Capítulo

3

Sentencias de iteración en C

En este capítulo se describen las sentencias para realizar la repetición de un conjunto instrucciones. Son las sentencias de iteración, que permitirán ir avanzando en el grado de dificultad de los programas realizados.

Bucles: while, do while y for

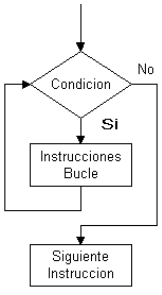
En el capítulo anterior vimos que la secuencialidad en la ejecución puede ser modificada con sentencias de control del tipo *if, if / else y switch / case*. Existen otras instrucciones, también de control, que permiten ejecutar repetidamente un grupo de sentencias, son los denominados bucles. Existen tres tipos en C: *while, do while y for*.

El bucle *while* es la sentencia más frecuente para controlar la repetición, de hecho con sólo esta instrucción y el *if* sería suficiente para programar estructuradamente. La estructura de la sentencia es la siguiente:

```
while ( expresión )
        sentencia(s);
```

Esto significa que *mientras (while*) una expresión sea distinta de 0 (TRUE) se ejecuten las siguientes instrucciones. Si el bucle tiene que realizar sólo una sentencia no será necesario encerrarla entre marcadores de bloque (llaves).

El funcionamiento es el siguiente: nada más entrar a ejecutar la sentencia *while*, se evaluará la expresión, si resulta cierta (distinta de 0) se ejecutan las sentencias internas al bucle (el cuerpo del *while*), si resulta falsa se salta la ejecución del mismo, pasándose a la siguiente sentencia. En el primer caso y una vez ejecutadas todas las sentencias internas, se vuelve a evaluar la condición de entrada, repitiéndose el proceso. Esto quiere decir, que en el cuerpo del bucle tiene que haber alguna instrucción que modifique en algo la evaluación de la expresión para que en un momento dado se puede salir de la ejecución del bucle.



El bucle *while* es útil cuando queremos repetir un grupo de instrucciones un número indeterminado de veces que no conocemos a priori, por ejemplo, si queremos dividir un entero por otro repetidamente, hasta que el resultado sea menor que el divisor (calcular los restos), ¿Cuántas veces tendremos que realizar esto?, en principio no se sabe, dependerá del número a dividir y del divisor, por lo tanto el número de iteraciones vendrá determinado por la condición “resultado < divisor”.

Para ver cómo funciona el bucle *while*, veamos el ejemplo anterior de la división, al que añadiremos que nos imprima los restos y además cuente el número de divisiones hechas:

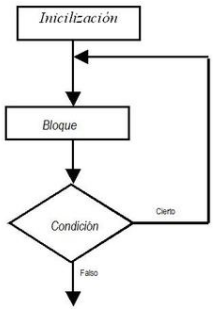
```
#include <stdio.h>
int main()
{
   int dividendo, divisor, resto;
   int contador = 0;
   printf ("Introduce dividendo y divisor : \n");
   scanf("%d %d", &dividendo, &divisor);
   while (dividendo >= divisor)   /* solo sigo si es mayor */
   {
         resto = dividendo % divisor;     /* hago la division */
         dividendo = dividendo / divisor;
         printf("%9d", resto);          /* a la derecha */
         contador++;  /* aumento contador de divisiones */
   }
   printf ("\nEl dividendo es %d\n", dividendo);
   printf("Numero de divisiones : %d \n", contador);
}
```

Lo primero que hacemos en el programa es inicializar la variable contador, ya que nada más ejecutar un programa todas sus variables tienen un valor aleatorio (dependiendo de como este la memoria del computador), es la única que inicializamos, ya que las demás toman un valor dado en el programa (dividendo y divisor en el *scanf* y resto al hacer el %).

Es muy frecuente al empezar a programar, dejar sin inicializar las variables, con lo que al utilizar instrucciones como: contador = contador + 1; se producen errores difíciles de detectar.

Una vez introducidos los datos, entro en el bucle *while* y realizo la primera evaluación de la expresión, si el dividendo es menor (¡hemos usado mayor o igual!) que el divisor se saltará hasta la siguiente instrucción fuera del bucle (*printf*) si no se realizará la primera división, calculando el resto y modificando el dividendo para la nueva evaluación del bucle, después se suma el contador y se vuelve a realizar la evaluación de la expresión de entrada, para ver si se hace una nueva iteración.

Para realizar iteraciones cuando no se conoce el número de ellas a priori, existe otro bucle que es el *while* de salida o *do-while* (en otros lenguajes es el repeat-until pero la condición es de permanencia no de salida). El bucle *while* se distingue del *do-while* en que la expresión se evalúa al comienzo de la iteración, con lo cual se puede salir del bucle sin que este se haya ejecutado su cuerpo, sin embargo, en el bucle *do-while* la expresión se realiza después de ejecutarse el cuerpo, con lo cual, siempre se entra al menos una vez en él. La estructura del bucle es:




```
do
        sentencia(s);
while ( expresión );
```

Anteriormente dijimos que con el bucle *while* teníamos suficiente para programar una iteración, entonces ¿Por qué en C existen dos tipos de bucles? La respuesta es que, en algunos casos, resultará más conveniente entrar directamente en el bucle y hacer la evaluación de la expresión de permanencia después de haber ejecutado una vez el cuerpo del bucle, que realizarla antes de entrar.

Veamos esto con un ejemplo, queremos leer una lista de números positivos y sumarlos hasta que se introduzca un número negativo, realicemos esta tarea usando los dos tipos de bucles. Vemos como el bucle *do-while*, en este caso, no funciona, porque lo que nos interesa, antes de realizar la suma, es efectuar el chequeo del número introducido. Si introducimos como primer número uno

negativo, en el primer caso no realiza ninguna sentencia del bucle, en el segundo caso, hace una suma errónea antes de darnos el resultado.

```
#include <stdio.h>
int main()
{
     int numero, suma = 0;
     scanf("%d", &numero);
     while (numero >= 0){
            suma += numero;
            scanf("%d", &numero);
     }
     printf ("La suma es : %d\n", suma);
}
```

```
#include <stdio.h>
int main()
{
    int numero, suma = 0;
    do {
        scanf("%d", &numero);
        suma += numero;
    }
    while (numero >= 0);
    printf ("La suma es : %d\n",suma);
}
```

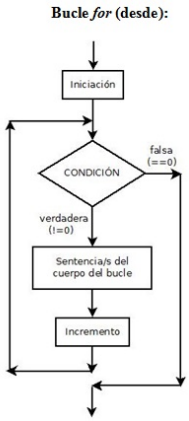
En este último ejemplo hemos utilizado el operador de la asignación aditiva. Lo mismo que para la suma, existen operadores para la resta, multiplicación, división y resto, son binarios y su precedencia es la de la asignación:

a = a - 20; <=> a -= 20; a = a / 20; <=> a /= 20;
a = a * 20; <=> a *= 20; a = a % 20 <=> a %= 20;

Existe un tercer bucle, que es el bucle *for*. Consta de tres campos separados por puntos y comas: por una lado está la inicialización, por otro está la expresión de permanencia (condición) y por último está la ejecución para la siguiente iteración (incremento). Su estructura es:

Bucle *for* en C:

```
for (iniciación; condición; incremento)
   {
      sentencia_1;
      . . .
      sentencia_n;
   }
```



Realmente es un bucle while encubierto que puede hacer la labor de él en un paso, su funcionamiento se compara con el while con este ejemplo:

```
cont = 5;                    <=>    for (cont = 5; cont <= numero; cont++)
while (cont <= numero)              1     printf("hola\n");
{
   printf("hola\n");
   cont++;
}
```

2,5,8… **4,7,10…** **3,6,9…**

La variable contadora al principio tomará el valor 5 (en rojo, paso 1), después se comprobará la condición de permanencia (en azul, paso 2), si es cierta se ejecutará la(s) sentencia(s) interior (en verde, paso 3), cuando se termine, se procederá a ejecutar la tercera parte del bucle, el incremento de *cont* (en morado, paso 4) y se volverá a comprobar la permanencia (y así sucesivamente). En algún momento determinado la condición de permanencia será FALSE y se pasará a la siguiente sentencia fuera del *for*.

En algunos casos es útil además de inicializar, declarar la variable (contadora) del for en el propio bucle. Eso significa que el ámbito de esa variable será el bucle y no puede usarse fuera de él.

```
for (int cont = 1; cont <= numero; cont++)
```

Se suele decir (hay diferentes escuelas) que un buen estilo de programación es colocar las variables lo más cerca posible de donde se usan, aunque en C K&R (clásico) esto no se hace así (se colocan al principio) en ANSI C y posteriores si se puede hacer, pero se debe tener en cuenta que el ámbito de la variable (dónde se puede usar) es sólo el del bloque ({ }) donde está declarada.

A diferencia de otros lenguajes, el bucle *for* de C tiene una gran flexibilidad y existen muchas maneras de utilizarlo:

1. Se puede utilizar para hacer bucles ascendentes y descendentes:

```
for (n = 10; n > 0; n--)
```

2. Para hacer bucles con iteraciones de paso a paso:

```
for (n = 2; n < 75; n += 15)
```

3. Con cualquier tipo de dato:

```
for (ch = 'a'; ch <= 'z'; ch++)
```

4. Con progresión geométrica de la variable (no siempre aritmética):

```
for (d = 1; divisas < 1500; divisas = divisas * 10)
```

5. De carácter general (con una expresión compleja):

```
for (x = 1; y <= 43; y = 4 * x++ + 11)
```

6. Debido a que la sentencia nula es cierta se pueden hacer bucles infinitos o dejar alguna parte del bucle vacía:

```
for ( ; ; )          for(x = 0; x != 22; )
```

7. Por último se puede utilizar cualquier expresión en cada parte:

```
for (printf("inicio\n"); scanf("%d",&n); printf("cuento\n"))
```

Todavía se puede ampliar más el bucle utilizando el operador coma (no es el que vimos en la declaración de variables), es binario y su precedencia es la más baja de todos los operadores, incluyendo la asignación:

```
for (g = 5, f = 4; g <= 50; g += 5, d += 7);
```

Veamos un ejemplo clásico de la suma de Zenón, es decir, 1 + 1/2 + 1/4 +... Esto se podría hacer en C con un bucle *for*:

```
for (suma = 0.0, x = 1.0, cont = 1; cont <= 20; cont ++, x *= 2.0)
       suma += 1.0 / x;
```

En este ejemplo se ve la filosofía del lenguaje C, que trata de compactar al máximo las sentencias usadas (normalmente a costa de la legibilidad del programa).

Hay que tener en cuenta con respecto a otros lenguajes como el Pascal, el BASIC o el Fortran, que su bucle FOR clásico se puede implementar de la siguiente manera:

```
for ( i = 1; i <= limite ; i ++)
```

y que ahora se puede cambiar la variable contadora dentro del bucle (aunque no es aconsejable), cosa que antes era sumamente arriesgado.

En los cuerpos de los tres tipos de bucles que hemos descrito, se pueden incluir otros bucles en al menos hasta15 niveles según el estándar ANSI.

Sentencias de control

Como vimos en el capítulo anterior hay otras sentencias de control que se pueden utilizar en los bucles. Por un lado, está **break**, que no sólo se puede utilizar en el *switch*, y que sirve para interrumpir el control de una instrucción y pasar a la siguiente. Por lo tanto se puede utilizar para abandonar un bucle (si fuera interno se saldría sólo del interno):

```
while ((ch = getchar()) != EOF)
{
      if (ch == '\n')
            break;
      putchar(ch);
}
```

En este caso, para abandonar el bucle hay dos condiciones, que se alcance el EOF (ver final del tema 1) o que el carácter leído sea una nueva línea. También se puede hacer lo mismo complicando la expresión del bucle:

```
while ((ch = getchar()) != EOF && ch != '\n')
```

Por otro lado, está **continue**, que se utiliza en los bucles de la misma forma que *break*, pero en vez de salirse definitivamente sólo se salta una iteración.

```
while ((ch = getchar()) != EOF)
{
      if (ch == '\n')
            continue;
      putchar(ch);
}
```

En este caso se podría sustituir en el cuerpo del bucle por:

```
if (ch != '\n')
   putchar(ch);
```

Estas dos sentencias son equivalentes al `goto` (de forma encubierta) y como sabemos, en la programación estructurada está denostado el uso de esta sentencia, ya que puede ser sustituida por otras que harán los programas más claros.

En el paradigma de la *programación estructurada*, todo algoritmo (propio[1]) debe expresarse sólo con sentencias secuenciales, iteraciones o condiciones, y ser dividido en subrutinas. El teorema de la programación estructurada o de Böhm-Jacopini demuestra que la instrucción goto[2] no es estrictamente necesaria y que para todo programa que la utilice, existe otro equivalente que no hace

[1] Entendemos que es un algoritmo propio aquel que cumple estas condiciones: 1. Tiene un único punto de entrada y un único punto de salida. 2. Todas las sentencias son alcanzables. 3. No hay bucles infinitos. 4. No hay ambigüedades. 5. Todos los posibles caminos llevan desde el punto de entrada al de salida. 6. El algoritmo acaba tras un número finito de pasos. 7. El algoritmo debe producir al menos una salida o un efecto. 8. Todas las sentencias del algoritmo deben poder realizarse de manera precisa en un tiempo finito.

[2] El inicio de la controversia que dio origen a la programación estructurada fue un artículo de Dijkstra a la ACM de marzo del 68 titulado “Go To Statement Considered Harmful” (la sentencia goto puede considerarse dañina). Donde indicaba todos los problemas a los que estaba dando lugar el uso de programación no estructurada. A este investigador le encontraréis en muchas materias de informática, como teoría de grafos, o programación concurrente (diseño de sección crítica y semáforos).

uso de dicha instrucción y puede expresarse con programación estructurada. En resumen un salto hacia adelante puede realizarse con una condición y un salto atrás con una iteración.

En mi opinión sólo son justificables en casos de detección de error y siempre acompañadas de una sentencia de condición.

Posteriormente a la programación estructurada, se han creado nuevos paradigmas de programación, como la programación modular, la programación orientada a objetos, la programación por capas, etc., y el desarrollo de entornos de programación que facilitan la programación de grandes aplicaciones y sistemas. Pero las piezas de las que están compuestos estos nuevos paradigmas siempre será recomendable que utilicen la programación estructurada para representar el algoritmo que intentan desarrollar.

Ejemplos

```c
/* Ejemplo 11.c
Comprende las operaciones que hay.
Introduce varios pares de números para ver que se dan diversas vueltas.
Cambia el programa para que sea un conversor de base.      */


#include <stdio.h>
int main()
{
    int dividendo, divisor, resto;
    int contador = 0;
    printf ("Introduce dividendo y divisor : \n");
    scanf("%d %d", &dividendo, &divisor);
    while (dividendo >= divisor)           /* solo sigo si es mayor */
    {
        resto = dividendo % divisor;       /* hago la division */
        dividendo = dividendo / divisor;   // /=
        printf("%d %5d | ", dividendo, resto);          /* a la derecha */
        contador++;                  /* aumento contador de divisiones */
    }
    printf ("\nEl resto es %d  ", dividendo);
    printf("El numero de divisiones : %d \n", contador);
    return 0;
}

```

```c
/* Ejemplo 12.c
Comprende las operaciones que hay.
Se trata de sumar los números introducidos hasta que se escriba uno negativo.
Date cuenta que si empiezas por uno negativo el do-while no vale.     */


#include <stdio.h>
int main()
{
    int numero, suma = 0;
    scanf("%d", &numero);
    while (numero >= 0)
    {
        suma += numero;
        scanf("%d", &numero);
    }
    printf("La suma es : %d\n", suma);
    return 0;
}

```

```c
/* Ejemplo 13.c
Comprende las operaciones que hay.
Se trata de sumar los números introducidos hasta que se escriba uno negativo.
Date cuenta que si empiezas por uno negativo el do-while no vale.     */


#include <stdio.h>
int main()
{
    int numero, suma = 0;
    do {
        scanf("%d", &numero);
        suma += numero;
    }
    while (numero >= 0);
    printf ("La suma es : %d\n", suma);
    return 0;
}

```

```c
/* Ejemplo 14.c
Comprende las operaciones que hay dentro de los bucles for.
Cambia algunos de los ejemplos    */


#include <stdio.h>
int main()
{
    int n, x, y;
    char ch;
    //Se puede utilizar para hacer bucles ascendentes y descendentes:
    printf("Descendiente  ");
    for (n = 10; n > 0; n--)
        printf("%4d ", n);
    //Para hacer bucles con iteraciones de paso a paso:
    printf("\nPaso a paso  ");
    for (n = 2; n < 75; n += 15)
        printf("%4d ", n);
    //Con cualquier tipo de dato:
    printf("\nOtros datos  ");
    for (ch = 'a'; ch <= 'z'; ch++)
        printf("%4c ", ch);
    //Con progresiÃ³n geomÃ©trica de la variable (no siempre aritmÃ©tica):
    printf("\nProgresiones  ");
    for (n = 1; n < 1500; n = n * 10)
        printf("%4d ", n);
    //De carÃ¡cter general (con una expresiÃ³n compleja):
    printf("\nDos variables  ");
    for (x = 1; y <= 43; y = 4 * x++ + 11)
        printf("%d %4d - ", x,y);
    //Debido a que la sentencia nula es cierta se pueden hacer bucles infinitos o dejar alguna parte del bucle vacÃa:
    //  for ( ; ; ) ;
    printf("\nBlanco  ");
        for(x = 0; x < 22; )
        printf("%d ", x++);
    //Utilizando el operador coma (mÃ¡s baja precedencia):
    printf("\nComa  ");
    for (x = 5, y = 4, n = 0; x <= 50; n += 5, x += 7)
        printf("%d %4d %4d / ", x,y,n);
    //En C99 la parte de inicializaciÃ³n puede ser de declaraciÃ³n. Compilarlo con c99
    printf("\nInicializado");
    for (char c = 'a'; c <= 'z'; c++)
        printf("%4c ", c);

    printf("\n");

    return 0;
}

```

```c
/* Ejemplo 15.c
Comprende las operaciones que hay dentro del bucle while.
Indica cuando terminará el programa,    */


#include <stdio.h>                              /* fichero de cabecera para el preprocesador */

int main ()
{
    char ch;

    while ((ch = getchar()) != EOF)
    {
        if (ch == 'a')
            break;
        putchar(ch);
    }

        return(0);
}


```

```c
/* Ejemplo 16.c
Indica que aparecerá y que no, cuando ejecutes el programa.
Observación: Estas dos sentencias son equivalentes al goto (de forma encubierta),
en la programación estructurada está denostado el uso de esta sentencia,
ya que puede ser sustituida por otras que harán los programas más claros.
En mi opinión sólo son justificables en casos de detección de error y acompañadas de una condición.     */


#include <stdio.h>                              /* fichero de cabecera para el preprocesador */

int main ()
{
    char ch;

    while ((ch = getchar()) != EOF)
    {
        if (ch == 'a')
            continue;
        putchar(ch);
    }

        return(0);
}
```