donde se encuentra el archivo.
   #include "c:\includes\misfunc.h"
2. Indicando que se encuentran en el directorio por defecto del compilador.
   #include <misfunc.h>

6. Operadores

6.1 Operadores aritméticos

Existen dos tipos de operadores aritméticos:

Los binarios:	
+	Suma
-	Resta
*****	Multiplicación
/	División

%	Módulo (resto)

Los unarios:

++	Incremento (suma 1)
- -	Decremento (resta 1)
-	Cambio de signo

Su sintaxis es:

- *binarios*: **<variable1><operador><variable2>**
- *unarios*: **<variable><operador>** y al revés, **<operador><variable>.**

Ejemplo:

```
/* Uso de los operadores aritméticos */

main()
{
   int a = 1, b = 2, c = 3, r;
   r = a + b;
   r = c - a;
   b++;
}
```

6.2 Operadores de asignación

La mayoría de los operadores aritméticos binarios explicados en el capítulo anterior tienen su correspondiente operador de asignación:

=	Asignación simple
+=	Suma
-=	Resta
***=**	Multiplicación
/=	División
%=	Módulo (resto)

Con estos operadores se pueden escribir, de forma más breve, expresiones del tipo

- **n = n + 3** se puede escribir **n += 3**
- **k = k * (x - 2)** lo podemos sustituir por **k *= x - 2**

Ejemplo:

```
/* Uso de los operadores de asignación */

main()
{
   int a = 1, b = 2, c = 3;
   a += 5;
   c -= 1;
   b *= 3;
```

```
}
```

6.3 Operadores relacionales

Los operadores relacionales se utilizan para comparar el contenido de dos variables. En C existen seis operadores relacionales básicos:

>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
==	Igual que
!=	Distinto que

El resultado que devuelven estos operadores es **1** para Verdadero y **0** para Falso. Si hay más de un operador se evalúan de izquierda a derecha. Además los operadores **==** y **!=** están por debajo del resto en cuanto al orden de precedencia.

6.4 Operadores lógicos

Los operadores lógicos básicos son tres:

&&	AND
||	OR
!	NOT (El valor contrario)

Estos operadores actúan sobre expresiones lógicas. Permiten unir expresiones lógicas simples formando otras más complejas.

6.5 Jerarquía de los operadores

Será importante tener en cuenta la precedencia de los operadores a la hora de trabajar con ellos:

( )	Mayor precedencia
++, --	
***, /, %**	
+, -	Menor precedencia

Las operaciones con mayor precedencia se realizan antes que las de menor precedencia. Si en una operación encontramos signos del mismo nivel de precedencia, dicha operación se realiza de izquierda a derecha. A continuación se muestra un ejemplo sobre ello:

a * b + c / d - e

1. a * b resultado = x
2. c / d resultado = y
3. x + y resultado = z
4. z - e

Fijarse que la multiplicación se resuelve antes que la división ya que está situada más a la izquierda en la operación. Lo mismo ocurre con la suma y la resta.

7.- Salida y Entrada

7.1 Sentencia printf( )

La rutina printf permite la aparición de valores numéricos, caracteres y cadenas de texto por pantalla. El prototipo de la sentencia *printf* es el siguiente:

printf(control, arg1, arg2...);

En la cadena de control indicamos la forma en que se mostrarán los argumentos posteriores. También podemos introducir una cadena de texto (sin necesidad de argumentos), o combinar ambas posibilidades, así como secuencias de escape.

En el caso de que utilicemos argumentos deberemos indicar en la cadena de control tantos

modificadores como argumentos vayamos a presentar. El modificador está compuesto por el caracter **%** seguido por un caracter de conversión, que indica de que tipo de dato se trata.

Ejemplo:

```
/* Uso de la sentencia printf() para imprimir por pantalla*/

#include <stdio.h> /*Librería necesaria para el uso del
printf*/

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);
}
```

Los **modificadores** más utilizados son:

%c	Un único caracter
%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

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 (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.
- *l/L*: utilizamos l cuando se trata de una variable de tipo long y L cuando es de tipo double.

7.2 Sentencia scanf( )

La rutina scanf permite entrar datos en la memoria del ordenador a través del teclado. El prototipo de la sentencia *scanf* es el siguiente:

scanf(control,arg1,arg2...);

En la cadena de control indicaremos, por regla general, los modificadores que harán referencia al tipo de dato de los argumentos. Al igual que en la sentencia *printf* los modificadores estarán formados por el caracter **%** seguido de un caracter de conversión. Los argumentos indicados serán, nuevamente, las variables.

La principal característica de la sentencia *scanf* es que necesita saber la posición de la memoria del ordenador en que se encuentra la variable para poder almacenar la información obtenida. Para indicarle esta posición utilizaremos el símbolo *ampersand* (**&**), que colocaremos delante del nombre de cada variable.

Ejemplo:

```
/* Uso de la sentencia scanf() para solicitar dos datos*/

#include <stdio.h>

main()
{
   int edad;
   printf("Introduce tu nombre: ");
   scanf("%s",nombre);
   printf("Introduce tu edad: ");
   scanf("%d",&edad);
}
```

8 Estructuras Condicionales

Este tipo de sentencias permiten variar el flujo del programa en base a unas determinadas condiciones. Existen varias estructuras diferentes:

8.1 Estructura IF...ELSE

Sintaxis:

```
if (condición)
   sentencia;
```

La sentencia solo se ejecuta si se cumple la condición. En caso contrario el programa sigue su curso sin ejecutar la sentencia. Otro formato:

```
if (condición)
```

```
   sentencia1;
else
   sentencia2;
```

Si se cumple la condición ejecutará la **sentencia1**, sino ejecutará la **sentencia2**. En cualquier caso, el programa continuará a partir de la **sentencia2**.

Ejemplo:

```
/* Uso de la sentencia condicional IF. */

#include <stdio.h>

main()
{
      int usuario, clave = 18276;
      printf("Introduce tu clave: ");
      scanf("%d",&usuario);
      if (usuario == clave)
         printf("Acceso permitido");
      else
         printf("Acceso denegado");
}
```

Otro formato:

```
if (condición)
   sentencia1;
else if (condición)
   sentencia2;
else if (condición)
   sentencia3;
else
   sentencia4;
```

Con este formato el flujo del programa únicamente entra en una de las condiciones. Si una de ellas se cumple, se ejecuta la sentencia correspondiente y salta hasta el final de la estructura para continuar con el programa. Existe la posibilidad de utilizar llaves para ejecutar más de una sentencia dentro de la misma condición.

Ejemplo:

```
/* Uso de la sentencia condicional ELSE...IF. */

#include <stdio.h>

main()
{
     int edad;
     printf("Introduce tu edad: ");
     scanf("%d", &edad);
     if (edad<1)
          printf("Lo siento, te has equivocado.");
     else if (edad<3)
```

```
            printf("Eres un bebé");
        else if (edad<13)
            printf("Eres un niño");
        else
            printf("Eres adulto");
}
```

8.2 Estructura SWITCH

Esta estructura se suele utilizar en los menús, de manera que según la opción seleccionada se ejecuten una serie de sentencias.

Su sintaxis es:

```
switch (variable){
        case contenido_variable1:
                sentencias;
                break;
        case contenido_variable2:
                sentencias;
                break;
        default:
                sentencias;
}
```

Cada case puede incluir una o más sentencias sin necesidad de ir entre llaves, ya que se ejecutan todas hasta que se encuentra la sentencia **BREAK**. La variable evaluada sólo puede ser de tipo **entero** o **caracter**. **default** ejecutará las sentencias que incluya, en caso de que la opción escogida no exista.

Ejemplo:

```
/* Uso de la sentencia condicional SWITCH. */

#include <stdio.h>

main()
{
      int dia;
      printf("Introduce el día: ");
      scanf("%d", &dia);

      switch(dia){
         case 1: printf("Lunes"); break;
         case 2: printf("Martes"); break;
         case 3: printf("Miércoles"); break;
         case 4: printf("Jueves"); break;
         case 5: printf("Viernes"); break;
         case 6: printf("Sábado"); break;
         case 7: printf("Domingo"); break;
         default: printf("Día incorrecto"); break;
      }
```

```
}
```

9. Estructuras cíclicas o bucles

Los bucles son estructuras que permiten ejecutar partes del código de forma repetida mientras se cumpla una condición. Esta condición puede ser simple o compuesta de otras condiciones unidas por operadores lógicos.

9.1 Estructura WHILE

Su sintaxis es:

```
while (condición)
    sentencia;
```

Con esta sentencia se controla la condición antes de entrar en el bucle. Si ésta no se cumple, el programa no entrará en el bucle. Naturalmente, si en el interior del bucle hay más de una sentencia, éstas deberán ir entre llaves para que se ejecuten como un bloque.

Ejemplo:

```
/* Uso de la sentencia WHILE. */

#include <stdio.h>

main()
{
       int numero = 1;
       while(numero<=10)
       {
          printf("%d\n",numero);
          numero++;
       }
}
```

9.2.- Estructura DO...WHILE

Su sintaxis es:

```
do{
       sentencia1;
       sentencia2;
}while (condición);
```

Con esta sentencia se controla la condición al final del bucle. Si ésta se cumple, el programa vuelve a ejecutar las sentencias del bucle.

La única diferencia entre las sentencias while y do...while es que con la segunda el cuerpo del bucle se ejecutará por lo menos una vez.

Ejemplo:

```
/* Uso de la sentencia DO...WHILE para hacer un menú*/

#include <stdio.h>

main()
{
   char seleccion;

   do{
      printf("1.- Comenzar\n");
      printf("2.- Abrir\n");
      printf("3.- Grabar\n");
      printf("4.- Salir\n");
      printf("Escoge una opción: ");
      seleccion = getchar(); /*función que lee un caracter*/

      switch(seleccion){
      case '1':printf("Opción 1");
         break;
      case '2':printf("Opción 2");
         break;
```

```
            case '3':printf("Opción 3");
             }

          }while(seleccion!='4');
      }
```

9.3.- Estructura FOR

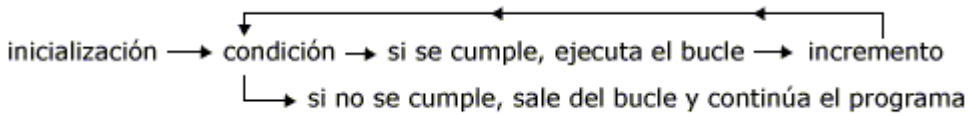
Su sintaxis es:

```
for (inicialización; condición; incremento){
        sentencia1;
        sentencia2;
}
```

La inicialización indica una variable (variable de control) que condiciona la repetición del bucle. Si hay más, van separadas por comas:

```
for (a=1, b=100; a!=b; a++, b- -){
```

El flujo del bucle **FOR** transcurre de la siguiente forma:



Ejemplo:

```
/* Uso de la sentencia FOR para escribir la tabla de mult.*/

#include <stdio.h>

main()
{
       int num, x, result;
       printf("Introduce un número: ");
       scanf("%d",&num);
       for (x=0; x<=10; x++){
          result=num*x;
          printf("\n%d por %d = %d\n", num, x, result);
       }
}
```

9.4.- Sentencia CONTINUE

Se utiliza dentro de un bucle. Cuando el programa llega a una sentencia **CONTINUE** no ejecuta las líneas de código que hay a continuación y salta a la siguiente iteración del bucle.

Existe otra sentencia, **GOTO**, que permite al programa saltar hacia un punto identificado con una etiqueta, pero el buen programador debe prescindir de su utilización. Es una sentencia muy mal vista en la programación en C.

Ejemplo:

```
/* Uso de la sentencia CONTINUE */

#include <stdio.h>

main()
{
       int numero=1;
       while(numero<=100)
       {
          if (numero==25)
          {
          numero++;
          continue;
          }
       printf("%d\n",numero);
       numero++;
       }
}
```

10. Funciones

10.1 Tiempo de vida de los datos

Según el lugar donde son declaradas puede haber dos tipos de variables.

- **Globales**: las variables permanecen activas durante todo el programa. Se crean al iniciarse éste y se destruyen de la memoria al finalizar. Pueden ser utilizadas en cualquier función.

- **Locales**: las variables son creadas cuando el programa llega a la función en la que están definidas. Al finalizar la función desaparecen de la memoria.

Si dos variables, una global y una local, tienen el mismo nombre, la local prevalecerá sobre la global dentro de la función en que ha sido declarada. Dos variables locales pueden tener el mismo nombre siempre que estén declaradas en funciones diferentes.

Ejemplo:

```
/* Variables globales y locales. */

#include <stdio.h>
```

```
int num1 = 1;

main()
{
   int num2 = 10;
   printf("%d\n",num1);
   printf("%d\n",num2);
}
```

10.2 ¿Qué es una función?

Las funciones son bloques de código utilizados para dividir un programa en partes más pequeñas, cada una de las cuáles tendrá una tarea determinada.

Su sintaxis es:

```
tipo_función nombre_función (tipo y nombre de argumentos)
{
        bloque de sentencias
}
```

- **tipo_función**: puede ser de cualquier tipo de los que conocemos. El valor devuelto por la función será de este tipo. Por defecto, es decir, si no indicamos el tipo, la función devolverá un valor de tipo entero (**int**). Si no queremos que retorne ningún valor deberemos indicar el tipo vacío (**void**).

- **nombre_función***:* es el nombre que le daremos a la función.

- **tipo y nombre de argumentos***:* son los parámetros que recibe la función. Los argumentos de una función no son más que variables locales que reciben un valor. Este valor se lo enviamos al hacer la llamada a la función. Pueden existir funciones que no reciban argumentos.

- **bloque de sentencias***:* es el conjunto de sentencias que serán ejecutadas cuando se realice la llamada a la función.

Las funciones pueden ser llamadas desde la función **main** o desde otras funciones. Nunca se debe llamar a la función **main** desde otro lugar del programa. Por último recalcar que los argumentos de la función y sus variables locales se destruirán al finalizar la ejecución de la misma.

10.3 Declaración de las funciones

Al igual que las variables, las funciones también han de ser declaradas. Esto es lo que se conoce como prototipo de una función. Para que un programa en C sea compatible entre distintos compiladores es imprescindible escribir los prototipos de las funciones.

Los prototipos de las funciones pueden escribirse antes de la función **main** o bien en otro archivo. En este último caso se lo indicaremos al compilador mediante la directiva **#include**.

En el ejemplo siguiente podremos ver la declaración de una función (prototipo). Al no recibir ni retornar ningún valor, está declarada como **void** en ambos lados. También vemos que existe una variable global llamada *num*. Esta variable es reconocible en todas las funciones del programa. Ya en la función **main** encontramos una variable local llamada *num*. Al ser una variable local, ésta tendrá preferencia sobre la global. Por tanto la función escribirá los números 10 y 5.

Ejemplo:

```
/* Declaración de funciones. */

#include <stdio.h>

void funcion(void); /* prototipo */
int num = 5; /* variable global */

main()
{
   int num=10; /* variable local */
   printf("%d\n",num);

   funcion(); /* llamada */
}

void funcion(void)
{
   printf("%d\n",num);
}
```

10.4 Paso de parámetros a una función

Como ya hemos visto, las funciones pueden retornar un valor. Esto se hace mediante la instrucción **return**, que finaliza la ejecución de la función, devolviendo o no un valor. En una misma función podemos tener más de una instrucción return. La forma de retornar un valor es la siguiente:

return (valor o expresión);

El valor devuelto por la función debe asignarse a una variable. De lo contrario, el valor se perderá. En el ejemplo puedes ver lo que ocurre si no guardamos el valor en una variable. Fíjate que a la hora de mostrar el resultado de la suma, en el **printf**, también podemos llamar a la función.

Ejemplo:

```
/* Paso de parámetros. */

#include <stdio.h>

int suma(int,int); /* prototipo */

main()
```

```
{
   int a = 10, b = 25, t;
   t = suma(a, b); /* guardamos el valor */
   printf("%d=%d", suma(a, b), t);
   suma(a, b); /* el valor se pierde */
}

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

11 Arreglos

Un arreglo es un identificador que referencia un conjunto de datos del mismo tipo. Imagina un tipo de dato **int**; podremos crear un conjunto de datos de ese tipo y utilizar uno u otro con sólo cambiar el índice que lo referencia. El índice será un valor entero y positivo. En C los arreglos comienzan por la posición **0**.

11.1 Arreglos unidimensionales (Vectores)

Un vector es un arreglo *unidimensional*, es decir, sólo utiliza un índice para referenciar a cada uno de los elementos. Su declaración será:

tipo nombre [tamaño];

El tipo puede ser cualquiera de los ya conocidos y el tamaño indica el número de elementos del vector (se debe indicar entre corchetes **[ ]**). En el ejemplo se puede observar que la variable **i** es utilizada como índice, el primer **for** sirve para rellenar el vector y el segundo para visualizarlo. Como se aprecia, las posiciones van de **0** a **9** (total 10 elementos).

Ejemplo:

/* Declaración de un arreglo. */

```
#include <stdio.h>

main()
{
    int vector[10],i;
    for (i=0;i<10;i++) vector[i]=i;
    for (i=0;i<10;i++) printf(" %d", vector[i]);
}
```

Podemos *inicializar* (asignarle valores) un vector en el momento de declararlo. Si lo hacemos así no es necesario indicar el tamaño. Su sintaxis es:

tipo nombre []={ valor 1, valor 2...}

Ejemplos:

```
int vector[]={1,2,3,4,5,6,7,8};
char vector[]="programador";
char vector[]={'p','r','o','g','r','a','m','a','d','o','r'};
```

Una particularidad con los vectores de tipo **char** (cadena de caracteres), es que deberemos indicar en que elemento se encuentra el fin de la cadena mediante el caracter nulo (**\0**). Esto no lo controla el compilador, y tendremos que ser nosotros los que insertemos este caracter al final de la cadena.

Por tanto, en un vector de 10 elementos de tipo **char** podremos rellenar un máximo de 9, es decir, hasta **vector[8]**. Si sólo rellenamos los 5 primeros, hasta **vector[4]**, debemos asignar el caracter nulo a **vector[5]**. Es muy sencillo: **vector[5]='\0';**

Ahora veremos un ejemplo de como se rellena un vector de tipo **char**.

```
/* Vector de tipo char. */

#include <stdio.h>

main()
{
    char cadena[20];
    int i;
    for (i=0;i<19 && cadena[i-1]!=13;i++)
          cadena[i]=getche( );
    if (i==19)
       cadena[i]='\0';
    else
       cadena[i-1]='\0';
    printf("\n%s",cadena);
}
```

Podemos ver que en el **for** se encuentran dos condiciones:

1.- Que no se hayan rellenado todos los elementos (**i<19**).
2.- Que el usuario no haya pulsado la tecla ENTER, cuyo código ASCII es **13**.

También podemos observar una nueva función llamada **getche( )**, que se encuentra en **conio.h**. Esta función permite la entrada de un caracter por teclado. Después se encuentra un **if**, que comprueba si se ha rellenado todo el vector. Si es cierto, coloca el caracter nulo en el elemento 20 (**cadena[19]**). En caso contrario tenemos el **else**, que asigna el caracter nulo al elemento que almacenó el caracter ENTER.

En resumen: al declarar una cadena deberemos reservar una posición más que la longitud que queremos que tenga dicha cadena.

Llamadas a funciones con arreglos

Como ya se comentó en el tema anterior, los arreglos únicamente pueden ser enviados a una función por referencia. Para ello deberemos enviar la dirección de memoria del primer elemento del arreglo. Por tanto, el argumento de la función deberá ser un puntero.

Ejemplo:

```
/* Envío de un arreglo a una función. */

#include <stdio.h>

void visualizar(int []); /* prototipo */

main()
{
      int array[25],i;
      for (i=0;i<25;i++)
      {
            printf("Elemento n° %d",i+1);
            scanf("%d",&array[i]);
      }
      visualizar(&array[0]);
}

void visualizar(int array[]) /* desarrollo */
{
      int i;
      for (i=0;i<25;i++) printf("%d",array[i]);
}
```

En el ejemplo se puede apreciar la forma de enviar un arreglo por referencia. La función se podía haber declarado de otra manera, aunque funciona exactamente igual:

- *declaración o prototipo*

void visualizar(int *);

- *desarrollo de la función*

void visualizar(int *arreglo)

11.2 Arreglos multidimensionales (Matrices)

Una matriz es un arreglo *multidimensional*. Se definen igual que los vectores excepto que se requiere un índice por cada dimensión.

Su sintaxis es la siguiente:

tipo nombre [tamaño 1][tamaño 2]...;

Una matriz *bidimensional* se podría representar gráficamente como una tabla con filas y columnas. La matriz *tridimensional* se utiliza, por ejemplo, para trabajos gráficos con objetos **3D**. En el ejemplo puedes ver como se rellena y visualiza una matriz *bidimensional*. Se necesitan dos bucles para cada una de las operaciones. Un bucle controla las filas y otro las columnas.

Ejemplo:

```
/* Matriz bidimensional. */

#include <stdio.h>

main()
{
   int x, i, numeros[3][4];

   /* rellenamos la matriz */

   for (x=0;x<3;x++)
        for (i=0;i<4;i++)
                scanf("%d",&numeros[x][i]);

   /* visualizamos la matriz */

   for (x=0;x<3;x++)
        for (i=0;i<4;i++)
                printf("%d",numeros[x][i]);
}
```

Si al declarar una matriz también queremos inicializarla, habrá que tener en cuenta el orden en el que los valores son asignados a los elementos de la matriz. Veamos algunos ejemplos:

int numeros[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};

Quedarían asignados de la siguiente manera:

numeros[0][0]=1 numeros[0][1]=2 numeros[0][2]=3 numeros[0][3]=4
numeros[1][0]=5 numeros[1][1]=6 numeros[1][2]=7 numeros[1][3]=8
numeros[2][0]=9 numeros[2][1]=10 numeros[2][2]=11 numeros[2][3]=12

También se pueden inicializar cadenas de texto:

char dias[7][10]={"lunes","martes", ... ,"viernes","sábado","domingo"};

Para referirnos a cada palabra bastaría con el primer índice:

```
printf("%s",dias[i]);
```

12 Registros

12.1 Concepto de registro

Un registro es un conjunto de una o más variables, de distinto tipo, agrupadas bajo un mismo nombre para que su manejo sea más sencillo. Su utilización más habitual es para la programación de bases de datos, ya que están especialmente indicadas para el trabajo con registros o fichas.

La sintaxis de su declaración es la siguiente:

```
struct tipo_estructura
{
        tipo_variable nombre_variable1;
        tipo_variable nombre_variable2;
        tipo_variable nombre_variable3;
};
```

Donde **tipo_estructura** es el nombre del nuevo tipo de dato que hemos creado. Por último, **tipo_variable** y **nombre_variable** son las variables que forman parte del registro. Para definir variables del tipo que acabamos de crear lo podemos hacer de varias maneras, aunque las dos más utilizadas son éstas:

Una forma de definir el registro es:

```
        struct trabajador
        {
                char nombre[20];
```

```
        char apellidos[40];
        int edad;
        char puesto[10];
};

struct trabajador fijo, temporal;
```

Otra forma es:

```
struct trabajador
{
        char nombre[20];
        char apellidos[40];
        int edad;
        char puesto[10];
} fijo, temporal;
```

En el primer caso declaramos el registro, y en el momento en que necesitamos las variables, las declaramos. En el segundo las declaramos al mismo tiempo que al registro. El problema del segundo método es que no podremos declarar más variables de este tipo a lo largo del programa. Para poder declarar una variable de tipo registro, el mismo tiene que estar declarado previamente. Se debe declarar antes de la función **main**.

El manejo de los registros es muy sencillo, así como el acceso a los campos (o variables) de estos registros. La forma de acceder a estos campos es la siguiente:

```
variable.campo;
```

Donde **variable** es el nombre de la variable de tipo *registro* que hemos creado, y **campo** es el nombre de la variable que forma parte del registro. Lo veremos mejor con un ejemplo basado en el registro definido anteriormente:

```
temporal.edad = 25;
```

Lo que estamos haciendo es almacenar el valor 25 en el campo **edad** de la variable **temporal** de tipo **trabajador**. Otra característica interesante de los registros es que permiten pasar el contenido de un registro a otro, siempre que sean del mismo tipo naturalmente:

```
fijo=temporal;
```

Al igual que con los otros tipos de datos, también es posible inicializar variables de tipo **registro** en el momento de su declaración:

```
struct trabajador fijo={"Pedro","Hernández Suárez", 32, "gerente"};
```

Si uno de los campos del registro es un arreglo de números, los valores de la inicialización deberán ir entre llaves:

```
struct notas
{
        char nombre[30];
        int notas[5];
```

};

struct notas alumno={"Carlos Pérez",{8,7,9,6,10}};

12.2 Registros y funciones

Podemos enviar un registro a una función de las dos maneras conocidas:

1.- Por valor: su declaración sería:

void visualizar(struct trabajador);

Después declararíamos la variable **fijo** y su llamada sería:

visualizar(fijo);

Por último, el desarrollo de la función sería:

void visualizar(struct trabajador datos)

Ejemplo:

```
/* Paso de un registro por valor. */

#include <stdio.h>

struct trabajador
{
      char nombre[20];
      char apellidos[40];
      int edad;
      char puesto[10];
};

void visualizar(struct trabajador);
main()
{
      struct trabajador fijo;
      printf("Nombre: ");
      scanf("%s",fijo.nombre);
      printf("\nApellidos: ");
      scanf("%s",fijo.apellidos);
      printf("\nEdad: ");
      scanf("%d",&fijo.edad);
      printf("\nPuesto: ");
      scanf("%s",fijo.puesto);
      visualizar(fijo);
}

void visualizar(struct trabajador datos)
{
       printf("Nombre: %s",datos.nombre);
      printf("\nApellidos: %s",datos.apellidos);
      printf("\nEdad: %d",datos.edad);
```

```
        printf("\nPuesto: %s",datos.puesto);
}
```

2.- Por referencia: su declaración sería:

void visualizar(struct trabajador *);

Después declararemos la variable **fijo** y su llamada será:

visualizar(&fijo);

Por último, el desarrollo de la función será:

void visualizar(struct trabajador *datos)

Fíjense que en la función **visualizar**, el acceso a los campos de la variable **datos** se realiza mediante el operador **->**, ya que lo tratamos con un puntero. En estos casos siempre utilizaremos el operador **->**. Se consigue con el signo *menos* seguido de *mayor que*.

Ejemplo:

```
/* Paso de un registro por referencia. */

#include <stdio.h>

struct trabajador
{
     char nombre[20];
     char apellidos[40];
     int edad;
        char puesto[10];
};

void visualizar(struct trabajador *);
main()
{
     struct trabajador fijo;
     printf("Nombre: ");
     scanf("%s",fijo.nombre);
     printf("\nApellidos: ");
     scanf("%s",fijo.apellidos);
     printf("\nEdad: ");
     scanf("%d",&fijo.edad);
     printf("\nPuesto: ");
     scanf("%s",fijo.puesto);
     visualizar(&fijo);
}

void visualizar(struct trabajador *datos)
{
     printf("Nombre: %s",datos->nombre);
     printf("\nApellidos: %s",datos->apellidos);
     printf("\nEdad: %d",datos->edad);
```

```
            printf("\nPuesto: %s",datos->puesto);
}
```

12.3 Arreglos de registros

Es posible agrupar un conjunto de elementos de tipo registro en un arreglo. Esto se conoce como *arreglo de registros:*

```
struct trabajador
{
    char nombre[20];
    char apellidos[40];
    int edad;
};

struct trabajador fijo[20];
```

Así podremos almacenar los datos de 20 trabajadores. Ejemplos sobre como acceder a los campos y sus elementos: para ver el nombre del cuarto trabajador, **fijo[3].nombre;**. Para ver la tercera letra del nombre del cuarto trabajador, **fijo[3].nombre[2];**. Para inicializar la variable en el momento de declararla lo haremos de esta manera:

```
struct trabajador fijo[20] = { {"José", "Herrero Martínez", 29 }, { "Luis", "García Sánchez", 46 } };
```

12.4 Definición de tipos

El lenguaje 'C' dispone de una declaración llamada **typedef** que permite la creación de nuevos tipos de datos. Ejemplos:

```
typedef int entero;       /*acabamos de crear un tipo de dato llamado entero*/
entero a, b = 3;          /*declaramos dos variables de este tipo*/
```

Su empleo con registros está especialmente indicado. Se puede hacer de varias formas:

Una forma de hacerlo:

```
struct trabajador
{
    char nombre[20];
    char apellidos[40];
    int edad;
};

typedef struct trabajador datos;
datos fijo, temporal;
```

Otra forma:

```
typedef struct
{
    char nombre[20];
    char apellidos[40];
    int edad;
}datos;

datos fijo, temporal;
```

13. Archivos

Por último veremos la forma de almacenar datos que podremos recuperar cuando deseemos. Estudiaremos los distintos modos en que podemos abrir un archivo, así como las funciones para leer y escribir en él.

13.1 Apertura

Antes de abrir un archivo necesitamos declarar un puntero de tipo **FILE**, con el que trabajaremos durante todo el proceso. Para abrir el archivo utilizaremos la función **fopen( )**.

Su sintaxis es:

```
FILE *puntero;
puntero = fopen ( nombre del archivo, "modo de apertura" );
```

donde **puntero** es la variable de tipo **FILE**, **nombre del archivo** es el nombre que daremos al archivo que queremos crear o abrir. Este nombre debe ir encerrado entre comillas. También podemos especificar la ruta donde se encuentra o utilizar un arreglo que contenga el nombre del archivo (en este caso no se pondrán las comillas).

Algunos ejemplos:

```
puntero = fopen("DATOS.DAT","r");
puntero = fopen("C:\\TXT\\SALUDO.TXT","w");
```

Un archivo puede ser abierto en dos modos diferentes, en modo texto o en modo binario. A continuación lo veremos con más detalle.

Modo texto

w	crea un archivo de escritura. Si ya existe lo crea de nuevo.
w+	crea un archivo de lectura y escritura. Si ya existe lo crea de nuevo.
a	abre o crea un archivo para añadir datos al final del mismo.
a+	abre o crea un archivo para leer y añadir datos al final del mismo.
r	abre un archivo de lectura.
r+	abre un archivo de lectura y escritura.

Modo binario

wb	crea un archivo de escritura. Si ya existe lo crea de nuevo.
w+b	crea un archivo de lectura y escritura. Si ya existe lo crea de nuevo.
ab	abre o crea un archivo para añadir datos al final del mismo.
a+b	abre o crea un archivo para leer y añadir datos al final del mismo.
rb	abre un archivo de lectura.
r+b	abre un archivo de lectura y escritura.

La función **fopen** devuelve, como ya hemos visto, un puntero de tipo **FILE**. Si al intentar abrir el archivo se produjese un error (por ejemplo si no existe y lo estamos abriendo en modo lectura), la función **fopen** devolvería **NULL**. Por esta razón es mejor controlar las posibles causas de error a la hora de programar.

Un ejemplo:

```
FILE *pf;
pf=fopen("datos.txt","r");
if (pf == NULL) printf("Error al abrir el archivo");
```

La función **freopen** cierra el archivo apuntado por el puntero y reasigna este puntero a un archivo que será abierto.

Su sintaxis es:

freopen(nombre del archivo,"modo de apertura",puntero);

donde **nombre del archivo** es el nombre del nuevo archivo que queremos abrir, luego el **modo de apertura**, y finalmente el puntero que va a ser reasignado.

13.2 Cierre

Una vez que hemos acabado nuestro trabajo con un archivo es recomendable cerrarlo. Los archivos se cierran al finalizar el programa pero el número de estos que pueden estar abiertos es limitado. Para cerrar los archivos utilizaremos la función **fclose( );**.

Esta función cierra el archivo, cuyo puntero le indicamos como parámetro. Si el archivo se cierra con éxito devuelve **0**.

fclose(puntero);

Un ejemplo ilustrativo es:

```
FILE *pf;
pf = fopen("AGENDA.DAT","rb");
```

```
if ( pf == NULL )
     printf ("Error al abrir el archivo");
else
     fclose(pf);
```

13.3 Escritura y lectura

A continuación veremos las funciones que se podrán utilizar dependiendo del dato que queramos escribir y/o leer en el archivo.

Un carácter:

fputc(variable_caracter , puntero_archivo);

Escribimos un caracter en un archivo (abierto en modo escritura).

Un ejemplo:

```
FILE *pf;
char letra='a';
if (!(pf=fopen("datos.txt","w"))) /* forma de controlar si hay un error */
{
        printf("Error al abrir el archivo");
        exit(0); /* abandonamos el programa */
}
else
     fputc(letra,pf);
fclose(pf);
```

La función **fgetc(puntero_archivo),** lee un caracter de un archivo (abierto en modo lectura). Deberemos guardarlo en una variable.

Un ejemplo:

```
FILE *pf;
char letra;
if (!(pf=fopen("datos.txt","r"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el archivo");
        exit(0); /* abandonamos el programa */
}
else
{
        letra=fgetc(pf);
        printf("%c",letra);
        fclose(pf);
}
```

Un número entero:

putw(variable_entera, puntero_archivo);

Escribe un número entero en formato binario en el archivo.

Ejemplo:

```
FILE *pf;
int num=3;
if (!(pf=fopen("datos.txt","wb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el archivo");
        exit(0); /* abandonamos el programa */
}
else
{
        fputw(num,pf); /* también podíamos haber hecho: fputw(3,pf); */
        fclose(pf);
}
```

La función **getw(puntero_archivo),** lee un número entero de un archivo, avanzando dos bytes después de cada lectura.

Un ejemplo:

```
FILE *pf;
int num;
if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el archivo");
        exit(0); /* abandonamos el programa */
}
else
{
        num=getw(pf);
        printf("%d",num);
        fclose(pf);
}
```

Una cadena de caracteres:

fputs(variable_array, puntero_archivo);

Escribe una cadena de caracteres en el archivo.

Ejemplo:

```
FILE *pf;
char cad="Me llamo Vicente";
if (!(pf=fopen("datos.txt","w"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el archivo");
        exit(0); /* abandonamos el programa */
}
else
{
        fputs(cad,pf); /* o también así: fputs("Me llamo Vicente",pf); */
```

```
        fclose(pf);
}
```

La función **fgets(variable_array, variable_entera, puntero_archivo), l**ee una cadena de caracteres del archivo y la almacena en variable_array. La variable_entera indica la longitud máxima de caracteres que puede leer.

Un ejemplo:

```
FILE *pf;
char cad[80];
if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el archivo");
        exit(0); /* abandonamos el programa */
}
else
{
        fgets(cad,80,pf);
        printf("%s",cad);
        fclose(pf);
}
```

Con formato:

fprintf(puntero_archivo, formato, argumentos);

Funciona igual que un **printf** pero guarda la salida en un archivo.

Ejemplo:

```
FILE *pf;
char nombre[20]="Santiago";
int edad=34;
if (!(pf=fopen("datos.txt","w"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el archivo");
        exit(0); /* abandonamos el programa */
}
else
{
        fprintf(pf,"%20s%2d\n",nombre,edad);
        fclose(pf);
}
```

La función **fscanf(puntero_archivo, formato, argumentos),** lee los argumentos del archivo. Al igual que con un **scanf**, deberemos indicar la dirección de memoria de los argumentos con el símbolo **&** (ampersand).

Un ejemplo:

```
FILE *pf;
char nombre[20];
```

```
int edad;
if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el archivo");
        exit(0); /* abandonamos el programa */
}
else
{
        fscanf(pf,"%20s%2d\",nombre,&edad);
        printf("Nombre: %s  Edad: %d",nombre,edad);
        fclose(pf);
}
```

Registros

fwrite(*buffer, tamaño, nº de veces, puntero_archivo);

Se utiliza para escribir bloques de texto o de datos, registros, en un archivo. En esta función, *buffer será la dirección de memoria de la cuál se recogerán los datos; **tamaño**, el tamaño en bytes que ocupan esos datos y **nº de veces**, será el número de elementos del tamaño indicado que se escribirán.

fread(*buffer, tamaño, nº de veces, puntero_archivo);

Se utiliza para leer bloques de texto o de datos de un archivo. En esta función, ***buffer** es la dirección de memoria en la que se almacenan los datos; **tamaño**, el tamaño en bytes que ocupan esos datos y **nº de veces**, será el número de elementos del tamaño indicado que se leerán.

Otras funciones para archivos:

- **rewind(puntero_archivo):** Sitúa el puntero al principio del archivo.

- **fseek(puntero_archivo, long posicion, int origen):** Sitúa el puntero en la **posicion** que le indiquemos. Como **origen** podremos poner:

- **0** = **SEEK_SET**, el principio del archivo
- **1** = **SEEK_CUR**, la posición actual
- **2** = **SEEK_END**, el final del archivo

- **rename(nombre1, nombre2):** Su función es exactamente la misma que la que conocemos en **MS-DOS**. Cambia el nombre del archivo **nombre1** por un nuevo nombre, **nombre2**.
- **remove(nombre):** Como la función del DOS **del**, podremos eliminar el archivo indicado en **nombre**.

Detección de final de archivo

La función **feof(puntero_archivo),** siempre deberemos controlar si hemos llegado al final de archivo cuando estemos leyendo, de lo contrario podrían producirse errores de lectura no deseados. Para este fin disponemos de la función **feof( )**. Esta función retorna **0** si no ha llegado al final, y un valor diferente de **0** si lo ha alcanzado.

Pues con esto llegamos al final del tema. Espero que no haya sido muy pesado. No es necesario que te aprendas todas las funciones de memoria. Céntrate sobre todo en las funciones **fputs( )**, **fgets( )**, **fprintf( )**, **fwrite( )** y **fread( )**. Con estas cinco se pueden gestionar los archivos perfectamente.

14. Apéndice

En este capítulo y para finalizar veremos los archivos de cabecera, donde están declaradas las funciones que utilizaremos más a menudo.

14.1 Librería stdio.h

printf
Función: Escribe en la salida estándar con formato.
Sintaxis: printf(formato , arg1 , ...);

scanf
Función: Lee de la salida estándar con formato.
Sintaxis: scanf(formato , arg1 , ...);

puts
Función: Escribe una cadena y salto de línea.
Sintaxis: puts(cadena);

gets
Función: Lee y guarda una cadena introducida por teclado.
Sintaxis: gets(cadena);

fopen
Función: Abre un fichero en el modo indicado.
Sintaxis: pf=fopen(fichero , modo);

fclose
Función: Cierra un fichero cuyo puntero le indicamos.
Sintaxis: fclose(pf);

fprintf
Función: Escribe con formato en un fichero.
Sintaxis: fprintf(pf , formato , arg1 , ...);

fgets
Función: Lee una cadena de un fichero.
Sintaxis: fgets(cadena , longitud , pf);

14.2 Librería stdlib.h

atof
Función: Convierte una cadena de texto en un valor de tipo float.
Sintaxis: numflo=atof(cadena);

atoi
Función: Convierte una cadena de texto en un valor de tipo entero.
Sintaxis: nument=atoi(cadena);

itoa
Función: Convierte un valor numérico entero en una cadena de texto. La base generalmente será 10, aunque se puede indicar otra distinta.
Sintaxis: itoa(número , cadena , base);

exit
Función: Termina la ejecución y abandona el programa.
Sintaxis: exit(estado); /* Normalmente el estado será 0 */

14.3 Librería conio.h

clrscr
Función: Borra la pantalla.
Sintaxis: clrscr( );

clreol
Función: Borra desde la posición del cursor hasta el final de la línea.
Sintaxis: clreol( );

gotoxy
Función: Cambia la posición del cursor a las coordenadas indicadas.
Sintaxis: gotoxy(columna , fila);

textcolor
Función: Selecciona el color de texto (0 - 15).
Sintaxis: textcolor(color);

textbackground
Función: Selecciona el color de fondo (0 - 7).
Sintaxis: textbackground(color);

wherex
Función: Retorna la columna en la que se encuentra el cursor.
Sintaxis: col=wherex( );

wherey
Función: Retorna la fila en la que se encuentra el cursor.
Sintaxis: fila=wherey( );

getch
Función: Lee y retorna un único caracter introducido mediante el teclado por el usuario. No muestra el caracter por la pantalla.
Sintaxis: letra=getch( );

getche
Función: Lee y retorna un único caracter introducido mediante el teclado por el usuario. Muestra el caracter por la pantalla.
Sintaxis: letra=getche( );

14.4 Librería string.h

strlen
Función: Calcula la longitud de una cadena.
Sintaxis: longitud=strlen(cadena);

strcpy
Función: Copia el contenido de una cadena sobre otra.
Sintaxis: strcpy(copia , original);

strcat
Función: Concatena dos cadenas.
Sintaxis: strcat(cadena1 , cadena2);

strcmp
Función: Compara el contenido de dos cadenas. Si **cadena1 < cadena2** retorna un número negativo. Si **cadena1 > cadena2**, un número positivo, y si **cadena1** es igual que **cadena2** retorna **0** (o **NULL**).
Sintaxis: valor=strcmp(cadena1 , cadena2);

14.5 Funciones interesantes

fflush(stdin)
Función: Limpia el buffer de teclado.
Sintaxis: fflush(stdin);
Prototipo: stdio.h

sizeof
Función: Operador que retorna el tamaño en bytes de una variable.
Sintaxis: tamaño=sizeof(variable);

cprintf
Función: Funciona como el printf pero escribe en el color que hayamos activado con la función textcolor sobre el color activado con textbackground.
Sintaxis: cprintf(formato , arg1 , ...);

Prototipo: conio.h

kbhit
Función: Espera la pulsación de una tecla para continuar la ejecución.
Sintaxis: while (!kbhit( )) /* Mientras no pulsemos una tecla... */
Prototipo: conio.h

random
Función: Retorna un valor aleatorio entre 0 y num-1.
Sintaxis: valor=random(num); /* También necesitamos la función randomize */
Prototipo: stdlib.h

randomize
Función: Inicializa el generador de números aleatorios. Deberemos llamarlo al inicio de la función en que utilicemos el random. También deberemos utilizar el include **time.h**, ya que randomize hace una llamada a la función **time**, incluida en este último archivo.
Sintaxis: randomize( );
Prototipo: stdio.h

system
Función: Ejecuta el comando indicado. Esto incluye tanto los comandos del sistema operativo, como cualquier programa que nosotros le indiquemos. Al acabar la ejecución del comando, volverá a la línea de código situada a continuación de la sentencia system.
Sintaxis: system(comando); /* p.ej: system("arj a programa"); */
Prototipo: stdlib.h

15.- Referencias

Aquí finaliza esta guía de **Programación Básica en C**. A lo largo de todas sus páginas se ha intentado describir los métodos, funciones, sentencias, operadores, etc. para poder programar en C.

Naturalmente el C no se acaba aquí, pero esperamos que con lo que se haya aprendido, se pueda comenzar a investigar, de forma que se comprenda el funcionamiento de cualquier código fuente que se presente.

Esta guía y los ejemplos no son más que una pequeña muestra de lo que se puede hacer en C. Por supuesto, como material introductorio, no se especifica con detalle aspectos importantes del lenguaje que todo buen programador debe manejar.

En la siguiente lista de referencias se pueden encontrar muchos materiales sobre C disponibles y sobre los cuales se baso este material. Además sobran los sitios en Internet con información, ejemplos y demás aspectos de este lenguaje.

- http://www.borland.com
- http://www.elrincondelc.com/index.php
- http://www.mundovb.net/mundoc/cursodec.htm