

## **Automatizar**

La automatización consiste en lograr que una tarea que anteriormente se realizaba a mano por los seres humanos, la realice en lo adelante un sistema artificial, con ninguna o muy escasa intervención humana.

Cuando decimos “a mano” no se trata de que sea literalmente con las manos. De hecho, la mente siempre interviene, y puede que lo hagan otras partes del cuerpo.

Lo básico es que es que la tarea es repetitiva y suficientemente bien estructurada que puede ser emulada por un sistema artificial.

La automatización es parcial cuando la tarea es llevada a cabo por una combinación hombre-máquina, como cuando realizamos operaciones numéricas con ayuda de una calculadora.

En este caso, se puede decir, que algunas subtareas son realizadas de forma automática por la máquina, pero la tarea global es guiada, paso a paso, por el humano.

La automatización existe en los propios cuerpo y mente humanos, de manera que nuestros órganos realizan de manera automática tareas como la respiración y la circulación de la sangre, y nuestro cerebro reconoce patrones y toma decisiones.

En un sentido, pues, la automatización es algo que está tanto en la naturaleza como en el hombre, y éste ha empezado emular por medio de sistemas artificiales.

## **Transferir**

En un sentido, automatizar es transferir a un sistema artificial la capacidad de emular una tarea natural, como realizar un cálculo aritmético, ordenar nombres, leer en voz alta texto escrito.

Por siglos el hombre había utilizado sistemas mecánicos para lograr la automatización, pero en las últimas décadas ha desarrollado matemática, electrónica, electromecánica, informática, para producir automatización más compleja.

Una clave de la automatización moderna es la programación de computadoras, pues éstas son sistemas artificiales extremadamente flexibles, que admiten la realización de innumerables tareas sin cambiar su estructura interna, procesando datos mediante instrucciones codificadas.

Si bien existe cierta dependencia entre la estructura física de la computadora, llamada hardware, y las instrucciones lógicas, llamada software, lo cierto es que ambos aspectos de la computación han logrado una gran autonomía en su desarrollo, tanto que se considera que el software está retrasado con respecto al hardware.

Esta relativa autonomía del software ha permitido diseñar sofisticadísimos sistemas que automatizan tareas sumamente complejas asumiendo un hardware estándar.

Se puede afirmar, pues, que con el software la humanidad ha logrado transferir el gesto mental humano a la máquina de manera sin precedentes. Pero no se ha dado el salto fundamental, permanecemos en la etapa artesanal.

Mientras la mecánica, la electromecánica, y la electrónica parecen haber encontrado sus ladrillos fundamentales de construcción, como son los tornillos, transformadores y transistores, la computación no ha logrado tal cosa, si bien se han hecho avances importantes.

Contamos con estructuras de datos y de control fundamentales, pero distan mucho de ser las contrapartes de los átomos en la química.

Cuando se encuentren estos átomos del software, entonces el transferir el gesto a la máquina habrá llegado a un nivel de madurez aceptable.

## Básico

Hay un nivel básico en la redacción de instrucciones para computadoras al que podemos llamar nivel de entrada, pues los comandos tienen una estructura accesible a cualquier persona que sepa leer y escribir.

De hecho, es el conjunto de instrucciones que suele cubrir un curso de introducción a la programación de computadoras digitales, con independencia del lenguaje de programación que se elija.

Estas instrucciones básicas son, a saber: Declarar, asignar, aceptar, desplegar, bifurcar, repetir, invocar. Claro, estas siete instrucciones básicas asumen un contexto de conceptos, entre ellos: Algoritmo, subrutina, valor, tipo, variable, expresión, operador, proposición, ejecución, memoria, entrada, salida, inicio, secuencia, finalización, encapsulación, paso, parámetro, prueba, depuración.

Resulta asombrosa la cantidad de problemas cotidianos, de ciencia y tecnología que pueden ser abordados de manera suficiente con esta reducida caja de herramientas.

Por supuesto, cuando se trata de que la velocidad es un imperativo, hay cajas de herramientas que corresponden a lenguajes de bajo nivel, como el ensamblador, y cuando se trata de abordar problemas de complejidad notable con un esfuerzo manejable, se tienen lenguajes de más alto nivel, como los orientados a objetos.

Existen también paradigmas de programación alternos, como la programación lógica, funcional, declarativa, entre otros.

Para el caso de aquellos que se introducen a la programación de computadoras, o que necesitan tener una idea somera de qué es la programación de computadoras digitales, la caja de herramientas básica esbozada aquí, es adecuada.

## C

Varios de los lenguajes más utilizados hoy en día en la industria del software, entre ellos C++, C# y Java, fueron derivados de un lenguaje desarrollado en la década de 1970: El lenguaje C..

Al parecer, los creadores del lenguaje C, Dennis MacAlistair Ritchie (n. en 1941) y Brian Wilson Kernighan (n. en 1942) dieron en el clavo, pues en la actualidad es el segundo lenguaje de programación más utilizado en la industria.

Todavía hoy en día se usan herramientas didácticas, como los diagramas de flujo y el pseudocódigo para introducir a la programación a los novatos programadores.

Con todo, entiendo que es perfectamente posible introducir a la programación usando C directamente, pues ya quedó atrás, a mi entender, la época en que se discutía si era mejor usar Basic o Pascal, en lugar de C.

C sigue latiendo como el corazón de lenguajes como C++, C# y Java, que es mucho decir. En lo que sigue, pues, asumiremos C en nuestros apuntes sobre transferir gesto a la máquina.

## Función

Un concepto fundamental en la programación C es el de función, que es un fragmento de código debidamente envuelto y que encapsula la realización de una tarea elemental.

Un programa C consiste de una o más funciones, de manera que el problema computacional que resuelve el programa queda dividido en partes, cada una asociada a una función que la acomete. La forma general de una función es:

```

tipo nombre(parámetros) {
    cuerpo
}

```

El tipo se refiere al tipo de un valor que la función retorna de manera implícita, el nombre a la palabra única utilizada para nombrar a la función, los parámetros son los distintos valores que la función retorna de manera explícita. Por ejemplo, una función que encapsule el cálculo del área de un triángulo, dadas las longitudes de la base y de la altura, podría tener la siguiente redacción:

```

float areaTriangulo(float b, float h) {
    return (b * h) / 2;
}

```

Note que se ha indicado que la función retorna un número real (float) y recibe dos números reales (float a y float b). La función calcula el valor del área a partir de la base y la altura y retorna de manera implícita, usando un return, el valor que produce la fórmula  $(b \cdot a) / 2$  a partir de los valores recibidos en b y h.

Cualquier entorno de programación C viene con varios conjuntos de funciones predefinidas, que no están incrustadas en el lenguaje mismo, sino son parte de los recursos disponibles para reutilización del programador en forma de archivos externos al núcleo del lenguaje.

De hecho, cualquier programador puede seguir extendiendo al lenguaje C creando nuevas funciones, organizándolas por temas y anadiéndolas al entorno del lenguaje, de manera que queden disponibles para reutilización.

## Tipo

Las funciones manipulan datos, por lo general, es típico que una función reciba uno o más valores y a partir de ellos calcule uno o más valores, y los retorne.

En el lenguaje C vienen predefinidos 4 tipos de datos básicos, a saber:

**int** (número entero: Positivo, negativo, o cero)

**float** (número real: Con parte entera y parte decimal; positivo, negativo o cero)

**double** (número real, pero con mayor rango que un float y con mayor cantidad de dígitos decimales)

**char** (caracter, es decir, letra, dígito, u otro símbolo unitario en el teclado)

Podemos realizar cálculos aritméticos con los datos de tipo int, float y double, pero los char no suelen tener un sentido numérico, sino que suelen usarse para datos no numéricos o alfanuméricos.

Se pueden crear nuevos tipos de datos a partir de estos tipos básicos, o de otros ya creados. Hay una palabra en C relacionada con los tipos de datos, y es **void**, que se usa para indicar que una función no retorna o no recibe valor.

Conviene aclarar que void no es un tipo de datos, sino una forma explícita de indicar que no hay dato en lo absoluto, como cuando en un libro se dice "esta página fue dejada deliberadamente en blanco".

Cada tipo de dato pertenece a una casta que es incompatible con las otras, aunque es posible promover o degradar un valor de tipo de datos a otro, mediante un mecanismo llamado type casting.

## Variable

Una variable es un lugar en la memoria del computador que puede almacenar o alojar un valor de un cierto tipo. Por tanto, una variable tiene siempre asociado un tipo de dato.

Cada variable es referenciada en el código mediante un nombre, el cual se crea de acuerdo a las siguientes reglas:

Debe consistir de una sola palabra, esto es, no puede contener espacios

Debe empezar con una letra

Sólo puede contener letras y/o dígitos, salvo el signo de subrayado (\_)

Las letras son del alfabeto inglés, esto es, ni acentos ni ñ están permitidos

No debe contener más de 32 caracteres.

Ejemplos de nombres válidos de variables son:

a  
B  
suma  
areaTriangulo  
prod1  
nuevo\_valor

*ecabrera, septiembre 2008.*

## Valores

Un valor es un dato que puede ser asignado a una variable, de acuerdo a su tipo. De manera que a una variable entera le podemos asignar números enteros, a una real números reales y a una caracter, caracteres.

Hay valores fijos o **constantes**, como 5, 3.1416 y 'x'.

Cuando una variable toma un valor, se convierte ella misma en un valor, por lo que puede asignarse a otra de ese mismo tipo. En este caso se dice que el valor es variable.

También se puede obtener un valor mediante el cálculo de una expresión aritmética o lógica.

Hay tres formas directas de asignar un valor a una variable:

Incondicional  
Genérica  
Condicional

La forma general de la asignación **incondicional** es:

variable = valor;

Ejemplos:

a = 5;  
a = b;  
a = b + 2;

La forma general de la asignación **genérica** (o pseudocódigo) es:

variable op= valor;

La partícula op que precede al signo = se puede cambiar por cualquiera de los símbolos =, -, \*, /, %. La asignación genérica es una forma de simplificar el código en ciertas asignaciones.

Ejemplos de asignaciones genéricas:

a += 2; (equivale a: a = a + 2;)  
 a -= 1; (equivale a: a = a - 1;)  
 a \*= b; (equivale a: a = a \* b;)  
 a /= c+2; (equivale a: a = a / (c+2);)  
 a %= 10; (equivale a: a = a % 10;)

La forma general de la asignación condicional es:

variable = (condición) ? valor1 : valor2;

Si la condición es verdadera, se asigna a la variable el valor1, sino el valor2.

Ejemplo:

a = (b > 0) ? 1 : 0;

Si el valor de b es positivo, se asigna uno a la variable, y cero en cualquier otro caso.

Es importante señalar que cualquier asignación es **destructiva**, es decir, el valor que había en una variable antes de la asignación es remplazado por el nuevo valor, y se pierde, a menos que haya sido salvado previamente en otra variable.

## Expresiones

Una expresión es un valor, ya sea fijo, variable u obtenido mediante el uso de operadores sobre valores. Un operador es un símbolo que permite obtener un nuevo valor a partir de otro (monario) y otros (binario). Hay distintos tipos de operadores, algunos de los cuales son:

**Aritméticos:** + (suma), - (resta), \* (multiplicación), / (división), % (módulo, o residuo)

**Relacionales:** < (menor que), <= (menor o igual a), == (igual a: Note los ==), >= (mayor o igual a), > (mayor que), != (no igual, o diferente)

**Lógicos:** && (Y, disyunción), || (O, conjunción), ! (NO, negación)

**Incremento y disminución:** ++ (suma 1 a la variable, antes o después de usar la variable),  
 -- (resta 1 a la variable, antes o después de usarla)

### Nota:

Los operadores de incremento a disminución tienen una sutileza interesante.

Por ejemplo, no es lo mismo c++ (primero usa a c y luego súmele 1)

++c (primero súmale 1 a c y luego úsala).

Ejemplos de expresiones:

```
a = a * 3;
a = a + 1;
a++;
a > b
a == b
a >= b
a || b
!(a == b)
(a > b) && (b > c)
```

*ecabrera, septiembre 2008.*

## Area

Podemos volver a considerar la función **areTriangulo()**:

```
float areaTriangulo(float b, float h) {
    return (b * h) / 2;
}
```

Vemos que la función retorna un número real (float); que recibe dos números reales (float b, float h); que se llama areaTriangulo; que retorna el valor que se obtiene al calcular la expresión  $(a * b) / 2$ .

Aquí b y h son valores variables (o simplemente variables), mientras que 2 es un valor fijo. Lo que esta función hace, vista a grandes rasgos, es recibir dos números reales cualesquiera, el primero en la variable b y el segundo en la variable h; calcular el valor de la expresión  $(b * h) / 2$  y retornar esta nuevo valor a la función que invoque a esta función areaTriangulo().

Una forma de invocar esta función (desde otra, por supuesto) podría ser:

```
area = areaTriangulo(2.0, 3.0);
```

En este caso el valor que tomaría la variable b sería 2.0, y el de h sería 3.0, lo cual haría que la expresión tomara el valor  $(2.0 * 3.0) / 2 = 6.0 / 2 = 3.0$ , que es el que se retornaría, y se asignaría a la variable **area**, la cual recibe el valor que la función retorna mediante la signación incondicional.

La función areaTriangulo(), pues, encapsula el proceso de calcular el área de un triángulo cualquiera a partir de los valores de la base y la altura, de manera que quien utilice esta función no tiene que conocer los detalles internos de cómo la función obtiene el valor que retorna, sino que basta saber qué tipo de valor retorna, cómo se llama, y los tipos de los valores que recibe.

Una versión más detallada de la función areaTriangulo() podría ser:

```
float areaTriangulo(float b, float h) {
    float area;
    area = (b * h) / 2;
    return area;
}
```

Note que esta versión tiene algunos cambios en los detalles internos, pero mantiene la misma forma para utilizarla: Retorna un valor real, y recibe dos valores reales.

## Programa

Habiendo discutido en qué consiste una función, pasemos a lo que es un programa C.

Un programa en lenguaje C es un conjunto de una o más funciones, donde nunca falta la función `main()`, pues ella constituye el programa en sí, y desde ella es que se invocan las otras funciones que pueda contener el programa para ayudarse a resolver el problema que tiene como propósito.

La forma más simple de un programa es:

```
comentario
librerías
int main() {
    cuerpo
}
```

El **comentario** no es obligatorio, pero se recomienda para explicitar al programador que lea información sobre el programador que redactó y el programa mismo. Hay dos formas de insertar el comentario:

```
// comentario de una línea
```

y

```
/* comentario de
varias líneas
*/
```

El comentario con `//` se usa para comentarios cortos que tienen una sola línea, o que se insertan luego del código ya escrito en una línea, como en el siguiente ejemplo:

```
int sum = 0; // suma, asegura en cero
```

El comentario de varias inicia con `/*` y puede prologarse tantas líneas como se desee y sólo termina cuando se coloque `*/`.

También se pueden usar los comentarios cuando se está probando un código, para que sea obviado por el compilador, como se ilustra:

```
// a = a * 7;
```

Esta línea, aunque tiene una asignación, no es tratada como tal, ya que al tener los símbolos `//` al inicio, es tratada como un comentario dirigido al programador, no a la máquina. El mismo efecto se obtendría con:

```
/* a = a * 7 */
```

Las librerías son recursos predefinidos que el programador puede reutilizar. Para incluir una librería se usa el siguiente formato:

```
#include <libreria.h>
```

o

```
#include "libreria.h"
```

La primera forma busca la librería en la carpeta include, mientras que la segunda la busca en distintos lugares, de acuerdo a la configuración del sistema operativo. Las librerías más utilizadas son:

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

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

La función main() es el programa en sí, y allí ocurre el control general de las acciones. Por ejemplo, un programa que despliegue en la pantalla "Hola, mundo.", podría ser:

```
// hola.c, despliega "Hola, mundo." en pantalla
```

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

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

```
int main() {
    printf("Hola, mundo.\n\n");
    system("pause");
    return 0;
}
```

El comentario indica que hola.c es el nombre con que se grabará el programa en el disco y que lo que hace es desplegar "Hola, mundo." en la pantalla. Las librerías a incluirse son **stdio.h** y **stdlib.h**. La función main() retorna un valor int, por lo que antes de concluir retorna un cero que le indica al sistema operativo que el programa terminó sin problemas.

La función printf(), que es invocada de la librería stdio.h, es la que despliega "hola, mundo." en la pantalla; mientras que system("pause") produce un pausa, hasta que el usuario pulse una tecla, de manera que pueda ver, en una ventana con fondo negro, el mensaje que se despliega. De paso, system("pause") genera un mensaje en idioma inglés, "Press any key to continue..." para que el usuario sepa que esperan pacientemente por él, hasta que quiera. Un programa completo que invoque nuestra función areaTriangulo() podría ser:

```
// areaTriangulo.c, calcula el area de un triángulo de base 6 y altura 5.
```

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

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

```
float areaTriangulo(float b, float h) {
    return (b * h) / 2;
}
```

```
int main() {
    printf("Area: %.2f\n\n", areaTriangulo(6, 5));
    system("pause");
    return 0;
}
```

Note que el código de la función `areaTriangulo()` se insertó antes del de la función `main()`, pero pudo haberse puesto después, o haberse declarado antes y detallarse después.

Note también que la función `printf()`, que es invocada de la librería `stdio.h`, despliega el valor que retorna la función `areaTriangulo`, que es un `float`, con dos decimales, pues usó el formato `%.2f`, que indica un número real con dos decimales. El número real que retorna la función `areaTriangulo()` es directamente desplegado, sin usar una variable intermedia.

## Escribir

Para desplegar (o escribir) un mensaje en la pantalla podemos usar la función **`printf()`**, cuyo formato es:

```
printf("formato/mensaje", valores);
```

Podemos usar sólo el mensaje, fijo, como:

```
printf("Hola, mundo.");
```

O combinar el mensaje con formatos:

```
printf("Hola, mundo.\n\n");
```

Cada secuencia `\n` hace saltar a la próxima línea en la pantalla, pues la **n** viene de **new line**. También podemos incluir un formato con valores:

```
printf("%d", 5);
```

Que despliega el valor fijo 5, pero a través del formato explícito para número entero. El siguiente ejemplo despliega un número real combinando mensaje con formato y valor no fijo:

```
printf("Suma: %f\n\n", sum);
```

El siguiente despliega tres valores enteros:

```
printf("%d %d %d\n\n", a, b, c);
```

Podemos especificar el número de decimales (4 en este caso) con que se despliega un valor real:

```
printf("%.4f", val);
```

**Leer**

Para permitir que el usuario introduzca valores en nuestro programa podemos usar la función `scanf()`, cuyo formato es:

```
scanf("formato", &nombre);
```

Note el uso del operador `&` antes del nombre de la variable, que es necesario para que C pueda acceder a la dirección de memoria que ocupa la variable. Se pueden especificar varias variables, siempre que se indique el formato correspondiente a cada una. Para leer tres enteros podemos:

```
scanf("%d %d %d", &a, &b, &c);
```

En este caso el usuario debe escribir cada número y dejar al menos un espacio entre cada dos de ellos y luego del último pulsar la tecla ENTER. Para leer un entero y un real:

```
scanf("%d %f", &e, &r);
```

Para ilustrar todo esto consideremos un programa que calcula el área de un triángulo pidiendo al usuario los valores de la base y de la altura, y desplegando el valor del área:

```
// areaTriangulo.c, calcula area de un triangulo dadas la base y la altura
#include <stdio.h>
#include <stdlib.h>
int main() {
    float b, h, a; // base , altura, area
    printf("Calcula el area de un triangulo dadas la base y la altura.\n\n");
    printf("Base: ");
    scanf("%f", &b);
    printf("Altura: ");
    scanf("%f", &h);
    a = (b * h) / 2;
    printf("Area: %.2f\n\n", a);
    system("pause");
    return 0;
}
```

**if()**

Para tomar decisiones en nuestro programa podemos usar la sentencia `if()`, cuyo formato más simple es:

```
if (condición) sentencia;
```

Si la condición es verdadera, se ejecuta la sentencia que sigue a la condición, si no, no se ejecuta. Ilustremos esto con una función que determine si un número es múltiplo de 3:

```
int multiplo3(int n) {
    // determina si n es multiplo de 3
    int sm = 0; // si multiplo, asume que no es
    if (n% 3 == 0) sm = 1;
    return sm;
}
```

Note que la función retorna un entero y recibe un entero, n, que declara una variable entera, sm, a la cual se asigna el valor 0 (falso), para asumir que el número recibido, n, no es múltiplo de 3.

La condición del if() se construye calculando el residuo que deja n al dividirlo por 3, y cuando este residuo es cero, se ejecuta el cambio del valor de sm de cero a uno, para indicar que n sí es múltiplo de 3.

Reiteremos que la función siempre retornará un cero en sm, a menos que la condición del if() sea verdadera, en cuyo caso el valor de sm cambia a 1, para indicar que n sí es múltiplo de 3.

Si se ejecuta esta función para n = 3, retorna un 1 en sm porque se ejecuta el if(); pero como no se ejecuta para n = 7, se retornaría un cero.

**if() { }**

Otro formato del if() es el siguiente:

```
if (condición) {
    sentencias
}
```

Note que ahora se pueden incluir más de una sentencia para que se ejecuten todas si la condición es verdadera, o no se ejecute ninguna si la condición es falsa. Así que este if() compuesto funciona de manera idéntica al if() simple, salvo porque se ejecutan varias sentencias en caso de darse como verdadera la condición. Ejemplo:

```
int a = 5, b;
if (a > 0) {
    printf(“%d es positivo.\n”, a);
    b = 1;
}
```

En este caso, como a es mayor que cero, se ejecutan tanto el printf() como la asignación de 1 a la variable b. Pero si cambiamos la asignación inicial (inicialización) de a con el valor fijo 5, no se ejecutan ninguna de las dos sentencias insertadas entre las llaves.

**if() else**

La sentencia if() permite una cláusula else (sino, de lo contrario) que permite que se tomen una (y sólo una) de dos acciones posibles, según que la condición sea verdadera o sea falsa. Su forma es:

```
if (condición)
    sentencia
else
    sentencia
```

Podemos ilustrarla con una función que determine si un número es positivo o no:

```
int positivo(int n) {
    // determina si n es positivo
    int sp; // si n es positivo
    if (n > 0 )
        sp = 1;
    else
        sp = 0;
    return sp;
}
```

Si n es mayor que cero se asigna a sp uno, pero en cualquier otro caso se le asigna cero. Nunca se pueden hacer ambas cosas y siempre se ejecuta una de las dos.

**if() if()**

Se puede anidar un if() en otro if(), de manera que se puedan manejar más de dos cursos de acción posibles. Sin embargo, no se recomienda usar demasiados niveles de anidamiento. Ilustremos esto con una función que determine el signo de un número real, el cual es -1 si el número es negativo, cero si el número es cero, y es 1 si el número es positivo:

```
int signo(float n) {
    // determina el signo de un número real: -1, 0, 1
    int sg; // signo de n

    if (n < 0)
        sg = -1;
    else
        if (n == 0)
            sg = 0;
        else
            sg = 1;

    return sg;
}
```

}

Note que la primera condición evalúa si  $n$  es negativo, y en tal caso le asigna a  $sg$   $-1$ , y con esto se sale del `if()` y se pasa a la sentencia `return`. Pero si esta primera condición es falsa, se pasa al `if()` que está luego del primer `else`, el cual tiene la condición de si  $n$  es igual a cero, en cuyo caso asigna  $0$  a  $sg$ , y si no es así, le asigna a  $sg$  el valor  $1$ .

Es interesante notar que no es necesario evaluar si el número es positivo, ya que se asume que si no es negativo ni cero, entonces ha de ser positivo, por lo que hay sólo dos condiciones, y no tres.

{ }

Todo el código que se coloca entre una llave abierta y otra cerrada, { }, se considera un bloque de código, una unidad lógica. Esto se ilustra en el formato de una función, el cuerpo de la cual siempre queda delimitado por las llaves que indican su comienzo y su fin. Lo mismo se aplica al `if()` compuesto, en el que las varias sentencias a ejecutarse en caso de la condición ser verdadera se delimitaron con llaves para hacerlas un todo monolítico.

Este uso de las llaves puede aplicarse en cualquier contexto en que se quiera sustituir una sentencia simple por varias sentencias. De esta manera, en el `if()` se pueden usar llaves antes del `else` para indicar que se ejecute un grupo de sentencias en caso de ser verdadera la condición, y también se pueden usar llaves para un grupo de sentencias que se deban ejecutar en caso de que la condición sea falsa.

Naturalmente, si se va a ejecutar una sola sentencia, las llaves son innecesarias, pero se pueden colocar sin que esto produzca efectos colaterales.

**for()**

Para repetir varias veces la ejecución de una o más sentencias se puede usar la sentencia `for()`:

```
for (inicialización; finalización; incremento)
    sentencia
```

El `for()` se recomienda cuando se tiene una variable, llamada contador, a la cual se puede asignar un valor inicial, se puede verificar que no pase de un valor final, y que se puede incrementar en un valor fijo para llegar desde el valor inicial hasta el final.

Para desplegar los primeros 9 números naturales podemos usar el código:

```
for (k = 1; k < 10; k++)
    printf("%d ", k);
```

Note que el contador  $k$  es inicializado en  $1$ , se verifica que no sea menor que  $10$  y se va incrementando de  $1$  en  $1$ . El funcionamiento es así: Se asigna  $1$  la variable  $k$ . Se verifica que este valor es menor que  $10$ , y como es el caso, se ejecuta la sentencia `print()` que aquí es el cuerpo del `for()`.

Luego de ejecutarse el despliegue del valor  $1$ , se incrementa a  $k$  en uno ( $k++$ ), lo cual la hace  $2$ , y se verifica nuevamente que este valor,  $2$ , es menor que  $10$ . Como lo es, se ejecuta nuevamente el despliegue, que es el de un  $2$ .

Se incrementa nuevamente a k, ahora 3, se verifica, se despliega, y se sigue este proceso de incrementar, verificar y desplegar hasta que k llega al valor 10, en cuyo caso la condición de finalización se hace falsa, y se abandona el for().

Es claro que el último valor de k desplegado es el 9, porque cuando k toma el valor 10, como la condición es falsa, se abandona el for().

Podríamos también ilustrar el uso del for con una función que obtenga la suma de los factores propios de un número entero positivo:

```
int sumaFactoresPropios(int n) {
// suma de los factores propios de n
    int s = 0; // suma , se asegura a cero
    int d ; // divisores, posibles factores de n
    for (d = 1; d <= n/2; d++)
        if (n %d == 0) s += d;
    return s;
}
```

Note que se declara a s para la suma, la cual se asegura en cero, y a d como contador para que recorra todos valores desde 1 hasta la mitad de n, pues ningún número tiene un factor superior a su mitad.

El for() empieza por asignar 1 a d. Se verifica que es menor que n, digamos 9, y si es así se ejecuta el cuerpo del for(), que en este caso verifica, con un if(), que n (9) es múltiplo de 1, y de ser así (que siempre es) , suma el valor de d, 1, a la suma.

Luego se incrementa a d, pasando a ser 2, y se verifica que es menor que la mitad de n (en este caso  $9/2 = 4.5$ ), y como es verdad, se ejecuta nuevamente el cuerpo del for(), pero como 9 no es divisible por 2, el 2 no se suma a s.

Se incrementa nuevamente a d, y ahora es 3, el cual es menor que 4.5, la mitad de n (9), y se procede a verificar si 9 es múltiplo de 3, y como lo es, se suma 3 a la suma, que ahora es  $0+1+3$ .

Luego se vuelve a incrementar a d, y ahora es 4, y como menor que 4.5, se verifica si 9 es múltiplo de 4, y como no lo es, no se suma el 4 a la suma.

Finalmente, se incrementa d a 5, y al verificar si menor que la mitad de 9, 4.5, resulta que no, y se abandona el for(), pasándose retornar la suma, que en este caso será un cuatro, la suma de los que resultaron ser factores propios de 9, es decir, de 1 y 3, solamente.

## while()

Otra sentencia para lograr la reiteración de una o más sentencias es while(), que es controlada sólo por una condición:

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

Si la condición es verdadera, se ejecuta la sentencia y se vuelve a verificar la condición, que si es nuevamente verdadera, se ejecuta sentencia. Este proceso sigue hasta que la condición se haga falsa.

Podemos ilustrar el uso del `while()` redactando una función que calcule la suma de los dígitos de un número entero positivo:

```
int sumDigitos(int n) {
    // obtiene suma de los digitos de n
    int s = 0; // suma , asegura cero
    while (n > 0) {
        s += n%10;
        n /= 10;
    }
    return s;
}
```

La función retorna un entero y recibe otro entero. Declara una variable `s`, asegurando su valor en cero para ir acumulando la suma de los dígitos a medida que los va encontrando.

La condición del `while()` es que termina cuando el valor de `n` deje de ser mayor que cero, lo cual verifica, y es siempre cierto la primera vez, si, como se asume, el número que se recibe es positivo.

El cuerpo del `while()` consiste de dos asignaciones. La primera le suma a `s` el residuo que deja `n` al dividirlo por dos, lo cual siempre da el último dígitos actual de `n`. Por ejemplo, si `n = 123`, el residuo de 123 al dividirlo por 10, `n%10`, da 3, que es último dígito de 123.

La segunda asignación en el `while()` susituye el valor actual de `n` por la parte entera de `n` al dividirlo por 10, lo cual equivale a "cortarle" su último dígito, de manera que 123 se sustituye por 12. Es claro que este de irle quitando sucesivamente el último dígito a `n` eventualmente llega a dar cero, que es cuando la condición del `while()` se hace falsa, y se abandona. Pero para entonces ya la suma de los dígitos, uno a uno, ha sido acumulada en `s`. Y esta suma es la que la función retorna.

### **do() while;**

Otra sentencia para reiterar la ejecución de una o más sentencias es `do() while`, similar a `while`, salvo que la condición se coloca y evalúa luego de jecutarse, al menos una vez, el cuerpo del `do() while`. Su formato es:

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

La podemos ilustrar con una función que permita aceptar un número entero dentro de un intervalo, entre `a` y `b`, asumiendo que `a <= b`:

```
int aceptaEntero(int a, int b) {
    // acepta entero entre a y b, ambos inclusive
    int n; // entero a retornar
    do
        scanf("%d", &n);
```

```

    while (n < a || n > b);
    return n;
}

```

Es claro que la sentencia `scanf()` se ejecuta al menos una vez ya que la condición está luego de ella. Si el valor de `n` aceptado es mayor o igual que `a` y menor o igual que `b`, se sale del `do()` `while` porque la condición es falsa. pero si el valor de `n` aceptado es menor que `a` o mayor que `b`, la condición es verdadera y se vuelve a ejecutar la sentencia `scanf()`, que en este caso constituye todo el cuerpo del `do()` `while`, y así se seguirá repitiendo su ejecución, hasta que el usuario inserte un valor para `n` que sea mayor o igual que `a` y menor o igual que `b`, que es lo que hace falsa la condición.

## arreglo

Una variable puede alojar múltiples valores a la vez y para ello al declararla se especifica hasta cuántos valores puede admitir. A una variable tal se le llama un arreglo. Para declarar un arreglo se usa el siguiente formato:

```
tipo nombre[tamaño];
```

Por ejemplo, si queremos alojar hasta 100 números enteros en un arreglo llamado `a`, podemos declarar:

```
int a[100];
```

Los arreglos permiten generalizar ciertas soluciones particulares a determinados problemas. Por ejemplo, si queremos redactar una función que sume 3 números reales, podemos escribir:

```
float suma3(float a, float b, float c) {
    // suma 3 reales
    float s = a + b + c;
    return s;
}

```

Es claro que si queremos que la función sume 5 reales, tendremos que usar 5 parámetros, y 20 si queremos que sume 20. Pero con un arreglo de números reales, la solución se escribe con mucho menos código, aparte de que la cantidad de valores a sumar puede ser cualquiera, sin tener que cambiar el código en lo más mínimo. Por esto se dice que los arreglos permiten **generalizar** soluciones.

Una función que permite sumar una cantidad arbitraria de números reales podría redactarse así:

```
float suma(float r[ ], int n) {
    // suma los n números reales que llegan en r
    float s = 0; // suma, asegura cero
    int p; // posición en r
    for (p = 0; p < n; p++)
        s += r[p];
    return s;
}

```

Hay que decir que la primera posición en un arreglo es siempre la cero, de manera que si un arreglo tiene tamaño 5, las posiciones van desde la cero hasta la 4; en general, desde la cero hasta la n-1.

Por eso el for() de esta función empieza en cero y termina en n-1, porque tan pronto p, la posición, llega a n, se sale del for() y no se llega a acceder a la posición n, sino que la última procesada es la n-1.

Dentro del for() lo único que se hace es sumar el valor que está en el arreglo r en la posición p, y el for() se asegura de recorrer todas las posiciones y esta asignación hace que todos se acumulen. Claro, previamente se aseguró de que la suma inicializara en cero, para que no quede adulterada con valores anteriores inciertos.

Un programa completo que utilice esta función suma() podría ser:

```
// suma.c, suma n valores, con un arreglo y una función
#include <stdio.h>
#include <stdlib.h>
float suma(float r[ ], int n) {
    // suma los n números reales que llegan en r
    float s = 0; // suma, asegura cero
    int p; // posición en r
    for (p = 0; p < n; p++)
        s += r[p];
    return s;
}
int main() {
    int lim = 100; // tamaño del arreglo
    float arr[lim]; // arreglo de lim valores
    float sum; // suma de los valores pedidos
    int can; // cantidad concreta de valores a sumar
    int k; // contador de cero a can-1
    printf("Suma N numeros reales dados por el usuario.\n\n");
    do {
        printf("Cantidad de valores a sumar (1 a %d): ", lim);
        scanf("%d", &can);
    } while (can < 1 || can > lim);
    for (k = 0; k < can; k++) {
        printf("Valor # %d: ", k+1);
        scanf("%f", &arr[k]);
    }
    sum = suma(arr, can);
    printf("Suma: %.2f\n\n", sum);
    system("pause");
    return 0;
}
```

La función `suma()` se ha insertado antes del código de la función `main()`, y en ésta se declara a la variable entera `lim` para establecer cuál es el tamaño (máximo de elementos) del arreglo, el cual se declara con el nombre `arr` y tamaño `lim` (100, en este caso).

Se declara la variable `sum` para la suma, la cual no es necesario inicializar a cero, porque su valor vendrá dado por lo que retorne la función `suma()` luego de ser invocada.

Se declara la variable `can` para especificar la cantidad concreta (entre 1 y `lim`) que el usuario quiere sumar cada vez que ejecute el programa. Tampoco se inicializa porque su valor será dado por el usuario.

Se declara la variable `k`, contador, para que, una vez el usuario haya especificado el la cantidad de valores a sumar, se procese desde cero hasta `can-1` para pedirle cada valor y colocarlo en el arreglo `arr`.

A seguidas se despliega en la pantalla el mensaje que le explica la usuario el propósito de este programa: Sumar una cantidad `N` de valores.

El ciclo `do while()` que sigue se asegura de que el usuario introduzca un valor entre 1 y `lim`, ambos inclusive, y si se sale de este rango, le vuelve a pedir la cantidad cada vez que se equivoque. Y es claro que si no se equivoca la primera vez, pasa como si el ciclo no estuviera allí, porque la condición se hace falsa esta primera vez. Pero si se sale del rango de 1 a `lim`, cada vez que se equivoque, le pide el valor de `can` nuevamente.

Luego hay un `for()` que le pide al usuario cada uno de los valores a sumar, colocando cada uno en el arreglo `arr` (`scanf("%f", &arr[p]);`) en su posición correspondiente, empezando desde la cero.

Lo que sigue es invocar la función `suma()`, asignando el valor que retorna a la variable `sum`, cuyo valor es a seguidas desplegado en el `printf()` que sigue. Note que la suma se despliega con dos decimales.

*ecabrera, octubre 2008.*