INTRODUCCIÓN

C proporciona una variedad de tipos de datos. Los tipos fundamentales son caracteres, enteros y números de punto flotante de varios tamaños. Además, existe una jerarquía de tipos de datos derivados, creados con apuntadores, arreglos, estructuras y uniones.

Las expresiones se forman a partir de operadores y operandos; cualquier expresión, incluyendo una asignación o una llamada a función, puede ser una proposición. Los apuntadores proporcionan una aritmética de direcciones independiente de la máquina.

Las funciones de un programa en C pueden existir en archivos separados, que se compilan de manera separada. Las variables pueden ser internas a una función, externas pero conocidas solo dentro de un archivo fuente, o visibles al programa completo.

C no proporciona operaciones para tratar directamente con objetos compuestos, tales como cadenas de caracteres, conjuntos, listas o arreglos.

C es relativamente pequeño, se puede describir en un pequeño espacio y aprender con rapidez. Un programador puede razonablemente esperar conocer, entender y utilizar en verdad la totalidad del lenguaje.

Esta introducción presenta muchos conceptos que se irán viendo detalladamente en secciones posteriores. Es una introducción rápida a C. Dado que la única manera de aprender un lenguaje de programación es escribiendo programas con él, en esta introducción se mostrarán algunos ejemplos.

En C, el programa para escribir "Hola, mundo" es el siguiente:

[hello_world.c]

```c
#include <stdio.h>

int main() {
    printf("Hola, mundo\n");

    return 0;
}
```

Para ejecutar este sencillo programa debemos primero compilarlo. Se debe guardar en un archivo con extensión .c y compilar con el comando 'cc hello_world.c' o 'gcc hello_world.c'. La compilación creará un archivo ejecutable llamado 'a.out'.

Expliquemos el programa presentado:

Un programa en C, cualquiera que sea su tamaño, consta de funciones y variables. Una función contiene proposiciones que especifican las operaciones de cálculo que se van a realizar, y las variables almacenan los valores utilizados durante los cálculos.

La función main():

Nuestro ejemplo es una función llamada main. Normalmente se tiene la libertad de dar cualquier nombre que se desee a una función, pero “main” es especial, el programa comienza a ejecutarse al principio de main(). Esto significa que todo

programa debe tener un main() en algún sitio. Por lo común main llamará a otras funciones que ayuden a realizar su trabajo, algunas que usted ya escribió, y otras de bibliotecas escritas previamente.

Todos los programas en C constan de una o más funciones, cada una de las cuales contiene una o más sentencias. Una función es una subrutina con nombre a la que pueden llamar otras partes del programa. Las funciones son los bloques constituyentes de C. Las sentencias se encuentran dentro de las funciones. Aunque un programa puede tener varias funciones, la única función que debe tener es main(). La función main(), como hemos dicho, es la función en la que comienza la ejecución del programa.

Un método para comunicar datos entre las funciones es que la función que llama proporciona una lista de valores, llamados argumentos (parámetros), a la función que está invocando. Los paréntesis que están después del nombre de la función encierran a la lista de argumentos. En este ejemplo, main() está definido para ser una función que no espera argumentos, lo cual está indicado por la lista vacía ().

int delante de main(), es decir 'int main()' indica el tipo de retorno de la función. Significa que la función retorna un entero. Esto es relevante a la hora de ejecutar un programa, ya que el tipo de retorno de la función main() aquí importa para tener información del estado de finalización del programa, ya que de ser entero, si vale 0 generalmente implica que la ejecución ha sido exitosa sin errores (return 0) y los demás valores posibles darán información de qué tipo de error se ha producido. void main() significa que la función no retorna valor alguno.

Actualmente el uso de int o void no afecta en nada salvo el uso de la palabra reservada "return" en caso de los programas iniciados con "int main(), al utilizar "void main()"" puedes saltar el retorno del valor a la función. Sin embargo, el uso de "int main()" o "void main()"" pueden estar restringidos a lo que el compilador permita hacer con el punto de entrada de los programas y puede que estés limitado al uso de una sola variante de entrada. Es preferible el uso de "int main()"" sobre "void main()"" ya que se tiene un mejor control sobre el estado del término del programa. El valor de retorno para main indica cómo salió el programa. La salida normal está representada por un valor de retorno 0 de main. La salida anormal se indica con un retorno distinto de cero, pero no existe un estándar sobre cómo se interpretan los códigos distintos de cero. void main()está prohibido por el estándar C++ y no debe utilizarse.

La función main() se puede dejar sin valor de retorno, es decir sin la sentencia return 0, los compiladores lo aceptarán, dado que el valor de retorno predeterminado es 0. Vale la pena señalar que en C ++, int main() se puede dejar sin una declaración de retorno, momento en el que devuelve 0 por defecto. Esto también es cierto con un programa C99. Si return 0; debe omitirse o no está abierto a debate. El rango de firmas principales válidas del programa C es mucho mayor.

Primera línea '#include <stdio.h>':

La primera línea indica al compilador que debe incluir información acerca de la biblioteca estándar 'stdio.h' de entrada/salida. Se llama archivo de cabecera. En C, la información sobre las funciones de biblioteca estándar se encuentra en varios archivos que se suministran con el compilador. Todos los archivos terminan con una extensión .h. Estos archivos se añaden al programa utilizando la directiva del preprocesar del compilador #define. Todos los compiladores de C utilizan como primera fase de compilación un preprocesador, que realiza varias manipulaciones en el archivo fuente antes de compilarlo realmente. Las líneas que comienzan con # son procesadas por el preprocesador antes de que el programa se compile. La directiva #include le dice al preprocesador que lea en

otro archivo y lo incluya en el programa. El archivo de cabecera que se requiere con más frecuencia es stdio.h.

La directiva #include no termina con un ';', la razón es que no es una palabra clave de C que pueda definir una sentencia (solo las sentencias terminan en punto y coma). En lugar de eso, es una instrucción para el propio compilador de C.
Si el archivo a incluir forma parte de la biblioteca estándar el modo de incluirlo es:

```
#include <file.h>
```

si es una biblioteca propia o de terceros en modo es:

```
#include "file.h"
```

printf:

Una función se invoca al nombrarla, seguida de una lista de argumentos entre paréntesis; de esta manera se está llamando a la función printf con el argumento "Hola, mundo\n". printf es una función de biblioteca que escribe en pantalla la salida, en este caso la cadena de caracteres que se encuentra entre comillas.

Se debe utilizar '\n' para incluir un carácter de nueva línea en el argumento de printf. Nótese que \n representa un solo carácter. Una secuencia de escape como \n proporciona un mecanismo general y extensible para representar caracteres invisibles o difíciles de escribir. Entre otros que C proporciona están \t para tabulación, \b para retroceso, \" para comillas, y \\ para la diagonal invertida.

Variables y expresiones aritméticas.

Sigue un ejemplo de código más sofisticado. Introduce varias ideas nuevas, incluyendo comentarios, declaraciones, variables, expresiones aritméticas, ciclos y salida con formato:

```
#include <stdio.h>

// Imprime los pares Fahrenheit - Celsius, desde 0 hasta 300 Fahrenheit.

int main(){

    int fahr, celsius;
    int lower, upper, step;

    lower = 0;              // Límite inferior.
    upper = 300;            // Límite superior.
    step = 20;              // Tamaño del incremento.

    fahr = lower;
    while (fahr <= upper){
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
    return 0;
}
```

Una variable es una posición de memoria con un nombre que puede guardar distintos valores. En C, todas las variables se deben declarar antes de poder utilizarlas. Con esto le decimos al compilador qué tipo de variable se está utilizando.

En C, se deben declarar todas las variables antes de su uso, generalmente al principio de la función y antes de cualquier proposición ejecutable. Una declaración notifica las propiedades de una variable; consta de un nombre de tipo y una lista de variables.

El tipo int significa que las variables de la lista son enteros, en contraste con float, que significa punto flotante, esto es, números que pueden tener una parte fraccionaria. El rango tanto de int como de float depende de la máquina que se está utilizando; los int de 16 bits, que están comprendidos entre el -32768 y +32767, son comunes, como lo son los int de 32 bits.

Además de int y float, C proporciona varios tipos de datos básicos, incluyendo:

char	carácter (un solo byte)
short	entero corto
long	entero largo
double	punto flotante de doble precisión
void	sin valor
otros	...

Una variable tipo char tiene 8 bits de longitud y se utiliza normalmente para guardar un solo carácter. dado que C es muy flexible, una variable tipo char también se puede utilizar como un "entero pequeño" si se desea.

IMPORTANTE: Los valores en punto flotante deben incluir un punto decimal. No es válido el uso de ',', es decir 100.01 es válido, pero 100,01 no lo es. Por ejemplo, para indicarle al compilador de C que 100 es un número en punto flotante, se debe utilizar 100.0.

En C, una declaración de variable es una sentencia y debe terminar con punto y coma.

Hay dos lugares donde se declaran las variables: dentro de una función o fuera de todas las funciones. Las variables declaradas fuera de todas las funciones se llaman variables globales y cualquier función del programa puede acceder a ellas. Las variables globales existen durante todo el tiempo de ejecución del programa. Las variables declaradas dentro de una función se llaman variables locales. A una variable local solamente la conoce, y puede acceder, la función donde está declarada.

También existen arreglos (arrays), estructuras y uniones de estos tipos básicos, apuntadores a ellos y funciones que regresan valores con esos tipos, todo lo cual se verá en el momento oportuno.

Los arrays son estructuras de datos que nos permiten almacenar otros datos dentro de este tipo. Es decir, es un contenedor que nos permite tener varios datos al mismo tiempo almacenados.

Las variables se inicializan, se les asigna un valor inicial. Se trata de proposiciones individuales, sentencias, que terminan con un ';'.

El ciclo 'While' puede tener una o más proposiciones encerradas entre llaves, como en el ejemplo anterior,

```
while (fahr <= upper){
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr * step;
    }
```

o solo una proposición sin llaves:

```
while (i < j)
      i = 2 * i;
```

El sangrado acentúa la estructura lógica del programa. Aunque a los compiladores de C no les importa la apariencia del programa, un sangrado y espaciamiento adecuados son muy importantes para hacer programas fáciles de leer.

printf es una función de propósito general para dar formato de salida. Su primer argumento es una cadena de caracteres que serán impresos, con cada % indicando en donde uno de los otros argumentos va a ser sustituido, y en qué forma será impreso. Por ejemplo, %d especifica un argumento entero. Cada construcción % en el primer argumento de printf está asociada con el correspondiente segundo argumento, tercero, etc., y deben corresponder apropiadamente en número y tipo, o se tendrán soluciones incorrectas. printf es una función de la biblioteca estándar.

Para la función printf, si se quiere especificar un valor de carácter, el especificador de formato es %c. Para especificar un valor en punto flotante, se utiliza %f, válido tanto para float como para double. Por ejemplo:

```
#include <stdio.h>

int main(){
    double a = 100.01;
    double b = 100.02;

    printf("%0.2f %0.2f", a, b); // '0.2' Número de decimales que imprimirá.

    return 0;
}
```

Hay otra función importante, 'scanf'. La función scanf es como printf, exceptuando que lee de la entrada en lugar de escribir a la salida. El uso general de scanf es el siguiente:

```
    scanf("%d", &nombre-var-entero); Por ejemplo: int num; scanf("%d", &num);
```

El & que precede al nombre de la variable es especial para el funcionamiento de scanf(). El signo & permite que una función coloque un valor en uno de sus argumentos.

Si se quiere introducir un float con scanf() se utiliza %f y si se quiere introducir un double, se utiliza %lf.

Hagamos algunos cambios en el programa anterior:

```c
#include <stdio.h>

// Imprime los pares Fahrenheit - Celsius, desde 0 hasta 330 Fahrenheit.

int main(){

    float fahr, celsius;        // Se declaran variables en punto flotante.
    int lower, upper, step;
                                // Se inicializan las variables.
    lower = 0;                  // Límite inferior.
    upper = 300;                // Límite superior.
    step = 20;                  // Tamaño del incremento.

    fahr = lower;
    while (fahr <= upper){
        celsius = (5.0/9.0) * (fahr-32.0);      // Aritmética en punto flotante.
        printf("%3.0f\t%.2f\n", fahr, celsius);    // .0 y .2 precisión decial.
        fahr = fahr + step;
    }
    return 0;
}
```

Si aumentamos a cada %d de la proposición printf una amplitud, los números impresos serán justificados hacia su derecha dentro de sus campos. Por ejemplo, 3.0f quiere decir: imprimir el número en un campo de tres caracteres con cero decimales. .2 quiere decir, imprimir el número con una precisión de dos decimales.

Anteriormente se uso aritmética de enteros, ahora en punto flotante. Si queremos resultados precisos debemos usar aritmética en punto flotante. No pudimos utilizar 5/9 en la versión anterior debido a que la división entera lo truncaría a cero (la división entera da como resultado un entero). Sin embargo, un punto decimal en una constante indica que ésta es de punto flotante, por lo que 5.0/9.0 no se trunca debido a que es una relación de dos valores de punto flotante.

Entre otros, printf también reconoce %o para octal, %x para hexadecimal, %c para carácter, %s para cadena de caracteres y %% para % en sí, es decir, esta última imprimiría el carácter %.

La proposición for.

Vamos a reescribir el programa con una proposición for.

```c
#include <stdio.h>

// Imprime los pares Fahrenheit - Celsius, desde 0 hasta 330 Fahrenheit.

int main(){

    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20){
        printf("%3.0d %6.2f\n", fahr, (5.0 / 9.0) * (fahr - 32));
    }

    return 0;
```

La proposición for es una forma generalizada de while. Dentro de los paréntesis existen tres secciones, separadas por punto y coma. La primera, la inicialización, que se ejecuta una vez, antes de entrar propiamente al ciclo. La segunda sección es la condición o prueba que controla el ciclo. Esta condición se evalúa; si es verdadera, el cuerpo del ciclo se ejecuta. Después una tercera sección con un incremento que se ejecuta y la condición se vuelve a evaluar.

Este programa también se pudo haber escrito del siguiente modo:

```c
#include <stdio.h>

int main(){

    for (int fahr = 0; fahr <= 300; fahr = fahr + 20){
        printf("%3.0d %6.2f\n", fahr, (5.0 / 9.0) * (fahr - 32));
    }

    return 0;
}
```

Como se puede ver la variable int se ha declarado dentro de for, en su primera porción, esto es algo válido y de mayor elegancia.

Constantes simbólicas.

Es una mala práctica asignar valores a variables constantes, ya que proporcionan muy poca información a quien tenga que leer el programa. En su lugar, es mejor el uso de constantes simbólicas. Las constantes simbólicas se definen del siguiente modo. Veamos un ejemplo:

```c
#include <stdio.h>

// Imprime los pares Fahrenheit - Celsius, desde 0 hasta 330 Fahrenheit.

#define LOWER   0       // Límite inferior.
#define UPPER   300     // Límite superior.
#define STEP    20      // Tamaño del incremento.

int main(){

    for (int fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3.0f %6.2f\n", fahr, (5.0 / 9.0) * (fahr-32));

    return 0;
}
```

Una línea #define, define un nombre simbólico o constante simbólica como una cadena de caracteres especial:

```c
    #define NOMBRE  Texto-de-reemplazo
```

El texto de reemplazo puede ser cualquier secuencia de caracteres; no está limitado a números.

Las cantidades LOWER, UPPER y STEP son constantes simbólicas, no variables, por lo que no aparecen entre las declaraciones. Los nombres de constantes

simbólicas, por convención, se escriben con letras mayúsculas, de modo que se puedan distinguir fácilmente de los nombres de variables escritos con minúsculas. Nótese que no hay punto y coma al final de una línea #define.

Entrada y salida de caracteres.

El modelo de entrada y salida manejado por la biblioteca estándar es muy simple. La entrada y salida de texto, sin importar dónde fue originada o hacia dónde se dirige, se tratan como flujos (streams) de caracteres. Un flujo de texto es una secuencia de caracteres divididos entre líneas, cada una de las cuales consta de cero o más caracteres seguidos de un carácter nueva línea. La biblioteca es responsable de hacer que cada secuencia de entrada o salida esté de acuerdo con este modelo; el programador de C que utiliza la biblioteca no necesita preocuparse de cómo están representadas las líneas fuera del programa.

La biblioteca estándar proporciona varias funciones para leer o escribir un carácter a la vez, de las cuales 'getchar' y 'putchar' son las más simples. Cada vez que se invoca, getchar lee el siguiente carácter de entrada de una secuencia de texto y lo devuelve como su valor. Esto es, después de 'c = getchar()', la variable c contiene el siguiente carácter de entrada.

La función putchar(c) escribe un carácter cada vez que se invoca. Las llamadas a putchar y printf pueden mezclarse.

Copia de archivos.

Veamos un ejemplo de programa que copia la entrada en la salida, un carácter a la vez:

```c
#include <stdio.h>

// Copia la entrada estándar a la salida estándar v.1.

int main(){

    int c;

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

    return 0;
}
```

¿Por qué hemos declarado c como entero y no como char?

El problema es distinguir el fin de la entrada de los datos válidos. La solución es que getchar() devuelva un valor distintivo cuando no hay más a la entrada, un valor que no puede ser confundido con ningún otro carácter. Este valor se llama EOF, por “end of file (fin de archivo)”. Se debe declarar c con un tipo que sea lo suficientemente grande para almacenar cualquier valor que le regrese get-char(). No se puede utilizar char puesto que c debe ser suficientemente grande como para mantener a EOF además de cualquier otro carácter. Por lo tanto, se emplea int.

EOF es un entero definido en <stdio.h>, pero el valor numérico específico no importa mientras que no sea el mismo que ningún valor tipo char. Utilizando la

constante simbólica, hemos asegurado que nada en el programa depende del valor numérico específico.

```c
#include <stdio.h>

// Imprime el carácter EOF (-1).

int main(){

    printf("%d", EOF);

    return 0;
}
```

Podemos reescribir el programa anterior:

```c
#include <stdio.h>

// Copia la entrada estándar a la salida estándar v.2.

int main(){

    int c;

    while((c = getchar()) != EOF){
        putchar(c);
    }
    return 0;
}
```

Conteo de caracteres.

Veamos un ejemplo de programa que cuenta caracteres:

```c
#include <stdio.h>

// Cuenta los caracteres de la entrada v.1

int main(){

    long nc;

    nc = 0;
    while(getchar() != EOF)
        ++nc;

    printf("%1d\n", nc);

    return 0;
}
```

En el ejemplo anterior hemos presentado un nuevo operador, ++, que significa incrementa en uno. Es lo mismo que escribir nc = nc + 1, pero la notación ++nc, es más concisa y muchas veces más eficiente. El operador -- disminuye en uno.

Los operadores ++ y -- pueden ser tanto prefijos '++nc' como sufijos 'nc++'; esas dos formas tienen diferentes valores dentro de las expresiones, pero ambos incrementan a nc.

La notación, en printf, de %ld, indica a printf que el argumento correspondiente es un entero long.

Como en la anterior ocasión, podemos reescribir el código con un bucle for:

```
#include <stdio.h>

// Cuenta los caracteres de la entrada v.2

int main(){

    double nc;

    for(nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);

    return 0;
}
```

Aquí trabajamos con números mayores, double, y hacemos uso de una proposición for. %.0f suprime la impresión del punto decimal y de la parte fraccionaria, que es cero.

IMPORTANTE: El cuerpo de este ciclo for está vacío, debido a que todo el trabajo se realiza en las secciones de prueba e incremento. Pero las reglas gramaticales de C requieren que una proposición for tenga un cuerpo. El punto y coma aislado se llama proposición nula, y está aquí para satisfacer este requisito. Lo colocamos en una línea aparte para que sea visible.

Conteo de líneas.

```
#include <stdio.h>

// Cuenta las líneas de la entrada.

int main(){

    int c, nl;

    nl= 0;
    while ((c = getchar()) != EOF)  // No abrimos llaves, solo una expresión [*]
       if (c == '\n')     // Tampoco aquí, le sigue solo una expresión.
            ++nl;
    printf("%d\n", nl);

    return 0;
}
```

[*] Esta, aunque válida, no es una buena práctica. Mejor abrir siempre llaves {}. Se pueden declarar más de una variable del mismo tipo utilizando una lista separada por comas, como en: int c, nl;.

IMPORTANTE: Un carácter escrito entre apóstrofos representa un valor entero igual al valor numérico del carácter en el conjunto de caracteres de la máquina. Esto se llama una constante de carácter, aunque solo es otra forma de escribir un pequeño entero. Así, por ejemplo 'A' es una constante de carácter; en el conjunto ASCII de caracteres su valor es 65, esto es, la representación interna del carácter A. Por supuesto 'A' es preferible que 65: su significado es obvio, y es independiente de un conjunto de caracteres en particular.

Las secuencias de escape que se utilizan en constantes de cadena también son legales en constantes de carácter; así, '\n' significa el valor del carácter nueva línea, el cual es 10 en código ASCII. Se debe notar cuidadosamente que '\n' es un carácter simple, y en expresiones es solo un entero; por otro lado, '\n' es una constante cadena que contiene solo un carácter.

Conteo de palabras.

Veamos un programa que cuenta las líneas, palabras y caracteres. esta es una versión reducida del programa wc de Linux.

```c
#include <stdio.h>

// Cuenta líneas, palabras y caracteres.

#define IN 1         // Dentro de una palabra.
#define OUT 0        // Fuera de una palabra.

//int main (){       // Ambas declaraciones de la función main() son correctas [*].
int main(void){

    int c, nl, nw, nc, state;   // Se declaran las variables.
    state = OUT;                // Se inicializan las variables.
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF){
        ++nc;
        if(c == '\n')
            ++nl;
        if(c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT){
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);

    return 0;
}
```

[*] La segunda definición de la función main(), con void, se considera técnicamente mejor, ya que especifica claramente que solo se puede llamar a 'main' sin ningún argumento.

Funciones.

Al escribir sus propias funciones, podrá devolver un valor a la rutina de llamada a la función utilizando la sentencia return. La sentencia return sigue el siguiente formato (en este ejemplo la función devuelve un valor entero):

```c
#include "stdio.h"      // También está permitido el uso de comillas.[*]

int func(); // Se ve más adelante.

int main(){

    int num;
    num = func();
    printf("%d", num);  // Imprime: 10.

    return 0;
}

func(){
    return 10;
}
```

Aunque se pueden crear funciones que devuelvan cualquier tipo de dato, por omisión las funciones devuelven datos del tipo int.

Nota: Puede haber más de un return en una función.

[*] <stdio.h> busca en ubicaciones de biblioteca C estándar, mientras que "stdio.h" también busca en el directorio actual. Idealmente, se usa <filename> para bibliotecas C estándar y "filename" para bibliotecas que escribe y están presentes en el directorio actual.

Ejemplo de una función de exponenciación (C no incluye la exponenciación). A continuación se presenta la función power() y un programa main() para utilizarla, de modo que se vea la estructura completa de una vez.

```c
#include "stdio.h"

int power (int m, int n);   // Declaración del prototipo de la función power().

// Prueba la función power.
int main(){

    for (int i = 0; i < 10; ++i){
        printf("%d %d %d\n", i, power(2, i), power (-3, i));
    }
    return 0;
}

// Eleva la base a la enésima potencia; n >= 0.
int power(int base, int n){

    int p = 1;

    for (int i = 1; i <=n; ++i){
```

```
        p = p * base;
    }
    return p;
}
```

Una definición de función tiene la siguiente forma:

```
tipo-de-retorno nombre-de-la-función (declaración de parámetros){
      // declaraciones
      // sentencias
}
```

Las definiciones de función pueden aparecer en cualquier orden.

La primera línea de la función power(),

```
int power(int base, int n)
```

declara los tipos y los nombre de los parámetros, así como el tipo de resultado que la función devuelve. Los nombre que emplea power() para sus parámetros son locales a la función y son invisibles para cualquier otra función, es decir, otras rutinas pueden usar los mismos nombres sin que exista ningún problema. Así, la 'i' de power no tiene nada que ver con la 'i' de main().

El valor que calcula power() regresa a main() a través de la sentencia return, como hemos visto, a la cual le puede seguir cualquier expresión:

```
return expresión;
```

Alguna bibliografía aconseja cerrar la función main() con un return, normalmente return 0, que indica que el programa terminó con éxito.

La declaración

```
int power(int m, int n);
```

precisamente antes de main(), indica que power es una función que espera dos argumentos int y devuelve un int. Esta declaración, a la cual se le llama función prototipo, debe coincidir con la definición y uso de power(). Es un error el que la definición de la función o cualquier uso que se haga de ella no coincida con su prototipo. Los nombres de los parámetros no necesitan coincidir, de hecho son optativos en el prototipo de la función, de modo que para el prototipo se pudo haber escrito

```
int power(int, int);
```

IMPORTANTE: Como el compilador trabaja solo con la información contenida en un único fichero, a menudo es preciso "informar" al compilador de que en otro fichero existe una función. Esto se consigue insertando, en lugar de la definición de la función (que ya está presente en otro fichero), su prototipo. El prototipo de una función es una línea similar a la primera de su declaración: tipo del resultado, seguido del nombre de la función y de la lista de tipos de datos de los parámetros separados por comas y rodeados por paréntesis. Toda función que se invoca debe ir precedida o de su definición o de su prototipo. La definición y el prototipo de la función pueden estar presentes en el mismo fichero.

INTRODUCCIÓN A LAS SENTENCIAS DE CONTROL

Sentencia if.

En C es verdadero cualquier valor que no sea 0 y es falso 0. Por eso, es perfectamente válido tener una sentencia if como la que se muestra a continuación:

```
#include "stdio.h"

int main(){

int count;

      if (count+1){ // Es una expresión válida en C y es verdadera -> Es válida.
            printf("Es verdadero.\n");
            return 0;
      }
}
```

Bucle for.

Su forma general es la siguiente:

```
      for (inicialización; comprobación condicional; incremento) {
            //sentencia
      }
```

La inicialización se utiliza para dar un valor inicial a la variable que controla el bucle.

El incremento de for se ejecuta al final de cada bucle; es decir el incremento se ejecuta después de que se haya ejecutado la sentencia o bloques de sentencias del bucle cada vez, pero antes que la comprobación condicional.

```
#include "stdio.h"

int main(){
    for(int num = 1; num < 11; ++num){
        printf("%d", num);
    }
    printf("El programa a terminado");
}
```

Nótese que la variable 'num' se ha declarado e inicializa dentro de la sentencia for. Es perfectamente válido. Es importante señalar que si la prueba es falsa desde el principio, el bucle no se ejecutará ni una sola vez. Por ejemplo este bucle no se ejecutará:

```
#include "stdio.h"

int main(){
    for(int num = 11; num < 11; ++num){
        printf("%d", num);
    }
    printf("El programa a terminado");
}
```

IMPORTANTE: Operadores de incremento y decremento '++ y --'. Estos operadores pueden seguir a la variable, por ejemplo: num++ o num-- o precederla, ++num o --num. El efecto en la variable es el mismo, pero la posición del operador afecta cuando la operación se lleva a cabo. Para verlo, examinemos este programa:

```
#include "stdio.h"

int main(){

    int i, j;
    i = 10;
    j = i++;    // Primero se asigna el valor a j, después se incrementa i.
    printf("i y j: %d %d, i, j");   // Imprime: 11 10.
}
```

En la expresión j = i++, primero se asigna el valor de i a j. Después se incrementa i. Por ejemplo, suponiendo que max tiene el valor 1, en la expresión:

```
    count = 10 * max++;
```

se asigna el valor 10 a count y luego max se incrementa en uno.

Por el contrario, si el operador de incremento o decremento precede a la variable, la operación se realiza primero, y después se asigna el valor de la expresión. Por ejemplo:

```
#include "stdio.h"

int main(){

    int i, j;
    i = 10;
    j = ++i;    // Primero se incrementa i, después se asigna el valor a j.
    printf("i y j: %d %d, i, j");   // Imprime: 11 11.
}
```

Si se usan operador de incremento o decremento simplemente para remplazar sentencias de asignación equivalentes, no importa si el operador precede o sigue a la variable. Este es un asunto de su propio estilo personal.

Operadores lógicos y relaciones.

En los valores resultantes de las operaciones lógicas (&&, ||, !) y relacionales (<, >, ==, !=, ...) el resultado es 0 o 1. Aunque C describa como verdadero cualquier valor distinto de 0, los operadores lógicos y relaciones siempre dan el valor 1 para verdadero. Los programas pueden hacer uso de este hecho.

Introducción de caracteres.

La versión original de C definía una función llamada getchar(), que devuelve un carácter (solo uno) introducido por el teclado. getchar() hace eco de la

pulsación de la tecla en pantalla y devuelve el valor de la tecla al solicitante. Está definida por ANSI y requiere del archivo de cabecera stdio.h.

```c
#include <stdio.h>

int main(){

    char ch;
    ch = getchar();
    printf("Usted escribió: %c\n", ch);

}
```

El uso de getchar() requiere que el usuario pulse la tecla enter, es decir, antes de que getchar() devuelva el carácter hay que pulsar la tecla enter.

Estudio de las variaciones del bucle for.

El bucle for es muy flexible. Las expresiones de inicialización, prueba condicional e incremento del bucle no están limitadas a estos simples papeles. El bucle for de C no establece límites al tipo de expresiones que se producen dentro de él.

Por ejemplo, no hay qué utilizar la sección de inicialización, necesariamente, para inicializar la variable de control del bucle. Además, no tiene por qué haber ninguna variable de control del bucle, ya que la expresión de prueba condicional puede utilizar otras formas de detener el bucle. Veamos un ejemplo:

```c
#include <stdio.h>

int main(){

    int i;

    printf("Introduzca un número entero: ");
    scanf("%d", &i);

    for( ; i; i--){ // La cuenta atrás llega a cero, es False, el bucle para.
        printf("%d ", i);
    }
}
```

Mismo ejemplo inicializando la variable de control:

```c
int main() {
    int numero;
    printf("Ingresa un número para comenzar la cuenta regresiva: ");
    scanf("%d", &numero);

    printf("Cuenta regresiva desde %d:\n", numero);

    for (int i = numero; i >= 0; i--) {
        printf("%d\n", i);
    }
    return 0;
}
```

Mismo ejemplo usando un bucle infinito:

```
#include <stdio.h>

int main(){

    int i;

    printf("Introduzca un número entero: ");
    scanf("%d", &i);

    for( ; i; i--){
        printf("%d ", i);
        if(i==1){
          i = 0; // La cuenta atrás no para en 0, continúa indefinidamente.
          printf("0\n");
        }
    }
}
```

Existe un modo más elegante de definir un bucle infinito, ahora la veremos.

El incremento, técnicamente, solo es una expresión que se evalúa cada vez que se repite el bucle. No tiene que incrementar o decrementar una variable. Por ejemplo:

```
#include <stdio.h>

int main(){

    char ch;

    for(ch=getchar() ; ch!='q'; ch=getchar());

    printf("Encontrada la 'q'\n");
}
```

Observe, además, que el objetivo de for está vacío y la razón de esto es porque C permite sentencias nulas.

En C, es perfectamente válido alterar la variable de control del bucle desde fuera de la sección de incremento. Por ejemplo, el siguiente programa incrementa manualmente i al final del bucle:

```
#include <stdio.h>

int main(){

    int i;

    for(i=0; i<10; ){
        printf("%d", i);
        i++;
    }
}
```

Bucle for infinito:

```c
#include <stdio.h>

int main(){

    for( ; ; ){

        printf("Bucle infinito\n");
    }
}
```

Como se puede ver, no hay expresiones en for. Cuando no hay expresión en la porción condicional, el compilador asume que es verdadero. Por eso, el bucle continúa ejecutándose.

Bucle While de C.

El bucle while funciona repitiendo su objetivo mientras la expresión sea cierta. Cuando esta es falsa el bucle se detiene. El valor de la expresión se comprueba al principio del bucle. Esto significa que si la expresión es falsa al iniciarse, el bucle no se ejecutará ni siquiera una vez.

Bucle Do While.

El formato general de Do While es el siguiente:

```c
do{
      //sentencias
} while(expresión);
```

El bucle do-while repite la sentencia o sentencias mientras la expresión es verdadera. Se detiene cuando la expresión se convierte en falsa. El bucle do-while es único, ya que siempre ejecuta el código de dentro del bucle al menos una vez, dado que la expresión que controla el bucle se comprueba al final del mismo.

Sentencia break.

La sentencia break permite salir de un bucle desde cualquier punto de su cuerpo, pasando por alto su expresión de finalización normal. Cuando la sentencia break se encuentra dentro de un bucle, el bucle termina inmediatamente y el control del programa continúa en la sentencia que le sigue.

Sentencia continue.

La sentencia continue obliga a que se produzca la siguiente iteración del bucle, saltando cualquier código entre ella y la condición de prueba del bucle. Por ejemplo, este programa nunca muestra ninguna salida:

```c
#include <stdio.h>

int main(){

    int x;
    for(x=0; x<100; x++){
        continue;
        printf("%d ", x); // Esto no se ejecuta nunca.
    }
}
```

En los bucles while y do-while, una sentencia continue hará que el control vaya directamente a la condición de prueba y que continúe después el proceso del bucle. En el caso de for, se lleva a cabo la parte de incremento del bucle, se ejecuta la prueba condicional y el bucle continúa.

continue se utiliza con poca frecuencia, no porque su utilización sea una práctica pobre, sino simplemente porque normalmente no hay buenas aplicaciones para él.

Selección de caminos con la sentencia switch.

La sentencia switch es la sentencia de selección múltiple de C. Funciona de la siguiente forma: se compara una variable con un alista de constantes enteras o de carácter; cuando concuerda con alguna, se ejecuta la secuencia de sentencias asociada con esa constante.

El formato general de switch es el siguiente:

```c
      switch(variable){
            case constante 1:
                  //secuencia de sentencias
                  break;
            ...
            case constante n:
                  //secuencia de sentencias
                  break;
            ...
            defaul:
                  //secuencia de sentencias
                  break;
}
```

La porción default es opcional y se ejecuta si no coincide con ninguna de las otras. Si no hay coincidencias y no existe default, no se lleva a cabo ninguna acción. Veamos un ejemplo:

```c
#include <stdio.h>

int main(){
    int i;
    printf("Introduzca un número del 1 al 4: ");
    scanf("%d", &i);
    switch (i){
        case 1:
            printf("El número introducido es uno");
```

```
            break;
        case 2:
            printf("El número introducido es dos");
            break;
        case 3:
            printf("El número introducido es tres");
            break;
        case 4:
            printf("El número introducido es cuatro");
            break;
        default:
            printf("Número no reconocido");
    }
}
```

La sentencia switch se diferencia de if, en que switch solamente puede verificar la igualdad, mientras que la expresión condicional de if puede ser de cualquier tipo.

switch solo funciona con tipos int o char.

La sentencia goto.

La sentencia goto está en desuso.

Ejemplo con lo aprendido hasta ahora:

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

int main() {
    char c;
    printf("Introduce caracteres (presiona Enter para terminar):\n");

    while ((c = getchar()) != '\n') {
        if (islower(c)) { // islower() devuelve verdadero si el carácter es una
                          // letra minúscula.
            c = toupper(c); // toupper() convierte el carácter a mayúscula.
        }
        putchar(c); // putchar() muestra el carácter en pantalla.
    }

    return 0;
}
```

En este programa, se comienza por imprimir un mensaje pidiendo al usuario que introduzca caracteres y se muestra cómo terminar el programa.

Luego se utiliza un ciclo while que lee los caracteres introducidos por el usuario, uno por uno, hasta que se pulse la tecla Enter. Dentro del ciclo, se verifica si el carácter es una letra minúscula utilizando la función islower() de la biblioteca ctype.h. Si el carácter es una letra minúscula, se convierte a mayúscula utilizando la función toupper() de la misma biblioteca. Luego se muestra el carácter en pantalla utilizando la función putchar(). Finalmente, el programa termina y se devuelve el valor 0 para indicar que se ha ejecutado sin errores.

TIPOS DE DATOS, VARIABLES Y EXPRESIONES

Utilización de los tipos de datos en C.

C tiene 6 tipos de datos básicos: void, char, int, float, double y bool(C99). Estos tipos básicos, excepto los tipos void y bool, se pueden modificar utilizando modificadores de tipo C para que se ajusten más adecuadamente a nuestras necesidades específicas. Estos modificadores son:

```
signed
unsigned
long
short
```

El modificador de tipo precede al nombre del tipo. Por ejemplo, esto declara un entero long:

```
long int i;
```

El modificador signed es redundante y se utiliza para char, para crear un entero pequeño con signo.

C permite un anotación abreviada para la declaración de enteros unsigned, short y long. Basta con utilizar la palabra unsigned, short o long sin el int. El int se sobreentiende. Por ejemplo:

```
unsigned count; --> unsigned int count;
short count;
long count;
```

Es muy importante entender que las variables tipo char se pueden utilizar para que contengan valores aparte del simple juego de caracteres ASCII. C establece una pequeña distinción entre un carácter y un entero, excepto por la magnitud de los valores que cada uno puede contener. Por eso, la variable char se puede utilizar también como un entero "pequeño", con un rango de -128 a 127 y se puede utilizar en lugar de un entero, cuando la situación no requiere de números grandes.

Para utilizar printf y que muestre un short se utiliza %hd, para que muestre un long se utiliza %ld y existen otros modificadores, aquí está la lista:

```
short               --> %hd
long                --> %ld
double              --> %lf
unsigned            --> %u
long double         --> %Lf
unsigned long int   --> %lu
```

La función scanf() opera de un modo similar a printf().

Aprendizaje de dónde se declaran las variables.

Son variables locales las que están dentro de una función y su alcance se limita a la función en la que están declaradas. Cuando se llama a una función se crean sus variable locales y al abandonarla se destruyen. Esto significa que las variables locales no pueden mantener sus valores entre llamadas.

El lenguaje C contiene la palabra reservada auto, que se puede utilizar para declarar variables locales. Sin embargo, dado que se asume que todas las variables locales son auto por omisión, no se utiliza prácticamente nunca.

A diferencia de la variables locales, las variables globales son conocidas a lo largo de todo el programa y pueden ser utilizadas por cualquier trozo de código del programa. Además, mantendrán su valor durante toda la ejecución del programa. Estas se declaran fuera de cualquier función.

Las variables que están en main(), son privadas o locales a ella. Debido a que son declaradas dentro de main(), ninguna otra función puede tener acceso directo a ellas.

Cada variable local de una función comienza a existir cuando se llama a la función y desaparece cuando la función termina.

Existen las variables externas a todas las funciones, esto es, variables a las que toda función puede tener acceso por su nombre. Una variable externa debe definirse, exactamente una vez, fuera de cualquier función; esto fija un espacio de almacenamiento para ella. La variable debe declararse también en cada función que desea tener acceso a ella; esto establece el tipo de variable: La declaración debe ser una proposición 'extern' explícita, o bien puede estar implícita en el contexto. Por ejemplo:

```c
#include <stdio.h>

int max = 100;

int main(){
    extern int max;
    printf("%d", max);

    return 0;
}
```

Una mayor aproximación a las constantes.

C hace suposiciones de tipo sobre las constantes, para asignarles el tipo adecuado. En los casos en los que la suposición de tipo que haga C sobre una constante numérica no sea la adecuada, C permite que se especifique el tipo exacto de una constante numérica utilizando un sufijo. Para los tipos en punto flotante, si se sigue el número con una "F", se trata al número como un float. Si se sigue con una "L", el número se convierte en un long double. Para los tipos enteros, el sufijo "U" significa unsigned y "L" long.

Aunque C permite definir constantes de cadena, formalmente no tiene un tipo de datos de cadena. En lugar de eso, como se verá más adelante, las cadenas se corresponden en C con los arrays de caracteres.

Inicialización de variables.

Esta sentencia declara a count como int y le da un valor inicial:

```c
    count int = 100;
```

En las variables que no se inicialicen se debe asumir que contienen valores desconocidos. Aunque algunos compiladores de C inicializan automáticamente a 0 las variables no inicializadas, no se debe contar con ello.

Mezcla de tipos en expresiones.

C permite la mezcla de tipos en expresiones, ya que tiene un juego específico de reglas de conversión que dictan cómo se resuelven las diferencias de tipo. Por ejemplo, este código en C es perfectamente válido:

```c
#include <stdio.h>

int main(){
    char ch = '0';
    int i = 10;
    float f = 10.2;
    double outcome;

    outcome = ch*i/f;    // Mezcla de tipos.
}
```

Programación con moldes de tipo.

Cuando se utiliza un molde, se origina un cambio de tipo temporal:

```
    (tipo) valor,
```

donde tipo es el nombre de un tipo válido de datos de C. Por ejemplo:

```c
#include <stdio.h>

int main(){
    float f = 10.2;
    printf("%d", (int) f); // Imprime f como entero. Molde (int).
}
```

EXPLORACIÓN DE ARRAYS Y CADENAS

Declaración de arrays unidimensionales.

En C, un array es una lista de variables del mismo tipo que se referencian por un nombre común. A una variable individual del array se le llama elemento del array.

Para declarar un array unidimensional, se utiliza la forma general:

```
tipo nombre_de_variable[tamaño]
```

donde tamaño especifica el número de elementos del array. Por ejemplo:

```
int mi_array[20];
```

```
#include <stdio.h>

int main(){
    int sqrs[10];

    for (int i = 0; i < 11; i++){
        sqrs[i - 1] = i * i;
    }
    for (int i = 0; i < 10; i++){
        printf("%d", sqrs[i]);
    }
    return 0;
}
```

En C no se puede asignar un array completo a otro. Por ejemplo, este fragmento es incorrecto:

```
char a1[10], a2[10];
a2 = a1;                // Esto está mal. No se puede hacer.
```

Si se quieren copiar los valores de todos los elementos de un array en otro array, se debe hacer copiando cada elemento por separado.

```
#include <stdio.h>

int main(){

    int a1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; /*Debe inicializarse al
                                                    declararse. Esto ocurre
                                                    solo con los arrays.*/
    int a2[10];

    for (int i = 0; i < 11; i++){
        a2[i] = a1[i];
    }
    for (int i = 0; i < 10; i++){
        printf("%d\n", a2[i]);
    }
    return 0;
}
```

Los arrays que se quieran inicializar, deben inicializarse al declarar el array. En el código anterior vemos dos arrays a1 y a2. a1 es inicializado en su declaración, si intentáramos inicializarlo después el compilador devolvería un error.

Utilización de cadenas.

El uso más común de un array unidimensional en C es la cadena. C no tiene incorporado el tipo cadena (string), soporta cadenas usando arrays unidimensionales de caracteres. Una cadena se define como un array de caracteres con un carácter de terminación nulo. En C, cualquier carácter nulo es 0. Esto significa que para declarar un array de caracteres es necesario que sea de un carácter más que la cadena más larga que tenga que contener, para dejar sitio al carácter nulo. En una constante de cadena, el compilador pone automáticamente un carácter nulo de terminación.

Para leer una cadena por teclado se usa fgets(), perteneciente a <stdio.h>. La función fgets() lee caracteres hasta que hay un salto de línea '\n'. El salto de línea también se almacena, la función agrega automáticamente el carácter de terminación NULL '\0', que termina la cadena. Su uso es del siguiente modo:

```
    fgets(nombre_cadena, núemro_de_caracteres, FILE);
```

por ejemplo:

```
    fgets(str, 80, stdin); // stdin, entrada estándar, teclado.
```

```
#include <stdio.h>

int main(){
    char str[80];

    printf("Introduzca una cadena de menos de 80 caracteres:\n");
    fgets(str, 80, stdin);
    for (int i = 0; str[i]; i++) // Imprime carácter a carácter.
        printf("%c", str[i]);
    return 0;
}
```

Este programa muestra la cadena carácter a carácter, pero hay un modo mejor de mostrar la cadena en pantalla:

```
#include <stdio.h>

int main(){
    char str[80];

    printf("Introduzca una cade de menos de 80 caracteres:\n");
    fgets(str, 80, stdin);
    printf(str); // Imprime la cadena completa.
    return 0;
}
```

Dado que el primer argumento para printf() es una cadena, simplemente se utiliza str sin ningún índice como el primer argumento para printf(). Si se quisiera mostrar una nueva línea, se podría mostrar así:

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

Ejercicio. Escriba un programa que introduzca una cadena y después la muestre en pantalla de atrás hacia delante.

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

int main(){
    char str[80];

    printf("Introduzca una cade de menos de 80 caracteres:\n");
    fgets(str, 80, stdin);
    printf("%d", strlen(str)); // Imprime la longitud de la cadena.
    for (int i = 0; str[i]; i++){
        printf("%c", str[(strlen(str)) - i - 1]); // Imprime la cadena al revés.
    }
    return 0;
}
```

En el ejercicio anterior hacemos uso de strlen(), que es parte de la biblioteca estándar en "string.h". Devuelve la longitud de la cadena.

La expresión 'strlen(str))-i-1' contiene un -1 porque fgets guarda como parte de la cadena el retorno de carro al pulsar intro.

Otras funciones básicas para trabajar con cadenas son las siguientes (todas requieren el archivo de cabecera string.h):

```
strcpy() --> strcpy(hacia, desde)
Copia los contenidos de desde en hacia.

strcat() --> strcat(hacia, desde)
Añade los contenidos de una cadena en otra.

strcmp() --> strcmp(c1, c2)
Compara dos cadenas.
```

strcmp devuelve 0 si las cadenas son iguales. Devuelve un valor negativo si c1 es menor que c2 y uno positivo si c1 es mayor que c2. Las cadenas se comparan por orden lexicográfico; es decir, siguiendo el orden del diccionario.

Arrays multidimensionales.

Los arrays se pueden crear de dos o más dimensiones. Por ejemplo para crear un array entero bidimensional de 10x12 llamado count, se utilizaría la sentencia:

```
    int count[10][12];
```

Se trata de un array de arrays unidimensionales. Es más fácil imaginarlo como una matriz.

```
#include <stdio.h>

int main(){
    int array[4][5]; // Array de cuatro filas y cinco columnas.
```

```
    int i, j;

    for (i = 0; i < 4; i++){      // Recorre las filas del array.
        for (j = 0; j < 5; j++){ // Recorre las columnas del array.
            array[i][j] = i * j; // Calcula productos de elementos del array.
        }
    }
    for (i = 0; i < 4; i++){
        for (j = 0; j < 5; j++){
            printf("%d", array[i][j]);
            printf("\n");
        }
    }
}
```

Inicialización de arrays. Ejemplos.

```
int i[5] = {1, 2, 3, 4, 5};
char a[3] = {'A', 'B', 'C'};
char nombre[8] = "Alberto";
```

En el tercer ejemplo no hay llaves alrededor de la cadena. No se utilizan en esta forma de inicialización. Dado que las cadenas en C deben terminar con un carácter nulo, se debe cerciorar de que el array que se declare es lo suficientemente largo para incluir el carácter nulo 'nombre[8]'. Cuando se utiliza una constante de cadena el compilador suministra automáticamente el carácter de terminación nulo.

```
int sqr[3][3] = {1, 2, 3 4, 5, 6 7, 8, 9};
int pwr[] = {1, 2, 3, 4, 5, 6, 7, 8};
```

Los array multidimensionales se inicializan del mismo modo que los unidimensionales.

Si se está inicializando un array unidimensional, no tiene qué especificar el tamaño del array (basta con no poner nada dentro del los corchetes).

Loa arrays que no tienen sus dimensiones especificadas explícitamente se llaman arrays de tamaño indeterminado: Un array de tamaño indeterminado es útil, ya que es más fácil cambiar el tamaño de la lista de inicialización sin tener que contarla y después cambiar el tamaño del array. Esto es explícitamente importante a la hora de inicializar cadenas. Aquí se inicializa un array de tamaño indeterminado:

```
char promt[] = "Introduzca su nombre: ";
```

Ejemplo: búsqueda de datos en un array bidimensional.

```
#include <stdio.h>

long cpu[][2] = {// La 1ª dimensión no es necesaria en este tipo de declaración.
      4,
      4,
      10,
      20,
      40};
int main(){
    long procesador;
    int i;
```

```
    printf("Introduzca el número de procesador: ");
    scanf("%ld", &procesador);

    // Examina la tabla.
    for (i = 0; i < 5; i++){
        if (procesador == cpu[i][0]){
            printf("La velocidad media es de %d MHz.\n", cpu[1][1]);
            break;
        }
    }
    // Comunicar error si no encuentra.
    if (i == 5){
        printf("Procesador no encontrado.\n");
    }
    return 0;
}
```

Los arrays son objetos mutables, se pueden cambiar sus elementos y/o el contenido entero. El siguiente programa muestra un ejemplo:

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

int main(){

    char str[] = "Me gusta C."; // No es necesario declarar su dimensión.
    strcpy(str, "Hola.");
    printf(str); // Imprime Holo (hemos modificado el array).

    return 0;
}
```

Construcción de arrays de cadenas.

Los arrays de cadenas, a menudo llamados tablas de cadenas, con muy comunes en la programación en C. Un atabla de cadenas bidimensional se crea como cualquier otro array. Por ejemplo:

```
    char nombres[10][40];
```

Esa declaración especifica una tabla que puede contener 10 cadenas, cada una de hasta 40 caracteres de longitud (incluyendo el carácter de terminación nulo). Para acceder a una cadena dentro de esta tabla, se especifica solamente el primer índice. Por ejemplo para leer desde el teclado en la tercera cadena de 'nombres', se utiliza esta sentencia:

```
    fgets(nombres[3], 40, stdin); // 40 caracteres de longitud.
```

Del mismo modo, para mostrar la primera cadena se utiliza esta sentencia printf():

```
    printf(nombres[0]);
```

La declaración que sigue crea una cadena tridimensional con tres listas de cadenas. Cada lista tiene cinco cadenas de longitud, y puede contener hasta 80 caracteres:

```
    char animales[3][5][80];
```

Para acceder a una cadena específica en esta situación, se tienen que especificar las dos primeras dimensiones. Por ejemplo, para acceder a la segunda cadena de la tercera lista, especifique:

```
animales[2][1]
```

Ejemplo con lo aprendido hasta ahora. Se trata de un programa que lee un conjunto de líneas de texto e imprime la de mayor longitud:

```
#include <stdio.h>
#define MAXLINE 1000 // Tamaño máximo de la línea de entrada.

int getln(char line[], int maxline); // Definición de funciones prototipo.
void copy(char to[], char from[]);

int main(){
    int len;                // Longitud actual de la línea.
    int max;                // máxima longitud vista hasta el momento.
    char line[MAXLINE];     // Línea de entrada actual.
    char longest[MAXLINE];  // La línea más larga se guarda aquí.
    max = 0;
    while ((len = getln(line, MAXLINE)) > 0){
        if (len > max){
            max = len;
            copy(longest, line);
        }
    }
    if (max > 0){
        printf("%s", longest);
    }
    return 0;
}
    // getln: lee una línea en s, devuelve su longitud.
    int getln(char s[], int lim){
        int c, i;
        for (i = 0; (i < lim - 1) && ((c = getchar()) != EOF) && (c != '\n');
             ++i){
            s[i] = c;
        }
        if (c == '\n'){
            s[i] = c;
            ++i;
        }
        s[i] = '\0';
        return i;
    }

    // copy: copia 'from' en 'to'; supone que 'to' es suficientemente grande.
    void copy(char to[], char from[]){
        int i = 0;
        while ((to[i] = from[i] != '\0')){
            ++i;
        }
    }
```

La función getln() extrae la siguiente línea de la entrada; cuando se encuentre una línea que es mayor que la anteriormente más larga, se debe guardar en algún lugar, esto lo hace una segunda función, copy(), que copia la nueva línea a un lugar seguro.

Finalmente se necesita main() para controlar getln() y copy(). main() y getln() se comunican a través de un par de argumentos y un valor de retorno.

En getln la longitud del arreglo s[] no es necesaria, puesto que su tamaño se fija en main() (el propósito de proporcionar el tamaño de un arreglo es fijar espacio de almacenamiento continuo). El getln() se utiliza un return para devolver un valor a quien la llama.

Algunas funciones devuelven un valor útil; otras, como copy(), se emplean únicamente por su efecto y no devuelven valor. El tipo de retorno de copy() es void, el cual establece explícitamente que ningún valor es devuelto.

UTILIZACIÓN DE PUNTEROS

Un puntero es, básicamente, la dirección de un objeto. En C, los punteros son muy importantes, y la razón es porque gran parte de la potencia del lenguaje C se deriva del modo exclusivo en que se implementan los punteros en él.

Como hemos dicho, un puntero es una variable que guarda la dirección de memoria de otro objeto. Por ejemplo, si una variable llamada p contiene la dirección de otra variable llamada q, entonces se dice que p apunta a q. Por eso, si q estuviese en la posición de memoria 100, entonces p tendría el valor de 100. En otras palabras, el puntero tiene como valor la posición de memoria del objeto al que apunta.

Para declarar una variable de puntero, se utiliza esta forma general:

```
tipo *nombre-de-varibale;
```

Aquí, tipo es el tipo base del puntero, el tipo base especifica el tipo del objeto al que puede apuntar el puntero. Por ejemplo, la siguiente sentencia crea un puntero de un entero:

```
int *p;
```

C contiene dos operadores de puntero especiales: * y &. El operador & devuelve la dirección de memoria de la variable a la que precede. El operador * devuelve el valor almacenado en la dirección de memoria a la que precede. Por ejemplo:

```
#include <stdio.h>

int main(){
    int *p, q;
    q = 100;    // Asigna 100 a 'q'.
    p = &q;     // Asigna a p la dirección de 'q'.

    printf("%d", *p);       // Muestra el valor de 'q' utilizando el puntero.
                            // Imprime 100.

    return 0;
}
```

Veamos paso a paso el programa:

Se definen dos variables: p, que se declara como un puntero a entero, y q, que es un entero. Después, se asigna el valor 100 a q. En la siguiente línea se asigna a p la dirección de q (se puede expresar el operador & como "dirección de"). Por último, se muestra el valor utilizando el operador * aplicado a p (el operador * se puede expresar como "en la dirección"). La sentencia printf() se puede leer como:

```
imprimir el valor que hay en la dirección q, que es 100.
```

Cuando el valor de una variable se referencia a través de un puntero, al proceso se le llama indirección.

Asignar un valor a una variable utilizando un puntero.

Es posible utilizar el operador * al lado izquierdo de una sentencia de asignación para asignar un nuevo valor a una variable utilizando un puntero a ella. Esta programa asigna indirectamente un valor a q utilizando el puntero p:

```c
#include <stdio.h>

int main(){
    int *p, q;

    p = &q;      // Obtener la dirección de q.
    *p = 199;    // Asignar a q un valor utilizando un puntero.

    printf("El valor de q es %d", q); // Imprime "El valor de q es 199."

    return 0;
}
```

En los ejemplos que se acaban de mostrar no hay ninguna razón para usar un puntero. Sin embargo, al ir aprendiendo más sobre C, entenderá por qué son importantes los punteros.

No se debe utilizar nunca un puntero de un tipo para apuntar a un objeto de un tipo distinto.

Expliquemos esto mejor.

Dirección de memoria de una variable.

El operador '&' devuelve la dirección de memoria de la variable sobre la que se aplica. En el siguiente ejemplo obtenemos la dirección de memoria de 'x' con el uso del operador '&'.

```c
#include <stdio.h>

int main(){
    int x;  // [*] No hace falta inicializar x para que obtenga su dirección.

    // Dirección de memoria.
    printf("La variable x esta almacenada en '%lld'\n", &x); /* Devuelve la
                                                                dirección. */
    return 0;
}
```

[*] No hace falta asignar un valor inicial para que la variable tenga una dirección de memoria.

Declaración de un puntero.

Un puntero se declara de la siguiente forma: indicando el tipo de datos de la variable a la que apunta e incluyendo el operador '*' antes del identificador.

Por ejemplo:

```c
int main(){
    // p1 apuntando a una variable tipo entero.
    int *p1;
    // p2 apuntando a una variable tipo carácter.
    char *p2;
    // p3 apuntando a una variable genérica.
    void *p3;

    return 0;
}
```

Un puntero es una variable de tipo int. El contenido de una variable puntero es una dirección de memoria, un valor tipo entero. Su tamaño depende del sistema. Para sistemas de 64 bits, tiene un tamaño de 8 bytes.

Asignación de los punteros.

En el siguiente ejemplo p1 apunta a x, p2 apunta a y y p3 apunta a la misma variable que p1, es decir a x.

```c
int main(){
  int x, y;
  // p1 apunta a x.
  int *p1 = &x, *p2, *p3; // A p1 se le asigna la dirección de memoria de x.
  // p2 apunta a y.
  p2 = &y;
  // p3 apunta a la misma variable que p1, es decir a x.
  p3 = p1;
}
```

Operador '*'.

El operador '*' aplicado a un puntero proporciona el valor de la variable a la que apunta el puntero. Es decir, si el puntero p (*p) apunta a x (p = &x), entonces *p y x tienen el mismo valor.

```c
#include <stdio.h>

int main(){
  int x = 10, *p;
  // Operador & para obtener la dirección de x.
  p = &x;
  // *p y x son lo mismo
  printf("La variable apuntada vale %d", *p);

  return 0;
}
```

En resumen. Por ejemplo:

```c
    int x;      // x es un entero.
    p = &x;     // p es una dirección de memoria
    *p == x;    // *p y x son lo mismo, mismo valor.
```

```c
#include <stdio.h>

int main(){
    int x, *p;
    p = &x;
    if(*p == x){
        printf("Tienen el mismo valor");
    }else{
        printf("Tienen valores diferentes");
    }
    return 0;
}
```

```c
#include <stdio.h>

int main(){
    int x, *p;
    p = &x;

    printf("%lld\n %lld", &x, &*p); // Imprime la misma dirección para ambos.
                                    // Luego son lo mismo.
    return 0;
}
```

En definitiva *p es un puntero de memoria que apunta a x, el contenido de p es la dirección de memoria de x. El valor de *p es el mismo que el de x.

Restricciones a las expresiones de punteros.

En general, los punteros se pueden usar como cualquier otra variable, pero hay una serie de restricciones. Solamente hay cuatro operadores que se puedan aplicar a las variables de puntero: +, ++, - y --. Es más, solo se pueden sumar o restar cantidades enteras (recordar que el puntero es un entero). No se puede, por ejemplo, sumar un número en coma flotante a un puntero. Aparte de sumar o restar un entero no se puede realizar ningún otro tipo de operación aritmética con punteros.

Se pueden aplicar los operadores de incremento y decremento bien al propio puntero o al objeto al que apunta. *p++ y (*p)++ no son lo mismo. El paréntesis hace que se incremente el valor al que apunta p.

```c
#include <stdio.h>

int main(){
    int x = 1, *p;
    p = &x;

    printf("%lld - %lld - %d \n", *p++, (*p)++, x);    // Imprime: 2 - 1 - 1.
    (*p)++; // Incrementa al objeto al que apunta.
    printf("%d\n", x);    // Imprime: 2.

    return 0;
}
```

La aritmética de punteros es uno de los componentes más valiosos del lenguaje C.

Utilización de punteros con arrays.

En C existe una estrecha relación entre los punteros y los arrays. De hecho, a menudo son intercambiables. Esta relación entre los dos es la que hace que su implementación sea única y potente.

Cuando se utiliza un nombre de array sin índice, se está generando un puntero al principio del array. Esto es por lo que no se utilizan índices cuando se lee una cadena utilizando fgets(), por ejemplo. Lo que se está pasando a fgets() no es un array, sino un puntero. De hecho en C no se puede pasar un array a una función, solo se puede pasar un puntero al array a dicha función. Este hecho es crucial para entender el lenguaje C. La función fgets() utiliza el puntero para cargar el array al que apunta con los caracteres que se introducen por teclado. Es decir, el identificador de un array es un puntero que apunta al primer elemento del array.

```c
#include <stdio.h>

int main(){
    int array[3] = {1, 2, 3};
    int *p;

    p = &array[0];
    printf("El primer elemento es %d\n", *p);   // Imprime: 1.

    p = array;  // Identificador del array (nombre del array sin índice).
                // Se genera un puntero al principio del array.
    printf("El primer elemento es %d\n", *p);   // Imprime: 1. Igualmente.

    return 0;

}
```

Puesto que un array sin índice es un puntero al principio del array, es razonable que se pueda asignar ese valor a otro puntero y que se acceda al array utilizando aritmética de punteros. Y de hecho, esto es exactamente lo que se puede hacer.

Mover un puntero por un array.

Podemos recorrer un array a través de su puntero usando sumas y restas:

```c
#include <stdio.h>

int array[10] = {0, 2, 3, 4, 5, 6, 7, 8, 9, 9};

int main(){
    int *p;
    p = array; // Asigna a p la dirección del principio del array.
   /* Imprime los elementos primero, segundo y tercero del array usando el
      puntero. */
    printf("%d %d %d\n", *p, *(p+1), *(p+2));
    // Esta línea hace lo mismo usando a.
    printf("%d %d %d\n", array[0], array[1], array[2]);

    return 0;
}
```

Otro modo de recorrer un array con un puntero:

```c
#include <stdio.h>
#define N 10

int main() {
    int array[N] = {1};     // Declaración e inicialización de array.
    int *pArr, *pFin;       // Declaración de punteros.

    // Puntero apuntando al segundo elemento.
    pArr = array + 1;

    // Puntero apuntando al ultimo elemento.
    pFin = array + N - 1;

    // Comparamos los punteros para avanzar.
    while (pArr <=  pFin) {

        // array[i] = array[i - 1] + 1
        *pArr = *(pArr - 1) + 1;

        printf("%d\t", *pArr);
        ++pArr; // Movemos el puntero por el array.
    }
    return 0;
}
```

Podemos modificar los elementos de un array usando un puntero. Veamos un ejemplo sencillo:

```c
#include <stdio.h>
int main(){
    int i, array[3];
    int *p;
    p = array;
    //array[0] = 1
    *p = 1;
    //array[1] = 2
    *(p + 1) = 2;
    //array[2] = 3
    *(p + 2) = 3;
    // Imprime: El array es: 1 2 3.
    printf("El array es: %d, %d, %d.\n", array[0], array[1], array[2]);

    return 0;
}
```

Indexación de punteros.

Los punteros y los arrays están vinculados por más que el simple hecho de que se pueda acceder a los elementos del array por aritmética de punteros. Se puede indexar un array como si fuese un puntero.

```c
#include <stdio.h>

char str[] = "Los punteros son divertidos";
int main(){
```

```
    char *p;
    int i;

    p = str;

    for (int i=0; p[i]; i++){ // Itera hasta encontrar un elemento nulo.
        printf("%c", p[i]);
    }
    return 0;
}
```

Solamente se puede indexar un puntero cuando ese puntero apunta a un array.

Dado que un nombre de array sin índice es un puntero al principio del array, si se quiere, se puede utilizar aritmética de punteros en lugar de indexación de arrays para acceder a los elementos de un array:

```
#include <stdio.h>

int main(){
    char str[80];
    *(str+3) = 'c';

    printf("%c", *(str+3)); // Aritmética de punteros. Imprime: c.
    printf("%c", str[3]);   // Indexación de arrays. Imprime lo mismo: c.

    return 0;
}
```

No se puede, sin embargo, modificar el valor del puntero que se ha generado utilizando un nombre de array. Por ejemplo, en el programa anterior

```
    str++;
```

sería una sentencia incorrecta.

IMPORTANTE: Se debe pensar en el puntero que genera str como una constante que siempre apunta al principio del array. Por eso no es válido modificarlo, el compilador informaría de un error.

Ejemplo con lo visto hasta ahora.

El siguiente programa pide una cadena al usuario y después imprime la cadena, primero en mayúsculas y después en minúsculas. Esta versión utiliza indexación de arrays para acceder a los caracteres de la cadena.

Se hace uso de dos funciones de la biblioteca estándar, toupper() y tolower(), a estas funciones se las llama utilizando un argumento de carácter. Si el carácter es una letra minúscula/mayúscula devuelven la letra en mayúscula/minúscula, o sin cambios en el caso contrario.

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

int main(){
```

```
    char str[80];
    int i;

    printf("Introduzca una cadena: ");
    fgets(str, 80, stdin);

    for(i=0; str[i]; i++){
        str[i] = toupper(str[i]);
    }

    printf("%s\n", str);    // Cadena en mayúsculas.

    for(i=0; str[i]; i++){
        str[i] = tolower(str[i]);
    }

    printf("%s\n", str);    // Cadena en minúsculas.

    return 0;
}
```

Ahora escribiremos el mismo programa haciendo uso de un puntero para acceder a la cadena. Este segundo enfoque es el modo en que se vería si lo escribiese un programador de C profesional, ya que incrementar un puntero es, por lo general, más rápido que indexar una array.

IMPORTANTE:

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

int main(){
    char str[80], *p;

    printf("Introduzca una cadena: ");
    fgets(str, 80, stdin);
    p = str;

    while(*p){
        *p++ = toupper(*p); // *p = toupper(*p); p++; [*]
    }

    printf("%s\n", str);    // Cadena en mayúsculas.

    p = str;    // Inicializar p.

    while(*p){
        *p++ = tolower(*p); // *p = tolower(*p); p++; [*]
    }

    printf("%s\n", str);    // Cadena en minúsculas.

    return 0;
}
```

[*] Dado que ++ sigue a la p, primero se obtiene el valor al que apunta p y después se incrementa p para que apunte al siguiente elemento.

Utilización de punteros a constantes de cadena.

IMPORTANTE:

```c
#include <stdio.h>

int main(){
    char *p;
    p = "uno dos tres";
    printf(p);          // Imprime: uno dos tres.
    printf("%c\n", *p); // Imprime: u. El principio de la cadena.

    return 0;
}
```

Escrito más eficientemente:

```c
#include <stdio.h>

char *p = "uno dos tres";

int main(){
    printf(p);          // Imprime: uno dos tres
    printf("%c\n", *p); // Imprime: u. El principio de la cadena.

    return 0;
}
```

Cadenas y punteros.

Como hemos dicho, el identificador de una cadena, es un puntero que apunta al primer elemento de la cadena:

```c
#include <stdio.h>

int main(){
    char letras[3] = {'a', 'b', 'c'};
    char *p;

    // p apunta al primer elemento.
    p = &letras[0];
    printf("%c\n", *p);    // Imprime: a.

    // Aquí se ve mejor: el identificador de una cadena es un puntero: 'p'.
    p = letras;
    printf("%c\n", *p);    // Imprime: a.

    return 0;
}
```

Recorrido de una cadena.

```c
#include <stdio.h>

int main(){
    char mensaje[] = "Hola, mundo";
    char *p = mensaje;
    int i = 0;
    // Movemos el puntero por la cadena.
```

```
    while (*p != '\0'){
        printf("%c", *p);
        p++;    // Incrementa el puntero para pasar al siguiente carácter.
    }
    printf("\n");

    return 0;
}
```

Aritmética de punteros con cadenas.

```
#include <stdio.h>

int main(){
    char texto[] = "Hola, mundo";
    char *p = texto;

    printf("%c", *p); // IMPORTANTE: Esto que quede claro, imprime: H.

    while(*p != '\0'){
        ++p;
    }
    // Imprime: La cadena texto tiene 11 caracteres.
    printf("La cadena texto tiene %d caracteres.\n", p-texto);

    return 0;
}
```

Creación de arrays de punteros.

Los punteros pueden estructurarse también como arrays, como cualquier otro tipo de datos. Por ejemplo:

```
    int *pa[20];    // Array de punteros a entero de 20 elementos.
```

Dado que 'pa' es un array de punteros, el único valor que pueden contener los elementos del array son las direcciones de variables enteras (int). Por ejemplo:

```
    pa[8] = &mivar; // La dirección de mivar se asigna al noveno elemento de pa.
```

Para asignar el valor 100 al entero al que apunta el tercer elemento de pa, se utiliza la sentencia:

```
    *pa[3] = 100;   // IMPORTANTE.
```

Indirección múltiple. Punteros que apuntan a punteros.

Es posible hacer que un puntero apunte a otro puntero, a esto se le llama indirección múltiple. El primer puntero contiene la dirección del segundo, el cual apunta a la posición (dirección) que contiene el objeto.

Para declarar un puntero a otro puntero se coloca un asterisco adicional delante del nombre del puntero. Por ejemplo, mp es un puntero a un puntero a carácter:

```
    char **mp;
```

Es importante entender que mp no es un puntero a carácter sino un puntero a puntero a carácter. El acceso al valor al que apunta un puntero a puntero se obtiene aplicando dos veces el operador '*':

```
#include <stdio.h>

int main(){
    char **mp, ch;
    p = &ch;     // Obtener la dirección de ch.
    mp = &p;     // Obtener la dirección de p.
    **mp = 'A'; // Asignar a ch el valor 'A' usando indirección múltiple.

    return 0;
}
```

A ch se le asigna un valor indirectamente usando dos punteros. La indirección múltiple es muy valiosa, aunque en este momento no se comprenda su alcance.

Ejemplo. El siguiente programa muestra cómo se puede introducir una cadena por medio de fgets() utilizando un puntero a puntero a la cadena:

```
#include <stdio.h>

int main(){
    char *p, **mp, str[80];
    p = str;
    mp = &p;

    printf("Introduzca su nombre: ");
    fgets(*mp, 80, stdin);
    printf("Hola, %s", *mp);

    return 0;
}
```

Observe que cuando utiliza mp como argumento para fgets() y printf() solo se utiliza un *. Esto es porque ambas funciones requieren un puntero a una cadena para su operación.

IMPORTANTE: Recuerde, **mp es un puntero a p. Sin embargo, p es un puntero a la cadena str, por tanto *mp es un puntero a str.

Utilización de punteros como parámetros.

Los punteros se pueden pasar a funciones como argumentos. Por ejemplo, cuando se llama a una función como strlen() con el nombre de una cadena, realmente se está pasando un puntero a la función. La función debe declararse como receptora de un puntero.

Cuando se pasa un puntero a una función, el código dentro de esa función tiene acceso a la variable a la que apunta el parámetro. Esto significa que la función puede cambiar la variable utilizada para llamar a la función. Ahora se puede entender por qué tiene que preceder un & al nombre de la variable cuando utiliza scanf(). Para que scanf() pueda modificar uno de sus argumentos, se debe pasar un puntero a ese argumento.

Arrays multidimensionales y punteros.

También es posible utilizar un puntero como medio de acceso a arrays multidimensionales, pero en este caso, es más sencillo utilizar indexación de arrays que aritmética de punteros.

UNA APROXIMACIÓN A LAS FUNCIONES

Una función puede devolver cualquier tipo de datos. Una función se define del modo que sigue:

```
tipo nombre-de-función(lista-de-parámetros){
    // sentencias
}
```

Aquí, tipo, especifica el tipo de valor que devuelve la función. Una función puede devolver cualquier tipo de datos excepto un array. Si no hay un especificador del tipo de datos, entonces el compilador de C, automáticamente, supone que la función devuelve un entero. Es decir, int es el tipo implícito cuando no aparece un especificador de tipo.

Si una función no devuelve ningún valor, se puede declarar la función como void. Esto impide que la función se utilice en la parte derecha de una sentencia de asignación. El siguiente programa utiliza una función void:

```c
#include <stdio.h>

void message();

int main(){
    message();

    return 0;
}

void message(){
    printf("Esto es un mensaje");
}
```

Muchos programadores declaran main() como void cuando no se utiliza un valor de retorno explícito. El valor de retorno por defecto de main, si el programa ha finalizado satisfactoriamente, es 0.

Con bastante frecuencia, cuando una función devuelve un carácter, se permite que el tipo de valor de retorno de la función sea, por omisión, un int. La razón de esto se encuentra en el hecho de que C gestiona muy limpiamente la conversión de caracteres a enteros y de enteros a caracteres. No se pierde información. Por ejemplo, el siguiente programa es perfectamente válido:

```c
#include <stdio.h>

int get_a_char();   // Función prototipo.

int main(){
    char ch;
    ch = get_a_char();
    printf("%c", ch);
}

int get_a_char(){
    return 'a';     // [*]
}
```

[*] Cuando se vuelve de get_a_char(), convierte el carácter 'a' a un entero añadiendo contenido de ceros al byte más significativo. Cuando este valor se asigna a ch en main() se quita el byte más significativo.

Utilización de prototipos de función.

Los prototipos permiten que C encuentre e informe sobre cualquier conversión ilegal de tipo entre el tipo de argumentos utilizados para llamar a una función y la definición de tipo de sus parámetros. Los prototipos, a su vez, también facultan al compilador para que informe cuando el número de argumentos de una función no es el mismo que el número de parámetros declarados en la función. La forma general de prototipo de función es la siguiente:

```
tipo nobre-de-función( tipo parm1, tipo parm2, ..., tipo parmN);
```

¿Cómo se escribe el prototipo de una función que no tiene parámetros? Por ejemplo:

```
void line(){
    for(int i=0; i<80; i++{
        printf(".";)
    }
}
```

En C se amplió el uso de la palabra reservada void. Cuando una función no tiene parámetros, su prototipo utiliza void dentro de los paréntesis. Es decir el prototipo adecuado para la función line() anterior es:

```
void line(void);
```

pero, claro, ahora line debe tener este aspecto:

```
void line(void){
    for(int i=0; i<80; i++{
        printf(".";)
    }
}
```

Muchos programadores declaran main() como sigue:

```
int main(void)
```

Sin embargo, main() puede tener algunos parámetros. Se verá más adelante.

Como hemos dicho, cuando se crean funciones que no toman parámetros, se debe utilizar 'void' entre paréntesis, en ambos sitios: en el prototipo de función y cuando la función es declarada. Este programa lo muestra:

```
#include <stdio.h>

int getnum(void);

int main(void){
    int i;

    i = getnum();
    printf("&d", i);
}
int getnum  (void){
```

```
    int x;

    printf("Introduzca un número: ");
    scanf("%d", &x);
    return  x;
}
```

Recursividad en las funciones.

Una función se puede llamar así misma. Este es el concepto de recursividad. Veamos un ejemplo:

```
#include <stdio.h>

void recursive(int);      // En el prototipo, la declaración del tipo en los parámetros
                          // es suficiente. Si se trata de punteros No.
int main(void){
    recursive(0);   // Imprime: 9876543210.
}

void recursive (int i){
    if (i<10){
        recursive(i+1);  // La función se llama a sí misma de forma recursiva.
        printf("%d", i);
    }
}
```

Es importante comprender que no existen múltiples copias de una función recursiva. Solamente existe una copia.

La recursividad es básicamente un nuevo tipo de mecanismo de control de programas. Ese es el motivo por el que toda función recursiva que se escriba tendrá una sentencia if que controle si la función se llamará otra vez a sí misma o regresará. Sin una sentencia así, una función recursiva se ejecutaría alocadamente, utilizando toda la memoria asignada a la pila y haciendo que el programa falle.

IMPORTANTE: Un mayor acercamiento a los parámetros.

Se pueden pasar parámetros a las funciones de dos formas:

    Llamada por valor
    Llamada por referencia.

En la llamada por valor, se copia el valor de un argumento en el parámetro formal de la función. Por lo tanto los cambios hechos en los parámetros de la subrutina no tienen efecto sobre las variables que se utilizan para llamarla.

La llamada por referencia. En este método se copia la dirección de un argumento en el parámetro. Dentro de la subrutina, se utiliza la dirección para acceder al argumento real que se utiliza en la llamada. Esto significa que los cambios hechos en el parámetro afectarán a la variable utilizada para llamar a la rutina.
Por omisión, C utiliza la llamada por valor para pasar argumentos. Esto significa, en general, que no se pueden alterar las variables utilizadas para

llamar a la función. Es decir, IMPORTANTE, lo que ocurra dentro de la función no tendrá efecto en la variable utilizada en la llamada. Sin embargo, es posible provocar una llamada por referencia pasando un puntero a un argumento. Dado que esto hace que se pase la dirección del argumento a la función, entonces es posible cambiar el valor del argumento con efectos fuera de la función.

Ejemplo de una función de llamada por referencia.

```
#include <stdio.h>

void intercambia(int *i, int *j);    // Si se trata de punteros no basta con el tipo.

int main(void){
    int num1, num2;

    num1 = 100;
    num2 = 800;

    printf("num1: %d num2: %d\n", num1, num2);
    intercambia(&num1, &num2);
        printf("num1:  %d  num2:  %d\n",  num1,  num2);    //  Imprime  valores intercambiados.

}

// Intercambia los valores a los que apuntan los dos punetros a enteros.
void intercambia(int *i, int *j){
    int temp = *i;
    *i = *j;
    *j = temp;
}
```

Cuando se utiliza un array como argumento a una función, solo se pasa la dirección del array, no una copia de todo el array, lo que implica llamada por valor. Esto significa que la declaración del parámetro debe ser de un tipo puntero compatible. Existen tres formas de declarar el parámetro como un array del mismo tipo y tamaño que el utilizado para llamar a la función:

Se puede declarar el parámetro como un array del mismo tipo y tamaño que el utilizado para llamar a la función.

Se puede especificar como un array, de tamaño indeterminado.

Por último, se puede especificar como un puntero al tipo base del array, que es la forma más común.

El siguiente programa muestra los tres métodos descritos:

```
#include <stdio.h>

void f1(int num[5]), f2(int num[]), f3(int *num);

int main(void){
    int count[5] = {1, 2, 3, 4, 5};
    f1(count);
    f2(count);
    f3(count);
}
```

```c
void f1(int num[5]){
    for(int i=0; i<5; i++){
        printf("%d", num[i]);     // Imprime: 12345.
    }
}
void f2(int num[]){
    for(int i=0; i<5; i++){
        printf("%d", num[i]);    // Imprime: 12345.
    }
}
void f3(int *num){
    for(int i=0; i<5; i++){
        printf("%d", num[i]);    // Imprime: 12345.
    }
}
```

IMPORTANTE: Aunque los tres métodos de declarar un parámetro de tipo array parezcan distintos, todos conducen al mismo resultado: un puntero.

Paso de argumentos a main().

Muchos programas permiten que se especifiquen argumentos de línea de comandos cuando se ejecutan. Los argumentos de línea de comandos se utilizan para pasar información al programa.

Los argumentos de línea de comandos se pasan a un programa en C a través de dos argumentos de la función main(). Los parámetros se llaman:

    argc
    argv

Estos parámetros, por supuesto, son opcionales y no se utilizan si no se trabaja con argumentos de línea de comandos.

El parámetro argc contiene el número de argumentos de línea de comandos y es un entero. Siempre será al menos 1, ya que el nombre del programa se cuenta como el primer argumento.

El parámetro argv es un array de punteros a cadena. Aquí se muestra el modo más común de declaración de argv:

```c
    char *argv[];
```

Los corchetes vacíos indican que es un array de longitud indeterminada. Todos los argumentos de línea de comandos se pasan a main() como cadenas. Para acceder a una cadena individual, se indexa argv. Por ejemplo, argv[0] apunta al nombre del programa y argv[1] apunta al primer argumento.

Ejemplo. Este programa muestra todos los argumentos de línea de comandos con los que se le llama:

```c
#include <stdio.h>

int main(int argc, char *argv[]){
    for(int i=1; i<argc; i++){      // i=1 (No mostrar el nombre del programa).
        printf("%s", argv[i]);
    }
}
```

Los nombres argc y argv son arbitrarios, pero es aconsejable usarlos para la identificación de los parámetros de la línea de comandos.

Ejemplos de uso de los argumentos de main().

Cuando se tiene que pasar datos numéricos a un programa, estos se recibirán en forma de cadena. El programa tiene que convertirlos al formato interno apropiado. Las funciones de la biblioteca estándar más comunes son las siguientes (usamos sus prototipos de función para presentarlas):

```
    int atoi (char *str);       // Cadena a entero.
    double atof (char *str);    // Cadena a double.
    long atol(char *str);       // Cadena a long.
```

Estas funciones forman parte del archivo de cabecera <stdlib.h>. Si se llama a una de estas funciones con una cadena que no sea un número válido, devolverá 0.

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

int main(int argc, char *argv[]){
    int i;
    double d;
    long l;

    i = atoi(argv[1]);
    d = atof(argv[2]);
    l = atol(argv[3]);

    printf("%d %lf %d", i, d, l);
}
```

ENTRADA Y SALIDA POR CONSOLA

La directiva del preprocesador #define.

El preprocesador de C realiza varias gestiones en el código de un programa antes de que se compile realmente. Una directiva al preprocesador es simplemente una instrucción al mismo.

La directiva #define se utiliza para decirle a preprocesador que sustituya una cadena por otra a lo largo de todo el programa:

```
#define nombre-de-la-cadena
```

Las directivas #define pueden ir en cualquier lugar del programa, una vez definidas, cualquier código a partir de ese punto puede tener acceso a ella. Reciben el nombre de macro. Por ejemplo:

```c
#include <stdio.h>

void f(void)

int main(){
    #define PROCESADOR 8086

    f();        // Imprime: 8086.
    return 0;
}

void f(void){
    printf("%ld", PROCESADOR);
}
```

Una vez definido un nombre de macro, se puede utilizar para definir otro nombre de macro, por ejemplo:

```c
#include <stdio.h>

#define SMALL 1
#define MEDIUM SMALL+1  // Usamos la macro SMALL para definir la macro MEDIUM.
#define LARGE MEDIUM+1

int main(void){
    printf("%d %d %d", SMALL MEDIUM, LARGE);    // Imprime: 1 2 3.
}
```

Examen de la entrada y salida de caracteres y de cadenas.

La función getchar() devuelve el siguiente carácter que se introduce por teclado (devuelve el siguiente carácter del flujo de entrada al que apunta stdin). Se lee como unsigned char convertido en un int. La razón por la que getchar() devuelve un entero es porque si se produce un error al leer la entrada, getchar() devuelve la macro EOF, que es el entero negativo -1. La macro EOF, definida en <stdio.h>, representa fin de archivo. Para permitir que se devuelva el valor EOF, getchar() debe devolver un entero. En la gran mayoría de las circunstancias, si se produce un error al leer del teclado, normalmente significa que la computadora se ha bloqueado. Por eso la mayoría de los programadores no se preocupan de comprobar la condición EOF cuando utilizan

getchar(), simplemente asignan el valor a un valor de carácter en lugar de a un entero. Es decir: getchar() devuelve un int, no un char. Esto es para que pueda devolver cualquier carácter válido (como un valor 0..255 para sistemas donde CHAR_BIT es 8) y un valor separado (generalmente -1) como EOF.

La función putchar() muestra un solo carácter en pantalla. Aunque su parámetro se declara de tipo int, la función lo convierte en un unsigned char. Si la operación de salida se ejecuta correctamente, putchar() devuelve el carácter escrito. Si se produce un error a la salida, devuelve EOF. Por el mismo motivo, la mayoría de programadores tampoco se ocupan de comprobar el valor de retorno de putchar() para los errores.

La razón por la que puedan querer utilizar putchar() en vez de printf() con el especificador %c para mostrar un carácter es que putchar() es más rápido y eficiente.

getchar y getch.

getchar() se implementa generalmente utilizando búfer de línea. Cuando la entrada es con búfer de línea, no se devuelven realmente los caracteres al programa que los llama hasta que el usuario pulsa 'intro'. Es decir: no se entrega ninguna entrada hasta que esté disponible una línea completa.

```c
#include <stdio.h>

int main(void){
    char ch;
    do{
        ch = getchar();
        putchar(',');       // Devuelve una coma por cada carácter introducido.
    }while(ch !='\n');
}
```

Cuando se introducen caracteres utilizando getchar(), pulsar 'intro' hará que se devuelva el carácter de salto de línea. Sin embargo, cuando se utilice una de las funciones no estándar alternativas, pulsar 'intro' hará que se devuelva el carácter '\r' de retorno de carro.

IMPORTANTE: Si desea leer líneas y procesar cada una de ellas, use fgets() para leer la línea y scanf() para analizar el resultado.

Examen de la función de consola no estándar getch().

Bien es sabido por los programadores, que la librería conio.h es exclusiva de Borland, y contiene las famosas funciones gotoxy(x,y), clrscr(), getch()… que son tan usadas en programas en modo texto. La librería conio.h no está disponible para GNU/Linux, pero la alternativa es la librería <curses.h> (ncurses). Esta librería no viene por defecto instalada, pero su instalación en distribuciones basadas en debian es bien sencilla, 'apt install ncurses-dev'.

Ejemplo de uso de getch() y ncurses. Lee los caracteres introducidos hasta que se pulsa 'q'.

```c
#include <ctype.h>
#include <curses.h> // Uso del compilador: gcc -Wall finlename.c -lcurses.
```

```
int main(void){
    char ch;

    do{
        // iniciar curses:
        initscr();

        ch = getch();    // getch() es parte de curses.h

        // finalizar curses:
        endwin();

        putchar(ch);
    }while(ch != 'q');
}
```

printf().

Especificadores de formato de printf():

%c	Carácter
%d	Enteros decimales con signo
%i	Enteros decimales con signo
%e	Notación científica ('e' minúscula)
%E	Notación científica ('E' mayúscula)
%f	Punto flotante decimal
%g	Utiliza %e o %f, el más corto
%G	Utiliza %E o %F, el más corto
%o	Octal sin signo
%s	Cadena de caracteres
%u	Enteros decimales sin signo
%x	Hexadecimales sin signo (letras minúsculas)
%X	Hexadecimales sin signo (letras mayúsculas)
%p	Muestra un puntero
%n	El argumento asociado es un puntero a entero, al que se le asigna el número de caracteres escritos hasta el momento.
%%	Imprime el signo %

Los especificadores pueden tener, así mismo, un especificador de longitud mínima de campo y/o especificador de precisión asociado a ellos. Ambos son cantidades int. El especificador de longitud mínima de campo se sitúa entre el signo % y el especificador de formato. El especificador de precisión sigue al especificador de longitud mínima de campo. Ambos están separados por un punto. Por ejemplo:

```
%15.2lf
```

Este especificador de formato le indica a printf() que muestre un valor double, utilizando una longitud de campo de 15, con dos dígitos decimales.

Scanf().

Especificadores de formato de scanf():

%c	Lee un único carácter
%d	Lee un entero decimal
%i	Lee un entero decimal
%e	Lee un número en punto flotante
%f	Lee un número en punto flotante
%g	Lee un número en punto flotante

%o	Lee un número octal sin signo
%s	Lee una cadena
%x	Lee un número hexadecimal sin signo
%p	Lee un puntero
%n	Recibe un valor entero igual al número de caracteres leídos hasta el momento
%u	Lee un entero sin signo
%[]	Inspecciona un juego de caracteres

La función scanf() devuelve el número de campos a los que se ha asignado un valor. Si se produce un error antes de que se haga cualquier asignación, devuelve EOF.

Los especificadores h (short) y l (long) su pueden añadir para modificar los anteriores cuando sena compatibles.

Se puede leer una cadena utilizando %s, pero probablemente no lo querrá hacer. Este es el motivo: cuando scanf() introduce una cadena, detiene la lectura de la cadena al encontrar el primer espacio en blanco. IMPORTANTE: Este es el motivo por el que generalmente se usa fgets() para introducir cadenas.

ENTRADA Y SALIDA POR ARCHIVOS

Aunque C no tiene ningún método incorporado para realizar E/S por archivo, la biblioteca estándar de C contiene un juego muy rico de funciones de E/S. El enfoque de E/S en C es eficiente potente y flexible.

Comprensión de los fundamentos de los flujos.

El concepto de flujo es un concepto muy importante en C. El flujo es una interfaz lógica común a los distintos dispositivos que comprende la computadora. En su forma más común es una interfaz lógica a un archivo. Aquí archivo se puede referir a un archivo de disco, a la pantalla, a un puerto, al teclado, etc. El flujo proporciona una interfaz consistente.

Un flujo se vincula a una archivo utilizando una operación de apertura y se desliga de un archivo utilizando una operación de cierre.

Existen dos tipos de flujo: de texto y binarios. Un flujo de texto se utiliza con caracteres ASCII. Cuando se utiliza un flujo de texto se pueden producir algunas conversiones de caracteres. Por ejemplo, el carácter de nueva línea se convierte en la secuencia de retorno de carro/salto de línea. Un flujo binario se puede usar con cualquier tipo de datos. No se produce conversión de caracteres, y hay una correspondencia directa entre lo que se envía al flujo y lo que el archivo contiene realmente.

Concepto de 'posición actual': es la posición del archivo en la que se producirá el siguiente acceso al archivo. Por ejemplo, si archivo tiene 100 bytes y se ha leído la mitad del archivo, la siguiente operación de lectura se producirá en el byte 50, que es la posición actual.

Fundamentos del sistema de archivos.

Aprenderemos a abrir y cerrar un archivo, así como a leer y escribir desde y hacia un archivo. Todas las funciones del sistema de archivos utilizan el archivo de cabecera <stdio.h>

Para abrir un archivo y asociarlo con un flujo, se utiliza fopen(). Aquí se muestra su prototipo:

```
FILE *fopen(char *fnombre, char *modo);
```

fnombre apunta al nombre del archivo a abrir. modo puede ser uno de los siguientes:

```
r       abre para lectura
w       abre para escritura
a       abre para añadir
rb      abre para binario lectura
wb      abre para binario escritura
ab      abre para binario añadir
r+      abre para lectura/escritura
w+      crea para lectura/escritura
a+      añade o crea para lectura/escritura
r+b     abre binario para lectura/escritura
w+b     crea binario para lectura/escritura
a+b     añadir en binario en modo lectura/escritura
```

Si la operación de apertura se ejecuta correctamente, fopen() devuelve un puntero a archivo válido. El tipo FILE está definido en <stdio.h>; es una

estructura que contiene distintas clases de información sobre el archivo, como su tamaño, posición actual y sus modos de acceso (una estructura es un grupo de variables a las que se accede bajo un nombre. Veremos esto con posterioridad).

La función fopen() devuelve un puntero a la estructura asociada con el archivo por el proceso de apertura. Se utiliza este puntero con todas las demás funciones que operan sobre el archivo. Sin embargo, nunca se deben alterar éste o el objeto al que apunta. Si la función fopen() falla, devuelve un puntero nulo (<stdio.h> define la macro NULL, que se define como un puntero nulo). En el siguiente ejemplo se muestra el modo apropiado para abrir un archivo:

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

FILE *fp;

int main(void){

    if((fp = fopen("my_file", "r")) == NULL) {
            printf("Error en la apertura del archivo\n");
        exit(1); // o sustituya su propio manejador de errores.
    }
}
```

Hay que tener en consideración que cuando se abre un archivo en modo escritura:

```
    fopen("my_file", "r");
```

si el archivo no existe, se creará. Si existe, los contenidos del archivo original se destruirán y se creará un nuevo archivo.

Para cerrar un archivo se usa fclose(), cuyo prototipo es:

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

La función fclose() cierra el archivo asociado a fp, que debe ser un puntero a archivo válido obtenido previamente utilizando fopen(), y desliga el flujo del archivo. Nunca se debe llamar a fclose() con un argumento no válido. Al hacerlo se dañará el sistema de archivos y probablemente se produzca una pérdida de datos.

La función fclose() devuelve 0 si se ejecuta correctamente. Si se produce algún error, devuelve EOF.

fgetc() y fputc().

Una vez que se ha abierto una archivo, dependiendo de su modo de apertura, se puede leer y/o escribir hacia o desde él utilizando estas dos funciones:

```
    int fgetc(FILE *fp);
    int fputc(int ch, FILE *fp);
```

La función fgetc() lee el siguiente byte del archivo descrito por fp como unsigned char y lo devuelve como un entero. La razón por la que devuelve un entero es porque si se produce algún error, devuelve EOF, que es un valor entero. También devuelve EOF cuando se alcanza el final del archivo.

La función fputc() escribe el byte contenido en ch en el archivo asociado con fp como unsigned char. Aunque ch está definido como un int, se le puede llamar usando un char, que es el procedimiento común. Devuelve el carácter escrito si se ejecuta correctamente o EOF si se produce algún error.

Nota: getc() y putc() son nombres antiguos para fgetc() y fputc().

Ejemplo. El siguiente ejemplo abre un archivo, escribe en el archivo una cadena; después cierra el archivo y lo vuelve a abrir para operaciones de lectura. Por último, muestra el contenido del archivo en pantalla y lo cierra nuevamente.

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

char str[80] = "Esto es una prueba del sistema de archivos.";

int main(void){
    FILE *fp;
    char *p;
    int i;

    // Abre my_file para la salida.
    if((fp = fopen("my_file", "w")) == NULL){
        printf("No se puede abrir el archivo.\n");
        exit(1);
    }
    // Escribe str en el disco.
    p = str;
    while(*p){
        if(fputc(*p, fp)==EOF){
            printf("Error en la escritura del archivo.\n");
            exit(1);
        }
        p++;
    }
    fclose(fp);
    // Abre my_file para la entrada.
    if((fp = fopen("my_file", "r")) == NULL){
        printf("No se puede abrir el archivo.\n");
        exit(1);
    }
    // Leer el archivo.
    for( ; ; ){
        i = fgetc(fp);
        if(i==EOF){
            break;
        }
        putchar(i);
    }
    fclose(fp);
}
```

Un enfoque más profesional del mismo programa se vería así:

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

char str[80] = "Esto es una prueba del sistema de archivos.";
```

```c
int main(void){
    FILE *fp;
    char ch, *p;

    // Abre my_file para la salida.
    if((fp = fopen("my_file", "w")) == NULL){
        printf("No se puede abrir el archivo.\n");
        exit(1);
    }
    // Escribe str en el disco.
    p = str;
    while(*p){
        if(fputc(*p++, fp)==EOF){
            printf("Error en la escritura del archivo.\n");
            exit(1);
        }
    }
    fclose(fp);
    // Abre my_file para la entrada.
    if((fp = fopen("my_file", "r")) == NULL){
        printf("No se puede abrir el archivo.\n");
        exit(1);
    }
    // Leer el archivo.
    while((ch = fgetc(fp)) != EOF){
        putchar(ch);
    }
    fclose(fp);
}
```

Ejemplo. El siguiente programa busca un carácter, especificado en la línea de comandos, en el archivo especificado, que también se pasa por comandos al ejecutar el programa. IMPORTANTE: es muy ilustrativo de varios conceptos.

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

int main(int argc, char *argv[]){
    FILE *fp;
    char ch;

// Comprueba el número de argumentos de la línea de comandos.
if(argc != 3){
    printf("Uso: search <nombre de archivo> <ch>.\n");
    exit(1);
}

// Abre el archivo para la entrada.
if((fp = fopen(argv[1], "r")) == NULL){
    printf("NO se puede abrir el archivo.\n");
    exit(1);
}

// Buscar el carácter.
while((ch = fgetc(fp)) != EOF){
    if(ch == *argv[2]){
        printf("%c encontrado", ch);
        break;
    }
```

```
}
fclose(fp);
}
```

feof() y ferror().

Cuando fgetc() devuelve EOF, bien se ha producido un error o bien se ha alcanzado el final del archivo, pero ¿cómo se sabe qué suceso ha ocurrido?. Además si está trabajando con un archivo binario, todos los valores son válidos. Esto significa que es posible que un byte tenga el mismo valor (cuando se eleva a un int) que EOF, por tanto, ¿cómo se sabe si se ha devuelto un dato válido o se ha alcanzado el final del archivo? La solución a estos problemas son las funciones feof() y ferror(). Estos son los prototipos de las funciones:

```
int feof(FILE *fp);
int ferror(FILE *fp);
```

La función feof() devuelve un valor distinto de 0 si el archivo asociado a fp ha llegado al final. En caso contrario devuelve 0. Funciona tanto para archivos de texto como para archivos binarios.

La función ferror() devuelve un valor distinto de cero si el archivo asociado a fp ha experimentado un error; en caso contrario devuelve 0.

Utilizando feof(), este fragmento de código muestra como leer hasta el final del archivo:

```
FILE *fp;
...
while(!feof(pf)){
      fgetc(fp);
}
```

Y aquí añadimos comprobación de errores:

```
FILE *fp;
...
while(!feof(pf)){
    fgetc(fp);
    if(ferror(fp)){
        printf("Error en el archivo.\n");
        break;
    }
}
```

ferror() solamente comunica el estado del sistema de archivos relativo al último acceso de archivo. Por eso, para proporcionar la comprobación de errores más completa, se le debe llamar después de cada operación de archivo.

Ejemplo. El siguiente programa copia cualquier tipo de archivo, binario o texto.

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

int main (int argc, char *argv[]){
    FILE *from, *to;
    char ch;
```

```
    //Ver que el número de argumentos de la línea de comandos es correcto.
    if(argc != 3){
        printf("Uso: copy <fuente> <destino>.\n");
        exit(1);
    }

    // Abrir archivo fuente.
    if((from = fopen(argv[1], "rb")) == NULL){
        printf("No se puede abrir el archivo fuente.\n");
        exit(1);
    }

    // Abrir el archivo destino.
    if((to = fopen(argv[2], "wb")) == NULL){
        printf("No se puede abrir el archivo destino.\n");
        exit(1);
    }
    // Copiar el archivo.
    while(!feof(from)){
        ch = fgetc(from);
        if(ferror(from)){
            printf("Error leyendo el archivo fuente.\n");
            exit(1);
        }
        if(!feof(from)){
            fputc(ch, to);
        }
        if(ferror(to)){
            printf("Error escribiendo el archivo destino.\n");
            exit(1);
        }
    }
    fclose(from);
    fclose(to);
}
```

fputs(), fgets(), fprintf() y fscanf().

C proporciona cuatro funciones que facilitan las operaciones con archivos, Los prototipos de las funciones son:

```
    int fputs(char *str, FILE *fp);
    char *fgets(char *str, int num, FILE *fp);

    int fprintf(FILE *fp, char *cadena de control, ...);
    int fscanf(FILE *fp, char *cadena de control, ...);
```

La función fputs() escribe la cadena a la que apunta str en el archivo asociado a fp. Devuelve EOF si se produce un error y un valor no negativo si se ejecuta correctamente. No se escribe el carácter nulo que termina str. Además, a diferencia de la función puts(), no añade automáticamente una secuencia de retorno de carro/salto de línea.

La función fgets() lee caracteres del archivo asociado a fp en la cadena a la que apunta str hasta que haya leído num-1 caracteres, se encuentre una nueva línea, o se alcance el final del archivo. A diferencia de la función gets(), se mantiene el carácter de nueva línea. La función devuelve str si se ha ejecutado correctamente y un puntero nulo si se produce un error.

fprintf() y fscanf() funcionan exactamente igual que printf() y scanf(), excepto que trabajan con archivos. En lugar de dirigir sus operaciones de E/S hacia la consola, estas funciones operan sobre el archivo especificado fp. Su ventaja es que facilitan la escritura de una gran variedad de datos en un archivo utilizando el formato de texto.

fread() y fwrite().

fread() y fwrite() pueden leer y escribir cualquier tipo de dato. Están a nuestra disposición dado que fscanf() fprintf(), no son necesariamente el modo más eficiente de leer y escribir datos en un archivo. La razón es porque ambas funciones realizan conversión de datos. Sus funciones prototipo son las siguientes:

```
size_t fread(void *buffer, size_t length, size_t count, FILE *filename);
size_t fwrite(void *buffer, size_t length, size_t count, FILE *filename);
```

fread() lee del archivo filename el número de elementos especificado en el argumento count en el búfer apuntado por buffer y considerando cada elemento con una longitud de length bytes. Devuelve el número de elementos leídos. Si el valor es 0, no se ha leído ningún elemento, y o bien se ha alcanzado el final del archivo o se ha producido un error. Se pueden utilizar feof() y ferror() para averiguarlo.

fwrite() es la opuesta de fread(). Escribe en el archivo asociado filename, el número de elementos especificados en count del búfer apuntado por buffer y considerando cada elemento con una longitud de length bytes. Devuelve el número de objetos escritos. Este valor será inferior al de count solamente si se ha producido un error de salida.

Veamos los conceptos implícitos en los prototipos de las funciones:

El primero es el puntero void. Un puntero void es un puntero que puede apuntar a cualquier tipo de dato sin utilizar un molde de tipo, se le conoce normalmente como puntero genérico. Como se dijo anteriormente, fread() y fwrite() se pueden utilizar para leer y escribir cualquier tipo de datos, por tanto, deben ser capaces de recibir cualquier tipo de datos a los que apunte buffer. Este es el motivo por el que se implementaron los punteros void. Un segundo propósito es para que una función devuelva un puntero de tipo genérico.

El segundo concepto nuevo es el tipo size_t. Este tipo está definido en el archivo de cabecera <stdio.h>. Una variable de este tipo se define como una variable capaz de contener un valor igual al tamaño del elemento de mayor longitud que soporte el compilador. Para nuestros propósitos se pueden considerar como unsigned o como unsigned long. El motivo por el que se utiliza size_t en lugar de un tipo incorporado es para permitir que los compiladores de C se ejecuten en distintos entornos.

Ejemplo. Se escribe el valor de un entero en un archivo llamado my_file, utilizando su representación a carácter interna.

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

int main(void){
    FILE *fp;
    int i = 100;
```

```c
    // Abrir archivo para salida
    if((fp = fopen("my_file", "w")) == NULL){
        printf("No se puede abrir el archivo.\n");
        exit(1);
    }
    // sizeof (int): devuelve el tamaño en bytes del tipo que le sigue.
    if(fwrite(&i, sizeof (int), 1, fp) != 1){  // 1: número de elementos leídos.
        printf("Error al escribir.\n");
        exit(1);
    }
    fclose(fp);

    // Abrir archivo para entrada.
     if((fp = fopen("my_file", "r")) == NULL){
        printf("No se puede abrir el archivo.\n");
        exit(1);
    }

    if(fread(&i, sizeof (int), 1, fp) != 1){
        printf("Error al leer.\n");
        exit(1);
    }

    fclose(fp);
    printf("i es %d", i);

    return 0;
}
```

remove() y rewind().

Se puede borrar un archivo utilizando remove(). remove() devuelve 0 si se ejecuta correctamente y un valor distinto de cero si se produce un error.

```c
    int remove(char *filename);
```

Se puede colocar el indicador de posición de un archivo al principio del mismo utilizando rewind(). La función no tiene valor de retorno.

```c
    void rewind(FILE *fp);
```

```c
#include <stdio.h>
#include <ctype.h>      // toupper
#include <string.h>     // strcspn

int main(void){
    char filename[80];

    printf("Introduzca el nombre del archivo a borrar: \n");
    fgets(filename, 80, stdin);
    printf("¿Está seguro? (S/N) ");

    if (toupper (getchar()) == 'S'){
        printf("Eliminando %s", filename);
        filename[strcspn(filename, "\n")] = 0; // [*] Eliminamos '\n'.
        remove(filename);
    }
    return 0;
}
```

[*] El prototipo de la función strcspn es:

```
size_t strcspn(const char *s, const char *rechaza);
```

La función strcspn() calcula la longitud del segmento inicial de 's' que consta únicamente de caracteres que no aparecen en 'rechaza'. devuelve el número de caracteres del segmento inicial de s que consta únicamente de caracteres que no están en 'rechaza'.

Flujos estándar.

Cuando comienza un programa en C, se abren automáticamente tres flujos ya disponibles para su uso: stdin, stdout y stderr. Por omisión están asociados a la consola, pero el sistema operativo los puede redireccionar a cualquier otro dispositivo.

Estos flujos estándar son a punteros FILE, y se pueden utilizar con cualquier función que utilice una variable tipo FILE *. Por ejemplo, se puede utilizar fprintf() para imprimir una salida formateada en la pantalla. Las dos sentencias siguientes hacen lo mismo:

```
fprintf("stdout, "%s", "esto es una cadena");
printf("%s", "esto es una cadena");
```

Sin embargo, stdin, stdout y stderr no son variables, no se les puede asignar un valor usando fopen() y no se deben intentar cerrar usando fclose(). A estas cadenas las mantiene internamente el compilador. Se pueden utilizar libremente, pero no se pueden cambiar.

ESTRUCTURAS Y UNIONES

Fundamentos de las estructuras.

Una estructura es un tipo de dato conglomerado que está compuesto por dos o más elementos relacionados. A diferencia de los arrays, cada elemento de la estructura puede tener su propio tipo, que puede diferir de los tipos de otros elementos. Esta es su forma general:

```
struct nombre{
    tipo elemento_1;
    tipo elemento_2;
    ...
    tipo elemento_n;
} lista_de_variables;
```

La lista de variables es donde se definen las variables utilizas por la estructura. Tanto el nombre como la lista_de_variables son opcionales, pero una de las dos tiene que estar presente. A los elementos de una estructura se les conoce como campos o miembros.

Por ejemplo, podríamos usar una estructura para guardar entradas de un catálogo de tarjetas de biblioteca, como en el ejemplo:

```
#include <stdio.h>

int main(void){

    struct catalog{
        char name[40];      // nombre del autor
        char title[40];     // título del libro
        char publisher[40]; // editorial
        unsigned date;      // fecha de publicación
        unsigned char ed;   // número de la edición
    } card;

    card.date = 2023;

    printf("Fecha de publicación: %u", card.date);
    return 0;
}
```

La única variable definida en la estructura es card; para acceder a un campo de una variable estructura hay que especificar tanto el nombre de la variable como el del campo, separados por un punto. Los programadores de C, frecuentemente, llaman al punto como el operador punto. Por ejemplo la sentencia anterior asigna al campo 'date' el valor 2023.

Una vez que tenga definido un tipo de estructura, puede crear más variables de ese tipo utilizando el formato general:

```
struct nombre lista_de_variables;
```

```
#include <stdio.h>

int main(void){

    struct catalog{
```

```
        char name[40];      // nombre del autor
        char title[40];     // título del libro
        char publisher[40]; // editorial
        unsigned date;      // fecha de publicación
        unsigned char ed;   // número de la edición
    } card;

    struct catalog var1, var2, var3;

    return 0;
}
```

Este es el motivo por el que no es necesario definir todas las variables cuando se declara la estructura.

Un ejemplo de estructura sin nombre es el siguiente:

```
    struct {
        int a;
        char ch;
    } var1, var2;
```

Las estructuras se pueden poner en forma de array, del mismo modo que otros tipos de datos. Por ejemplo, la siguiente definición de estructura crea un array de estructuras de 100 elementos de tipo catalog:

```
    struct catalog cat[100];
```

Para acceder a una estructura individual del array, tiene que acceder por índice al nombre del array. Por ejemplo, la siguiente sentencia accede a la primera estructura:

```
    cat[0];
```

Para acceder a un elemento de una estructura específica, ponga detrás del índice un punto y el nombre del elemento que desee. Por ejemplo:

```
    cat[33].ed = 2;
```

Las estructuras se pueden pasar como parámetros a funciones, como cualquier otro tipo de datos. Una función también puede devolver una estructura.

Ejemplo: Catálogo de tarjetas.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define MAX 100

int menu(void);
void display(int i);
void author_search(void), title_search(void);
void intro(void), save(void), load(void);

struct catalog{
        char name[80];      // nombre del autor
        char title[80];     // título del libro
        char publisher[80]; // editorial
```

```
        unsigned date;      // fecha de publicación
        unsigned char ed;   // número de la edición
    } cat[MAX];

int top = 0;    // Última posición usada.

int main(void){
    int choice;

    load();     // Leer el catálogo.

    do{
        choice = menu();
        switch (choice){
            case 1: intro();            // Introducción de libros.
                break;
            case 2: author_search();    // Búsqueda por autor.
                break;
            case 3: title_search();     // Búsqueda por título.
                break;
            case 4: save();             // Guardar catálogo.
        }
    }while(choice!=5);
}

// Devolver una selección de menú.
int menu(void){
    int i;
    char str[80];

    printf("Tarjeta:\n");
    printf(" 1. Entrada de registro\n");
    printf(" 2. Búsqueda por autor\n");
    printf(" 3. Búsqueda por título\n");
    printf(" 4. Guardar catálogo\n");
    printf(" 5. Salir\n");

    do{
        printf("Elija una opción: ");
        fgets(str, 80, stdin);
        str[strcspn(str, "\n")] = 0;
        i = atoi(str);
    }while(i<1 || i>5);

    return i;
}

// Introducir libros en la base de datos.
void intro(void){
    int i;
    char temp[80];

    for(i=top; i<MAX; i++){
        printf("Introduzca en nombre del autor ('intro' para salir): ");
        fgets(cat[i].name, 80, stdin);
        if(*cat[i].name == '\n'){
            break;
        }
        printf("Introduzca el título: ");
        fgets(cat[i].title, 80, stdin);
        printf("Introduzca la editorial: ");
```

```
        fgets(cat[i].publisher, 80, stdin);
        printf("Introduzca la fecha de publicación: ");
        fgets(temp, 80, stdin);
        cat[i].date = (unsigned) atoi(temp);
        printf("Introduzca la edición: ");
        fgets(temp, 80, stdin);
        cat[i].ed = (unsigned char) atoi(temp);
    }
    top = i;
}

//Búsqueda por autor.
void author_search(void){
    char name[80];
    int i, found;

    printf("Nombre:");
    fgets(name, 80, stdin);

    found = 0;
    for(i=0; i<top; i++){
        if(!strcmp(name, cat[i].name)){
            display(i);
            found = 1;
            printf("\n");
        }
    }
    if(!found){
        printf("Autor no encontrado");
    }
}

//Búsqueda por título.
void title_search(void){
    char title[80];
    int i, found;

    printf("Título:");
    fgets(title, 80, stdin);

    found = 0;
    for(i=0; i<top; i++){
        if(!strcmp(title, cat[i].title)){
            display(i);
            found = 1;
            printf("\n");
        }
    }
    if(!found){
        printf("Título no encontrado");
    }
}

// Visualización de entradas de catálogo.
void display(int i){
    printf("%s\n", cat[i].title);
    printf("por %s\n", cat[i].name);
    printf("Editorial %s\n", cat[i].publisher);
    printf("Editado en %u, %u edición\n", cat[i].date, cat[i].ed);
}
```

```c
// Recuperar el archivo de catálogo.
void load(void){
    FILE *fp;

    if((fp = fopen("catalog", "r")) == NULL){
        printf("Archivo de catálogo no disponible\n");
        return;
    }
    fread(&top, sizeof(top), 1, fp);    // Contador de lectura.
    fread(cat, sizeof(cat), 1, fp);

    fclose(fp);
}

// Guardar el archivo de catálogo.
void save(void){
    FILE *fp;

    if((fp = fopen("catalog", "w")) == NULL){
        printf("Archivo de catálogo no disponible\n");
        exit(1);
    }
    fwrite(&top, sizeof(top), 1, fp);
    fwrite(cat, sizeof(cat), 1, fp);

    fclose(fp);
}
```

Los nombres de los elementos de las estructuras no entran en conflicto con otras variables que utilicen los mismos nombres. Por ejemplo:

```c
#include <stdio.h>

int main (void){
    struct tipo_s{
        int i;
        int j;
    } s;

    int i;

    i = 10;
    s.i = 100;
    s.j = 101;

    printf("%d - %d - %d", i, s.i, s.j);    // Imprime: 10 - 100 - 101.

    return 0;
}
```

Declaración de punteros a estructuras.

Es muy común acceder a una estructura mediante un puntero. Un puntero a estructura se declara del mismo modo que se declara un puntero a cualquier otro tipo de variable. Por ejemplo, en la siguiente estructura 'tipo_s', se declaran dos variables. La primera, s, es una variable de estructura real. La segunda, p, es un puntero a la estructura 'tipo_s'.

```
    struct tipo_s{
        int i;
        char str[80];
    } s, *p;
```

Dada esta declaración, la siguiente sentencia asigna a p la dirección de s:

```
    p = &s;
```

ahora que p apunta a s se puede acceder a s a través de p. Sin embargo, para acceder a un elemento individual de s usando p no se puede usar el operador punto. En este caso, se debe usar el operador flecha, como se muestra aquí:

```
    p->i = 1;
```

Esta sentencia asigna el valor 1 al elemento i de s a través de p.

IMPORTANTE: Cuando se accede a un elemento de estructura usando una variable de estructura, se utiliza el operador '.'. Sin embargo, cuando se accede a un elemento de estructura usando un puntero, se debe utilizar el operador '->'.

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

struct tipo_s{
    int i;
    char str[80];
} s, *p;

int main(void){
    p = &s;

    s.i = 10;
    p->i = 10;
    strcpy(p->str, "Esto es una copia de cadena a str.");

    printf("%d %d %s", s.i, p->i, p->str);// 10 10 es una copia de cadena a str.

    return 0;
}
```

Ejemplo de uso de estructuras con funciones.

```
#include <stdio.h>

void operaciones (float x, float y,
              float *s, float *p, float *d);

int main(){
  float a = 1.0, b = 2.0; // Datos
  float suma, producto, division; // Resultados
  operaciones(a, b, &suma, &producto, &division);
  printf("S: %f \t P: %f \t D: %f \t",
       suma, producto, division);
}
//Función con varios resultados
void operaciones (float x, float y,
              float *s, float *p, float *d)
{
```

```
// Cada puntero sirve para un resultado
  *s = x + y;
  *p = x * y;
  *d = x / y;
}
```

Estructuras anidadas.

Los elementos de una estructura también pueden ser otras estructuras. A esto se le llama estructuras anidadas. Por ejemplo, el siguiente programa usa estructuras anidadas para guardar información del rendimiento de dos cadenas de ensamblaje, cada una con diez trabajadores. Para asignar el valor 12 a 'unidades_por_hora' de la segunda estructura 'trab' de 'linea1' se usa la sentencia dada en el ejemplo:

```
#include <stdio.h>

#define NUN_ON_LINE 10

struct trabajador{
    char nombre[80];
    int unidades_por_hora;
    int defectuosas_por_hora;
};

struct linea_ensamblaje{
    int codigo_producto;
    double coste_material;
    struct trabajador trab[NUN_ON_LINE];
} linea1, linea2;

// IMPORTANTE: Esto estaría mal.
/*
struct linea_ensamblaje{
    int codigo_producto;
    double coste_material;
    struct trabajador{
        trab[NUN_ON_LINE];
    }
} linea1, linea2;
*/

int main(void){

    linea1.trab[1].unidades_por_hora = 12;
    printf("%d", linea1.trab[1].unidades_por_hora); // Imprime: 12.

    return 0;
}
```

Uniones.

En C, una unión es una posición individual de memoria compartida por dos o más variables. Las variables que comparten la memoria pueden ser de diferentes tipos. Sin embargo, solo una variable puede estar en uso en un momento dado. Una unión se define de un modo muy similar a una estructura:

```
union nombre{
    tipo elemento_1;
    tipo elemento_2;
    ...
    tipo elemento_n;
} lista_de_variables;
```

Como en las estructuras, tanto el nombre (etiqueta) como 'lista_de_variables' se pueden suprimir. Los elementos pueden ser cualquier tipo de dato válido en C. Veamos un ejemplo:

```c
#include <stdio.h>

int main(void){

    union tipo_u{
        int i;
        char c[2];
        double d;
    } ejemplo, *p;

    p = &ejemplo;       // IMPORTANTE: Esta declaración es imprescindible.
    ejemplo.d = 123.098;
    p->i = 101;

    printf("%6.2f %i\n", ejemplo.i, p->d); // Imprime: 123.10 101
    printf("%6.2f %i\n", p->i, ejemplo.d); // Imprime: 123.10 101

    return 0;
}
```

TIPOS DE DATOS Y OPERADORES AVANZADOS

Especificadores de tipo.

C define cuatro especificadores de tipo que afectan al almacenamiento de una variable:

- auto (en desuso)
- extern
- register
- static

Cuando un programa crece se puede dividir en dos o más archivos. Se pueden compilar separadamente y después enlazarlos. Los datos globales solo se declaran una vez, debido a que dos o más archivos pueden necesitar acceder a los datos globales, se tiene que disponer de un método para informar al compilador sobre los datos globales usados por el programa.

Consideremos estos dos archivos pertenecientes a un mismo programa:

```c
/* FILE #1 */
#include <stdio.h>

int count       // count declarada en FILE #1

void f1(void);

int main(void){

    f1();
    for (int i=0; i<count; i++){
        printf("%d", i);
    }
}

/* FILE #2 */
#include <stdlib.h>

//int count     // Esto sería incorrecto. Error de duplicación en el enlazador.
extern int count;       // Uso del especificador extern.

void f1(void){
    count = rand();
}
```

En FILE #2 estamos indicando al compilador que count es un entero declarado en otra parte.

El especificador 'register' indica al compilador que se quiere acceder a la variable lo más rápido posible. Las variables se guardan de modo que se minimiza el tiempo de acceso. Estas variables se guardan en los registros de la CPU, de ahí su nombre, pero estos son limitados. Una buena opción es transformar una variable muy utilizada, como la variable que controla un bucle, en una variable register. Debido a que una variable register puede estar almacenada en un registro de la CPU, puede no tener una dirección de memoria. Esto implica que no pueda usar & para encontrar la dirección de una variable register.

El especificador static puede proteger el contenido de una variable local entre llamadas. Además, a diferencia de las variables locales normales, que se inicializan cada vez que se introduce una función, una variable local static solo se inicializa una vez. Por ejemplo, en el siguiente programa, count mantiene su valor entre las sucesivas llamadas a f():

```c
#include <stdio.h>

void f(void);

int main(void){

    for (int i=0; i<10; i++){
        f();
    }

    return 0;
}

void f(void){
    static int count = 0;
    count++;
    printf("count es %d\n", count);
}
```

La ventaja de usar una variable local static frente a usar una variable global es que la variable local static es conocida y accesible solo por la función en la que está declarada.

El modificador static se puede usar también para variables globales, cuando se utiliza con estas, provoca que la variable global solo se conozca y esté accesible por las funciones del mismo archivo en que está declarada. Es decir, una función no declarada en el mismo archivo que una variable global static no puede acceder a esa variable global, además ni siquiera conoce su nombre. Esto quiere decir que no se producen conflictos con los nombres cuando una variable global static de un archivo tiene el mismo nombre que otra variable global de otro archivo del mismo programa.

Modificadores de acceso.

C incluye dos modificadores de tipo que afectan al modo de acceso de las variables, tanto por el programa como por el compilador. Estos modificadores son:

    const
    volatile

Si se precede un tipo de una variable con const, se protege a esa variable frente a una modificación por parte del programa. Sin embargo, a la variable se le puede dar un valor inicial cuando se declara, mediante una inicialización. Una variable const puede cambiar su valor por medios dependientes del hardware.

El modificador const tiene un segundo uso. Puede evitar que una función modifique el objeto al que apunta un parámetro. Es decir, cuando un parámetro puntero está precedido por const, ninguna sentencia de la función puede modificar la variable apuntada por ese parámetro.

Cuando se precede un tipo de variable con volatile, se le está indicando al compilador que el valor de la variable se puede cambiar de modo no definido explícitamente en el programa. Por ejemplo, se podría dar una dirección de variable a una rutina de servicio de interrupción y su valor cambiaría cada vez que se produjese una interrupción.

Definición de enumeraciones.

En C se puede definir una lista de constantes enteras con nombre llamada enumeración. Estas constantes se podrían usar en cualquier sitio en que se pueda usar un entero. Para definir una enumeración se usa la forma general:

```
enum etiqueta {lista de enumeración} lista_de_variables;
```

Por ejemplo:

```
enum tipo_color {rojo, verde, amarillo} color;
```

Tanto la etiqueta como la 'lista_de_variables' son opcionales. La etiqueta es el nombre de la enumeración. Las variables de la enumeración solo pueden contener los valores que están definidos en la enumeración.

Por omisión, el compilador asigna valores enteros a las constantes, empezando por 0 en la parte más a la izquierda de la lista. Cada constante a la derecha es mayor en una unidad que la constante que la precede. En el ejemplo anterior rojo es 0, verde es 1 y amarillo es 2. Sin embargo se pueden redefinir los valores de todas o algunas de ellas, por ejemplo:

```
enum tipo_color {rojo, verde=9, amarillo} color;
```

Ahora rojo es 0, verde es 9 y amarillo 10 (también cambian los siguientes elementos).

Una vez que se ha declarado/definido una enumeración, se puede usar su etiqueta para declarar variables de enumeración en otros puntos del programa: Por ejemplo, suponiendo la enumeración 'tipo_color', esta sentencia es perfectamente válida y declara 'mi_color' como una variable de 'tipo_color'.

```
enum tipo_color mi_color;
```

Los principales propósitos de una enumeración son ayudar a proporcionar código autodocumentado y clarificar la estructura del programa.

```c
#include <stdio.h>

enum computadora {teclado, CPU, pantalla, impresora};

int main(void){
    enum computadora comp;
    comp = CPU;
    printf("%d", comp); // Imprime: 1. Segundo elemento de la enumeración.
}
```

Observe, en el ejemplo anterior, cómo se usa la etiqueta de la enumeración para declarar comp como una variable de enumeración, separadamente de la declaración inicial de computadora.

IMPORTANTE: Las constantes enumeradas no son cadenas, son constantes enteras con nombre. De ahí que en el ejemplo anterior se imprimiera '1'.

typedef.

En C se puede crear un nuevo nombre para un tipo existente usando typedef. La forma general de typedef es:

```
typedef nombre_viejo nombre_nuevo;
```

El nuevo nombre se puede usar para declarar variables. Por ejemplo, en el siguiente programa 'smallint' es el nuevo nombre de 'signed char' y se usa para declarar i:

```
#include <stdio.h>

typedef signed char smallint;

int main(void){

    smallint i;

    for (i=0; i<10; i++){
        printf("%d", i);
    }
    return 0;
}
```

typedef no provoca la desactivación del nombre original, es decir, en el ejemplo anterior, el uso de signed char, seguiría siendo válido. Además, se pueden usar varias sentencias typedef para crear varios nombres nuevos diferentes para el mismo tipo.

Operador '?'.

C contiene un operador ternario: '?'. Requiere tres operandos. Este operador se utiliza para reemplazar sentencias como esta:

```
if(condición){
    var = exp1;
}else {
    var = exp2;
}
```

o lo que es lo mismo:

```
if(condición) var = exp1;
else var = exp2;
```

La forma general del operador '?' es:

```
var = condición ? exp1: exp2;
```

Aquí la condición es una expresión que se evalúa a verdadero o falso. Si es verdadero, se le asigna a var el valor de exp1. Si es falso se le asigna el valor de exp2.