CAPÍTULO 5

Flujo de control II: Estructuras repetitivas

5.1. Estructuras repetitivas
5.2. Estructura **`mientras ("while")`**
5.3. Estructura **`hacer-mientras ("do-while")`**
5.4. Diferencias entre **`mientras`** (**`while`**) y **`hacer-mientras`** (**`do-while`**): una aplicación en C++
5.5. Estructura **`repetir`** (**`"repeat"`**)
5.6. Estructura **`desde/para`** (**`"for"`**)
5.7. Salidas internas de los bucles
5.8. Sentencias de salto **`interrumpir`** (**`break`**) y **`continuar`** (**`continue`**)
5.9. Comparación de bucles **`while`**, **`for`** y **`do-while`**: una aplicación en C++
5.10. Diseño de bucles (lazos)
5.11. Estructuras repetitivas anidadas
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS
REFERENCIAS BIBLIOGRÁFICAS

INTRODUCCIÓN

Los programas utilizados hasta este momento han examinado conceptos de programación, tales como entradas, salidas, asignaciones, expresiones y operaciones, sentencias secuenciales y de selección. Sin embargo, muchos problemas requieren de características de repetición, en las que algunos cálculos o secuencia de instrucciones se repiten una y otra vez, utilizando diferentes conjuntos de datos. Ejemplos de tales tareas repetitivas incluyen verificaciones (chequeos) de entradas de datos de usuarios hasta que se introduce una entrada aceptable, tal como una contraseña válida; conteo y acumulación de totales parciales; aceptación constante de entradas de datos y recálculos de valores de salida, cuyo proceso sólo se detiene cuando se introduce o se presenta un valor centinela.

Este capítulo examina los diferentes métodos que tilizan los programadores para construir secciones de código repetitivas. Se describe y analiza el concepto de **bucle** como la sección de código que se repite y que se denomina así ya que cuando termina la ejecución de la última sentencia el flujo de control vuelve a la primera sentencia y comienza otra repetición de las sentencias del código. Cada repetición se conoce como *iteración* o *pasada a través del bucle*.

Se estudian los bucles más típicos, tales como **`mientras`**, **`hacer-mientras`**, **`repetir-hasta que`** y **`desde`** (o **`para`**).

5.1. ESTRUCTURAS REPETITIVAS

Las computadoras están especialmente diseñadas para todas aquellas aplicaciones en las cuales una operación o conjunto de ellas deben repetirse muchas veces. Un tipo muy importante de estructura es el algoritmo necesario para repetir una o varias acciones un número determinado de veces. Un programa que lee una lista de números puede repetir la misma secuencia de mensajes al usuario e instrucciones de lectura hasta que todos los números de un fichero se lean.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan *bucles* y se denomina *iteración* al hecho de repetir la ejecución de una secuencia de acciones. Un ejemplo aclarará la cuestión.

Supongamos que se desea sumar una lista de números escritos desde teclado —por ejemplo, calificaciones de los alumnos de una clase—. El medio conocido hasta ahora es leer los números y añadir sus valores a una variable `SUMA` que contenga las sucesivas sumas parciales. La variable `SUMA` se hace igual a cero y a continuación se incrementa en el valor del número cada vez que uno de ellos se lea. El algoritmo que resuelve este problema es:

```
algoritmo suma
var
  entero : SUMA, NUMERO
inicio
  SUMA ← 0
  leer(numero)
  SUMA ←SUMA + numero
  leer(numero)
  SUMA ← SUMA + numero
  leer(numero)
fin
```

y así sucesivamente para cada número de la lista. En otras palabras, el algoritmo repite muchas veces las acciones.

```
leer(numero)
SUMA ← SUMA + numero
```

Tales opciones repetidas se denominan *bucles* o *lazos*. La acción (o acciones) que se repite en un bucle se denomina *iteración*. Las dos principales preguntas a realizarse en el diseño de un bucle son ¿qué contiene el bucle? y ¿cuántas veces se debe repetir?

Cuando se utiliza un bucle para sumar una lista de números, se necesita saber cuántos números se han de sumar. Para ello necesitaremos conocer algún medio para *detener* el bucle. En el ejemplo anterior usaremos la técnica de solicitar al usuario el número que desea, por ejemplo, `N`. Existen dos procedimientos para contar el número de iteraciones, usar una variable `TOTAL` que se inicializa a la cantidad de números que se desea y a continuación se decrementa en uno cada vez que el bucle se repite (este procedimiento añade una acción más al cuerpo del bucle: `TOTAL ← TOTAL - 1`), o bien inicializar la variable `TOTAL` en `0` o en `1` e ir incrementando en uno a cada iteración hasta llegar al número deseado.

```
algoritmo suma_numero
var
  entero : N, TOTAL
  real : NUMERO, SUMA
inicio
  leer(N)
  TOTAL ← N
  SUMA ← 0
  mientras TOTAL > 0 hacer
    leer(NUMERO)
    SUMA ← SUMA + NUMERO
    TOTAL ← TOTAL - 1
  fin_mientras
  escribir('La suma de los', N, 'números es', SUMA)
fin
```

El bucle podrá también haberse terminado poniendo cualquiera de estas condiciones:

- *hasta_que* TOTAL sea cero
- *desde* 1 *hasta* N

Para detener la ejecución de los bucles se utiliza una condición de parada. El pseudocódigo de una estructura repetitiva tendrá siempre este formato:

```
inicio
//inicialización de variables
repetir
  acciones S1, S2, ...
  salir según condición
  acciones Sn, Sn+1, ...
fin_repetir
```

Aunque la condición de salida se indica en el formato anterior en el interior del bucle —y existen lenguajes que así la contienen expresamente[1]—, lo normal es que *la condición se indique al final o al principio del bucle*, y así se consideran tres tipos de instrucciones o estructuras repetitivas o iterativas generales y una particular que denominaremos **iterar**, que contiene la salida en el interior del bucle.

```
iterar           (loop)
mientras         (while)
hacer-mientras   (do-while)
repetir          (repeat)
desde            (for)
```

El algoritmo de suma anterior podría expresarse en pseudocódigo estándar así:

```
algoritmo SUMA_numeros
var
  entero : N, TOTAL
  real : NUMERO, SUMA
inicio
  leer(N)
  TOTAL ← N
  SUMA ← 0
  repetir
    leer(NUMERO)
    SUMA ← SUMA + NUMERO
    TOTAL ← TOTAL - 1
  hasta_que TOTAL = 0
  escribir('La suma es', SUMA)
fin
```

Los tres casos generales de estructuras repetitivas dependen de la situación y modo de la condición. La condición se evalúa tan pronto se encuentra en el algoritmo y su resultado producirá los tres tipos de estructuras citadas.

1. La condición de salida del bucle se realiza al principio del bucle (estructura **mientras**).

```
algoritmo SUMA1
inicio
```

[1] Modula-2 entre otros.

```
  //Inicializar K, S a cero
    K ← 0
    S ← 0
  leer(n)
  mientras K < n hacer
    K ← K + 1
    S ← S + K
  fin_mientras
  escribir (S)
fin
```

Se ejecuta el bucle *mientras* se verifica una condición `(K < n)`.

2. La condición de salida se origina al final del bucle; el bucle se ejecuta *hasta que* se verifica una cierta condición.

```
repetir
  K ← K + 1
  S ← S + K
hasta_que K > n
```

3. La condición de salida se realiza con un contador que cuenta el número de iteraciones.

```
desde i = vi hasta vf hacer
S ← S + i
fin_desde
```

`i` es un contador que cuenta desde el valor inicial `(vi)` hasta el valor final `(vf)` con los incrementos que se consideren; si no se indica nada, el incremento es 1.

5.2. ESTRUCTURA `mientras ("while")`

La estructura repetitiva **`mientras`** (en inglés **`while`** o **`dowhile`**: *hacer mientras*) es aquella en que el cuerpo del bucle se repite mientras se cumple una determinada condición. Cuando se ejecuta la instrucción **`mientras`**, la primera cosa que sucede es que se evalúa la condición (una expresión *booleana*). Si se evalúa *falsa*, no se toma ninguna acción y el programa prosigue en la siguiente instrucción del bucle. Si la expresión *booleana* es *verdadera*, entonces se ejecuta el cuerpo del bucle, después de lo cual se evalúa de nuevo la expresión booleana. Este proceso se repite una y otra vez **`mientras`** la expresión *booleana* (condición) sea verdadera. El ejemplo anterior quedaría así y sus representaciones gráficas como las mostradas en la Figura 5.1.

EJEMPLO 5.1

Leer por teclado un número que represente una cantidad de números que a su vez se leerán también por teclado. Calcular la suma de todos esos números.

```
algoritmo suma_numeros
var
  entero : N, TOTAL
  real : numero, SUMA
inicio
  leer(N)
  {leer numero total N}
  TOTAL ← N
  SUMA ← 0
```

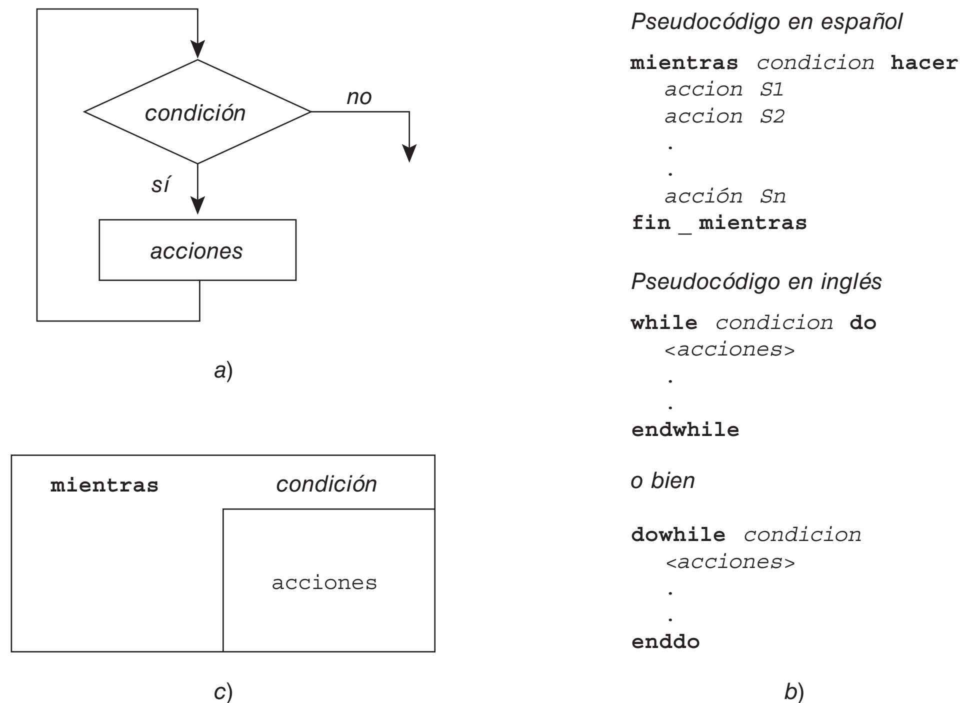


Figura 5.1. Estructura **mientras**: *a*) diagrama de flujo; *b*) pseudocódigo; *c*) diagrama N-S.

```
  mientras TOTAL > 0 hacer
    leer(numero)
    SUMA ← SUMA + numero
    TOTAL ← TOTAL - 1
  fin_mientras
  escribir('La suma de los', N, 'numeros es', SUMA)
fin
```

En el caso anterior, como la variable `TOTAL` se va decrementando y su valor inicial era `N`, cuando tome el valor `0`, significará que se han realizado `N` iteraciones, o, lo que es igual, se han sumado `N` números y el bucle se debe parar o terminar.

Ejemplo 5.2

Contar los números enteros positivos introducidos por teclado. Se consideran dos variables enteras `numero` *y* `contador` *(contará el número de enteros positivos). Se supone que se leen números positivos y se detiene el bucle cuando se lee un número negativo o cero.*

```
algoritmo cuenta_enteros
var
  entero : numero, contador
inicio
  contador ← 0
  leer(numero)
  mientras numero > 0 hacer
    leer(numero)
    contador ← contador + 1
  fin_mientras
  escribir('El numero de enteros positivos es', contador)
fin
```

inicio
contador ← 0
leer numero
mientras numero > 0
leer numero contador ← contador + 1
escribir 'numeros enteros', contador
fin

La secuencia de las acciones de este algoritmo se puede reflejar en el siguiente pseudocódigo:

Paso	Pseudocódigo	Significado
1	`contador ← 0`	inicializar contador a 0
2	**leer**(numero)	leer primer número
3	**mientras** numero > 0 **hacer**	comprobar si `numero > 0`. Si es así, continuar con el paso 4. Si no, continuar con el paso 7
4	`sumar 1 a contador`	incrementar `contador`
5	**leer**(numero)	leer siguiente `numero`
6	`regresar al paso 3`	evaluar y comprobar la expresión booleana
7	**escribir**(contador)	visualizar resultados

Obsérvese que los pasos 3 a 6 se ejecutarán mientras los números de entrada sean positivos. Cuando se lea –15 (después de 4 pasos), la expresión `numero > 0` produce un resultado falso y se transfiere el control a la acción **`escribir`** y el valor del contador será 4.

5.2.1. Ejecución de un bucle cero veces

Obsérvese que en una estructura **`mientras`** la primera cosa que sucede es la evaluación de la expresión booleana; si se evalúa *falsa* en ese punto, entonces del cuerpo del bucle nunca se ejecuta. Puede parecer *inútil* ejecutar el cuerpo del bucle *cero veces*, ya que no tendrá efecto en ningún valor o salida. Sin embargo, a veces es la acción deseada.

```
inicio
  n ← 5
  s ← 0
  mientras n <= 4 hacer
    leer(x)
    s ← s + x
  fin_mientras
fin
```

En el ejemplo anterior se aprecia que nunca se cumplirá la condición (expresión booleana `n <= 4`), por lo cual se ejecutará la acción **`fin`** y no se ejecutará ninguna acción del bucle.

EJEMPLO 5.3

El siguiente bucle no se ejecutará si el primer número leído es negativo o cero.

```
C ← 0
leer(numero)
mientras numero > 0 hacer
  C ← C + 1
  leer(numero)
fin_mientras
```

5.2.2. Bucles infinitos

Algunos bucles no exigen fin y otros no encuentran el fin por error en su diseño. Por ejemplo, un sistema de reservas de líneas aéreas puede repetir un bucle que permita al usuario añadir o borrar reservas. El programa y el bucle corren siempre, o al menos hasta que la computadora se apaga. En otras ocasiones un bucle no se termina nunca porque nunca se cumple la condición.

Un bucle que nunca se termina se denomina *bucle infinito* o *sin fin*. Los bucles sin fin no intencionados son perjudiciales para la programación y se deben evitar siempre.

Consideremos el siguiente bucle que visualiza el interés producido por un capital a las tasas de interés comprendidos en el rango desde 10 a 20 por 100.

```
leer(capital)
tasa ← 10
mientras tasa <> 20 hacer
  interes ← tasa*0.01*capital // tasa*capital/100=tasa*0.01*capital
  escribir('interes producido', interes)
  tasa ← tasa + 2
fin_mientras
escribir('continuacion')
```

Los sucesivos valores de la tasa serán `10`, `12`, `14`, `16`, `18`, `20`, de modo que al tomar *tasa* el valor `20` se detendrá el bucle y se escribirá el mensaje `'continuación'`. Supongamos que se cambia la línea última del bucle por

```
tasa ← tasa + 3
```

El problema es que el valor de la tasa salta ahora de `19` a `22` y nunca será igual a `20` (`10`, `13`, `16`, `19`, `22`,...). El bucle sería infinito, la expresión booleana para terminar el bucle será:

`tasa < 20` *o bien* `tasa <= 20`

Regla práctica

Las pruebas o test en las expresiones booleanas es conveniente que sean *mayor* o *menor que* en lugar de pruebas de *igualdad* o *desigualdad*. En el caso de la codificación en un lenguaje de programación, esta regla debe seguirse rígidamente en el caso de comparación de *números reales*, ya que como esos valores se almacenan en cantidades aproximadas las comparaciones de igualdad de valores reales normalmente plantean problemas. Siempre que realice comparaciones de números reales use las relaciones <, <=, > o >=.

5.2.3. Terminación de bucles con datos de entrada

Si su algoritmo o programa está leyendo una lista de valores con un bucle **`mientras`**, se debe incluir algún tipo de mecanismo para terminar el bucle. Existen cuatro métodos típicos para terminar un bucle de entrada:

1. preguntar antes de la iteración,
2. encabezar la lista de datos con su tamaño,
3. finalizar la lista con su valor de entrada,
4. agotar los datos de entrada.

Examinémoslos por orden. El primer método simplemente solicita con un mensaje al usuario si existen más entradas.

```
Suma ← 0
escribir('Existen mas numeros en la lista s/n')
leer(Resp)   //variable Resp, tipo carácter
mientras(Resp = 'S') o (Resp = 's') hacer
  escribir('numero')
  leer(N)
  Suma ← Suma + N
  escribir('Existen mas numeros (s/n)')
  leer(Resp)
fin_mientras
```

Este método a veces es aceptable y es muy útil en ciertas ocasiones, pero suele ser tedioso para listas grandes; en este caso, es preferible incluir una señal de parada. El método de conocer en la cabecera del bucle el tamaño o el número de iteraciones ya ha sido visto en ejemplos anteriores.

Tal vez el método más correcto para terminar un bucle que lee una lista de valores es con un *centinela*. Un *valor centinela* es un valor especial usado para indicar el final de una lista de datos. Por ejemplo, supongamos que se tienen unas calificaciones de unos tests (cada calificación comprendida entre 0 y 100); un valor centinela en esta lista puede ser –999, ya que nunca será una calificación válida y cuando aparezca este valor se terminará el bucle. Si la lista de datos son números positivos, un valor centinela puede ser un número negativo que indique el final de la lista. El siguiente ejemplo realiza la suma de todos los números positivos introducidos desde el terminal.

```
suma ← 0
leer(numero)
mientras numero >= 0 hacer
  suma ← suma+numero
  leer(numero)
fin_mientras
```

Obsérvese que el último número leído de la lista no se añade a la suma si es negativo, ya que se sale fuera del bucle. Si se desea sumar los números 1, 2, 3, 4 y 5 con el bucle anterior, el usuario debe introducir, por ejemplo:

```
1 2 3 4 5 -1
```

el valor final –1 se lee, pero no se añade a la suma. Nótese también que cuando se usa un valor centinela se invierte el orden de las instrucciones de lectura y suma con un valor centinela, éste debe leerse al final del bucle y se debe tener la instrucción `leer` al final del mismo.

El último método de agotamiento de datos de entrada es comprobar simplemente que no existen más datos de entrada. Este sistema suele depender del tipo de lenguaje; por ejemplo, Pascal puede detectar el final de una línea; en los archivos secuenciales se puede detectar el final físico del archivo (*eof*, **end of file**).

EJEMPLO 5.4

Considere los siguientes algoritmos. ¿Qué visualizará y cuántas veces se ejecuta el bucle?

1.
```
i ← 0
mientras i < 6 hacer
  escribir(i)
  i ← i + 1
fin_mientras
```

La salida es el valor de la variable de control *i* al principio de cada ejecución del cuerpo del bucle: 0, 1, 2, 3, 4 y 5. El bucle se ejecuta seis veces.

2.
```
i ← 0
mientras i < 6 hacer
  i ← i + 1
  escribir(i)
fin_mientras
```

La salida será entonces `1`, `2`, `3`, `4`, `5` y `6`. El cuerpo del bucle se ejecuta también seis veces. Obsérvese que cuando `i = 5`, la expresión *booleana* es verdadera y el cuerpo del bucle se ejecuta; con `i = 6` la sentencia **`escribir`** se ejecuta, pero a continuación se evalúa la expresión *booleana* y se termina el bucle.

EJEMPLO 5.5

Calcular la media de un conjunto de notas de alumnos. Pondremos un valor centinela de –99 que detecte el fin del bucle.

```
inicio
  total ← 0
  n ← 0     //numero de alumnos
  leer(nota) //la primera nota debe ser distinta de -99
  mientras nota <> -99 hacer
    total ← total + nota
    n ← n + 1
    leer (nota)
  fin_mientras
  media ← total / n
  escribir('La media es', media)
fin
```

Obsérvese que `total` y `n` se inicializan a cero antes de la instrucción **`mientras`**. Cuando el bucle termina, la variable `total` contiene la suma de todas las notas y, por consiguiente, `total/n`, siendo `n` el número de alumnos, será la media de la clase.

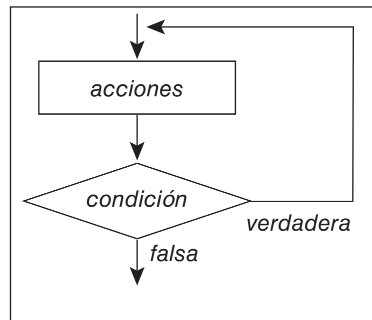
5.3. ESTRUCTURA `hacer-mientras ("do-while")`

El bucle `mientras` al igual que el bucle `desde` que se verá con posterioridad evalúan la expresión al comienzo del bucle de repetición; siempre se utilizan para crear bucle *pre-test*. Los bucles *pre-test* se denominan también bucles controlados por la entrada. En numerosas ocasiones se necesita que el conjunto de sentencias que componen el cuerpo del bucle se ejecuten al menos una vez sea cual sea el valor de la expresión o condición de evaluación. Estos bucles se denominan bucles post-test o bucles controlados por la salida. Un caso típico es el bucle **`hacer-mientras (do-while)`** existente en lenguajes como **C/C++**, **Java** o **C#**.

El bucle **`hacer-mientras`** es análogo al bucle **`mientras`** y el cuerpo del bucle se ejecuta una y otra vez mientras la condición (expresión *booleana*) sea verdadera. Existe, sin embargo, una gran diferencia y es que el cuerpo del bucle está encerrado entre las palabras reservadas hacer y mientras, de modo que las sentencias de dicho cuerpo se ejecutan, al menos una vez, antes de que se evalúe la expresión booleana. En otras palabras, el cuerpo del bucle siempre se ejecuta, al menos una vez, incluso aunque la expresión *booleana* sea falsa.

Regla

El bucle **`hacer-mientras`** se termina de ejecutar cuando el valor de la condición es falsa. La elección entre un bucle **`mientras`** y un bucle **`hacer-mientras`** depende del problema de cómputo a resolver. En la mayoría de los casos, la condición de entrada del bucle mientras es la elección correcta. Por ejemplo, si el bucle se utiliza para recorrer una lista de números (o una lista de cualquier tipo de objetos), la lista puede estar vacía, en cuyo caso las sentencias del bucle nunca se ejecutarán. Si se aplica un bucle **`hacer-mientras`** nos conduce a un código de errores.




```
hacer
  <acciones>
mientras (<expresión>)
```

a) Diagrama de flujo de una sentencia **hacer-mientras**

b) Pseudocódigo de una sentencia **hacer-mientras**

Figura 5.2. Estructura **hacer-mientras**: *a*) diagrama de flujo; *b*) pseudocódigo.

Al igual que en el caso del bucle **mientras** la sentencia en el interior del bucle puede ser simple o compuesta. Todas las sentencias en el interior del bucle se ejecutan al menos una vez antes de que la expresión o condición se evalúe. Entonces, si la expresión es **verdadera** (un valor distinto de cero, en C/C++) las sentencias del cuerpo del bucle se ejecutan una vez más. El proceso continúa hasta que la expresión evaluada toma el valor **falso** (valor cero en C/C++). El diagrama de control del flujo se ilustra en la Figura 5.2, donde se muestra el funcionamiento de la sentencia **hacer-mientras**. La Figura 5.3 representa un diagrama de sintaxis con notación BNF de la sentencia **hacer-mientras**.

```
Sentencia hacer-mientras::=
      hacer
        <cuerpo del bucle>
      mientras (<condición_del_bucle>)

donde

<cuerpo del bucle>::= <sentencia>
                  ::= <sentencia_compuesta>

<condición del bucle>::= <expresión booleana>
```

Nota: el cuerpo del bucle se repite mientras <condición del bucle> sea verdadero.

Figura 5.3. Diagrama de sintaxis de la sentencia **hacer-mientras**.

EJEMPLO 5.6

Obtener un algoritmo que lea un número (por ejemplo, 198) y obtenga el número inverso (por ejemplo, 891).

```
algoritmo invertirnummero
var
  entero: num, digitoSig
inicio
  num ← 198
  escribir ('Número: ← ', num)
  escribir ('Número en orden inverso: ')
  hacer
      digitoSig = num MOD 10
```

```
      escribir(digitoSig)
      num = num DIV 10
  mientras num > 0
fin
```

La salida de este programa se muestra a continuación:

```
Número:   198
Número en orden inverso:     891
```

Análisis del ejemplo anterior

Con cada iteración se obtiene el dígito más a la derecha, ya que es el resto de la división entera del valor del número (num) por 10. Así en la primera iteración digitoSig vale 8 ya que es el resto de la división entera de 198 entre 10 (cociente 19 y resto 8). Se visualiza el valor 8. A continuación se divide 198 entre 10 y se toma el cociente entero 19, que se asigna a la variable num.

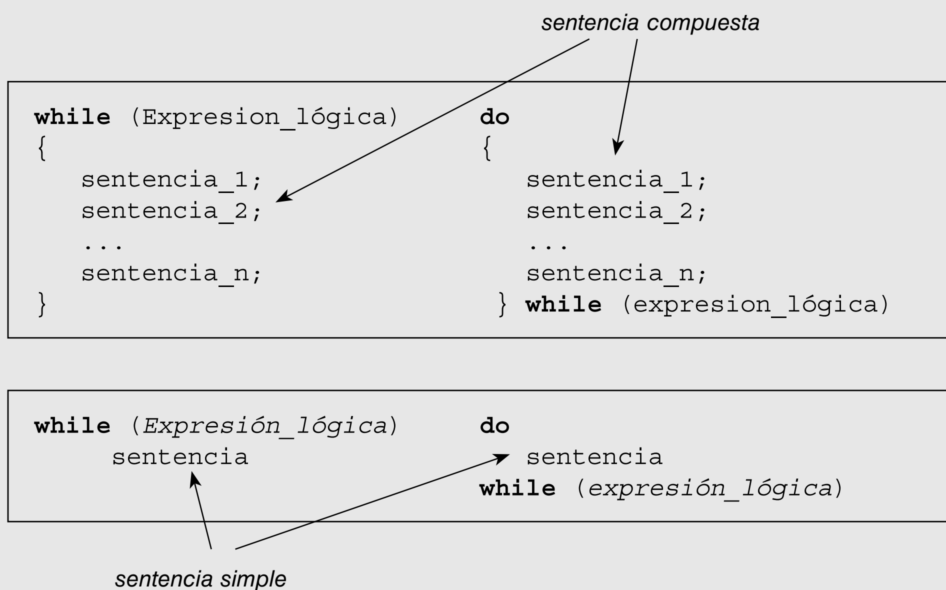
En la siguiente iteración se divide 19 por 10 (cociente entero 1, resto 9) y se visualiza, por consiguiente, el valor del resto, digitoSig, es decir el dígito 9; a continuación se divide 19 por 10 y se toma el cociente entero, es decir, 1.

En la tercera y última iteración se divide 1 por 10 y se toma el resto (digitoSig) que es el dígito 1. Se visualiza el dígito 1 a continuación de 89 y como resultado final aparece 891. A continuación se efectúa la división de nuevo por 10 y entonces el cociente entero es 0 que se asigna a num que al no ser ya mayor que cero hace que se termine el bucle y el algoritmo correspondiente.

5.4. DIFERENCIAS ENTRE `mientras` (`while`) Y `hacer-mientras` (`do-while`): UNA APLICACIÓN EN C++

Una sentencia **do-while** es similar a una sentencia **while**, excepto que el cuerpo del bucle se ejecuta siempre al menos una vez.

Sintaxis



Ejemplo 1

```
// cuenta a 10
int x = 0;
```

```
do
   cout << "X:" << x++;
while (x < 10)
```

Ejemplo 2

```
// imprimir letras minúsculas del alfabeto
char car = 'a';
do
{
   cout << car << '';
   car++;
}while (car <= 'z');
```

EJEMPLO 5.7

Visualizar las potencias de dos cuerpos cuyos valores estén en el rango 1 a 1.000.

```
// ejercicio con while
potencia = 1;
while (potencia < 1000)
{
   cout << potencia << endl;
   potencia *= 2
} // fin de while
```

```
// ejercicio con do-while
potencia = 1;
do
{
   cout << potencia << endl;
      potencia *= 2;
} while (potencia < 1000);
```

5.5. ESTRUCTURA `repetir ("repeat")`

Existen muchas situaciones en las que se desea que un bucle se ejecute al menos una vez *antes* de comprobar la condición de repetición. En la estructura **`mientras`** si el valor de la expresión booleana es inicialmente falso, el cuerpo del bucle no se ejecutará; por ello, se necesitan otros tipos de estructuras repetitivas.

La estructura **`repetir`** (**`repeat`**) se ejecuta hasta que se cumpla una condición determinada que se comprueba al final del bucle (Figura 5.4).

El bucle **`repetir-hasta_que`** se repite mientras el valor de la expresión *booleana* de la condición sea *falsa*, justo la opuesta de la sentencia **`mientras`**.

```
algoritmo repetir
var
  real : numero
  entero: contador
inicio
  contador ← 1
  repetir
    leer(numero)
    contador ← contador + 1
  hasta_que contador > 30
  escribir('Numeros leidos 30')
fin
```

En el ejemplo anterior el bucle se repite hasta que el valor de la variable `contador` exceda a 30, lo que sucederá después de 30 ejecuciones del cuerpo del bucle.

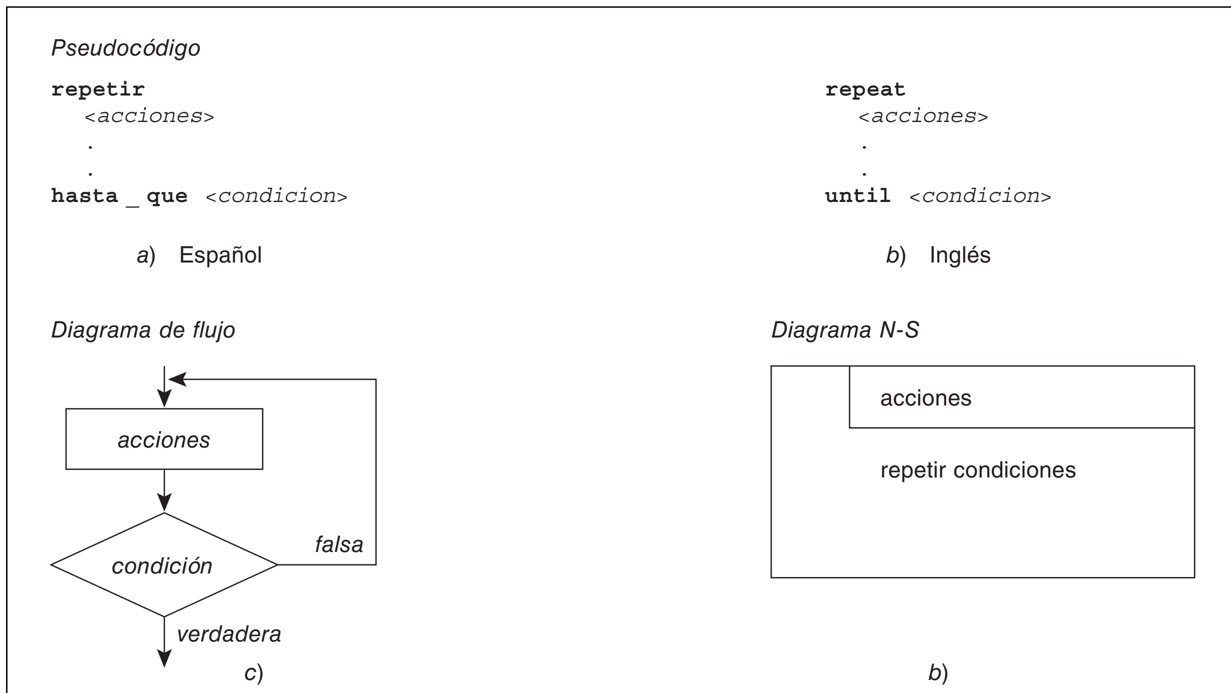


Figura 5.4. Estructura **repetir**: pseudocódigo, diagrama de flujo, diagrama N-S.

EJEMPLO 5.8

Desarrollar el algoritmo necesario para calcular el factorial de un número N que responda a la fórmula:

$N! = N * (N - 1) * (N - 2), ..., 3 * 2 * 1$

El algoritmo correspondiente es:

```
algoritmo factorial
var
  entero : I, N
  real : Factorial
inicio
  leer(N)          // N > = 1
  Factorial ← 1
  I ← 1
  repetir
      Factorial ← Factorial * I
      I ← I + 1
  hasta_que I = N + 1
  escribir('El factorial del numero', N, 'es', Factorial)
fin
```

Con una estructura **repetir** el cuerpo del bucle *se ejecuta siempre al menos una vez*. Cuando una instrucción **repetir** se ejecuta, lo primero que sucede es la ejecución del bucle y, a continuación, se evalúa la expresión *booleana* resultante de la condición. Si se evalúa como falsa, el cuerpo del bucle se repite y la expresión *booleana* se evalúa una vez. Después de cada iteración del cuerpo del bucle, la expresión *booleana* se evalúa; si es *verdadera*, el bucle termina y el programa sigue en la siguiente instrucción a **hasta_que**.

Diferencias de las estructuras mientras y repetir

- La estructura **`mientras`** termina cuando la condición es falsa, mientras que **`repetir`** termina cuando la condición es verdadera.
- En la estructura **`repetir`** el cuerpo del bucle se ejecuta siempre al menos una vez; por el contrario, **`mientras`** es más general y permite la posibilidad de que el bucle pueda no ser ejecutado. Para usar la estructura **`repetir`** debe estar seguro de que el cuerpo del bucle —bajo cualquier circunstancia— se repetirá al menos una vez.

EJEMPLO 5.9

Encontrar el entero positivo más pequeño (num) para el cual la suma 1+2+3+...+num es menor o igual que límite.

1. Introducir `limite`.
2. Inicializar `num` y `suma` a `0`.
3. Repetir las acciones siguientes hasta que `suma` > `limite`
   - incrementar `num` en `1`,
   - añadir `num` a `suma`.
4. Visualizar `num` y `suma`.

El pseudocódigo de este algoritmo es:

```
algoritmo mas_pequeño
var
  entero : num, limite, suma
inicio
  leer(limite)
  num ← 0
  suma ← 0
  repetir
    num ← num + 1
    suma ← suma + num
  hasta_que suma > limite
  escribir(num, suma)
fin
```

EJEMPLO 5.10

Escribir los números 1 a 100.

```
algoritmo uno_cien
var
  entero : num
inicio
  num ← 1
  repetir
    escribir(num)
    num ← num + 1
  hasta_que num > 100
fin
```

EJEMPLO 5.11

Es muy frecuente tener que realizar validación de entrada de datos en la mayoría de las aplicaciones. Este ejemplo detecta cualquier entrada comprendida entre 1 y 12, rechazando las restantes, ya que se trata de leer los números correspondientes a los meses del año.

```
algoritmo validar_mes
var
  entero : mes
inicio
  escribir('Introducir numero de mes')
  repetir
    leer(mes)
    si (mes < 1) o (mes > 12) entonces
      escribir('Valor entre 1 y 12')
    fin_si
  hasta_que (mes >=1) y (mes <= 12)
fin
```

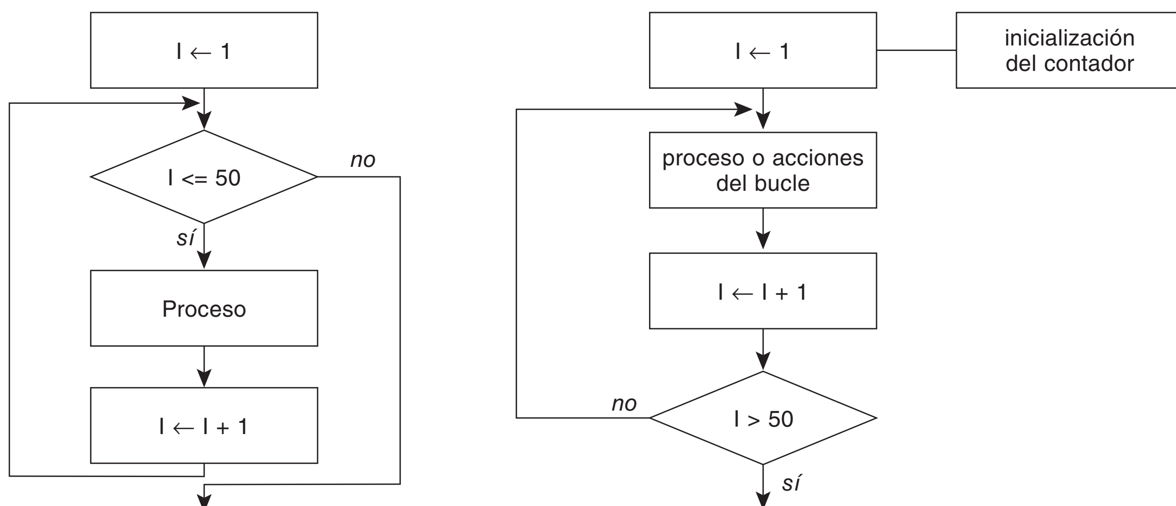
Este sistema es conocido como *interactivo* por entablar un *«diálogo imaginario»* entre la computadora y el programador que se produce «en tiempo real» entre ambas partes, es decir, «interactivo» con el usuario.

5.6. ESTRUCTURA `desde/para` ("`for`")

En muchas ocasiones se conoce de antemano el número de veces que se desean ejecutar las acciones de un bucle. En estos casos, en el que el número de iteraciones es fijo, se debe usar la estructura **`desde`** o **`para`** (**`for`**, en inglés). La estructura **`desde`** ejecuta las acciones del cuerpo del bucle un número especificado de veces y de modo automático controla el número de iteraciones o pasos a través del cuerpo del bucle. Las herramientas de programación de la estructura **`desde`** o **`para`** se muestran en la página siguiente junto a la Figura 5.5.

5.6.1. Otras representaciones de estructuras repetitivas `desde/para` (`for`)

Un bucle **`desde`** (**`for`**) se representa con los símbolos de proceso y de decisión mediante un contador. Así, por ejemplo, en el caso de un bucle de lectura de cincuenta números para tratar de calcular su suma:



Pseudocódigo estructura **desde**

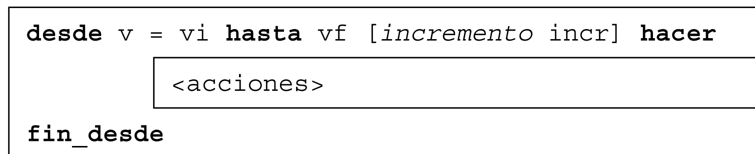
```
desde v ← vi hasta vf [incremento incr] hacer
  <acciones>
    .
    .
    .
fin_desde
v: variable indice
vi, vf: valores inicial y final de la variable
```

a) Modelo 1

```
para v ← vi hasta vf [incremento incr] hacer
  <acciones>
    .
    .
    .
fin_para
```

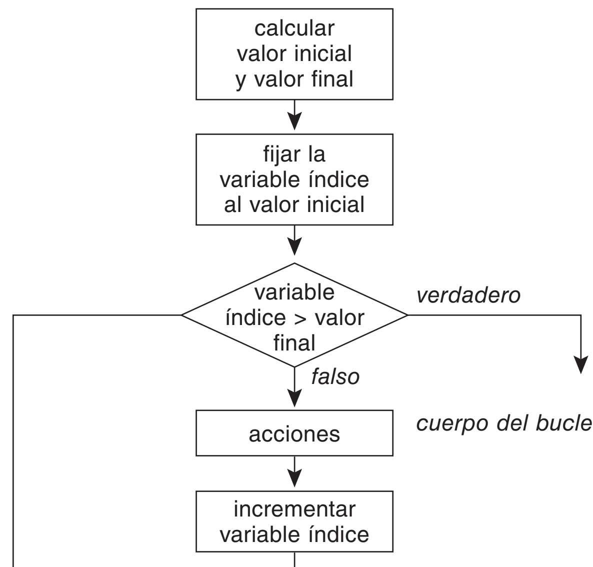
b) Modelo 2

Diagrama N-S, estructura **desde**



b) Modelo 3

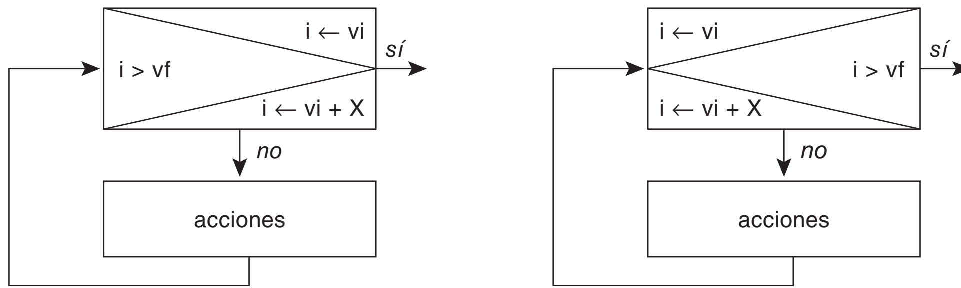
Diagrama de flujo, estructura, **desde**



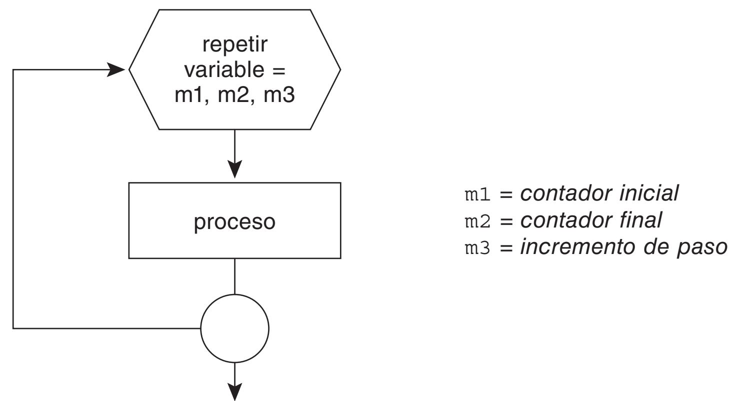
c) Modelo 4

Figura 5.5. Estructura **desde (for)**: *a*) pseudocódigo, *b*) diagrama N-S, *c*) diagrama de flujo.

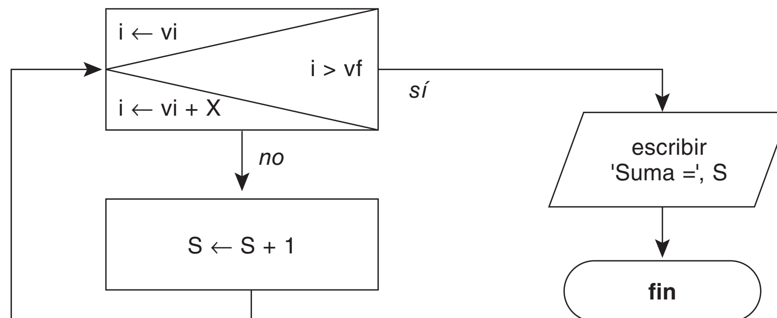
Es posible representar el bucle con símbolos propios



o bien mediante este otro símbolo



Como aplicación, *calcular la suma de los N primeros enteros.*



equivale a

```
algoritmo suma
var
  entero : I, N, S
inicio
  S ← 0,
  leer (N)
  desde I ← 1 hasta N hacer
    S ← S + I
  fin_desde
  escribir('Suma =', S)
fin
```

La estructura **desde** comienza con un valor inicial de la variable índice y las acciones especificadas se ejecutan, a menos que el valor inicial sea mayor que el valor final. La variable índice se incrementa en uno y si este nuevo valor no excede al final, se ejecutan de nuevo las acciones. Por consiguiente, las acciones específicas en el bucle se ejecutan para cada valor de la variable índice desde el valor inicial hasta el valor final con el incremento de uno en uno.

El incremento de la variable índice siempre es 1 si no se indica expresamente lo contrario. Dependiendo del tipo de lenguaje, es posible que el incremento sea distinto de uno, positivo o negativo. Así, por ejemplo, FORTRAN admite diferentes valores positivos o negativos del incremento, y Pascal sólo admite incrementos cuyo tamaño es la unidad: bien positivos, bien negativos. La variable índice o de control normalmente será de tipo entero y es normal emplear como nombres las letras `I`, `J`, `K`.

El formato de la estructura **desde** varía si se desea un incremento distinto a 1, bien positivo, bien negativo (decremento).

```
desde v ← vi hasta vf          inc paso hacer      {inc, incremento}
                               dec                 {dec, decremento}
  <acciones>
     .
     .
     .
fin_desde
```

Si el valor inicial de la variable índice es menor que el valor final, los incrementos deben ser positivos, ya que en caso contrario la secuencia de acciones no se ejecutaría. De igual modo, si el valor inicial es mayor que el valor final, el incremento debe ser en este caso negativo, es decir, *decremento*. Al incremento se le suele denominar también *paso* (“*step*”, en inglés). Es decir,

```
desde i ← 20 hasta 10 hacer
  <acciones>
fin_desde
```

no se ejecutaría, ya que el valor inicial es `20` y el valor final `10`, y como se supone un incremento positivo, de valor 1, se produciría un error. El pseudocódigo correcto debería ser

```
desde i ← 20 hasta 10 decremento 1 hacer
  <acciones>
fin_desde
```

5.6.2. Realización de una estructura `desde` con estructura `mientras`

Es posible, como ya se ha mencionado en apartados anteriores, sustituir una estructura **desde** por una **mientras**; en las líneas siguientes se indican dos formas para ello:

1. *Estructura* **desde** *con incrementos de la variable índice positivos.*

```
v ← vi
mientras v <= vf hacer
  <acciones>
  v ← v + incremento
fin_mientras
```

2. *Estructura* **desde** *con incrementos de la variable índice negativos.*

```
v ← vi
mientras v >= vf hacer
  <acciones>
  v ← v - decremento
fin_mientras
```

La estructura **desde** puede realizarse con algoritmos basados en estructura **mientras** y **repetir**, por lo que pueden ser intercambiables cuando así lo desee. Las estructuras equivalentes a **desde** son las siguientes:

a)
```
inicio
  i ← n
  mientras i > 0 hacer
    <acciones>
    i ← i - 1
  fin_mientras
fin
```

b)
```
inicio
  i ← 1
  mientras i <= n hacer
    <acciones>
    i ← i + 1
  fin_mientras
fin
```

c)
```
inicio
  i ← 0
  repetir
    <acciones>
    i ← i+1
  hasta_que i = n
fin
```

d)
```
inicio
  i ← 1
  repetir
    <acciones>
    i ← i+1
  hasta_que i > n
fin
```

e)
```
inicio
  i ← n + 1
  repetir
    <acciones>
    i ← i - 1
  hasta_que i = 1
fin
```

f)
```
inicio
  i ← n
  repetir
    <acciones>
    i ← i - 1
  hasta_que i < 1
fin
```

5.7. SALIDAS INTERNAS DE LOS BUCLES

Aunque no se incluye dentro de las estructuras básicas de la programación estructurada, en ocasiones es necesario disponer de una estructura repetitiva que permita la salida en un punto intermedio del bucle cuando se cumpla una condición. Esta nueva estructura sólo está disponible en algunos lenguajes de programación específicos; la denominaremos **iterar** para diferenciarlo de **repetir_hasta** ya conocida. Las salidas de bucles suelen ser válidas en estructuras **mientras, repetir** y **desde**.

El formato de la estructura es

```
iterar
  <acciones>
  si <condicion> entonces
    salir_bucle
  fin_si
  <acciones>
fin_iterar
```

En general, la instrucción **iterar** no produce un programa legible y comprensible como lo hacen **mientras** y **repetir**. La razón para esta ausencia de claridad es que la salida de un bucle ocurre en el medio del bucle, mientras que normalmente la salida del bucle es al principio o al final del mismo. Le recomendamos no recurra a esta opción —aunque la tenga su lenguaje— más que cuando no exista otra alternativa o disponga de la estructura **iterar** (*loop*).

EJEMPLO 5.12

Una aplicación de un posible uso de la instrucción **salir** *se puede dar cuando se incluyen mensajes de petición en el algoritmo para la introducción sucesiva de informaciones.*

Algoritmo 1

```
leer(informacion)
repetir
  procesar (informacion)
  leer(informacion)
hasta_que fin_de_lectura
```

Algoritmo 2

```
leer(informacion)
mientras_no fin_de_lectura
  procesar (informacion)
  leer(informacion)
fin_mientras
```

En los algoritmos anteriores cada entrada (lectura) de información va acompañada de su correspondiente proceso, pero la primera lectura está fuera del bucle. Se pueden incluir en el interior del bucle todas las lecturas de información si se posee una estructura **`salir (exit)`**. Un ejemplo de ello es la estructura siguiente:

```
iterar
  leer(informacion)
  si fin_de_lectura entonces
    salir_bucle
  fin_si
  procesar (informacion)
fin_iterar
```

5.8. SENTENCIAS DE SALTO `interrumpir` (`break`) y `continuar` (`continue`)

Las secciones siguientes examinan las sentencias de salto (*jump*) que se utilizan para influir en el flujo de ejecución durante la ejecución de una sentencia de bucle.

5.8.1. Sentencia `interrumpir` (`break`)

En ocasiones, los programadores desean terminar un bucle en un lugar determinado del cuerpo del bucle en vez de esperar que el bucle termine de modo natural por su entrada o por su salida. Un método de conseguir esta acción —siempre utilizada con precaución y con un control completo del bucle— es mediante la sentencia `interrumpir (break)` que se suele utilizar en la sentencia `según_sea` (`switch`).

La sentencia `interrumpir` se puede utilizar para terminar una sentencia de iteración y cuando se ejecuta produce que el flujo de control salte fuera a la siguiente sentencia inmediatamente a continuación de la sentencia de iteración. La sentencia `interrumpir` se puede colocar en el interior del cuerpo del bucle para implementar este efecto.

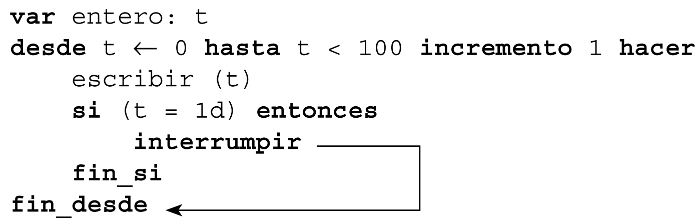
Sintaxis

```
interrumpir
sentencia_interrumpir::= interrumpir
```

EJEMPLO 5.13

```
hacer
    escribir ('Introduzca un número de identificiación')
    leer (numId)
    si (numId < 1000 o numId > 1999)entonces
      escribir ('Número no válido ')
      escribir ('Por favor, introduzca otro número')
    si-no
      interrumpir ─────────────────────┐
    fin_si                             │
mientras (expresión cuyo valor sea siempre verdadero) ◄┘
```

EJEMPLO 5.14



```
var entero: t
desde t ← 0 hasta t < 100 incremento 1 hacer
    escribir (t)
    si (t = 1d) entonces
        interrumpir
    fin_si
fin_desde
```

Regla

La sentencia **`interrumpir`** (**`break`**) se utiliza frecuentemente junto con una sentencia **`si`** (**`if`**) actuando como una condición interna del bucle.

5.8.2. Sentencia `continuar` (`continue`)

La sentencia **`continuar`** (**`continue`**) hace que el flujo de ejecución salte el resto de un cuerpo del bucle para continuar con el siguiente bucle o iteración. Esta característica suele ser útil en algunas circunstancias.

Sintaxis

```
continuar
Sentencia_continuar::= continuar
```

La sentencia **`continuar`** sólo se puede utilizar dentro de una *iteración de un bucle*. La sentencia `continuar` no interfiere con el número de veces que se repite el cuerpo del bucle como sucede con interrumpir, sino que simplemente influye en el flujo de control en cualquier iteración específica.

EJEMPLO 5.15

```
i = 0
desde i = 0 hasta 20 inc 1 hacer
    si (i mod 4 = 0) entonces
        continuar
    fin_si
    escribir (i, ', ')
fin_desde
```

Al ejecutar el bucle anterior se producen estos resultados

```
1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15, 17, 18, 19
```

Un análisis del algoritmo nos proporciona la razón de los resultados anteriores:

1. La variable `i` se declara igual a cero, como valor inicial.
2. El bucle `i` se incrementa en cada iteración en 1 hasta llegar a 21, momento en que se termina la ejecución del bucle.
3. Siempre que `i` es múltiplo de 4 (`i mod 4`) se ejecuta la sentencia **`continuar`** y salta el flujo del programa sobre el resto del cuerpo del bucle, se termina la iteración en curso y comienza una nueva iteración (en ese

caso no se escribe el valor de i). En consecuencia, no se visualiza el valor de i correspondiente (múltiplo de 4).

4. Como resultado final, se visualizan todos los números comprendidos entre 0 y 20, excepto los múltiplos de 4; es decir, 4, 8, 12, 16 y 20.

5.9. COMPARACIÓN DE BUCLES while, for Y do-while: UNA APLICACIÓN EN C++

C++ proporciona tres sentencias para el control de bucles: **while**, **for** y **do-while**. El bucle **while** se repite *mientras* su condición de repetición del bucle es verdadera; el bucle for se utiliza normalmente cuando el conteo esté implicado, o bien el control del bucle **for**, en donde el número de iteraciones requeridas se puede determinar al principio de la ejecución del bucle, o simplemente cuando existe una necesidad de seguir el número de veces que un suceso particular tiene lugar. El bucle **do-while** se ejecuta de un modo similar a **while** excepto que las sentencias del cuerpo del bucle se ejecutan siempre al menos una vez.

La Tabla 5.1 describe cuándo se usa cada uno de los tres bucles. En C++, el bucle **for** es el más frecuentemente utilizado de los tres. Es relativamente fácil reescribir un bucle **do-while** como un bucle while, insertando una asignación inicial de la variable condicional. Sin embargo, no todos los bucles while se pueden expresar de modo adecuado como bucles **do-while**, ya que un bucle **do-while** se ejecutará siempre al menos una vez y el bucle **while** puede no ejecutarse. Por esta razón, un bucle while suele preferirse a un bucle **do-while**, a menos que esté claro que se debe ejecutar una iteración como mínimo.

Tabla 5.1. Formatos de los bucles en C++

while	El uso más frecuente es cuando la repetición no está controlada por contador; el test de condición precede a cada repetición del bucle; el cuerpo del bucle puede no ser ejecutado. Se debe utilizar cuando se desea saltar el bucle si la condición es falsa.
for	Bucle de conteo cuando el número de repeticiones se conoce por anticipado y puede ser controlado por un contador; también es adecuado para bucles que implican control no contable del bucle con simples etapas de inicialización y de actualización; el test de la condición precede a la ejecución del cuerpo del bucle.
do-while	Es adecuada cuando se debe asegurar que al menos se ejecuta el bucle una vez.

Comparación de tres bucles

```
cuenta = valor_inicial;
while (cuenta < valor_parada)
{
   ...
   cuenta++;
} // fin de while

for(cuenta=valor_inicial; cuenta<valor_parada; cuenta++)
{
   ...
} // fin de for

cuenta = valor_inicial;
if (valor_inicial < valor_parada)
   do
   {
      ...
      cuenta++;
   } while (cuenta < valor_parada);
```

5.10. DISEÑO DE BUCLES (LAZOS)

El diseño de un bucle requiere tres partes:

1. El cuerpo del bucle.
2. Las sentencias de inicialización.
3. Las condiciones para la terminación del bucle.

5.10.1. Bucles para diseño de sumas y productos

Muchas tareas frecuentes implican la lectura de una lista de números y calculan su suma. Si se conoce cuántos números habrá, tal tarea se puede ejecutar fácilmente por el siguiente pseudocódigo. El valor de la variable `total` es el número de números que se suman. La suma se acumula en la variable `suma`.

```
suma ← 0;
repetir lo siguiente total veces:
   cin >> siguiente;
   suma ← suma + siguiente;
fin_bucle
```

Este código se implementa fácilmente con un bucle `for` en C++.

```
int suma = 0;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
   cin >> siguiente;
   suma = suma + siguiente;
}
```

Obsérvese que la variable `suma` se espera tome un valor cuando se ejecuta la siguiente sentencia

```
suma = suma + siguiente;
```

Dado que `suma` debe tener un valor la primera vez que la sentencia se ejecuta, `suma` debe estar inicializada a algún valor antes de que se ejecute el bucle. Con el objeto de determinar el valor correcto de inicialización de `suma` se debe pensar sobre qué sucede después de una iteración del bucle. Después de añadir el primer número, el valor de `suma` debe ser ese número. Esto es, la primera vez que se ejecute el bucle, el valor de `suma + siguiente` sea igual a `siguiente`. Para hacer esta operación *true* (verdadero), el valor de `suma` debe ser inicializado a 0.

Si en lugar de `suma`, se desea realizar productos de una lista de números, la técnica a utilizar es:

```
int producto = 1;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
   cin >> siguiente;
   producto = producto * siguiente;
}
```

La variable `producto` debe tener un valor inicial. No se debe suponer que todas las variables se deben inicializar a cero. Si `producto` se inicializara a cero, seguiría siendo cero después de que el bucle anterior se terminara.

5.10.2. Fin de un bucle

Existen cuatro métodos utilizados normalmente para terminar un bucle de entrada. Estos cuatro métodos son[2]:

[2] Estos métodos son descritos en Savitch, Walter, *Problem Solving with C++, The Object of Programming*, 2.ª edición, Reading, Massachussetts, Addison-Wesley, 1999.

1. *Lista encabezada por tamaño.*
2. *Preguntar antes de la iteración.*
3. *Lista terminada con un valor centinela.*
4. *Agotamiento de la entrada.*

Lista encabezada por el tamaño

Si su programa puede determinar el tamaño de una lista de entrada por anticipado, bien preguntando al usuario o por algún otro método, se puede utilizar un bucle "`repetir n veces`" para leer la entrada exactamente `n` veces, en donde `n` es el tamaño de la lista.

Preguntar antes de la iteración

El segundo método para la terminación de un bucle de entrada es preguntar, simplemente, al usuario, después de cada iteración del bucle, si el bucle debe ser o no iterado de nuevo. Por ejemplo:

```
suma = 0;
cout << "¿Existen números en la lista?:\n"
      << "teclee S para Sí, N para No y Final, Intro):";
char resp;
cin >> resp;
while ((resp == 'S')|| (resp == 's'))
{
   cout << "Introduzca un número: ";
   cin >> número;
   suma = suma + numero;
   cout << "¿Existen más números?:\n";
         << "S para Sí, N para No. Final con Intro:";
   cin >> resp;
}
```

Este método es muy tedioso para listas grandes de números. Cuando se lea una lista larga es preferible incluir una única señal de parada, como se incluye en el método siguiente.

Valor centinela

El método más práctico y eficiente para terminar un bucle que lee una lista de valores del teclado es mediante un valor centinela. Un **valor centinela** es aquél que es totalmente distinto de todos los valores posibles de la lista que se está leyendo y de este modo sirve para indicar el final de la lista. Un ejemplo típico se presenta cuando se lee una lista de números positivos; un número negativo se puede utilizar como un valor centinela para indicar el final de la lista.

```
// ejemplo de valor centinela (número negativo)
...
cout << "Introduzca una lista de enteros positivos" << endl;
     << "Termine la lista con un número negativo" << endl;
suma = 0;
cin >> numero;
while (numero >= 0)
{
   suma = suma + numero;
   cin >> numero;
}
cout << "La suma es: " << suma;
```

Si al ejecutar el segmento de programa anterior se introduce la lista

```
4       8       15      -99
```

el valor de la suma será `27`. Es decir, `-99`, último número de la entrada de datos no se añade a suma. `-99` es el último dato de la lista que actúa como centinela y no forma parte de la lista de entrada de números.

Agotamiento de la entrada

Cuando se leen entradas de un archivo, se puede utilizar un valor centinela. Aunque el método más frecuente es comprobar simplemente si todas las entradas del archivo se han leído y se alcanza el final del bucle cuando no hay más entradas a leer. Éste es el método usual en la lectura de archivos, que suele utilizar una marca al final de archivo, `eof`. En el capítulo de archivos se dedicará una atención especial a la lectura de archivos con una marca de final de archivo.

5.11. ESTRUCTURAS REPETITIVAS ANIDADAS

De igual forma que se pueden anidar o encajar estructuras de selección, es posible insertar un bucle dentro de otro. Las reglas para construir estructuras repetitivas anidadas son iguales en ambos casos: la estructura interna debe estar incluida totalmente dentro de la externa y no puede existir solapamiento. La representación gráfica se indica en la Figura 5.6.

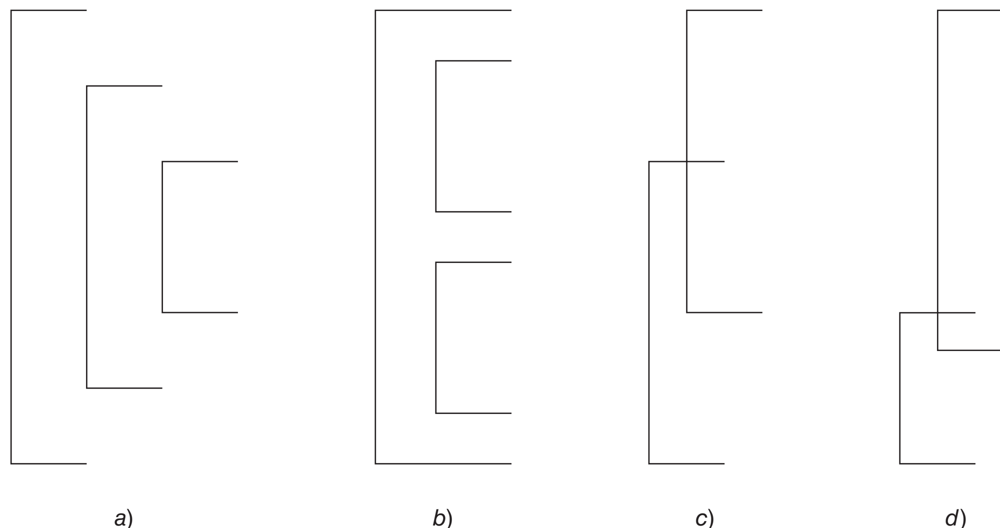


Figura 5.6. Bucles anidados: *a*) y *b*), correctos; *c*) y *d*), incorrectos.

Las variables índices o de control de los bucles toman valores de modo tal que por cada valor de la variable índice del ciclo externo se debe ejecutar totalmente el bucle interno. Es posible anidar cualquier tipo de estructura repetitiva con tal que cumpla las condiciones de la Figura 5.5.

EJEMPLO 5.16

Se conoce la población de cada una de las veinticinco ciudades más grandes de las ocho provincias de Andalucía y se desea identificar y visualizar la población de la ciudad más grande de cada provincia.

El problema consistirá, en primer lugar, en la obtención de la población mayor de cada provincia y realizar esta operación ocho veces, una para cada provincia.

1. Encontrar y visualizar la ciudad mayor de una provincia.
2. Repetir el paso 1 para cada una de las ocho provincias andaluzas.

El procedimiento para deducir la ciudad más grande de entre las veinticinco de una provincia se consigue creando una variable auxiliar `MAYOR` —inicialmente de valor 0— que se va comparando sucesivamente con los veinticinco valores de cada ciudad, de modo tal que, según el resultado de comparación, se intercambian valores de la ciudad por el de la variable `MAYOR`. El algoritmo correspondiente sería:

```
algoritmo CIUDADMAYOR
var
  entero : i      //contador de provincias
  entero : j      //contador de ciudades
  entero : MAYOR  //ciudad de mayor población
  entero : CIUDAD //población de la ciudad
inicio
  i ← 1
  mientras i <= 8 hacer
    MAYOR ← 0
    j ← 1
    mientras j <= 25 hacer
      leer(CIUDAD)
      si CIUDAD > MAYOR entonces
        MAYOR ← CIUDAD
      fin_si
      j ← j + 1
    fin_mientras
    escribir('La ciudad mayor es', MAYOR)
    i ← i + 1
  fin_mientras
fin
```

EJEMPLO 5.17

Calcular el factorial de n números leídos del terminal.

El problema consistirá en realizar una estructura repetitiva de *n* iteraciones del algoritmo del problema ya conocido del cálculo del factorial de un entero.

```
algoritmo factorial2
var
  entero : i, NUMERO, n
  real : FACTORIAL
inicio
  {lectura de la cantidad de números}
  leer(n)
  desde i ← 1 hasta n hacer
    leer(NUMERO)
    FACTORIAL ← 1
    desde j ← 1 hasta NUMERO hacer
      FACTORIAL ← FACTORIAL * j
    fin_desde
    escribir('El factorial del numero', NUMERO, 'es', FACTORIAL)
  fin_desde
fin
```

Ejemplo 5.18

Imprimir todos los número primos entre 2 y 100 inclusive.

```
algoritmo Primos
var entero : i, divisor
    logico : primo
inicio
  desde i ← hasta 100 hacer
    primo ← verdad
    divisor ← 2
    mientras (divisor <= raiz2(i)) y primo hacer
      si i mod divisor = 0 entonces
        primo ← falso
      si_no
        divisor ← divisor + 1
      fin_si
    fin_mientras
    si primo entonces
      escribir(i,' ')
    fin_si
  fin_desde
fin
```

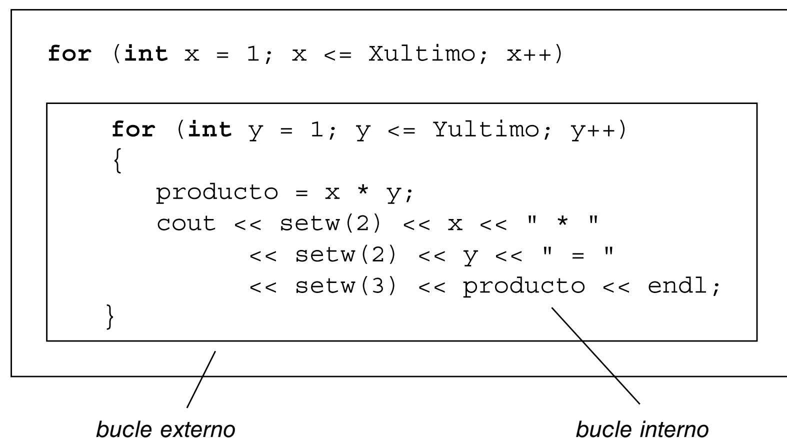
5.11.1. Bucles (lazos) anidados: una aplicación en C++

Es posible *anidar* bucles. Los bucles anidados constan de un bucle externo con uno o más bucles internos. Cada vez que se repite el bucle externo, los bucles internos se repiten, se reevalúan los componentes de control y se ejecutan todas las iteraciones requeridas.

Ejemplo 5.19

El segmento de programa siguiente visualiza una tabla de multiplicación por cálculo y visualización de productos de la forma `x * y` *para cada* `x` *en el rango de* `1` *a* `Xultimo` *y desde cada* `y` *en el rango* `1` *a* `Yultimo` *(donde* `Xultimo`*, e* `Yultimo` *son enteros prefijados). La tabla que se desea obtener es*

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
        ...
```



El bucle que tiene `x` como variable de control se denomina **bucle externo** y el bucle que tiene `y` como variable de control se denomina **bucle interno**.

EJEMPLO 5.20

```
// Aplicación de bucles anidados

#include <iostream>
#include <iomanip.h>      // necesario para cin y cout
using namespace std;      // necesario para setw

void main()
{
   // cabecera de impresión
   cout << setw(12) << " i " << setw(6) << " j " << endl;

   for (int i = 0; i < 4; i++)
   {
      cout << "Externo " << setw(7) << i << endl;
      for (int j = 0; j < i; j++)
         cout << "Interno " << setw(10) << j << endl;
   } // fin del bucle externo
}
```

La salida del programa es

```
              i    j
Externo       0
Externo       1
   Interno         0
Externo       2
   Interno         0
   Interno         1
Externo       3
   Interno         0
   Interno         1
   Interno         2
```

EJERCICIO 5.1

Escribir un programa que visualice un triángulo isósceles.

```
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
```

El triángulo isósceles se realiza mediante un bucle externo y dos bucles internos. Cada vez que se repite el bucle externo se ejecutan los dos bucles internos. El bucle externo se repite cinco veces (cinco filas); el número de repeticiones realizadas por los bucles internos se basan en el valor de la variable `fila`. El primer bucle interno visualiza los espacios en blanco no significativos; el segundo bucle interno visualiza uno o más asteriscos.

```
// archivo triángulo.cpp
#include <iostream>
using namespace std;
```

```
void main()
{
   // datos locales...
   const int num_lineas = 5;
   const char blanco = '';
   const char asterisco = '*';

   // comienzo de una nueva línea
   cout << endl;

   // dibujar cada línea: bucle externo
   for (int fila = 1; fila <= num_lineas; fila++)
   {
      // imprimir espacios en blanco: primer bucle interno
      for (int blancos = num_lineas –fila; blancos > 0;
         blancos--)
         cout << blanco;

      for (int cuenta_as = 1; cuenta_as < 2 * fila;
           cuenta_as ++)
           cout << asterisco;

      // terminar línea
      cout << endl;
   } // fin del bucle externo
}
```

El bucle externo se repite cinco veces, uno por línea o fila; el número de repeticiones ejecutadas por los bucles internos se basa en el valor de `fila`. La primera fila consta de un asterisco y cuatro blancos, la fila 2 consta de tres blancos y tres asteriscos, y así sucesivamente; la fila 5 tendrá 9 asteriscos ($2 \times 5 - 1$).

EJERCICIO 5.2

Ejecutar y visualizar el programa siguiente que imprime una tabla de m filas por n columnas y un carácter prefijado.

```
1:    //Listado
2:    //ilustra bucles for anidados
3:
4:    int main()
5:    {
6:       int filas, columnas;
7:       char elCar;
8:       cout << "¿Cuántas filas?";
9:       cin >> filas;
10:      cout << "¿Cuántas columnas?";
11:      cin >> columnas;
12:      cout << "¿Qué carácter?";
13:      cin >> elCar;
14:      for (int i = 0; i < filas; i++)
15:      {
16:         for (int j = 0; j < columnas; j++)
17:            cout << elCar;
18:      cout << "\n";
19:      }
20:      return 0;
21:   }
```

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

5.1. *Calcular el factorial de un número N utilizando la estructura* **desde**.

Solución

Recordemos que factorial de `N` responde a la fórmula

```
N! = N · (N – 1) · (N – 2) · (N – 3) · ... · 3 · 2 · 1
```

El algoritmo **`desde`** supone conocer el número de iteraciones:

```
var
  entero : I, N
  real : FACTORIAL
inicio
  leer(N)
  FACTORIAL ← 1
  desde I ← 1 hasta N hacer
    FACTORIAL ← FACTORIAL * I
  fin_desde
  escribir('El factorial de', N, 'es', FACTORIAL)
fin
```

5.2. *Imprimir las treinta primeras potencias de 4, es decir, 4 elevado a 1, 4 elevado a 2, etc.*

Solución

```
algoritmo potencias4
var
  entero : n
inicio
  desde n ← 1 hasta 30 hacer
    escribir(4 ^ n)
  fin_desde
fin
```

5.3. *Calcular la suma de los n primeros números enteros utilizando la estructura* **`desde`**.

Solución

$S = 1 + 2 + 3 + ... + n$

El pseudocódigo correspondiente es

```
algoritmo sumaNenteros
var
  entero : i, n
  real : suma
inicio
  leer(n)
  suma ← 0
  desde i ← 1 hasta n hacer
    suma ← suma + 1
  fin_desde
  {escribir el resultado de suma}
  escribir(suma)
fin
```

5.4. *Diseñar el algoritmo para imprimir la suma de los números impares menores o iguales que n.*

Solución

Los números impares son 1, 3, 5, 7, ..., *n*. El pseudocódigo es

```
algoritmo sumaimparesmenores
var
  entero : i, n
  real : S
inicio
  S ← 0
  leer(n)
  desde i ← 1 hasta n inc 2 hacer
    S ← S + i
  fin_desde
  escribir(S)
fin
```

5.5. *Dados dos números enteros, realizar el algoritmo que calcule su cociente y su resto.*

Solución

Sean los números `M` y `N`. El método para obtener el cociente y el resto es por restas sucesivas; el método sería restar sucesivamente el divisor del dividendo hasta obtener un resultado menor que el divisor, que será el resto de la división; el número de restas efectuadas será el cociente

50	13	50 – 13 = 37	C = 1
11	3	37 – 13 = 24	C = 2
		24 – 13 = 11	C = 3

Como 11 es menor que el divisor 13, se terminarán las restas sucesivas y entonces 11 será el resto y 3 (número de restas) el cociente. Por consiguiente, el algoritmo será el siguiente:

```
algoritmo cociente
var
  entero : M, N, Q, R
inicio
  leer(M, N)      {M, dividendo / N, divisor}
  R ← M
  Q ← 0
  repetir
    R ← R - N
    Q ← Q + 1
  hasta_que R < N
  escribir('dividendo',M, 'divisor',N, 'cociente',Q, 'resto',R)
fin
```

5.6. *Realizar el algoritmo para obtener la suma de los números pares hasta 1.000 inclusive.*

Solución

Método 1

S = 2 + 4 + 6 + 8 + ... + 1.000

```
algoritmo sumapares
var
  real : NUMERO, SUMA
```

```
inicio
  SUMA ← 2
  NUMERO ← 4
  mientras NUMERO <= 1.000 hacer
    SUMA ← SUMA + NUMERO
    NUMERO ← NUMERO + 2
  fin_mientras
fin
```

Método 2

```
{idéntica cabecera y declaraciones}
inicio
  SUMA ← 2
  NUMERO ← 4
  repetir
    SUMA ← SUMA + NUMERO
    NUMERO ← NUMERO + 2
  hasta_que NUMERO > 1000
fin
```

5.7. *Buscar y escribir la primera vocal leída del teclado. (Se supone que se leen, uno a uno, caracteres desde el teclado.)*

Solución

```
algoritmo buscar_vocal
var
  carácter: p
inicio
  repetir
    leer(p)
  hasta_que p = 'a' o p = 'e' o p = 'i' o p = 'o' o p = 'u'
  escribir('Primero', p)
fin
```

5.8. *Se desea leer de una consola a una serie de números hasta obtener un número inferior a 100.*

Solución

```
algoritmo menor_100
var
  real : numero
inicio
  repetir
    escribir('Teclear un numero')
    leer(numero)
  hasta_que numero < 100
  escribir('El numero es', numero)
fin
```

5.9. *Escribir un algoritmo que permita escribir en una pantalla la frase '¿Desea continuar?* `S/N`*' hasta que la respuesta sea* `'S'` *o* `'N'`.

Solución

```
algoritmo SN
var
  carácter : respuesta
```

```
inicio
  repetir
    escribir('Desea continuar S/N')
    leer(respuesta)
  hasta_que(respuesta = 'S') o (respuesta = 'N')
fin
```

5.10. *Leer sucesivamente números del teclado hasta que aparezca un número comprendido entre 1 y 5.*

Solución

```
algoritmo numero1_5
var
  entero : numero
inicio
  repetir
    escribir('Numero comprendido entre 1 y 5')
    leer(numero)
  hasta_que(numero >= 1) y (numero <= 5)
  escribir('Numero encontrado', numero)
fin
```

5.11. *Calcular el factorial de un número n con métodos diferentes al Ejercicio 5.1.*

Solución

$n! = n \times (n - 1) \times (n - 2) \times ... \times 3 \times 2 \times 1$

es decir,

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
$4! = 4 \times 3 \times 2 \times 1 = 24$
$3! = 3 \times 2 \times 1 = 6$
$2! = 2 \times 1 = 2$
$1! = 1 = 1$

Para codificar estas operaciones basta pensar que

$$(n + 1)! = (n + 1) \times n \times \underbrace{(n - 1) \times (n - 2) \times ... \times 3 \times 2 \times 1}_{n!}$$

$$(n + 1)! = (n + 1) \times n!$$

Por consiguiente, para calcular el factorial `FACTORIAL` de un número, necesitaremos un contador `i` que cuente de uno en uno y aplicar la fórmula

```
FACTORIAL = FACTORIAL * i
```

inicializando los valores de `FACTORIAL` e `i` a 1 y realizando un bucle en el que `i` se incremente en 1 a cada iteración, es decir,

Algoritmo 1 de Factorial de n

```
FACTORIAL ← 1
i ← 1
repetir
  FACTORIAL ← FACTORIAL * i
  i ← i + 1
hasta_que i = n + 1
```

Algoritmo 2 de Factorial de n

```
FACTORIAL ← 1
i ← 1
repetir
  FACTORIAL ← FACTORIAL * (i + 1)
  i ← i + 1
hasta_que i = n
```

Algoritmo 3 de Factorial de n

```
FACTORIAL ← 1
i ← 1
repetir
  FACTORIAL ← FACTORIAL * (i + 1)
  i ← i + 1
hasta_que i > n - 1
```

Algoritmo 4 de factorial de n

```
FACTORIAL ← 1
i ← 1
desde i ← 1 hasta n - 1 hacer
FACTORIAL ← FACTORIAL * (i + 1)
fin_desde
```

Un algoritmo completo con lectura del número `n` por teclado podría ser el siguiente:

```
algoritmo factorial
var
  entero : i, n
  real : f
inicio
  f ← 1
  i ← 1
  leer(n)
  repetir
    f ← f * i
    i ← i + 1
  hasta_que i = n + 1
  escribir('Factorial de', n, 'es', f)
fin
```

5.12. *Calcular el valor máximo de una serie de 100 números.*

Solución

Para resolver este problema necesitaremos un contador que cuente de 1 a 100 para contabilizar los sucesivos números. El algoritmo que calcula el valor máximo será repetido y partiremos considerando que el primer número leído es el valor máximo, por lo cual se realizará una primera asignación del número 1 a la variable *máximo*.

La siguiente acción del algoritmo será realizar comparaciones sucesivas:

- leer un nuevo número
- compararlo con el valor máximo
- si es *inferior*, implica que el valor máximo es el antiguo;
- si es *superior*, implica que el valor máximo es el recientemente leído, por lo que éste se convertirá en *máximo* mediante una asignación;
- repetir las acciones anteriores hasta que $n = 100$.

```
algoritmo maximo
var
  entero : n, numero, maximo
inicio
  leer(numero)
  n ← 1
  maximo ← numero
  repetir
    n ← n+1
    leer(numero)
    si numero > maximo entonces
      maximo ← numero
    fin_si
  hasta_que n = 100
  escribir('Numero mayor o maximo', maximo)
fin
```

Otras soluciones

1.
```
algoritmo otromaximo
  var
    entero : n, numero, maximo
  inicio
    leer(numero)
    maximo ← numero
    n ← 2
    repetir
      n ← n + 1
      leer(numero)
      si numero > maximo entonces
        maximo ← numero
      fin_si
    hasta_que n > 100
    escribir('Numero mayor o maximo', maximo)
  fin
```

2.
```
algoritmo otromaximo
  var
    entero : n, numero, maximo
  inicio
    leer(numero)
    maximo ← numero
    para n = 2 hasta 100 hacer        //pseudocódigo sustituto de desde
      leer(numero)
      si numero > maximo entonces
        maximo ← numero
      fin_si
   fin_para
    escribir('Maximo,', maximo)
  fin
```

NOTA: Los programas anteriores suponen que los números pueden ser positivos o negativos; si se desea comparar sólo números positivos, los programas correspondientes serían:

1.
```
algoritmo otromaximo
  var
    entero : n, numero, maximo
  inicio
    n ← 0
    maximo ← 0
```

```
    repetir
      leer(numero)
      n = n + 1
      si numero > maximo entonces
        maximo ← numero
      fin_si
    hasta_que n = 100
    escribir('Maximo numero', maximo)
  fin
```

2.
```
algoritmo otromaximo
  var
    entero : n, numero, maximo
  inicio
    n ← 0
    maximo ← 0
    para N ← 1 hasta 100 hacer
      leer(numero)
      si numero > maximo entonces
        maximo ← numero
      fin_si
    fin_para
    escribir('Maximo numero =', maximo)
  fin
```

5.13. *Bucles anidados. Las estructuras de control tipo bucles pueden anidarse internamente, es decir, se puede situar un bucle en el interior de otro bucle.*

Solución

La anidación puede ser:

- bucles **repetir** dentro de bucles **repetir**,
- bucles **para (desde)** dentro de bucles **repetir**,
- etc.

Ejemplo 1. Bucle **para** en el interior de un bucle **repetir-hasta_que**

```
repetir
  leer(n)
  para i ← 1 hasta hacer 5
    escribir(n * n)
  fin_para
hasta_que n = 0
escribir('Fin')
```

Si ejecutamos estas instrucciones, se obtendrá para:

```
n = 5       resultados       25
                             25
                             25
                             25
                             25

n = 2       resultados       4
                             4
                             4
                             4
                             4
```

```
n = 0       resultados       0
                             0
                             0
                             0
                             0
            fin
```

Ejemplo 2. Bucles anidados **para (desde)**

```
para i ← 1 hasta 3 hacer
  para j ← 1 hasta 10 hacer
    escribir(i, 'por', j, '=', i*j)
  fin_para
fin_para
```

Los valores sucesivos de `i, j, i*j`, serán

```
i = 1  j = 1     i * j = 1 x 1 = 1
       j = 2     i * j = 1 x 2 = 2
       j = 3     i * j = 1 x 3 = 3
       j = 4     i * j = 1 x 4 = 4
..................................
       j = 10    i * j = 1 x 10 = 10

i = 2  j = 1     i * j = 2 x 1 = 2
       j = 2     i * j = 2 x 2 = 4
       j = 3     i * j = 2 x 3 = 6
       j = 4     i * j = 2 x 4 = 8
..................................
       j = 10    i * j = 2 x 10 = 20

i = 3  j = 1     i * j = 3 x 1 = 3
       j = 2     i * j = 3 x 2 = 6
       j = 3     i * j = 3 x 3 = 9
       j = 4     i * j = 3 x 4 = 12
..................................
       j = 10    i * j = 3 x 10 = 30
```

Es decir, se obtendrá la tabla de multiplicar de 1, 2 y 3.

Ejemplo 3. La codificación completa para obtener la popular tabla de multiplicar de los escolares sería la siguiente:

```
algoritmo Tabla_de_multiplicar
var
  entero : i, j, producto
inicio
  para ← 1 hasta 9 hacer
    escribir('Tabla del', i)
    para j ← 1 hasta 10 hacer
      producto ← i * j
      escribir(i, 'por', j, '=', producto)
    fin_para
  fin_para
fin
```

5.14. *Se dispone de una lista de* `N` *números. Se desea calcular el valor del número mayor.*

Solución

```
algoritmo
var
  entero : I
```

```
  real : NUM, MAX
  entero : N
inicio
  leer(N)
  leer(NUM)
  MAX ← NUM
  desde I ← 2 hasta 100 hacer
    leer(NUM)
    si NUM > MAX entonces
      MAX ← NUM
    fin_si
  fin_desde
fin
```

5.15. *Determinar simultáneamente los valores máximo y mínimo de una lista de 100 números.*

Solución

```
algoritmo max_min
var
  entero : I
  real : MAX, MIN, NUMERO
inicio
  leer(NUMERO)
  MAX ← NUMERO
  MIN ← NUMERO
  desde I ← 2 hasta 100 hacer
    leer(NUMERO)
    si NUMERO > MAX entonces
      MAX ← NUMERO
    si_no
      si NUMERO < MIN entonces
        MIN ← NUMERO
      fin_si
    fin_si
  fin_desde
  escribir('Maximo', MAX, 'Minimo', MIN)
fin
```

5.16. *Se dispone de un cierto número de valores de los cuales el último es el 999 y se desea determinar el valor máximo de las medias correspondientes a parejas de valores sucesivos.*

Solución

```
algoritmo media_parejas
var
  entero : N1, N2
  real : M, MAX
inicio
  leer(N1, N2)
  MAX ← (N1 + N2)/ 2
  mientras (N2 <> 999) o (N1 <> 999) hacer
    leer(N1, N2)
    M ← (N1 + N2)/2
    si M > MAX entonces
      MAX ← M
    fin_si
  fin_mientras
  escribir('Media maxima =' MAX)
fin
```

5.17. *Detección de entradas numéricas —enteros— erróneas.*

Solución

Análisis

Este algoritmo es una aplicación sencilla de «interruptor». Se sitúa el valor inicial del interruptor `(SW = 0)` antes de recibir la entrada de datos.

La detección de números no enteros se realizará con una estructura repetitiva **`mientras`** que se realizará si `SW = 0`. La instrucción que detecta si un número leído desde el dispositivo de entradas es entero:

```
leer(N)
```

Realizará la comparación de `N` y parte entera de `N`:

- si son iguales, `N` es entero,
- si son diferentes, `N` no es entero.

Un método para calcular la parte entera es utilizar la función estándar **`ent (int)`** existente en muchos lenguajes de programación.

Pseudocódigo

```
algoritmo error
var
  entero : SW
  real : N
inicio
  SW ← 0
  mientras SW = 0 hacer
    leer(N)
    si N <> ent(N) entonces
      escribir('Dato no valido')
      escribir('Ejecute nuevamente')
      SW ← 1
    si_no
      escribir('Correcto', N, 'es entero')
    fin_si
  fin_mientras
fin
```

5.18. *Calcular el factorial de un número dado (otro nuevo método).*

Solución

Análisis

El factorial de un número N (N!) es el conjunto de productos sucesivos siguientes:

$N! = N * (N - 1) * (N - 2) * (N - 3) * ... * 3 * 2 * 1$

Los factoriales de los primeros números son:

1! = 1
2! = 2 * 1 = 2 * 1!
3! = 3 * 2 * 1 = 3 * 2!
4! = 4 * 3 * 2 * 1 = 4 * 3!
.
.
.
$N! = N * (N - 1) * (N - 2) * ... * 2 * 1 = N * (N - 1)!$

Los cálculos anteriores significan que el factorial de un número se obtiene con el producto del número *N* por el factorial de $(N - 1)!$

Como comienzan los productos en 1, un sistema de cálculo puede ser asignar a la variable *factorial* el valor 1. Se necesita otra variable I que tome los valores sucesivos de 1 a *N* para poder ir efectuando los productos sucesivos. Dado que en los números negativos no se puede definir el factorial, se deberá incluir en el algoritmo una condición para verificación de error, caso de que se introduzcan números negativos desde el terminal de entrada ($N < 0$).

La solución del problema se realiza por dos métodos:

1. Con la estructura **repetir** (**repeat**).
2. Con la estructura **desde** (**for**).

Pseudocódigo

Método 1 (estructura **repetir**)

```
algoritmo FACTORIAL
var
  entero : I, N
  real : factorial
inicio
  repetir
    leer(N)
  hasta_que N > 0
  factorial ← 1
  I ← 1
  repetir
    factorial ← factorial * I
    I ← I + 1
  hasta_que I = N + 1
  escribir(factorial)
fin
```

Método 2 (estructura **desde**)

```
algoritmo FACTORIAL
var
  entero : K, N
  real : factorial
inicio
  leer(N)
  si n < 0 entonces
    escribir('El numero sera positivo')
  si_no
    factorial ← 1
    si N > 1 entonces
      desde K ← 2 hasta N hacer
        factorial ← factorial * K
      fin_desde
    fin_si
    escribir('Factorial de', N, '=', factorial)
  fin_si
fin
```

5.19. *Se tienen las calificaciones de los alumnos de un curso de informática correspondiente a las asignaturas BASIC, Pascal, FORTRAN. Diseñar un algoritmo que calcule la media de cada alumno.*

Solución

Análisis

Asignaturas: `C`
`Pascal`
`FORTRAN`

Media: $\frac{(C + Pascal + FORTRAN)}{3}$

Se desconoce el número de alumnos N de la clase; por consiguiente, se utilizará una marca final del archivo `ALUMNOS`. La marca final es '***' y se asignará a la variable *nombre*.

Pseudocódigo

```
algoritmo media
var
  cadena : nombre
  real : media
  real : BASIC, Pascal, FORTRAN
inicio
  {entrada datos de alumnos}
  leer(nombre)
  mientras nombre <> '***' hacer
    leer(BASIC, Pascal, FORTRAN)
    media ← (BASIC + Pascal + FORTRAN) / 3
    escribir(nombre, media)
    leer(nombre)
  fin_mientras
fin
```

CONCEPTOS CLAVE

- bucle.
- bucle anidado.
- bucle infinito.
- bucle sin fin.
- centinela.
- iteración.
- pasada.
- programación estructurada.
- sentencia **`continuar`**.
- sentencia **`ir_a`**.
- sentencia **`interrumpir`**.
- sentencia **`de repetición`**.
- sentencia **`desde`**.
- sentencia **`hacer-mientras`**.
- sentencia **`mientras`**.
- sentencia **`nula`**.
- sentencia **`repetir-hasta_que`**.

RESUMEN

Este capítulo examina los aspectos fundamentales de la iteración y el modo de implementar esta herramienta de programación esencial utilizando los cuatro tipos fundamentales de sentencias de iteración: **`mientras`**, **`hacer-mientras`**, **`repetir-hasta_que`** y **`desde (para)`**.

1. Una sección de código repetitivo se conoce como bucle. El bucle se controla por una sentencia de repetición que comprueba una condición para determinar si el código se ejecutará. Cada pasada a través del bucle se conoce como una iteración o repetición. Si la condición se evalúa como falsa en la primera iteración, el bucle se termina y se habrán ejecutado las sentencias del cuerpo del bucle una sola vez. Si la condición se evalúa como verdadera la primera vez que se ejecuta el bucle, será necesa-

rio que se modifiquen alguna/s sentencias del interior del bucle para que se altere la condición correspondiente.

2. Existen cuatro tipos básicos de bucles: **`mientras`**, **`hacer-mientras`**, **`repetir-hasta_que`** y **`desde`**. Los bucles mientras y desde son bucles controlados por la entrada o pretest. En este tipo de bucles, la condición comprobada se evalúa al principio del bucle, que requiere que la condición sea comprobada explícitamente antes de la entrada al bucle. Si la condición es verdadera, las repeticiones del bucle comienzan; en caso contrario, no se introduce al bucle. Las iteraciones continúan mientras que la condición permanece verdadera. En la mayoría de los lenguajes, estas sentencias se construyen utilizando, respectivamente, las sentencias **`while`** y **`for`**. Los bucles **`hacer-mientras`** y **`repetir-hasta_que`** son bucles controlados *por salida* o *posttest*, en los que la condición a evaluar se comprueba al final del bucle. El cuerpo del bucle se ejecuta siempre al menos una vez. El bucle hacer-mientras se ejecuta siempre que la condición sea verdadera y se termina cuando la condición se hace falsa; por el contrario, el bucle repetir-hasta_que se realiza siempre que la condición es falsa y se termina cuando la condición se hace verdadera.
3. Los bucles también se clasifican en función de la condición probada. En un bucle de conteo fijo, la condición sirve para fijar cuantas iteraciones se realizarán. En un bucle con condición variable (**`mientras, hacer-mientras`** y **`repetir-hasta_que`**), la condición comprobada está basada en que una variable puede cambiar interactivamente con cada iteración a través del bucle.
4. Un bucle mientras es un bucle con condición de entrada, de modo que puede darse el caso de que su cuerpo de sentencias no se ejecute nunca si la condición es falsa en el momento de entrar al bucle. Por el contrario, los bucles hacer-mientras y repetir-hasta_que son bucles de salida y, por consiguiente, las sentencias del cuerpo del bucle al menos se ejecutarán una vez.
5. La sintaxis de la sentencia mientras es:

```
mientras                 cuenta = 1
  <sentencias>           mientras (cuenta <= 10) hacer
                             cuenta = cuenta + 1
fin_mientras             fin_mientras
```

6. La sentencia desde **`(for)`** realiza las mismas funciones que la sentencia mientras pero utiliza un formato diferente. En muchas situaciones, especialmente aquellas que utilizan una condición de conteo fijo, la sentencia desde es más fácil de utilizar que la sentencia mientras equivalente.

```
desde v ← vi hasta of [inc/dec] hacer
    <sentencias>
fin_desde
```

7. La sentencia **`hacer_mientras`** se utiliza para crear bucles *posttest*, ya que comprueba su expresión al final del bucle. Esta característica asegura que el cuerpo de un bucle hacer se ejecuta al menos una vez. Dentro de un bucle hacer debe haber al menos una sentencia que modifique el valor de la expresión comprobada.
8. La programación estructurada utiliza las sentencias explicadas en este capítulo. Esta programación se centra en el modo de escribir las partes detalladas de programas de una computadora como módulos independientes. Su filosofía básica es muy simple: «Utilice sólo construcciones que tengan un punto de entrada y un punto de salida». Esta regla básica se puede romper fácilmente si se utiliza la sentencia de salto **`ir_a`**, por lo que no es recomendable su uso, excepto en situaciones excepcionales.

EJERCICIOS

5.1. Determinar la media de una lista indefinida de números positivos, terminados con un número negativo.

5.2. Dado el nombre de un mes y si el año es o no bisiesto, deducir el número de días del mes.

5.3. Sumar los números enteros de 1 a 100 mediante: *a*) estructura **`repetir`**; *b*) estructura **`mientras`**; *c*) estructura **`desde`**.

5.4. Determinar la media de una lista de números positivos terminada con un número no positivo después del último número válido.

5.5. Imprimir todos los números primos entre 2 y 1.000 inclusive.

5.6. Se desea leer las calificaciones de una clase de informática y contar el número total de aprobados (5 o mayor que 5).

5.7. Leer las notas de una clase de informática y deducir todas aquellas que son NOTABLES (>= 7 y < 9).

5.8. Leer 100 números. Determinar la media de los números positivos y la media de los números negativos.

5.9. Un comercio dispone de dos tipos de artículos en fichas correspondientes a diversas sucursales con los siguientes campos:

- código del artículo A o B,
- precio unitario del artículo,
- número de artículos.

La última ficha del archivo de artículos tiene un código de artículo, una letra X. Se pide:

- el número de artículos existentes de cada categoría,
- el importe total de los artículos de cada categoría.

5.10. Una estación climática proporciona un par de temperaturas diarias (máxima, mínima) (no es posible que alguna o ambas temperaturas sea 9 grados). La pareja fin de temperaturas es 0,0. Se pide determinar el número de días, cuyas temperaturas se han proporcionado, las medias máxima y mínima, el número de errores —temperaturas de 9°— y el porcentaje que representaban.

5.11. Calcular:

$$E(x) = 1 + x = \frac{x^2}{2!} + ... + \frac{n^n}{n!}$$

a) Para N que es un entero leído por teclado.
b) Hasta que N sea tal que $x^n/n < E$ (por ejemplo, $E = 10^{-4}$).

5.12. Calcular el enésimo término de la serie de Fibonacci definida por:

$$A_1 = 1 \quad A_2 = 2 \quad A_3 = 1 + 2 = A_1 + A_2$$
$$A_n = A_{n-1} + A_{n-2} \; (n >= 3)$$

5.13. Se pretende leer todos los empleados de una empresa —situados en un archivo EMPRESA— y a la terminación de la lectura del archivo se debe visualizar un mensaje «existen trabajadores mayores de 65 años en un número de ...» y el número de trabajadores mayores de 65 años.

5.14. Un capital C está situado a un tipo de interés R. ¿Al término de cuántos años se doblará?

5.15. Se desea conocer una serie de datos de una empresa con 50 empleados: *a*) ¿Cuántos empleados ganan más de 300.000 pesetas al mes (salarios altos); *b*) entre 100.000 y 300.000 pesetas (salarios medios); y *c*) menos de 100.000 pesetas (salarios bajos y empleados a tiempo parcial)?

5.16. Imprimir una tabla de multiplicar como

	1	2	3	4	...	15
**	**	**	**	**	...	**
1*	1	2	3	4	...	15
2*	2	4	6	8	...	30
3*	3	6	9	12	...	45
4*	4	8	12	16	...	60
.						
.						
.						
15*	15	30	45	60	...	225

5.17. Dado un entero positivo n (> 1), comprobar si es primo o compuesto.

REFERENCIAS BIBLIOGRÁFICAS

DIJKSTRA, E. W.: «Goto Statement Considered Harmful», *Communications of the ACM*, vol. 11, núm. 3, marzo 1968, 147-148, 538, 541.

KNUTH, D. E.: «Structured Programming with goto Statements», *Computing Surveys*, vol. 6, núm. 4, diciembre 1974, 261.