Curso de programación en C.

Miquel A Garcies. Servei de Càlcul i Informatització. UIB.

http://massanella.uib.es/c-calculo/scimgs/cursc/index.html

INDICE

Introducción. 2
Programas. 2
El desarrollo de un programa. 4
Tipos básicos y variables. 5
Funciones. 6
Expresiones y operadores. 7
Conversión de tipos. 9
Control de flujo. 10
Definición y prototipos de funciones. 12
Construcción de tipos. 13
Ámbito de funciones y variables. 17
Punteros. 18
El preprocesador. 21
Funciones de entrada y salida por pantalla. 22
Funciones de asignación de memoria. 25
Funciones matemáticas. 26
Operaciones con ficheros. 27
Bibliografía y referencias. 31

Introducción.

C es un lenguaje de programación de propósito general que ofrece economía sintáctica, control de flujo y estructuras sencillas y un buen conjunto de operadores. No es un lenguaje de muy alto nivel y más bien un lenguaje pequeño, sencillo y no está especializado en ningún tipo de aplicación. Esto lo hace un lenguaje potente, con un campo de aplicación ilimitado y sobre todo, se aprende rápidamente. En poco tiempo, un programador puede utilizar la totalidad del lenguaje.

Este lenguaje ha sido estrechamente ligado al sistema operativo UNIX, puesto que fueron desarrollados conjuntamente. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ninguna máquina concreta. Se le suele llamar *lenguaje de programación de sistemas* debido a su utilidad para escribir compiladores y sistemas operativos, aunque de igual forma se pueden desarrollar cualquier tipo de aplicación.

La base del C proviene del BCPL, escrito por Martin Richards, y del B escrito por Ken Thompson en 1970 para el primer sistema UNIX en un DEC PDP-7. Estos son lenguajes sin tipos, al contrario que el C que proporciona varios tipos de datos. Los tipos que ofrece son caracteres, números enteros y en coma flotante, de varios tamaños. Además se pueden crear tipos derivados mediante la utilización de punteros, vectores, registros y uniones. El primer compilador de C fue escrito por Dennis Ritchie para un DEC PDP-11 y escribió el propio sistema operativo en C.

C trabaja con tipos de datos que son directamente tratables por el hardware de la mayoría de computadoras actuales, como son los caracteres, números y direcciones. Estos tipos de datos pueden ser manipulados por las operaciones aritméticas que proporcionan las computadoras. No proporciona mecanismos para tratar tipos de datos que no sean los básicos, debiendo ser el programador el que los desarrolle. Esto permite que el código generado sea muy eficiente y de ahí el éxito que ha tenido como lenguaje de desarrollo de sistemas. No proporciona otros mecanismos de almacenamiento de datos que no sea el estático y no proporciona mecanismos de entrada ni salida. Ello permite que el lenguaje sea reducido y los compiladores de fácil implementación en distintos sistemas. Por contra, estas carencias se compensan mediante la inclusión de funciones de librería para realizar todas estas tareas, que normalmente dependen del sistema operativo.

Originariamente, el manual de referencia del lenguaje para el gran público fue el libro [1] de Kernighan y Ritchie, escrito en 1977. Es un libro que explica y justifica totalmente el desarrollo de aplicaciones en C, aunque en él se utilizaban construcciones, en la definición de funciones, que podían provocar confusión y errores de programación que no eran detectados por el compilador. Como los tiempos cambian y las necesidades también, en 1983 ANSI establece el comité X3J11 para que desarrolle una definición moderna y comprensible del C. El estándar está basado en el manual de referencia original de 1972 y se desarrolla con el mismo espíritu de sus creadores originales. La primera versión de estándar se publicó en 1988 y actualmente todos los compiladores utilizan la nueva definición. Una aportación muy importante de ANSI consiste en la definición de un conjunto de librerías que acompañan al compilador y de las funciones contenidas en ellas. Muchas de las operaciones comunes con el sistema operativo se realizan a través de estas funciones. Una colección de ficheros de encabezamiento, *headers*, en los que se definen los tipos de datos y funciones incluidas en cada librería. Los programas que utilizan estas bibliotecas para interactuar con el sistema operativo obtendrán un comportamiento equivalente en otro sistema.

Programas.

La mejor forma de aprender un lenguaje es programando con él. El programa más sencillo que se puede escribir en C es el siguiente:

```
main()
{
}
```

Como nos podemos imaginar, este programa no hace nada, pero contiene la parte más importante de cualquier programa C y además, es el más pequeño que se puede escribir y que se compile correctamente. En el se define la función `main`, que es la que ejecuta el sistema operativo al llamar a un programa C. El nombre de una función C siempre va seguida de paréntesis, tanto si tiene argumentos como si no. La definición de la función está formada por un bloque de sentencias, que esta encerrado entre llaves `{}`.

Un programa algo más complicado es el siguiente:

```
#include <stdio.h>
main()
{
printf("Hola amigos!\n");
}
```

Con el visualizamos el mensaje `Hola amigos!` en el terminal. En la primera línea indica que se tengan en cuenta las funciones y tipos definidos en la librería `stdio` (*standard input/output*). Estas definiciones se encuentran en el fichero *header* `stdio.h`. Ahora, en la función `main` se incluye una única sentencia que llama a la función `printf`. Esta toma como argumento una cadena de caracteres, que se imprimen van encerradas entre dobles comillas `" "`. El símbolo `\n` indica un cambio de línea. Hay un grupo de símbolos, que son tratados como caracteres individuales, que especifican algunos caracteres especiales del código ASCII. Los más importantes son:

`\a`	*alert*
`\b`	*backspace*
`\f`	*formfeed*
`\n`	*newline*
`\r`	*carriage return*
`\t`	*horizontal tab*
`\v`	*vertical tab*
`\\`	*backslash*
`\'`	*single quote*
`\"`	*double quote*
`\OOO`	visualiza un carácter cuyo código ASCII es OOO en octal.
`\xHHH`	visualiza un carácter cuyo código ASCII es HHH en hexadecimal.

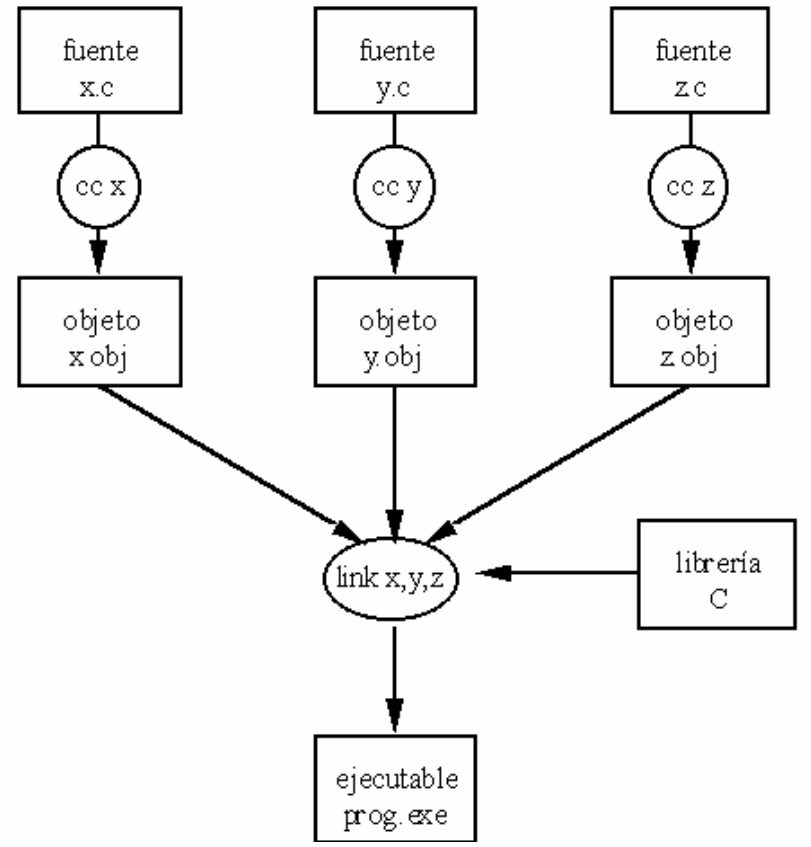
Las funciones de entrada y salida y los formatos utilizados los explicaremos con más detalle en otro capítulo.

El desarrollo de un programa.

Un programa C puede estar formado por diferentes módulos o fuentes. Es conveniente mantener los fuentes de un tamaño no muy grande, para que la compilación sea rápida. También, al dividirse un programa en partes, puede facilitar la legibilidad del programa y su estructuración. Los diferentes fuentes son compilados de forma separada, únicamente los fuentes que han sido modificados desde la última compilación, y después combinados con las librerías necesarias para formar el programa en su versión ejecutable.

Los comandos necesarios para compilar, *linkar* y ejecutar un programa dependen del sistema operativo y debemos dirigirnos a los manuales correspondientes para conocer la sintaxis exacta. Como forma más común podemos dar la siguiente:

```
cc prog
cc modulo1, modulo2
link prog, modulo2, modulo2
prog
```



Creación de un programa en C.

Tipos básicos y variables.

Los tipos de datos básicos definidos por C son caracteres, números enteros y números en coma flotante. Los caracteres son representados por `char`, los enteros por `short`, `int`, `long` y los números en coma flotante por `float` y `double`. Los tipos básicos disponibles y su tamaño son:

`char`	Carácter	(normalmente 8 bits)
`short`	Entero corto con signo	(normalmente 16 bits)
`int`	Entero con signo	(depende de la implementación)
`unsigned`	Entero sin signo	(depende de la implementación)
`long`	Entero largo con signo	(normalmente 32 bits)
`float`	Flotante simple	(normalmente 32 bits)
`double`	Flotante doble	(normalmente 64 bits)

La palabra `unsigned` en realidad es un modificador aplicable a tipos enteros, aunque si no se especifica un tipo se supone `int`. Un modificador es una palabra clave de C que indica que una variable, o función, no se comporta de la forma normal. Hay también un modificador `signed`, pero como los tipos son por defecto con signo, casi no se utiliza.

Las variables son definidas utilizando un identificador de tipo seguido del nombre de la variable. Veamos el siguiente programa:

```
#include <stdio.h>
main()
{
float cels, farh;

farh = 35.0;
cels = 5.0 * ( farh - 32.0 ) / 9.0;
printf("-> %f F son %f C\n", farh, cels );
}
```

En el programa anterior se definen dos variables `float`, se asigna un valor a la primera y se calcula la segunda mediante una expresión aritmética. Las asignaciones en C también son una expresión, por lo que se pueden utilizar como parte de otra expresión, pero según que prácticas de este tipo no son muy recomendables ya que reducen la legibilidad del programa. En la instrucción `printf`, el símbolo `%f` indica que se imprime un número en coma flotante.

Hay un tipo muy importante que se representa por `void` que puede significar dos cosas distintas, según su utilización. Puede significar nada, o sea que si una función devuelve un valor de tipo `void` no

devuelve ningún resultado, o puede significar cualquier cosa, como puede ser un puntero a `void` es un puntero genérico a cualquier tipo de dato. Más adelante veremos su utilización.

Funciones.

Un programa C está formado por un conjunto de funciones que al menos contiene la función `main`. Una función se declara con el nombre de la función precedido del tipo de valor que retorna y una lista de argumentos encerrados entre paréntesis. El cuerpo de la función está formado por un conjunto de declaraciones y de sentencias comprendidas entre llaves. Veamos un ejemplo de utilización de funciones:

```
#include <stdio.h>
#define VALOR 5
#define FACT 120

int fact_i ( int v )
{
int r = 1, i = 0;

while ( i <= v ) {
r = r * i;
i = i + 1;
}
return r;
}

int fact_r ( int v )
{
if ( v == 0 )
return 1;
else
return v * fact_r(v-1);
}

main()
{
int r, valor = VALOR;

if ( (r = fact_i(valor)) != fact_r(valor) )
printf("Codificación errónea!!.\n");
else
if ( r == FACT )
printf("Codificación correcta.\n");
else
printf("Algo falla!!.\n");
}
```

Se definen dos funciones, `fact_i` y `fact_r`, además de la función `main`. Ambas toman como parámetro un valor entero y devuelven otro entero. La primera calcula el factorial de un número de forma iterativa, mientras que la segunda hace lo mismo de forma recursiva.

Todas las líneas que comienzan con el símbolo `#` indican una directiva del precompilador. Antes de realizar la compilación en C se llama a un precompilador cuya misión es procesar el texto y realizar ciertas sustituciones textuales. Hemos visto que la directiva `#include` incluye el texto contenido en un fichero en el fuente que estamos compilando. De forma parecida, `#define` *nombre texto* sustituye todas las apariciones de *nombre* por *texto*. Así, en el fuente, la palabra VALOR se sustituye por el número 5.

El valor que debe devolver una función se indica con la palabra `return`. La evaluación de la expresión debe dar una valor del mismo tipo de dato que el que se ha definido como resultado. La declaración de una variable puede incluir una inicialización en la misma declaración. Se debe tener muy en cuenta que en C todos los argumentos son pasados 'por valor'. No existe el concepto de paso de parámetros 'por variable' o 'por referencia'. Veamos un ejemplo:

```
int incr ( int v )
{
return v + 1;
}

main()
{
int a, b;

b = 3;
a = incr(b);
/*
a = 4 mientras que
b = 3. No ha cambiado después de la llamada.
*/
}
```

En el ejemplo anterior el valor del parámetro de la función `incr`, aunque se modifique dentro de la función, no cambia el valor de la variable `b` de la función `main`. Todo el texto comprendido entre los caracteres `/*` y `*/` son comentarios al programa y son ignorados por el compilador. En un fuente C los comentarios no se pueden anidar.

Expresiones y operadores.

Los distintos operadores permiten formar expresiones tanto aritméticas como lógicas. Los operadores aritméticos y lógicos son:

`+`, `-`	suma, resta
`++`, `--`	incremento, decremento

`*`, `/`, `%`	multiplicación, división, módulo
>>, <<	rotación de bits a la derecha, izquierda.
`&`	AND *booleano*
`|`	OR *booleano*
`^`	EXOR *booleano*
~	complemento a 1
!	complemento a 2, NOT lógico
==, `!=`	igualdad, desigualdad
`&&`, `||`	AND, OR lógico
<, <=	menor, menor o igual
>, >=	mayor, mayor o igual

En estos operadores deben tenerse en cuenta la precedencia de operadores y las reglas de asociatividad, que son las normales en la mayoría de lenguajes. Se debe consultar el manual de referencia para obtener una explicación detallada. Además hay toda una serie de operadores aritméticos con asignación, como pueden ser += y ^=.

En la evaluación de expresiones lógicas, los compiladores normalmente utilizan técnicas de evaluación rápida. Para decidir si una expresión lógica es cierta o falsa muchas veces no es necesario evaluarla completamente. Por ejemplo una expresión formada <exp1> `||` <exp2>, el compilador evalúa primero <exp1> y si es cierta, no evalúa <exp2>. Por ello se deben evitar construcciones en las que se modifiquen valores de datos en la propia expresión, pues su comportamiento puede depender de la implementación del compilador o de la optimización utilizada en una compilación o en otra. Estos son errores que se pueden cometer fácilmente en C ya que una asignación es también una expresión.

Debemos evitar:

```
if (( x++ > 3 ) || ( x < y ))
```

y escribir en su lugar:

```
x++;
if (( x > 3 ) || ( x < y ))
```

Hay un tipo especial de expresión en C que se denomina expresión condicional y está representada por los operadores `?` `:` . Su utilización es como sigue: <e> ? <x> `:` <y>. Se evalúa si *e* entonces *x*; si no, *y*.

```
int mayor ( int a, int b )
{
return ( a > b ) ? TRUE : FALSE;
}

waste_time ()
{
float a, b = 0.0;

( b > 0.0 ) ? sin(M_PI / 8) : cos(M_PI / 4);
}
```

Conversión de tipos.

Cuando escribimos una expresión aritmética a+b, en la cual hay variables o valores de distintos tipos, el compilador realiza determinadas conversiones antes de que evalúe la expresión. Estas conversiones pueden ser para 'aumentar' o 'disminuir' la precisión del tipo al que se convierten los elementos de la expresión. Un ejemplo claro, es la comparación de una variable de tipo `int` con una variable de tipo `double`. En este caso, la de tipo `int` es convertida a `double` para poder realizar la comparación.

Los tipos pequeños son convertidos de la forma siguiente: un tipo `char` se convierte a `int`, con el modificador `signed` si los caracteres son con signo, o `unsigned` si los caracteres son sin signo. Un `unsigned char` es convertido a `int` con los bits más altos puestos a cero. Un `signed char` es convertido a `int` con los bits más altos puestos a uno o cero, dependiendo del valor de la variable.

Para los tipos de mayor tamaño: si un operando es de tipo `double`, el otro es convertido a `double`. Si un operando es de tipo `float`, el otro es convertido a `float`. Si un operando es de tipo `unsigned long`, el otro es convertido a `unsigned long`. Si un operando es de tipo `long`, el otro es convertido a `long`. Si un operando es de tipo `unsigned`, el otro es convertido a `unsigned`. Si no, los operandos son te tipo `int`.

Una variable o expresión de un tipo se puede convertir explícitamente a otro tipo, anteponiéndole el tipo entre paréntesis.

```
void cambio_tipo (void)
{
float a;
int b;

b = 10;
a = 0.5;

if ( a <= (float) b )
menor();
}
```

Control de flujo.

La sentencia de control básica es `if (<e>) then <s> else <t>`. En ella se evalúa una expresión condicional y si se cumple, se ejecuta la sentencia *s*; si no, se ejecuta la sentencia *t*. La segunda parte de la condición, `else <t>`, es opcional.

```
int cero ( double a )
{
if ( a == 0.0 )
return (TRUE);
else
return (FALSE);
}
```

En el caso que <e> no sea una expresión condicional y sea aritmética, se considera falso si vale 0; y si no, verdadero. Hay casos en los que se deben evaluar múltiples condiciones y únicamente se debe evaluar una de ellas. Se puede programar con un grupo de sentencias `if then else` anidadas, aunque ello puede ser farragoso y de complicada lectura. Para evitarlo nos puede ayudar la sentencia `switch`. Su utilización es:

```
switch (valor) {
case valor1: <sentencias>
case valor2: <sentencias>
...
default: <sentencias>
}
```

Cuando se encuentra una sentencia `case` que concuerda con el valor del `switch` se ejecutan las sentencias que le siguen y todas las demás a partir de ahí, a no ser que se introduzca una sentencia `break` para salir de la sentencia `switch`. Por ejemplo,

```
ver_opcion ( char c )
{
switch(c){
case 'a': printf("Op A\n"); break;
case 'b': printf("Op B\n"); break;
case 'c':
case 'd': printf("Op C o D\n"); break;
default: printf("Op ?\n");
}
```

```
}
```

Otras sentencias de control de flujo son las que nos permiten realizar iteraciones sobre un conjunto de sentencias. En C tenemos tres formas principales de realizar iteraciones. La sentencia `while (<e>)` <s> es seguramente la más utilizada. La sentencia, o grupo de sentencias <s> se ejecuta mientras la evaluación de la expresión <e> sea verdadera.

```
long raiz ( long valor )
{
long r = 1;

while ( r * r <= valor )
r++;

return r;
}
```

Otra sentencia iterativa, que permite inicializar los controles del bucle se la sentencia `for (` <i>`;` <e>`;` <p> `)` <s>. La sentencia `for` se puede escribir también como:

```
<i>;

while ( <e> ) {

<s>;

<p>;

}
```

El ejemplo anterior se podría escribir como:

```
long raiz ( long valor )
{
long r;

for ( r = 1; r * r <= valor; r++ )
;

return r;
}
```

Una variación de la sentencia `while` es: `do` <s> `while (` <e> `);` En ella la sentencia se ejecuta al menos una vez, antes de que se evalúe la expresión condicional.

Otras sentencias interesantes, aunque menos utilizadas son `break` y `continue`. `break` provoca que se termine la ejecución de una iteración o para salir de la sentencia `switch`, como ya hemos visto. En cambio, `continue` provoca que se comience una nueva iteración, evaluándose la expresión de control. Veamos dos ejemplos:

```
void final_countdown (void)
{
int count = 10;

while ( count-- > 1 ) {
if ( count == 4 )
start_engines();
if ( status() == WARNING )
break;
printf("%d ", count );
}
if ( count == 0 ) {
launch();
printf("Shuttle launched\n");
}
else {
printf("WARNING condition received.\n");
printf("Count held at T - %d\n", count );
}
}

d2 ()
{
int f;

for ( f = 1; f <= 50; f++ ) {
if ( f % 2 == 0 )
continue;
printf("%d", f );
}
}
```

Definición y prototipos de funciones.

Los programas sencillos, como los ejemplo planteados hasta ahora, normalmente no necesitan un nivel de estructuración elevado. Pero cuando éstos crecen un poco necesitamos estructurarlos adecuadamente para mantenerlos legibles, facilitar su mantenimiento y para poder reutilizar ciertas porciones de código. El mecanismo C que nos permite esto son las funciones. Con los compiladores, los fabricantes nos proporcionan un conjunto importante de funciones de librería. A veces, nos puede interesar construir nuestras propias librerías. Ya hemos utilizado funciones, pero veamos cómo debemos definirlas.

Los prototipos de funciones son una característica clave de la recomendación ANSI del C. Un prototipo es una declaración que toma la forma:

tipo_resultado nombre_función (*tipo_parámetro nombre_parámetro ...*) ;

Aquí tenemos varios ejemplos:

```
int fact_i ( int v );
int mayor ( int a, int b );
int cero ( double a );
long raiz ( long valor );
void final_countdown ( void );
int main ( int argc, char **argv );
```

Observando el prototipo de una función podemos decir exactamente que tipo de parámetros necesita y que resultado devuelve. Si una función tiene como argumento `void`, quiere decir que no tiene argumentos, al igual que si el resultado es `void`, no devuelve ningún valor.

En la vieja definición de Kernighan y Ritchie el tipo que devolvía una función se declaraba únicamente si era distinto de `int`. Similarmente, los parámetros eran declarados en el cuerpo de la función, en lugar de utilizar la lista de parámetros. Por ejemplo:

```
mayor ( a, b )
int a;
int b;
{
...
}
```

Las funciones al viejo estilo se compilan correctamente en muchos compiladores actuales. Por contra, proporcionan menos información sobre sus parámetros y errores que afecten al tipo de parámetros de llamada a las funciones no pueden ser detectados automáticamente. Por tanto, la declaración de una función debe escribirse igual que su prototipo pero sin el punto y coma final. El cuerpo de la función le sigue encerrado entre llaves.

En un programa que esté formado por distintas partes bien diferenciadas es conveniente utilizar múltiples ficheros fuente. Cada fuente agrupa las funciones semejantes, como por ejemplo en un compilador podríamos tener un fuente para el análisis léxico, otro para el sintáctico y otro para la generación de código. Pero en un fuente necesitaremos funciones que se han definido en otro. Para ello, escribiremos un fichero de cabecera (*header*), que contendrá las declaraciones que podemos necesitar en otros fuente. Así, en el fuente que implementa el analizador sintáctico pondremos una línea `#include "lexic.h"`. De esta forma al compilar el módulo sintáctico tendremos todos los prototipos de las funciones del léxico y el compilador podrá detectar malas utilizaciones de las funciones allí definidas.

Construcción de tipos.

Los datos del mundo real, normalmente no están formados por variables escalares de tipos los tipos básicos. Por ejemplo, nos puede interesar saber cuántos módulos en C hemos escrito cada semana, a lo

largo del año. O también nos interesa tener los datos de cada planeta del Sistema Solar, masa, posición, velocidad y aceleración, para un programa de simulación de la ley de gravitación de Newton. Para resolver el primer caso, C nos permite declarar una variable que sea de tipo vector. Para el segundo, podemos definir un registro para cada elemento.

Un vector es una porción de memoria que es utilizada para almacenar un grupo de elementos del mismo tipo Un vector se declara: *tipo nombre* `[`*tamaño*`];`. Por ejemplo, `int modulo[52];`. Aquí 'modulo' es un vector de 52 elementos enteros.

```
main()
{
int f, modulo[52];

for ( f = 0; f < 52; f++ )
modulo[f] = 0;
...
}
```

Cada elemento de un vector es accedido mediante un número de índice y se comporta como una variable del tipo base del vector. Los elementos de un vector son accedidos por índices que van desde 0 hasta N-1 para un vector de N elementos. Los elementos de un vector pueden ser inicializados en la misma declaración:

```
char vocal[5] = { 'a', 'e', 'i', 'o', 'u' };
float n_Bode[5] = { 0.4, 0.7, 1, 1.6, 2.8 };
```

También podemos definir vectores multidimensionales. C no impone ninguna limitación al número de dimensiones de un vector. Existe, en cambio, la limitación del tamaño de memoria que podamos utilizar en nuestro ordenador. Por ejemplo, para la declaración de un vector multidimensional podemos escribir:

```
int video[25][80][2];
```

El tamaño de la variable `video` es proporcional al tamaño del tipo `int` y al tamaño de cada dimensión. Existe un operador C que nos permite obtener el tamaño de un tipo o de una variable. Este es `sizeof()` y nos proporciona el tamaño en bytes.

```
if ( sizeof(video) == 80 * 25 * 2 * sizeof(int) )
printf("OK!\n");
else
printf("Algo no funciona.\n");
```

Un tipo vector muy utilizado es la cadena de caracteres (*string*). Si queremos asignar espacio para un string podemos hacer:

```
char nombre[60], direccion[80];
```

Es un vector C pero con la particularidad de que el propio lenguaje utiliza un carácter especial como marca de final de string. Así en un vector de caracteres de tamaño N podremos almacenar una cadena de N-1 caracteres, cuyo último carácter estará en la posición N-2 y la marca de final de string en la N-1. Veamos un ejemplo:

```
char servei[6] = "SCI";
```

La posición 0 contiene el carácter 'S'; la 1 el 'C'; la 2 el 'I'; la 3 el '\0', marca de final de string. El resto de componentes no están definidas. En la inicialización de strings no se debe indicar el final; ya lo hace el compilador. Para la manipulación de cadenas de caracteres ANSI proporciona el fichero `string.h` que contiene las declaraciones de un conjunto de funciones proporcionadas con la librería del compilador.

Un registro agrupa distintos tipos de datos en una misma estructura. Los registros son definidos de la forma:

`struct` *nombre* `{` *lista de declaraciones* `};`

Los campos de cada registro pueden ser tipos básicos u otros registros. Por ejemplo:

```
struct planeta {
struct 3D r, v, a;
double masa;
char nom[10];
};

struct 3D {
double x,y,z;
};
```

Los campos de cada registro son accesibles mediante el nombre del registro seguido de punto y el nombre del campo, como por ejemplo `venus.r.x = 1.0;`. Cada campo se comporta como lo hace su tipo básico. C no proporciona mecanismos de inicialización ni copia de registros, por lo que debe ser el programador el que los implemente.

A veces los datos se ajustan a series ordenadas en las cuales un elemento sigue, o precede, a otro. Un caso típico son los días de la semana. Si se desea realizar iteraciones con los días de la semana una forma es, por ejemplo, asignar un número a cada día con `#define`. C proporciona un mecanismo compacto para

realizar esto; son las enumeraciones. Una enumeración toma la forma: `enum` nombre `{` lista de elementos `};`. Veamos un ejemplo:

```
void planning ( void )
{
enum diasemana { lunes, martes, miercoles,
jueves, viernes, sabado, domingo };
int dia;

for ( dia = lunes; dia <= viernes; dia++ )
trabajar(dia);
if ( dia == sabado )
salir();
}
```

A cada elemento de la enumeración se le asigna un valor consecutivo, comenzando por 0. Si se desea que el valor asignado sea distinto se puede hacer de la siguiente forma:

```
enum puntos { t_6_25 = 3, t_zona = 2, t_libre = 1 };
```

Muchas veces es conveniente renombrar tipos de datos para que la escritura del programa se nos haga más sencilla y la lectura también. Esto se puede conseguir con la palabra `typedef`. Con ella damos un nombre a cierto tipo, o combinación de ellos.

```
typedef struct planeta PLANETA;

PLANETA mercurio, venus, tierra, marte;
```

Al igual que podemos inicializar las variables de tipos básicos en la misma declaración, también lo podemos hacer con los registros. Los valores de cada campo de un registro van separados por comas y encerrados entre llaves.

```
PLANETA mercurio = { { 0.350, 0, 0 },
{ 0, 0, 0 },
{ 0, 0, 0 },
100, "Mercurio" };
```

Ámbito de funciones y variables.

El ámbito, o visibilidad, de una variable nos indica en que lugares del programa está activa esa variable. Hasta ahora, en los ejemplos que hemos visto, se han utilizado variables definidas en el cuerpo de funciones. Estas variables se crean en la memoria del ordenador cuando se llama a la función y se destruyen cuando la función termina de ejecutarse. Es necesario a veces, que una variable tenga un valor que pueda ser accesible desde todas las funciones de un mismo fuente, e incluso desde otros fuentes.

En C, el ámbito de las variables depende de dónde han sido declaradas y si se les ha aplicado algún modificador. Una variable definida en una función es, por defecto, una variable local. Esto es, que sólo existe y puede ser accedida dentro de la función. Para que una variable sea visible desde una función cualquiera del mismo fuente debe declararse fuera de cualquier función. Esta variable sólo será visible en las funciones definidas después de su declaración. Por esto, el lugar más común para definir las variables globales es antes de la definición de ninguna función. Por defecto, una variable global es visible desde otro fuente. Para definir que existe una variable global que está definida en otro fuente tenemos que anteponer la palabra `extern` a su declaración. Esta declaración únicamente indica al compilador que se hará referencia a una variable declarada en un módulo distinto al que se compila.

Las variables locales llevan implícito el modificador `auto`. Este indica que se crean al inicio de la ejecución de la función y se destruyen al final. En un programa sería muy ineficiente en términos de almacenamiento que se crearan todas las variables al inicio de la ejecución. Por contra, en algunos casos es deseable. Esto se consigue anteponiendo el modificador `static` a una variable local. Si una función necesita una variable que únicamente sea accedida por la misma función y que conserve su valor a través de sucesivas llamadas, es el caso adecuado para que sea declarada local a la función con el modificador `static`. El modificador `static` se puede aplicar también a variables globales. Una variable global es por defecto accesible desde cualquier fuente del programa. Si, por cualquier motivo, se desea que una de estas variables no se visible desde otro fuente se le debe aplicar el modificador `static`. Lo mismo ocurre con las funciones. Las funciones definidas en un fuente son utilizables desde cualquier otro. En este caso conviene incluir los prototipos de las funciones del otro fuente. Si no se desea que alguna función pueda ser llamada desde fuera del fuente en la que está definida se le debe anteponer el modificador `static`.

```
void contar ( void )
{
static long cuenta = 0;

cuenta++;
printf("Llamada %ld veces\n", cuenta );
}
```

Un modificador muy importante es `const`. Con él se pueden definir variables cuyo valor debe permanecer constante durante toda la ejecución del programa. También se puede utilizar con argumentos de funciones. En esta caso se indica que el argumento en cuestión es un parámetro y su valor no debe ser modificado. Si al programar la función, modificamos ese parámetro, el compilador nos indicará el error.

```
#define EULER 2.71828
const double pi = 3.14159;

double lcercle ( const double r )
{
```

```
return 2.0 * pi * r;
}

double EXP ( const double x )
{
return pow ( EULER, x );
}

double sinh ( const double x )
{
return (exp(x) - exp(-x)) / 2.0;
}
```

Debemos fijarnos que en el ejemplo anterior `pi` es una variable, la cual no podemos modificar. Por ello `pi` sólo puede aparecer en un único fuente. Si la definimos en varios, al linkar el programa se nos generará un error por tener una variable duplicada. En el caso en que queramos acceder a ella desde otro fuente, debemos declararla con el modificador `extern`.

Otro modificador utilizado algunas veces es el `register`. Este modificador es aplicable únicamente a variables locales e indica al compilador que esta variable debe ser almacenada permanentemente en un registro del procesador del ordenador. Este modificador es herencia de los viejos tiempos, cuando las tecnologías de optimización de código no estaban muy desarrolladas y se debía indicar qué variable era muy utilizada en la función. Hoy en día casi todos los compiladores realizan un estudio de qué variables locales son las más adecuadas para ser almacenadas en registros, y las asignan automáticamente. Con los compiladores modernos se puede dar el caso de que una declaración `register` inadecuada disminuya la velocidad de ejecución de la función, en lugar de aumentarla. Por ello, hoy en día, la utilización de este modificador está en desuso, hasta el punto de que algunos compiladores lo ignoran. Se debe tener en cuenta que de una variable declarada como `register` no se puede obtener su dirección, ya que está almacenada en un registro y no en memoria.

Punteros.

Cada variable de un programa tiene una dirección en la memoria del ordenador. Esta dirección indica la posición del primer byte que la variable ocupa. En el caso de una estructura es la dirección del primer campo. En los ordenadores actuales la dirección de inicio se considera la dirección baja de memoria. Como en cualquier caso las variables son almacenadas ordenadamente y de una forma predecible, es posible acceder a estas y manipularlas mediante otra variables que contenga su dirección. A este tipo de variables se les denomina punteros.

Los punteros C son el tipo más potente y seguramente la otra clave del éxito del lenguaje. La primera ventaja que obtenemos de los punteros es la posibilidad que nos dan de poder tratar con datos de un tamaño arbitrario sin tener que moverlos por la memoria. Esto puede ahorrar un tiempo de computación muy importante en algunos tipos de aplicaciones. También permiten que una función reciba y cambie el valor de una variable. Recordemos que todas las funciones C únicamente aceptan parámetros por valor. Mediante un puntero a una variable podemos modificarla indirectamente desde una función cualquiera.

Un puntero se declara de la forma: *tipo `*`nombre;*

```
float *pf;
PLANETA *pp;
char *pc;
```

Para manipular un puntero, como variable que es, se utiliza su nombre; pero para acceder a la variable a la que apunta se le debe preceder de `*`. A este proceso se le llama indirección. Accedemos indirectamente a una variable. Para trabajar con punteros existe un operador, `&`, que indica 'dirección de'. Con él se puede asignar a un puntero la dirección de una variable, o pasar como parámetro a una función.

```
void prova_punter ( void )
{
long edat;
long *p;

p = &edat;
edad = 50;
printf("La edat es %ld\n", edat );
*p = *p / 2;
printf("La edat es %ld\n", edat );
}

void imprimir_string ( char string[] )
{
char *p;

for ( p = string; *p != '\0'; p++ )
imprimir_char(*p);
}
```

Los punteros también se pueden utilizar con los registros. Para ello se utiliza la notación `->` en lugar del punto que utilizábamos anteriormente. Si `p` es un puntero a `PLANETA`, y queremos conocer su masa, debemos escribir `p->masa`. Un puntero se puede utilizar para almacenar la dirección de cualquier tipo de datos, tanto simple como un vector, como un registro. De cómo lo definimos y lo utilizamos depende su comportamiento. Las componentes de un vector, por ejemplo pueden ser referenciadas por un puntero al tipo de cada componente. Veamos un ejemplo:

```
#define N_PLA 9

static PLANETA SSolar[N_PLA];

void init_SistemaSolar ( void )
{
PLANETA *p;

for ( p = SSolar; p < SSolar[N_PLA]; p++ )
init_planeta(p);
}
```

```
void init_planeta ( PLANETA *p )
{
p->masa = 0;
p->nom = "";
init_co(&(p->r));
init_co(&(p->v));
init_co(&(p->a));
}

void init_co ( struct co *c )
{
c->x = c->y = c->z = 0;
}
```

Definimos un vector de `N_PLA` componentes de tipo `PLANETA`. Este tipo está formado por un registro. Vemos que en la función de inicialización del vector el puntero a la primera componente se inicializa con el nombre del vector. Esto es una característica importante de C. La dirección de la primera componente de un vector se puede direccionar con el nombre del vector. Esto es debido a que en la memoria del ordenador, los distintos elementos están ordenados de forma ascendente. Así, `SSolar` se puede utilizar como `&SSolar[0]`. A cada iteración llamamos a una función que nos inicializará los datos de cada planeta. A esta función le pasamos como argumento el puntero a la componente en curso para que, utilizando la notación `->`, pueda asignar los valores adecuados a cada campo del registro. Debemos fijarnos en el incremento del puntero de control de la iteración, `p++`. Con los punteros se pueden realizar determinadas operaciones aritméticas aunque, a parte del incremento y decremento, no son muy frecuentes. Cuando incrementamos un puntero el compilador le suma la cantidad necesaria para que apunte al siguiente elemento de la memoria. Debemos fijarnos que esto es aplicable sólo siempre que haya distintas variables o elementos situados consecutivamente en la memoria, como ocurre con los vectores.

De forma similar se pueden utilizar funciones que tengan como parámetros punteros, para cambiar el valor de una variable. Veamos:

```
void intercambio ( void )
{
int a, b;

a = 1;
b = 2;
swap( &a, &b );
printf(" a = %d b = %d\n", a, b );
}

void swap ( int *x, int *y )
{
int tmp;

tmp = *x;
```

```
*x = *y;
*y = tmp;
}
```

La sintaxis de C puede, a veces, provocar confusión. Se debe distinguir lo que es un prototipo de una función de lo que es una declaración de una variable. Así mismo, un puntero a un vector de punteros, etc...

`int f1();`	función que devuelve un entero
`int *p1;`	puntero a entero
`int *f2();`	función que devuelve un puntero a entero
`int (*pf)(int);`	puntero a función que toma y devuelve un entero
`int (*pf2)(int *pi);`	puntero a función que toma un puntero a entero y devuelve un entero
`int a[3];`	vector de tres enteros
`int *ap[3];`	vector de tres punteros a entero
`int *(ap[3]);`	vector de tres punteros a entero
`int (*pa)[3];`	puntero a vector de tres enteros
`int (*apf[5])(int *pi);`	vector de 5 punteros a función que toman un puntero a entero y devuelven un entero

En los programas que se escriban se debe intentar evitar declaraciones complejas que dificulten la legibilidad del programa. Una forma de conseguirlo es utilizando `typedef` para redefinir/renombrar tipos.

```
typedef int *intptr;
typedef intptr (*fptr) ( intptr );
fptr f1, f2;
```

El preprocesador.

El preprocesador es una parte del compilador que se ejecuta en primer lugar, cuando se compila un fuente C y que realiza unas determinadas operaciones, independientes del propio lenguaje C. Estas operaciones se realizan a nivel léxico y son la inclusión de otros textos en un punto del fuente, realizar sustituciones o eliminar ciertas partes del fuente. Debemos tener en cuenta que el preprocesador trabaja únicamente con el texto del fuente y no tiene en cuenta ningún aspecto sintáctico ni semántico del lenguaje.

El control del preprocesador se realiza mediante determinadas directivas incluidas en el fuente. Una directiva es una palabra que interpreta el preprocesador, que siempre va precedida por el símbolo # y que está situada a principio de línea.

La directiva `#define` se utiliza para definir una macro. Las macros proporcionan principalmente un mecanismo para la sustitución léxica. Una macro se define de la forma `#define` *id secuencia*. Cada ocurrencia de *id* en el fuente es sustituida por *secuencia*. Puede definirse una macro sin una secuencia de caracteres. Una macro se puede "indefinir" mediante la directiva `#undef`.

```
#define MSG01 "SCI-I-START: Starting system kernel\n"
#define MSG02 "SCI-I-STOP: Stopping system kernel\n"

void print_msg ( void )
{
if ( check_state() == START )
printf(MSG01);
else
printf(MSG02);
}
```

El estado de una macro, si está definida o no, se puede comprobar mediante las directivas `#ifdef` y `#ifndef`. Estas dos directivas se deben completar con una `#endif` y, el texto comprendido entre ambas es procesado si la macro está definida. Todas las directivas deben ser completadas en el mismo fuente y pueden ser anidadas.

```
#ifndef M_PI
#define M_PI 3.1415927
#endif
```

El preprocesador nos permite también incluir también otros ficheros en un fuente C. Esto se consigue con la directiva `#include`. Esta puede tomar tres formas: `#include` <*fichero*>, `#include` "*fichero*" y `#include` *macro*. La diferencia entre la primera y la segunda está en el lugar dónde se buscará el fichero en cuestión. Normalmente se utiliza la primera para ficheros proporcionados por la librería del compilador y la segunda para ficheros creados por el programador.

Funciones de entrada y salida por pantalla.

En este apartado y los siguientes vamos a ver algunas de las funciones más importantes que nos proporcionan las librerías definidas por ANSI y su utilización. Como hemos visto hasta ahora, el lenguaje C no proporciona ningún mecanismo de comunicación ni con el usuario ni con el sistema operativo. Ello es realizado a través de las funciones de librería proporcionadas por el compilador.

El fichero de declaraciones que normalmente más se utiliza es el `stdio.h`. Vamos a ver algunas funciones definidas en él.

Una función que ya hemos utilizado y que, ella y sus variantes, es la más utilizadas para la salida de información es `printf`. Esta permite dar formato y enviar datos a la salida estándar del sistema operativo.

```
#include <stdio.h>

int printf ( const char *format [, argumentos, ...] );
```

Acepta un string de formato y cualquier número de argumentos. Estos argumentos se aplican a cada uno de los especificadores de formato contenidos en *format*. Un especificador de formato toma la forma %[*flags*][*width*][.*prec*][h|l] *type*. El tipo puede ser:

`d`, `i`	entero decimal con signo
`o`	entero octal sin signo
`u`	entero decimal sin signo
`x`	entero hexadecimal sin signo (en minúsculas)
`X`	entero hexadecimal sin signo (en mayúsculas)
`f`	coma flotante en la forma [-]dddd.dddd
`e`	coma flotante en la forma [-]d.dddd e[+/-]ddd
`g`	coma flotante según el valor
`E`	como `e` pero en mayúsculas
`G`	como `g` pero en mayúsculas
`c`	un carácter
`s`	cadena de caracteres terminada en `'\0'`
`%`	imprime el carácter %
`p`	puntero

Los *flags* pueden ser los caracteres:

`+`	siempre se imprime el signo, tanto + como -
`-`	justifica a la izquierda el resultado, añadiendo espacios al final
blank	si es positivo, imprime un espacio en lugar de un signo +

#	especifica la forma alternativa

En el campo *width* se especifica la anchura mínima de la forma:

n	se imprimen al menos *n* caracteres.
0*n*	se imprimen al menos *n* caracteres y si la salida es menor, se anteponen ceros
*	la lista de parámetros proporciona el valor

Hay dos modificadores de tamaño, para los tipos enteros:

l	imprime un entero `long`
h	imprime un entero `short`

Otra función similar a `printf` pero para la entrada de datos es `scanf`. Esta toma los datos de la entrada estándar del sistema operativo. En este caso, la lista de argumentos debe estar formada por punteros, que indican dónde depositar los valores.

```
#include <stdio.h>

int scanf ( const char *format [, argumentos, ...] );
```

Hay dos funciones que trabajan con strings. La primera lee un string de la entrada estándar y la segunda lo imprime en el dispositivo de salida estándar.

```
#include <stdio.h>
char *gets ( char *s );
int puts ( char *s );
```

También hay funciones de lectura y escritura de caracteres individuales.

```
#include <stdio.h>
int getchar ( void );
int putchar ( int c );
```

Veamos, por ejemplo, un programa que copia la entrada estándar a la salida estándar del sistema operativo, carácter a carácter.

```
#include <stdio.h>

main()
{
int c;

while ( (c = getchar()) != EOF )
putchar(c);
}
```

Funciones de asignación de memoria.

Hemos visto que en C, las variables estáticas pueden estar creadas al inicio de la ejecución del programa, o bien son variables locales automáticas que se crean al iniciarse la ejecución de una función. En muchas aplicaciones en necesario la utilización de estructuras de datos dinámicas, o bien variables que deben existir durante un tiempo no determinado a priori. Para asignar memoria de forma dinámica, C utiliza varias funciones definidas en `stdlib.h`. Vamos a ver la utilización de estas funciones.

La función de asignación de memoria más utilizada es `malloc`. Esta toma como parámetro el tamaño en bytes y devuelve un puntero al bloque de memoria asignado. Si no hay memoria suficiente para asignar el bloque devuelve `NULL`.

```
#include <stdlib.h>
void *malloc ( size_t size );
```

El tipo `size_t` está definido normalmente como `unsigned` y se utiliza en todas las funciones que necesitan un tamaño en bytes. Otra función similar es:

```
#include <stdlib.h>
void *calloc ( size_t nitems, size_t size );
```

En este caso se le pasa como parámetro el número de elementos consecutivos que se desean. A diferencia de `malloc`, `calloc` inicializa el contenido de la memoria asignada al valor 0. La función que se utiliza para devolver memoria dinámica previamente asignada es `free`. Esta toma como parámetro un puntero previamente obtenido con `malloc` o `calloc`

```
#include <stdlib.h>
void free ( void *block );
```

Hay que tener en cuenta, que la función `free` no cambia el valor del parámetro. El puntero al cual se había asignado memoria y ahora se ha liberado sigue almacenando la dirección de memoria, pero esta ya no existe después de llamar a `free`. Es misión del programador actualizar el valor del puntero, si es necesario. Como ejemplo mostraremos una función que asigna memoria para un vector, lo inicializa y lo libera.

```
void alloc_array ( void )
{
int *v, f;

if ((v = (int *) calloc ( 10, sizeof(int) )) == NULL)
printf("No hay memoria\n");
else {
for ( f = 0; f < 10; f++ )
v[f] = 0;

free(v);
v = NULL;
}
}
```

Debemos observar la conversión de tipo realizada al valor que devuelve `calloc`. Todas las funciones devuelven punteros a `void`, por lo que se deben convertir al tipo de nuestra variable.

Funciones matemáticas.

La utilización de las funciones matemáticas definidas en el ANSI C requieren la inclusión del fichero `math.h`. Todas ellas trabajan con el tipo `double`, por lo que si los argumentos o resultados son del tipo `float` el propio compilador se encarga de convertirlos al formato adecuado. En ANSI se está trabajando para proporcionar funciones con argumentos de tipo `float` e introducir el tipo `long float`. Casi todas la funciones tienen la forma `double` *nombre* `( double x );`.

`atan2`	toma dos argumentos x e y y devuelve la arcotangente de y/x en radianes.
`exp`	devuelve el valor e elevado a x.
`acos`	retorna el arco coseno del parámetro x.
`asin`	retorna el arco seno del parámetro x.
`atan`	retorna el valor de la arco tangente del parámetro x.
`cos`	retorna el coseno del ángulo x.
`cosh`	retorna el coseno hiperbólico del parámetro x.

`sin`	retorna el seno del ángulo x.
`sinh`	retorna el seno hiperbólico del parámetro x.
`tan`	retorna la tangente del ángulo x.
`tanh`	retorna la tangente hiperbólica del parámetro x.
`log`	retorna el logaritmo natural del parámetro x.
`log10`	retorna el logaritmo en base 10 del parámetro x.
`pow`	toma dos parámetros *x* e *y* y devuelve el valor *xy*
`sqrt`	retorna la raiz cuadrada del parámetro x.

Operaciones con ficheros.

La entrada y salida a ficheros es uno de los aspectos más delicados de cualquier lenguaje de programación, pues suelen estar estrechamente integradas con el sistema operativo. Los servicios ofrecidos por los sistemas operativos varían enormemente de un sistema a otro. Las librerías del C proporcionan un gran conjunto de funciones, muchas de ellas descritas en el libro de Kernighan y Ritchie y otras derivadas de los servicios que ofrece el Unix.

En C hay dos tipos de funciones de entrada/salida a ficheros. Las primeras son derivadas del SO Unix y trabajan sin buffer. Las segundas son las que fueron estandarizadas por ANSI y utilizan un buffer intermedio. Además, hacen distinciones si trabajan con ficheros binarios o de texto. Veremos las segundas, que son las más utilizadas.

Las funciones del C no hacen distinción si trabajan con un terminal, cinta o ficheros situados en un disco. Todas las operaciones se realizan a través de *streams*. Un *stream* está formado por una serie ordenada de bytes. Leer o escribir de un fichero implica leer o escribir del *stream*. Para realizar operaciones se debe asociar un *stream* con un fichero, mediante la declaración de un puntero a una estructura `FILE`. En esta estructura se almacena toda la información para interactuar con el SO. Este puntero es inicializado mediante la llamada a la función `fopen()`, para abrir un fichero.

Cuando se ejecuta todo programa desarrollado en C hay tres *streams* abiertos automáticamente. Estos son *stdin*, *stdout* y *stderr*. Normalmente estos *streams* trabajan con el terminal, aunque el sistema operativo permite redireccionarlos a otros dispositivos. Las funciones `printf()` y `scanf()` que hemos visto, utilizan *stdout* y *stdin* respectivamente.

Los datos de los ficheros pueden ser accedidos en uno de los dos formatos: texto o binario. Un *text stream* consiste en una serie de lineas de texto acabadas con un carácter *newline*. En modo binario un fichero es una colección de bytes sin ninguna estructura especial.

Hay que tener muy en cuenta que respecto a la velocidad de la memoria y de la CPU, los dispositivos de entrada y salida con muy lentos. Puede haber cinco o seis órdenes de magnitud entre la velocidad de la CPU y la de un disco duro. Además una operación de entrada y salida puede consumir una cantidad importante de recursos del sistema. Por ello, conviene reducir en número de lecturas y escrituras a disco. La mejor forma de realizar esto es mediante un *buffer*. Un buffer es una área de memoria en la cual los datos son almacenados temporalmente, antes de ser enviados a su destino. Por ejemplo, las operaciones de escritura de caracteres a un fichero se realizan sobre el *buffer* del *stream*. Unicamente cuando se llena el *buffer* se escriben todos los caracteres sobre el disco de una vez. Esto ahorra un buen número de

operaciones sobre el disco. Las funciones del C nos permiten modificar el tamaño y comportamiento del *buffer* de un *stream*.

Para utilizar las funciones de ficheros se debe incluir el fichero `stdio.h`. Este define los prototipos de todas las funciones, la declaración de la estructura `FILE` y algunas macros y definiciones. Una definición importante es `EOF`, que es el valor devuelto por muchas funciones cuando se llega al final de fichero.

Los pasos a seguir para operar con un fichero son: abrir, realizar el tratamiento y cerrar. Para abrir un fichero se utiliza la función `fopen`. Esta toma dos *strings* como parámetros. El primero indica el nombre del fichero que deseamos abrir y el segundo indica el modo de acceso.

```
#include <stdio.h>
FILE *fopen ( const char *filename, const char *mode );
```

Los modos de acceso para *streams* de texto son los siguientes:

`"r"`	Abre un fichero que ya existe para lectura. La lectura se realiza al inicio del fichero.
`"w"`	Se crea un nuevo fichero para escribir. Si el fichero existe se inicializa y sobreescribe.
`"a"`	Abre un fichero que ya existe para añadir información por el final. Sólo se puede escribir a partir del final.
`"r+"`	Abre un fichero que ya existe para actualizarlo (tanto para lectura como para escritura).
`"w+"`	Crea un nuevo fichero para actualizarlo (lectura y escritura) si existe, lo sobreescribe.
`"a+"`	Abre un fichero para añadir información al final. Si no existe lo crea.

Si se desea especificar un fichero binario, se añade una `b` al modo: `"wb+"`. Si el fichero se abre correctamente, la función devuelve un puntero a una estructura `FILE`. Si no, devuelve `NULL`.

La función `fprintf` se comporta exactamente a `printf`, excepto en que toma una argumento más que indica el *stream* por el que se debe realizar la salida. De hecho, la llamada `printf("x")` es equivalente a `fprintf ( stdout, "x")`.

```
FILE *f;

if ((f = fopen( "login.com", "r" )) == NULL )
printf("ERROR: no puedo abrir el fichero\n");
```

Para cerrar un fichero se utiliza la función `fclose`. Esta toma como argumento el puntero que nos proporcionó la función `fopen`.

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

```
int fclose ( FILE *stream );
```

Devuelve 0 si el fichero se cierra correctamente, `EOF` si se produce algún error.

Una vez conocemos como abrir y cerrar ficheros, vamos a ver cómo leer y escribir en ellos. Hay funciones para trabajar con caracteres, lineas y bloques.

Las funciones que trabajan con caracteres son `fgetc` y `fputc`. La primera lee un carácter de un *stream* y la segunda lo estribe.

```
#include <stdio.h>
int fgetc ( FILE *stream );
int fputc ( int c, FILE *stream );
```

La primera lee el siguiente carácter del *stream* y los devuelve convertido a entero sin signo. Si no hay carácter, devuelve `EOF`. La segunda, `fputc`, devuelve el propio carácter si no hay error. Si lo hay, devuelve el carácter `EOF`. Hay una tercera función, `feof` que devuelve cero si no se ha llegado al final del *stream*. Veamos un ejemplo de cómo se copia un fichero carácter a carácter.

```
while ( !feof(infile))
fputc ( fgetc ( infile ), outfile );
```

Suponemos que `infile` y `outfile` son los *streams* asociados al fichero de entrada y al de salida, que están abiertos. Luego debemos cerrarlos para vaciar los *buffers* y completar los cambios realizados.

Otras funciones nos permiten realizar operaciones con ficheros de texto trabajando línea a línea.

```
#include <stdio.h>
char *fgets ( char *s, int n, FILE *stream );
int fputs ( const char *s, FILE *stream );
```

La función `fgets` lee caracteres del *stream* hasta que encuentra un final de línea o se lee el carácter n-1. Mantiene el carácter `\n` en el string y añade el carácter `\0`. Devuelve la dirección del string o NULL si se produce algún error. La función `fputs` copia el string al *stream*, no añade ni elimina caracteres `\n` y no copia la marca de final de string `\0`.

Hay dos funciones que nos permiten trabajan en bloques. Podemos considerar un bloque como un array. Debemos especificar el tamaño de cada elemento y el número de elementos.

```
#include <stdio.h>
size_t fread ( void *p, size_t s, size_t n, FILE *f );
size_t fwrite ( void *p, size_t s, size_t n, FILE *f );
```

A las anteriores funciones se les pasa un puntero genérico `p` con la dirección del área de datos que se desea leer o escribir, el tamaño de cada elemento `s`, y el número de elementos `n`, además del *stream*. Ambas devuelven el número de elementos leídos o escritos, que debe ser el mismo que le hemos indicado, en el caso en que no se haya producido ningún error.

Hasta ahora hemos visto funciones de tratamiento de ficheros que nos van bien para realizar un tratamiento secuencial de éstos. Debemos tener en cuenta que cuando debemos realizar operaciones de entrada y de salida alternadas, se deben vaciar los *buffers* de los *streams* para que las operaciones se realicen correctamente sobre los ficheros. Esto se consigue con las funciones `fflush` y `flushall`. La primera es aplicable al *stream* que deseemos y la segunda vacía los *buffers* de todos los *streams* abiertos.

```
#include <stdio.h>
int fflush ( FILE *stream );
int flushall ( void );
```

Si deseamos hacer un acceso aleatorio a un fichero disponemos de las funciones `fseek` y `ftell`. La primera mueve el cursor, indicador de posición, a un lugar determinado dentro del fichero. Este lugar viene determinado por el parámetro `whence`. Este puede ser `SEEK_SET`, si es desde el inicio del fichero, `SEEK_CUR` si es desde la posición actual y `SEEK_END` si es desde el final del fichero. Devuelve 0 si la operación se realiza correctamente.

```
#include <stdio.h>
int fseek ( FILE *stream, long offset, int whence );
long ftell ( FILE *stream );
```

La segunda función devuelve la posición del cursor dentro del fichero. Este valor indica el número de bytes desde el inicio del fichero, si el fichero es binario.

```
int escribir_planetas ( char *nombre )
{
PLANETA *p;
FILE *f;

if ( ( f = fopen ( "w+", nombre )) == NULL )
return ERROR;

if (N_PLA != fwrite ( p, sizeof(PLANETA), N_PLA, f ))
return ERROR;

if ( fclose(f) != 0 )
return ERROR;
else
return OK;
}
```

Bibliografía y referencias.

[1] El lenguaje de programación C.
Brian W. Kernighan, Dennis M. Ritchie.
Prentice-Hall Hispanoamericana, 1985.
ISBN 968-880-024-4

[2] American National Standard for Information Systems -- Programming Language C.
American National Standards Institute.
1430 Broadway
New York, NY 10018

[3] El lenguaje de programación C, segunda edición.
Brian W. Kernighan, Dennis M. Ritchie.
Prentice-Hall Hispanoamericana, 1991.
ISBN 968-880-205-0

[4] C. A software engineering approach.
Peter A Darnell, Philip E. Margolis.
Springer-Verlag, 1991
ISBN 3-540-97389-3