CAPÍTULO 5

Estructuras de control: bucles

Contenido

INTRODUCCIÓN

Una de las características de las computadoras que aumentan considerablemente su potencia es su capacidad para ejecutar una tarea muchas (*repetidas*) veces con gran velocidad, precisión y fiabilidad. Las tareas repetitivas es algo que los humanos encontramos difíciles y tediosas de realizar. En este capítulo se estudian las *estructuras de control iterativas* o *repetitivas* que realizan la repetición o iteración de acciones. C++ soporta tres tipos de estructuras de control: los bucles **`while`**, **`for`** y **`do-while`**. Estas estructuras de control o sentencias repetitivas controlan el número de veces que una sentencia o listas de sentencias se ejecutan.

CONCEPTOS CLAVE

- Bucle.
- Comparación de `while`, `for` y `do`.
- Control de bucles.
- Iteracción/repetición.
- Optimización de bucles.
- Sentencia **`break`**.
- Sentencia **`do-while`**.
- Sentencia **`for`**.
- Sentencia **`while`**.
- Terminación de un bucle.

5.1. LA SENTENCIA `WHILE`

Un bucle es cualquier construcción de programa que repite una sentencia o secuencia de sentencias un número de veces. La sentencia (o grupo de sentencias) que se repiten en un bloque se denomina **cuerpo** del bucle y cada repetición del cuerpo del bucle se llama **iteración** del bucle. Las dos principales cuestiones de diseño en la construcción del bucle son: ¿Cuál es el cuerpo del bucle? ¿Cuántas veces se iterará el cuerpo del bucle?

Un bucle `while` tiene una *condición* del bucle (una expresión lógica) que controla la secuencia de repetición. La posición de esta condición del bucle es delante del cuerpo del bucle y significa que un bucle `while` es un bucle *pretest* de modo que cuando se ejecuta el mismo, se evalúa la condición *antes* de que se ejecute el cuerpo del bucle. La Figura 5.1 representa el diagrama del bucle `while`.

El diagrama de la Figura 5.1 indica que la ejecución de la sentencia o sentencias expresadas se repite *mientras* la condición del bucle permanece verdadera y termina cuando se hace falsa. También indica el diagrama anterior que la condición del bucle se evalúa antes de que se ejecute el cuerpo del bucle y, por consiguiente, si esta condición es inicialmente falsa, el cuerpo del bucle no se ejecutará. En otras palabras, el cuerpo de un bucle `while` se ejecutará *cero* o *más veces*.

Sintaxis

```
1 while (condición_bucle)
     sentencia;                          cuerpo

2 while (condición_bucle)
  {
     sentencia-1;
     sentencia-2;
           .                             cuerpo
           .
           .
     sentencia-n;
}
```

`while` — *palabra reservada C++*
`condición_bucle` — *expresión lógica o booleana*
`sentencia` — *sentencia simple o compuesta*

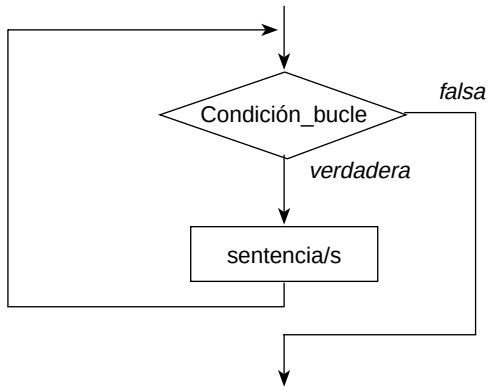


Figura 5.1. Diagrama de flujo de un bucle `while`.

El *comportamiento* o *funcionamiento* de una sentencia (bucle) `while` es:

1. Se evalúa la *condición_bucle*.
2. Si *condición_bucle* es verdadera:
   a) La *sentencia* especificada, denominada **cuerpo** del bucle, se ejecuta.
   b) Vuelve el control al paso 1.
3. En caso contrario:
   El control se transfiere a la sentencia siguiente al bucle o sentencia `while`.

El cuerpo del bucle se repite **mientras** que la expresión lógica (condición del bucle) sea verdadera. Cuando se evalúa la expresión lógica y resulta falsa, se termina y se *sale* del bucle y se ejecuta la siguiente sentencia del programa después del cuerpo de la sentencia `while`.

```
// cuenta hasta 10 (1 a 10)
int x = 1;
while (x <= 10)
   cout << "x: " << ++x;
```

Ejemplo

```
// visualizar n asteriscos
contador = 0; <------------------------------ inicialización
while (contador < n) <----------------------- prueba/condición
{
   cout << " * ";
   contador++; <------------------------------ actualización (incrementa en
                                                1 contador)
} // fin de while
```

La variable que representa la condición del bucle se denomina también **variable de control del bucle** debido a que su valor determina si el cuerpo del bucle se repite. La variable de control del bucle debe ser: (1) inicializada, (2) comprobada, y (3) actualizada para que el cuerpo del bucle se ejecute adecuadamente. Cada etapa se resume así:

1. *Inicialización*. `contador` se establece a un valor inicial (se inicializa a cero, aunque podría ser otro su valor) antes de que se alcance la sentencia `while`.
2. *Prueba/condición*. Se comprueba el valor de `contador` antes de que comience la repetición de cada bucle (denominada *iteración* o *pasada*).
3. *Actualización*. `Contador` se actualiza (su valor se incrementa en 1, mediante el operador ++) durante cada iteración.

Si la variable de control no actualiza el bucle, éste se ejecutará «siempre». Tal bucle se denomina **bucle infinito**. En otras palabras, un bucle infinito (sin terminación) se producirá cuando la condición del bucle permanece y no se hace falsa en ninguna iteración.

```
// bucle infinito
contador = 1;
while (contador < 100)
{
   cout << contador << endl;
   contador--; <------------- decrementa en 1 contador
}
```

`contador` se inicializa a 1 (menor de 100) y como `contador--` decrementa en 1 el valor de `contador` en cada iteración, el valor del contador nunca llegará a valer 100, valor necesario de `contador` para que la condición del bucle sea falsa. Por consiguiente, la condición `contador < 100` siempre será verdadera, resultando un bucle infinito, cuya salida será:

```
 1
 0
-1
-2
-3
-4
 .
 .
 .
```

Ejemplo

```
// Bucle de muestra con while

int main()
{
   int contador = 0;    // inicializa la condición

   while(contador < 5)  // condición de prueba
   {
      contador ++;        // cuerpo del bucle
      cout << "contador: " << contador << "\n";
   }

   cout << "Terminado.Contador: " << contador << "\n";
   return 0;
}
```

Ejecución

```
contador:  1
contador:  2
contador:  3
contador:  4
contador:  5
Terminado.Contador: 5
```

5.1.1. Operadores de incremento y decremento (++, --)

C++ ofrece los operadores de incremento (++) y decremento (--) que soporta una sintaxis abreviada para añadir (incrementar) o restar (decrementar) `1` al valor de una variable. Recordemos del Capítulo 3 la sintaxis de ambos operadores:

```
++nombreVariable            // preincremento
nombreVariable++            // postincremento
```

```
--nombreVariable            // predecremento
nombreVariable--            // postdecremento
```

Ejemplo 5.1

Si `i` es una variable entera cuyo valor es 3, las variables `k` e `i` toman los valores sucesivos que se indican en las sentencias siguientes:

```
k = i++;            // asigna el valor 3 a k y 4 a i
k = ++i;            // asigna el valor 5 a k y 5 a i
k = i--;            // asigna el valor 5 a k y 4 a i
k = --i;            // asigna el valor 4 a k y 3 a i
```

Ejemplo 5.2

Uso del operador de incremento ++ para controlar la iteración de un bucle (una de las aplicaciones más usuales de ++).

```
// programa cálculo de calorías
#include <iostream>
using namespace std;

int main()
{
   int num_de_elementos, cuenta,
       calorias_por_alimento, calorias_total;

   cout << "¿Cuántos alimentos ha comido hoy?";
   cin >> num_de_elementos;

   calorias_total = 0;
   cuenta = 1;
   cout << "Introducir el número de calorias de cada uno de los"
        << num_elementos << " alimentos tomados:\n";

   while (cuenta++ <= numero_de_elementos)
   {
      cin >> calorias_por_alimento;
      calorias_total = calorias_total + calorias_por_alimento;
   }

   cout << "Las calorias totales consumidas hoy son = "
        << calorias_total << endl;
   return 0;
}
```

Ejecución de muestra

```
¿Cuántos alimentos ha comido hoy? 8.
Introducir el número de calorías de cada uno de los 8 alimentos tomados:
500   50   1400   700   10   5   250   100
Las calorías totales consumidas hoy son = 3015
```

5.1.2. Terminaciones anormales de un ciclo

Un error típico en el diseño de una sentencia `while` se produce cuando el bucle sólo tiene una sentencia en lugar de varias sentencias como se planeó. El código siguiente:

```
contador = 1;
while (contador < 25)
   cout << contador << endl;
   contador++;
```

visualizará infinitas veces el valor 1. Es decir, entra en un bucle infinito del que nunca sale porque no se actualiza (modifica) la variable de control `contador`.

La razón es que el punto y coma al final de la línea `cout << contador << endl;` hace que el bucle termine en ese punto y coma, aunque aparentemente el sangrado pueda dar la sensación de que el cuerpo de `while` contiene dos sentencias: `cout` ... y `contador++;`.

El error se hubiera detectado rápidamente si el bucle se hubiera escrito correctamente con una sangría.

```
contador = 1;
while (contador < 25)
   cout << contador << endl;
contador++;
```

La solución es muy sencilla, utilizar las llaves de la sentencia compuesta:

```
contador = 1;
while (contador < 25)
{
   cout << contador << endl;
   contador++;
}
```

5.1.3. Diseño eficiente de bucles

Una cosa es analizar la operación de un bucle y otra diseñar eficientemente sus propios bucles. Los principios a considerar son: primero, analizar los requisitos de un nuevo bucle con el objetivo de determinar su inicialización, prueba (condición) y actualización de la variable de control del bucle; segundo, desarrollar *patrones estructurales* de los bucles que se utilizan frecuentemente.

5.1.4. Bucles `while` con cero iteraciones

El cuerpo de un bucle no se ejecuta nunca si la prueba o condición de repetición del bucle no se cumple (es falsa) cuando se alcanza `while` la primera vez.

```
contador = 10
while (contador > 100)
{
   ...
}
```

El bucle anterior nunca se ejecutará ya que la condición del bucle (`contador > 100`) es falsa la primera vez que se ejecuta. El cuerpo del bucle nunca se ejecutará.

5.1.5. Bucles controlados por centinela

Normalmente, no se conoce con exactitud cuántos elementos de datos se procesarán antes de comenzar su ejecucción. Esto se produce bien porque hay muchos datos a contar manualmente o porque el número de datos a procesar depende de cómo prosigue el proceso de cálculo.

Un medio para manejar esta situación es instruir al usuario a introducir un único dato definido y especificado denominado *valor centinela* como último dato. La condición del bucle comprueba cada dato y termina cuando se lee el valor centinela. El valor centinela se debe seleccionar con mucho cuidado y debe ser un valor que no pueda producirse como dato. En realidad el centinela es un valor que sirve para terminar el proceso del bucle.

```
// entrada de datos numéricos
// centinela –1
const int centinela = -1
cout << "Introduzca primera nota";
cin >> nota;
while (nota != centinela)
{
   cuenta++;
   suma += nota;
   cout << "Introduzca la siguiente nota:";
   cin >> nota;
} // fin de while
cout << "Final"
```

Si se lee el primer valor de `nota`, por ejemplo 25, y luego se ejecuta el bucle, la salida podría ser ésta:

```
Introduzca primera nota: 25
Introduzca siguiente nota: 30
Introduzca siguiente nota: 90
Introduzca siguiente nota: -1      // valor del centinela
Final
```

5.1.6. Bucles controlados por indicadores (banderas)

Las variables tipo `bool` se utilizan con frecuencia como *indicadores* o *banderas de estado* para controlar la ejecución de un bucle. El valor del indicador se inicializa (normalmente a falso `"false"`)antes de la entrada al bucle y se redefine (normalmente a verdadero `"true"`) cuando un suceso específico ocurre dentro del bucle. Un *bucle controlado por bandera* o *indicador* se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador.

Ejemplo 5.3

Se desea leer diversos datos tipo carácter introducidos por teclado mediante un bucle `while` y se debe terminar el bucle cuando se lea un dato tipo dígito (rango '0'a '9').

La variable bandera `digito_leido` se utiliza como un indicador que representa cuando un dígito se ha introducido por teclado.

Variable bandera	*Significado*
`digito_leido`	Su valor es falso antes de entrar en el bucle y mientras el dato leído sea un carácter y es verdadero cuando el dato leído es un dígito.

El problema que se desea resolver es la lectura de datos carácter y la lectura debe detenerse cuando el dato leído sea numérico (un dígito de '0' a '9'). Por consiguiente, antes de que el bucle se ejecute y se lean los datos de entrada, la variable `digito_leido` se inicializa a falso (`false`). Cuando se ejecuta el bucle, éste debe continuar ejecutándose mientras el dato leído sea un carácter y, en consecuencia, la variable `digito_leido` toma el valor falso y se debe detener el bucle cuando el dato leído sea un dígito y, en este caso, el valor de la variable `digito_leido` se debe cambiar a verdadero. En consecuencia, la condición del bucle debe ser `¡digito_leido` ya que esta condición es verdadera cuando `digito_leido` es falso. El bucle `while` será similar a:

```
char car;
digito_leido = false;       // no se ha leido ningún dato
while (!digito_leido)
{
   cout << "Introduzca un caracter:";
   cin >> car;
   digito_leido = (('0'<= car) && (car <= '9'));
   ...
}  // fin de while
```

El bucle funciona de la siguiente forma:

1. Entrada del bucle: la variable `digito_leido` tiene por valor «falso».
2. La condición del bucle `!digito_leido` es verdadera, por consiguiente se ejecutan las sentencias del interior del bucle.
3. Se introduce por teclado un dato que se almacena en la variable `car`. Si el dato leído es un carácter la variable `digito_leido` se mantiene con valor falso, ya que ése es el resultado de la sentencia de asignación.

   ```
   digito_leido = (('0'<= car) && (car <= '9'));
   ```

   Si el dato leído es un dígito, entonces la variable `digito_leido` toma el valor verdadero, resultante de la sentencia de asignación anterior.
4. El bucle se termina cuando se lee un dato tipo dígito (`'0'` a `'9'`) ya que la condición del bucle es falsa.

Modelo de bucle controlado por un indicador

El formato general de un bucle controlado por indicador es el siguiente:

1. Establecer el indicador de control del bucle a «falso» o «verdadero» con el objeto de que se ejecute el bucle `while` correctamente la primera vez (normalmente se suele inicializar a «falso»).
2. Mientras la condición de control del bucle sea verdadera:
   2.1. Realizar las sentencias del cuerpo del bucle.
   2.2. Cuando se produzca la condición de salida (en el ejemplo anterior, que el dato carácter leído fuese un dígito) se cambia el valor de la variable indicador o bandera, con el objeto de que entonces la condición de control se haga falsa y, por tanto, el bucle se termina.
3. Ejecución de las sentencias siguientes al bucle.

5.1.7. Bucles `while` (true)

La condición que se comprueba en el bucle `while` puede ser cualquier expresión válida C++. Mientras que la condición permanezca *verdadera* (cierta), el bucle `while` continuará ejecutándose. Se puede crear un bucle que nunca termine utilizando el valor `true` (verdadero) para la condición que se comprueba.

```
1:    //Listado while (true)
2:    #include <iostream>
3:    using namespace std;
4:    int main()
5:    {
6:       int contador = 0;

7:       while (contador = 0);  //también, while (true)
8:       {
9:          contador++;
10:          if (contador > 10)
11:            break;
12:      }
13:      cout << "Contador: " << contador << "\n";
14:      return 0;
15:    }
```

Salida

```
Contador: 11
```

Análisis

En la línea 6, un bucle `while` se establece con una condición que nunca puede ser falsa. El bucle incrementa la variable `contador` en la línea 8 y, a continuación, la línea 9 comprueba a ver si el contador es mayor que 10. Si no es así, el bucle se itera de nuevo. Si `contador` es mayor que 10, la sentencia `break` de la línea 10 termina el bucle `while`, y la ejecución del programa pasa a la línea 12.

Ejercicio 5.1

Calcular la media aritmética de seis números.

El cálculo típico de una media de valores numéricos es: leer sucesivamente los valores, sumarlos y dividir la suma total por el número de valores leídos. El código más simple podría ser:

```
float Num1;
float Num2;
float Num3;
float Num4;
float Num5;
float Num6;
cin >> Num1 >> Num2 >> Num3 >> Num4 >> Num5 >> Num6;
float Media = (Num1+Num2+Num3+Num4+Num5+Num6)/6;
```

Evidentemente, si en lugar de 6 valores fueran 1.000, la modificación del código no sólo sería enorme de longitud sino que la labor repetitiva de escritura sería tediosa. Por ello, la necesidad de utilizar un bucle. El algoritmo más simple sería:

```
definir número de elementos como constante de valor 6
Inicializar contador de números
Inicializar acumulador (sumador) de números
Mensaje de petición de datos
mientras no estén leídos todos los datos hacer
   Leer número
   Acumular valor del número a variable acumulador
   Incrementar contador de números
fin_mientras
Calcular media (Acumulador/Total número)
Visualizar valor de la media
Fin
```

El código en C++ es:

```
// Cálculo de la media de seis números
#include <iostream>
#include <string>
using namespace std;

int main()
{
   const int TotalNum = 6;
   int ContadorNum = 0;
   float SumaNum = 0;
   cout << " Introduzca " << TotalNum << " números " << endl;
   while (ContadorNum < TotalNum)
   {
      // valores a procesar
      float Numero;
      cin >> Número;     // leer siguiente número
      SumaNum += Número; // añadir valor a Acumulador
      ++ContadorNum;     // incrementar números leídos
}
   float Media = SumaNum / ContadorNum;
   cout << "Media:" << Media << endl;
   return 0;
}
```

5.2. REPETICIÓN: EL BUCLE FOR

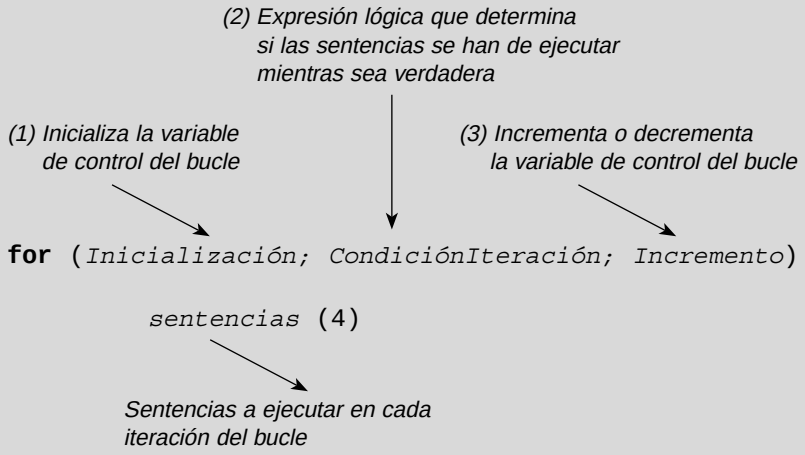
C++ ha heredado la notación de iteración/repetición de C y la ha mejorado ligeramente. El bucle `for` de C++ es superior a los bucles `for` de otros lenguajes de programación tales como BASIC, Pascal y C, ya que ofrece más control sobre la inicialización e incrementación de las variables de control del bucle.

Además del bucle `for`, C++ proporciona otros dos tipos de bucles, `for` y `do`. El bucle `for` que se estudia en esta sección es el más adecuado para implementar *bucles controlados por contador*, que son bucles en los que un conjunto de sentencias se ejecutan una vez por cada valor de un rango especificado, de acuerdo al algoritmo:

```
por cada valor de una variable_contador de un rango específico:
    ejecutar sentencias
```

La sentencia **`for`** (bucle **`for`**) es un método para ejecutar un bloque de sentencias un número fijo de veces. El bucle `for` se diferencia del bucle `while` en que las operaciones de control del bucle se sitúan en un solo sitio: la cabecera de la sentencia.

Sintaxis



El bucle `for` contiene las cuatro partes siguientes:

- *Parte de inicialización*, que inicializa las variables de control del bucle. Se pueden utilizar variables de control del bucle simples o múltiples.
- *Parte de iteración*, que contiene una expresión lógica que hace que el bucle realice las iteraciones de las sentencias, mientras que la expresión sea verdadera.
- *Parte de incremento*, que incrementa o decrementa la variable o variables de control del bucle.
- *Sentencias*, acciones o sentencias que se ejecutarán por cada iteración del bucle.

La sentencia `for` es equivalente al siguiente código `while`:

```
inicialización;
while (condiciónIteración)
{
    sentencias del bucle for
    incremento;
}
```

Ejemplo

```
// imprimir Hola 10 veces
for (int i = 0; i < 10; i++)
   cout << "Hola!";
```

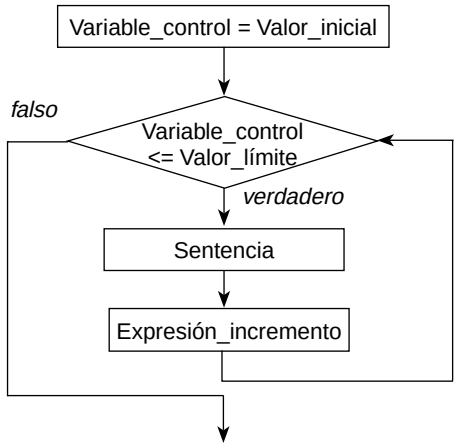


Figura 5.2. Diagrama de sintaxis de un bucle `for`.

Ejemplo

```
for (int i = 0; i < 10; i++)
{
   cout << "Hola!" << endl;
   cout << "el valor de i es: " << i << endl;
}
```

El diagrama de sintaxis de la sentencia `for` es el que se muestra en la Figura 5.2.

Existen dos formas de implementar la sentencia `for` que se utilizan normalmente para implementar bucles de conteo: *formato ascendente*, en el que la variable de control se incrementa y *formato descendente*, en el que la variable de control se decrementa.

```
for (int var_control = valor_inicial; var_control <=
     valor_límite; exp_incremento)
   sentencia
```

formato ascendente

formato descendente

```
for (int varcontrol = valor_inicial; var_control <=
     valor_límite; exp_decremento)
   sentencia
```

Ejemplo de formato ascendente

```
for (int n = 1; n <= 10; n++)
   cout << n << '\t' << n * n << endl;
```

La variable de control es `n` y su valor inicial es `1` de tipo entero, el valor límite es `10` y la expresión de incremento es `n++`. Esto significa que el bucle ejecuta la sentencia del cuerpo del bucle una vez por

cada valor de n en orden ascendente 1 a 10. En la primera iteración (pasada) n tomará el valor 1; en la segunda iteración el valor 2, y así sucesivamente hasta que n toma el valor 10. La salida que se producirá al ejecutarse el bucle será:

```
1    1
2    4
3    9
4    16
5    25
6    36
7    49
8    64
9    81
10   100
```

Ejemplo de formato descendente

```
for (int n = 10; n > 5; n--)
   cout << n << '\t'<< n * n << endl;
```

La salida de este bucle es:

```
10  100
9   81
8   64
7   49
6   36
```

debido a que el valor inicial de la variable de control es 10, y el límite que se ha puesto es n > 5 (es decir, verdadera cuando n = 10, 9, 8, 7, 6); la expresión de decremento es el operador de decremento n--, que decrementa en 1 el valor de n tras la ejecución de cada iteración.

Otros intervalos de incremento/decremento

Los rangos de incremento/decremento de la variable o expresión de control del bucle pueden ser cualquier valor y no siempre 1, es decir, 5, 10, 20, –4, ..., dependiendo de los intervalos que se necesiten. Así, el bucle

```
for (int n = 0; n <= 100; n += 20)
   cout << n << '\t' << n * n << endl;
```

utiliza la expresión de incremento

```
n += 20
```

que incrementa el valor de n en 20, dado que equivale a n = n + 20. Así, la salida que producirá la ejecución del bucle es:

```
0     0
20    400
40    1600
60    3600
80    6400
100   10000
```

Ejemplos

```
// ejemplo 1
for (i = 0; i < 10; i++)
   cout << i << "\n";

// ejemplo 2
for (i = 9; i >= 0; i -= 3)
   cout << ( i * i) << "\n";

// ejemplo 3
for (int i = 1; i < 100; i++)
   cout << i << "\n";

// ejemplo 4
for (int i = 0, j = MAX, i < j; i++, j--)
   cout << (i + 2 * j) << endl;
```

El primer ejemplo inicializa la variable de control del bucle `i` a `0` e itera mientras que el valor de la variable `i` es menor que 10. La parte de incremento del bucle incrementa el valor de la variable en `1`. Por consiguiente, el bucle se realiza diez veces con valores de la variable `i` que van de `0` a `9`.

El segundo ejemplo muestra un bucle descendente que inicializa la variable de control a `9`. El bucle se realiza mientras que `i` no sea negativo, como la variable se decrementa en `3`, el bucle se ejecuta cuatro veces con el valor de la variable de control `i, 9, 6, 3` y `0`.

El Ejemplo 3 muestra la característica de C++, ya explicada, de que se puede declarar e inicializar la variable de control `i`, en este caso a `1`. La variable se incrementa en 1, por consiguiente, el bucle realiza 99 iteraciones, para `i` de 1 a 99.

El Ejemplo 4 declara dos variables de control, `i` y `j,` y las inicializa a `0` y la constante `MAX`. El bucle se ejecutará mientras `i` sea menor que `j`. Las variables de control `i, j,` se incrementan ambas en `1`.

Ejemplo 5.4

Suma de los 10 primeros números.

```
// demo de un bucle for
#include <iostream>
using namespace std;

int main()
{
   int suma = 0;
   for (int n = 1; n <= 10; n++)
      suma = suma + n;

   cout << "La suma de los números 1 a 10 es "
        << suma << endl;
   return 0;
}
```

La salida del programa es

```
La suma de los números 1 a 10 es 55
```

5.2.1. Diferentes usos de bucles for

ANSI C++ Estándar requiere que las variables sean declaradas en la expresión de inicialización de un bucle `for` sean locales al bloque del bucle `for`. De igual modo permite que:

- El valor de la variable de control se puede modificar en valores diferentes de 1.
- Se pueden utilizar más de una variable de control.

Ejemplo de declaración local de variables de control

Cuando un bucle `for` declara una variable de control, esa variable permanece hasta el final del ámbito que contiene a `for`. Por consiguiente, el siguiente código contiene sentencias válidas.

```
for (int i != 0; i < 10; i++)
   cout << i << endl;
cout << " valor de la variable de control"
     << " después de que termina el bucle es " << i << endl;
```

Ejemplos de incrementos/decrementos con variables de control diferentes

Las variables de control se pueden incrementar o decrementar en valores de tipo `int`, pero también es posible en valores de tipo `double` y, en consecuencia, se incrementaría o decrementaría en una cantidad decimal.

```
int n;
for ( n = 1; n <= 10; n = n + 2)
   cout << "n es ahora igual a " << n << endl;

for (n = 0; n >= 100; n = n - 5)
   cout << " n es ahora igual a " << n << endl;

for (double long = 0.75; long <= 5; long = long + 0.05)
   cout << " longitud es ahora igual a " << long << endl;
```

La expresión de incremento en ANSI C++ no necesita ser una suma o una resta. Tampoco se requiere que la inicialización de una variable de control sea igual a una constante. Se puede inicializar y cambiar una variable de control del bucle en cualquier cantidad que se desee. Naturalmente, cuando la variable de control no sea de tipo `int`, se tendrán menos garantías de precisión. Por ejemplo, el siguiente código muestra un medio más para arrancar un bucle `for`.

```
for (double x = pow(y, 3.0); x > 2.0; x = sqrt(x))
   cout << "x es ahora igual a "  << x << endl;
```

5.3. PRECAUCIONES EN EL USO DE F0R

Un bucle `for` se debe construir con gran precaución, asegurándose de que la expresión de inicialización, la condición del bucle y la expresión de incremento harán que la condición del bucle se convierta en falsa en algún momento. En particular: *«si el cuerpo de un bucle de conteo modifica los valores de cualquier variable implicada en la condición del bucle, entonces el número de repeticiones se puede modificar»*.

Esta regla anterior es importante, ya que su aplicación se considera una mala práctica de programación. Es decir, no es recomendable modificar el valor de cualquier variable de la condición del bucle

dentro del cuerpo de un bucle `for`, ya que se pueden producir resultados imprevistos. Por ejemplo, la ejecución de

```
int limite = 1;
for (int i = 0; i <= limite; i++)
{
   cout << i << endl;
   limite++;
}
```

produce una secuencia infinita de enteros (puede terminar si el compilador tiene constantes `MAXINT`, con máximos valores enteros, entonces la ejecución terminará cuando `i` sea `MAXINT` y `limite` sea `MAXINT+1 = MININT`).

```
0
1
2
3
.
.
.
```

ya que a cada iteración, la expresión `limite++` incrementa `limite` en `1`, antes de que `i++` incremente `i`. A consecuencia de ello, la condición del bucle `i <= limite` siempre es cierta.

Otro ejemplo es el bucle

```
int limite = 1;
for (int i = 0; i <= limite; i++)
{
   cout << i << endl;
   i--;
}
```

que producirá infinitos ceros

```
0
0
0
.
.
```

ya que en este caso la expresión `i--` del cuerpo del bucle decrementa `i` en `1` antes de que se incremente la expresión `i++` de la cabecera del bucle en `1`. Como resultado `i` es siempre `0` cuando el bucle se comprueba.

5.3.1. Bucles infinitos

El uso principal de un bucle `for` es implementar bucles de conteo en el que el número de repeticiones se conoce por anticipado. Por ejemplo, la suma de enteros de 1 a n. Sin embargo, existen muchos problemas en los que el número de repeticiones no se pueden determinar por anticipado. Para estas situa-

ciones algunos lenguajes tienen sentencias específicas tales como las sentencias `LOOP` de Modula-2 y Modula-3, el bucle `DO` de FORTRAN 90 o el bucle `loop` del de Ada. C++ no soporta una sentencia que realice esa tarea, pero existe una variante de la sintasix de `for` que permite implementar **bucles infinitos**, que son aquellos bucles que, en principio, no tienen fin.

Sintaxis

```
for (;;)
  sentencia
```

`sentencia` se ejecuta indefinidamente a menos que se utilice una sentencia `return` o `break` (normalmente una combinación `if-break` o `if-return`).

La razón de que el bucle se ejecute indefinidamente es que se ha eliminado la expresión de inicialización, la condición del bucle y la expresión de incremento; al no existir una condición de bucle que especifique cuál es la condición para terminar la repetición de sentencias, éstas se ejecutarán indefinidamente. Así, el bucle

```
for (;;)
   cout << "Siempre así, te llamamos siempre así...";
```

producirá la salida

```
Siempre así, te llamamos siempre así...
Siempre así, te llamamos siempre así...
...
```

un número ilimitado de veces, a menos que el usuario interrumpa la ejecución (normalmente pulsando las teclas `Ctrl` y `C` en ambientes PC).

Para evitar esta situación, se requiere el diseño del bucle `for` de la forma siguiente:

1. El cuerpo del bucle ha de contener todas las sentencias que se desean ejecutar repetidamente.
2. Una sentencia terminará la ejecución del bucle cuando se cumpla una determinada condición.

La sentencia de terminación suele ser `if-break` con la sintaxis

```
if (condición) break;
```

condición *es una expresión lógica*
`break` *termina la ejecucción del bucle y transfiere el control a la sentencia siguiente al bucle*

y la sintaxis completa

```
for (;;)          // bucle
{
      lista_sentencias1
      if (condición_terminación) break;
      lista_sentencias2
}                 // fin del bucle
```

lista_sentencias *puede ser vacía, simple o compuesta*

Ejemplo 5.5

```
for (;;)
{
   cout << "Introduzca un número";
   cin >> num;
   if (num = -999) break;
   ...
}
```

5.3.2. Los bucles `for` vacíos

Tenga cuidado de situar un punto y coma después del paréntesis inicial del bucle `for`. Es decir, el bucle

```
for (int i = 1; i<= 10; i++); ←———————— problema
   cout << "Sierra Magina" << endl;
```

no se ejecuta correctamente, ni se visualiza la frase `"Sierra Magina"` diez veces como era de esperar, ni se produce un mensaje de error por parte del compilador.

En realidad, lo que sucede es que se visualiza una vez la frase `"Sierra Magina"` ya que la sentencia `for` es una sentencia vacía al terminar con un punto y coma (;). Entonces, lo que sucede es que la sentencia `for` no hace absolutamente nada y tras 10 iteraciones y, por tanto; después de que el bucle `for` se ha terminado, se ejecuta la siguiente sentencia `cout` y se escribe `"Sierra Magina"`.

El bucle `for` con cuerpos vacíos puede tener algunas aplicaciones, especialmente cuando se requieren ralentizaciones o temporizaciones de tiempo.

5.3.3. Sentencias nulas en bucles `for`

Cualquiera o todas las sentencias de un bucle `for` pueden ser nulas. Para ejecutar esta acción se utiliza el punto y coma (;) para marcar la sentencia vacía. Si se desea crear un bucle `for` que actúe exactamente como un bucle `while`, se deben incluir las primeras y terceras sentencias vacías.

```
1:    // Listado
2:    // bucles for con sentencias nulas
3:
4:    #include <iostream>
5:    using namespace std;
6:    int main()
7:    {
8:       int contador = 0;
9:
10:      for (; contador < 5;)
11:      {
12:         contador++;
13:         cout << "¡Bucle! ";
14:      }
15:
```

```
16:      cout << "\n Contador: " << contador << " \n";
17:      return 0;
18:   }
```

Salida

```
¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle!
Contador: 5
```

Análisis

En la línea 8 se inicializa la variable del contador. La sentencia `for` en la línea 10 no inicializa ningún valor, pero incluye una prueba de `contador < 5`. No existe ninguna sentencia de incrementación, de modo que el bucle se comporta exactamente como la sentencia siguiente.

```
while(contador < 5)
```

5.4. REPETICIÓN: EL BUCLE DO-WHILE

La sentencia **`do-while`** se utiliza para especificar un bucle condicional que se ejecuta al menos una vez. Esta situación se suele dar en algunas circunstancias en las que se ha de tener la seguridad de que una determinada acción se ejecutará una o varias veces, pero al menos una vez.

Sintaxis

Acción (sentencia) a ejecutar al menos una vez

Expresión lógica que determina si la acción se repite

1.

```
do   sentencia   while   (expresión)
```

2.

```
do
   sentencia
while (expresión)
```

La construcción **do** comienza ejecutando *sentencia*. Se evalúa a continuación *expresión*. Si *expresión* es verdadera, entonces se repite la ejecución de sentencia. Este proceso continúa hasta que expresión es falsa. La semántica del bucle do se representa gráficamente en la Figura 5.3.

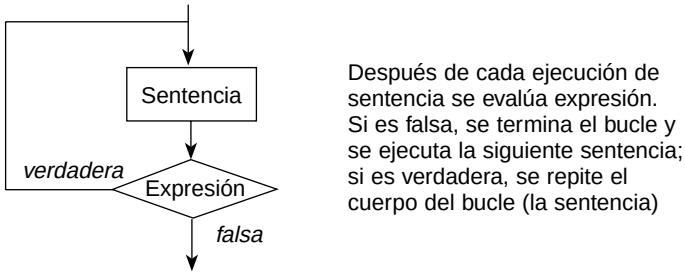


Figura 5.3. Diagrama de flujo de la sentencia **`do-while`**.

Ejemplo 5.6

```
do
{
   cout << "Introduzca un dígito (0-9): ";
   cin >> digito;
} while ((digito < '0') || ('9'< digito));
```

Este bucle se realiza mientras se introduzcan dígitos y se termina cuando se introduzca un carácter que no sea un dígito de `'0'` a `'9'`.

Ejercicio 5.2

Aplicación simple de un bucle `while`: seleccionar una opción de saludo al usuario dentro de un programa.

```
#include <iostream>
using namespace std;

int main()
{
   char opcion;
   do
   {
         cout << "Hola" << endl;
         cout << "¿Desea otro tipo de saludo?\n";
              << "Pulse s para si y n para no,\n"
              << "y a continuacion pulse intro: ";
         cin >> opcion;
   } while (opcion == 's'|| opcion == 'S');
   cout << "Adios\n";
   return 0;
}
```

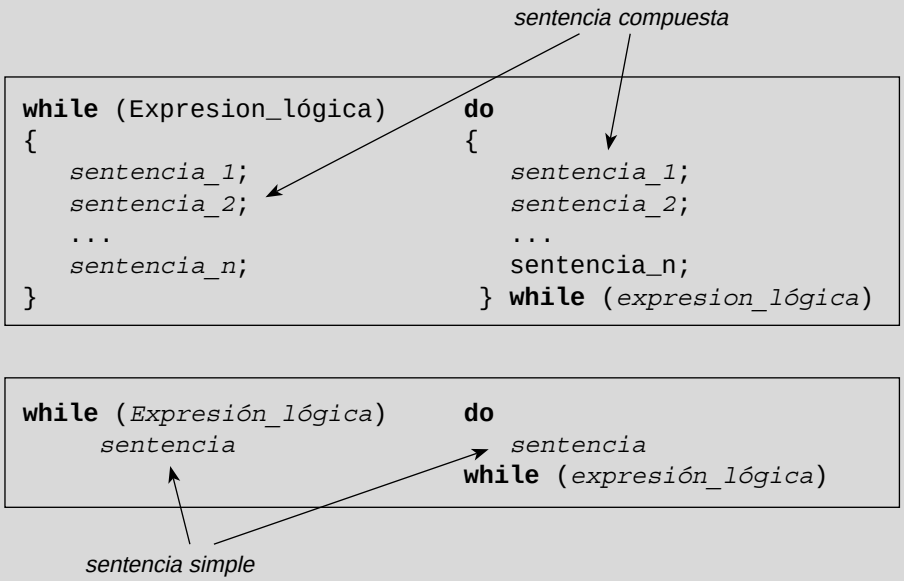
Salida de muestra

```
Hola
¿Desea otro tipo de saludo?
Pulse s para sí y n para no
y a continuación pulse intro: S
Hola
¿Desea otro tipo de saludo?
Pulse s para sí y n para no
y a continuación pulse intro: N
Adiós
```

5.4.1. Diferencias entre `while` y `do-while`

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 2 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. COMPARACIÓN DE BUCLES `WHILE`, `FOR` Y `DO-WHILE`

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.

`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);
```

5.6. DISEÑO DE BUCLES

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.

5.6.1. 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`.

```
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.

5.6.2. Fin de un bucle

Existen cuatro métodos utilizados normalmente para terminar un bucle de entrada. Estos cuatro métodos son[1]:

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)
...
```

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

```
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.

5.6.3. OTRAS TÉCNICAS DE TERMINACIÓN DE BUCLE

Las técnicas más usuales para la terminación de bucles de cualquier tipo son:

1. Bucles controlados por contador.
2. Preguntar antes de iterar.
3. Salir con una condición bandera.

Un bucle **controlado por contador** es cualquier bucle que determina el número de iteraciones antes de que el bucle comience y, a continuación, repite (itera) el cuerpo del bucle esas iteraciones. La técnica de la lista encabezada por tamaño es un ejemplo de un bucle controlado por contador.

La técnica de **preguntar antes de iterar** se puede utilizar para bucles distintos de los bucles de entrada, pero el uso más común de esta técnica es para procesar la entrada.

La técnica del valor centinela es una técnica conocida también como **salida con una condición bandera** o **señalizadora**. Una variable que cambia su valor para indicar que algún suceso o evento ha tenido lugar, se denomina normalmente **bandera** o **indicador**. En el ejemplo anterior de suma de números, la variable bandera es `numero` de modo que cuando toma un valor negativo significa que indica que la lista de entrada ha terminado.

5.6.4. Bucles `for` vacíos

La sentencia nula `(;)` es una sentencia que está en el cuerpo del bucle y no hace nada.

```
1:    // Listado
2:    // Demostración de la sentencia nula
```

```
3:    // como cuerpo del bucle for
4:
5:    #include <iostream>
6:    using namespace std;
7:    int main()
8:    {
9:       for (int i = 0; i < 5; cout << "i: " << i++ << endl;
11:       ;
12:      return 0;
13:   }
```

Salida

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

Análisis

El bucle `for` de la línea 8 incluye tres sentencias: la sentencia de *inicialización* establece el contador `i` y se inicializa a `0`. La sentencia de *condición* comprueba `i < 5`, y la sentencia *`acción`* imprime el valor de `i` y lo incrementa.

5.6.5. Ruptura de control en bucles

En numerosas ocasiones, es conveniente disponer de la posibilidad de abandonar o salir de una iteración durante su ejecución, o dicho de otro modo, saltar sobre partes del código. El flujo de control en bucles se puede alterar de dos modos: insertar una sentencia `break` y una sentencia `continue`. La sentencia `break` termina el bucle; la sentencia `continue` termina la iteración actual del cuerpo del bucle. Además de estas sentencias, C++ dispone también de la sentencia `goto`, aunque este último caso no es muy recomendable usar.

Sentencias para alteración del flujo de control

- `break` (ruptura).
- `continue` (continuar).
- `goto` (ir_a).

`break`

La sentencia `break` transfiere el control del programa al final del bucle (o en el caso de sentencia `swith` de selección, la sentencia más interna que lo encierra).

Sintaxis

```
break;
```

Se puede utilizar para salir de un bucle infinito en lugar de comprobar el número en la expresión del bucle `while` o bucle `for`.

```
while (true)
{
   int a;

   cout << "a= ";
   cin >> a;
   if (!cin || a == b)
      break;
   cout << "número de la suerte: " << a << endl;
}
```

continue

La sentencia `continue` consta de la palabra reservada `continue` seguida por un punto y coma. Cuando se ejecuta, la sentencia `continue` termina la iteración actual del cuerpo del bucle actual más cercano que lo circunda.

Sintaxis

```
continue;
```

La sentencia provoca el retorno a la expresión de control del bucle `while`, o a la tercera expresión del bucle `for`, saltándose así el resto del cuerpo del bucle. Su uso es menos frecuente que `break`.

Uso de `continue`

```
while (expresión)
{
   sentencia1;
   if (car == '\n')
      continue;
   sentencia 2;
}
sentencia3;
```

Salta el resto del cuerpo del bucle y comienza una nueva iteración

Uso de `break`

```
while (cin.get (car))
{
   sentencia1;
   if (car == '\n')
      break;
   sentencia2;
}
sentencia n;
```

`break` *salta el resto del bucle y va a la sentencia siguiente*

Sentencia goto

C++, como otros lenguajes de programación, incluye la sentencia `goto` (ir_a), pero nunca se debe utilizar. Sólo en casos muy extremos se debe recurrir a ella.

Sintaxis

```
goto etiqueta;
goto excepción;
...
excepción;
```

Un caso extremo puede ser, salir de varios bucles anidados, ya que en este caso excepcional `break` sólo saca al bucle inmediatamente superior. Existen otros casos en que se requiere máxima eficiencia; por ejemplo, en un bucle interno de alguna solución en tiempo real o en el núcleo de un sistema operativo.

La sentencia break en bucles

La sentencia **`break`** se utiliza para realizar una terminación anormal del bucle. Dicho de otro modo, una terminación antes de lo previsto. Su sintaxis es:

```
break;
```

y se utiliza para salir de un bucle `while` o `do-while`, aunque también se puede utilizar dentro de una sentencia `switch`, siendo éste un uso muy frecuente:

```
while (condición)
{
    if (condición2)
       break;
    // sentencias
}
```

Ejemplo 5.8

El siguiente código extrae y visualiza valores de entrada desde el flujo de entrada `cin` hasta que se encuentra un valor especificado

```
int Clave;
cin >> Clave;
int Entrada;
while (cin >> Entrada) {
   if (Entrada != Clave)
      cout << Entrada << endl;
   else
      break;
}
```

Precaución

El uso de **break** en un bucle no es muy recomendable ya que puede hacer difícil la comprensión del comportamiento del programa. En particular, suele hacer muy difícil verificar los invariantes de los bucles. Por otra parte, suele ser fácil la reescritura de los bucles sin la sentencia **break**. El bucle anterior escrito sin la sentencia `break`.

```
int Clave;
cin >> Clave;
int Entrada;
while ((cin >> Entrada) && (Entrada != Clave))
{
   cout << Entrada << end1;
}
```

Sentencias `break` y `continue`

Las sentencias `break` y `continue` terminan la ejecución de un bucle `while` de modo similar a los otros bucles.

```
// Listado
// sentencias bucles for vacíos

#include <iostream>
using namospace std;

int main()
{
   int contador = 0;          // inicialización
   int max;
   cout << "Cuántos holas?";
   cin >> max;
   for (;;)         // bucle for que no termina nunca
   {
      if(contador < max)     // test
      {
         cout << "Hola!\n";
         contador++;         // incremento
      }
      else
        break;
   }
   return 0;
}
```

Salida

```
Cuántos holas? 3
Hola!
Hola!
Hola!
```

5.7. BUCLES ANIDADOS

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 5.9

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`*, y* `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
        ...
```

```
for (int x = 1; x <= Xultimo; x++)

   for (int y = 1; y <= Yultimo; y++)
   {
      producto = x * y;
      cout << setw(2) << x << " * "
           << setw(2) << y << " = "
           << setw(3) << producto << endl;
   }
```

bucle externo *bucle interno*

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 5.10

```
// 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 5.3

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 ∂ 5 – 1).

Ejercicio 5.4

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:   }
```

Salida

```
¿Cuántas filas? 4
¿Cuántas columnas? 12
¿Qué carácter? *
************
************
************
************
```

Análisis

El usuario solicita el número de filas y columnas y un carácter a imprimir. El primer bucle `for` de la línea 14 inicializa un contador (`i`) a `0` y, a continuación, se ejecuta el cuerpo del bucle `for` externo.

En la línea 16 se inicializa otro bucle `for` y un segundo contador `j` se inicializa a `0` y se ejecuta el cuerpo del bucle interno. En la línea 17 se imprime el carácter `elCar` (un `*`, en la ejecucción). Se evalúa la condición (`j < columnas`) y si se evalúa a *true* (*verdadero*), `j` se incrementa y se imprime el siguiente carácter. Esta acción se repite hasta que `j` sea igual al número de columnas.

El bucle interno imprime 12 caracteres asterisco en una misma fila y el bucle externo repite cuatro veces (número de filas) la fila de caracteres.

RESUMEN

Un bucle es un grupo de instrucciones a la computadora que se ejecutan repetidamente hasta que una condición de terminación se cumple. Los bucles representan estructuras repetitivas y una estructura repetitiva es aquella que especifica que una acción (sentencia) se repite mientras que una condición especificada permanece verdadera (es cierta). Existen muchas formas de diseñar un bucle (lazo o ciclo) dentro de un programa C++, pero las más importantes son: repetición controlada por contador y repetición controlada por centinela.

- Un contador de bucle se utiliza para contar las repeticiones de un grupo de instrucciones. Se incrementa (o decrementa) normalmente en 1 cada vez que se ejecuta el grupo de instrucciones.
- Los valores centinela se utilizan para controlar la repetición cuando el número de repeticiones (iteraciones) no se conoce por adelantado y el bucle contiene sentencias que obtienen datos cada vez que se ejecuta. Un valor centinela se introduce después de que se han proporcionado todos los datos válidos al programa y debe ser distinto de los datos válidos.
- Los bucles `while` comprueban una condición, y si es verdadera (*true*), se ejecutan las sentencias del cuerpo del bucle.
- Los bucles `do-while` ejecutan el cuerpo del bucle y comprueban a continuación la condición.
- Los bucles `for` inicializan un valor y, a continuación, comprueban una expresión; si la expresión es verdadera se ejecutan las sentencias del cuerpo del bucle y se comprueba la expresión cada vez que termina la iteración. Cuando la expresión es falsa se termina el bucle y se sale del mismo.
- La sentencia `break` produce la salida inmediata del bucle.
- La sentencia `continue` cuando se ejecuta salta todas las sentencias que vienen a continuación y prosigue con la siguiente iteración del bucle.
- La sentencia `switch` maneja una serie de decisiones en las que se comprueban los valores de una variable o expresión determinada y en función de su valor realiza una acción diferente.