4

Control de programa

Objetivos

- Ser capaz de utilizar las estructuras de repetición **`for`** y **`do/while`**.
- Comprensión de la selección múltiple utilizando la estructura de selección **`switch`**.
- Ser capaz de utilizar los enunciados de control de programa **`break`** y **`continue`**.
- Ser capaz de utilizar los operadores lógicos.

¿Quién puede controlar su destino?
William Shakespeare
Othello

La llave que siempre se usa está brillante.
Benjamín Franklin

El ser humano es un animal que fabrica herramientas.
Benjamín Franklin

La inteligencia ...es la facultad de poder fabricar objetos artificiales, en especial herramientas para fabricar herramientas.
Henry Bergson.

Sinopsis

4.1 Introducción

Llegado a este punto, el lector debería ya estar cómodo escribiendo programas en C simples pero completos. En este capítulo, se estudia con mayor detalle la repetición, y se presentan estructuras adicionales de control de repetición, es decir, la estructura **`for`** y la estructura **`do/while`**. También se presenta la estructura de selección múltiple **`switch`**. Se analiza el enunciado **`break`**, para salir de inmediato y rápidamente de ciertas estructuras de control, así como el enunciado **`continue`**, para saltarse el resto del cuerpo de una estructura de repetición, continuando con la siguiente iteración del ciclo. El capítulo analiza operadores lógicos utilizados en la combinación de condiciones, y concluye con un resumen de los principios de la programación estructurada, tal y como se presentan en los capítulos 3 y 4.

4.2 Lo esencial de la repetición

La mayor parte de los programas incluyen repeticiones o *ciclos.* Un ciclo es un grupo de instrucciones que la computadora ejecuta en forma repetida, en tanto se conserve verdadera alguna *condición de continuación del ciclo*. Hemos analizado dos procedimientos de repetición:

1. Repetición controlada por contado
2. Repetición controlada por centinela

La repetición controlada por contador se denomina a veces *repetición definida,* porque con anticipación se sabe con exactitud cuántas veces se ejecutará el ciclo. La repetición controlada por centinela a veces se denomina *repetición indefinida,* porque no se sabe con anticipación cuantas veces el ciclo se ejecutará.

En la repetición controlada por contador, se utiliza una *variable de control* para contar el número de repeticiones. La variable de control es incrementada (normalmente en 1), cada vez que se ejecuta el grupo de instrucciones. Cuando el valor de la variable de control indica que se ha ejecutado el número correcto de repeticiones, se termina el ciclo y la computadora continúa ejecutando el enunciado siguiente al de la estructura de repetición.

Los valores centinela se utilizan para controlar la repetición cuando:

1. El número preciso de repeticiones, no es conocido con anticipación, y
2. El ciclo incluye enunciados que deben obtener datos cada vez que éste se ejecuta.

El valor centinela indica "fin de datos". El centinela es introducido una vez que al programa se le han proporcionado todos los elementos normales de datos. Los centinelas deben ser diferentes a los elementos normales de datos.

4.3 Repetición controlada por contador

La repetición controlada por contador requiere:

1. El *nombre* de una variable de control (o contador del ciclo).
2. El *valor inicial* de la variable de control.
3. El *incremento* (o *decremento*) con el cual, cada vez que se termine un ciclo, la variable de control será modificada.
4. La condición que compruebe la existencia del *valor final* de la variable de control (es decir, si se debe o no seguir con el ciclo).

Considere el programa sencillo mostrado en la figura 4.1, que imprime los números de 1 a 10. La declaración

```
int counter = 1
```

le da *nombre* a la variable de control (**`counter`**), la declara como un entero, reserva espacio para la misma, y ajusta su *valor inicial* a **`1`**. Esta declaración no es un enunciado ejecutable.

La declaración e inicialización de **`counter`** también podía haber sido hecha mediante los enunciados

```
int counter;

counter = 1;
```

La declaración no es ejecutable, pero la asignación si lo es. Se utilizarán ambos métodos para inicialización de variables.

El enunciado

```
++counter;
```

```
/* Counter-controlled repetition */
#include <stdio.h>

main()
{
   int counter = 1;              /* initialization */

   while (counter <= 10) {       /* repetition condition */
      printf ("%d\n", counter);
      ++counter;                 /* increment */
   }

   return 0;
}
```

```
1
2
3
4
5
6
7
8
9
10
```

Fig. 4.1 Repetición controlada por contador.

incrementa en 1 el contador del ciclo, cada vez que este ciclo se ejecuta. La condición de continuación del ciclo existente en la estructura **while** prueba si el valor de la variable de control es menor o igual a **10** (el valor final en el cual la condición se hace verdadera). Note que el cuerpo de este **while** se ejecuta, aun cuando la variable de control llegue a **10**. El ciclo se terminará cuando la variable de control exceda de **10** (es decir, **counter** se convierte en **11**).

Los programadores de C por lo regular elaborarían la programación de la figura 4.1 de manera más concisa, inicializando **counter** a **0** y remplazando la estructura **while** con

```
While (++counter <= 10 )
   printf ("%d/n", counter);
```

Este código ahorra un enunciado porque el incremento se efectúa de forma directa en la condición **while**, antes de que la condición sea probada. También, este código elimina las llaves alrededor del cuerpo del **while**, porque el **while** ahora sólo contiene un enunciado. Para codificar de esta forma tan concisa y condensada, se requiere de algo de práctica.

Error común de programación 4.1

Dado que los valores en punto flotante pueden ser aproximados, el control de contador de ciclos con variables de punto flotante puede dar como resultado valores de contador no precisos y pruebas no exactas de terminación.

Práctica sana de programación 4.1

Controlar el contador de ciclos con valores enteros.

Práctica sana de programación 4.2

Hacer sangría en los enunciados en el cuerpo de cada estructura de control.

Práctica sana de programación 4.3

Colocar una línea en blanco antes y después de cada estructura de control principal, para que se destaque en el programa.

Práctica sana de programación 4.4

Demasiados niveles anidados pueden dificultar la comprensión de un programa. Como regla general, procure evitar el uso de más de tres niveles de sangrías.

Práctica sana de programación 4.5

La combinación de espaciamiento vertical antes y después de las estructuras de control y las sangrías de los cuerpos de las estructuras de control dentro de los encabezados de las estructuras de control, le da a los programas una apariencia bidimensional que mejora de manera significativa la legibilidad del programa.

4.4 La estructura de repetición for

La estructura de repetición **for** maneja de manera automática todos los detalles de la repetición controlada por contador. Para ilustrar los poderes de **for**, volvamos a escribir el programa de la figura 4.1. El resultado se muestra en la figura 4.2.

El programa funciona como sigue: cuando la estructura **for** inicia su ejecución, la variable de control **counter** se inicializa a **1**. A continuación, se verifica la condición de continuación de ciclo **counter <= 10**. Dado que el valor inicial de counter es **1**, se satisface esta condición, por lo que el enunciado **printf** imprime el valor de counter, es decir **1**.

```
/* Counter-controlled repetition with the for structure */
#include <stdio.h>

main()
{
   int counter;

   /* initialization, repetition condition, and increment */
   /* are all included in the for structure header        */
   for (counter = 1; counter <= 10; counter++)
      printf("%d\n", counter);

   return 0;
}
```

Fig. 4.2 Repetición controlada por contador con la estructura **for**.

A continuación la variable de control **counter** es incrementada por la expresión **counter++**, y el ciclo empieza otra vez, con la prueba de continuación de ciclo. Dado que la variable de control ahora es igual a **2**, el valor final aún no es excedido, y el programa por lo tanto ejecuta otra vez el enunciado **printf**. Este proceso continúa hasta que la variable de control **counter** es incrementada a su valor final de 11 lo que hará que falle la prueba de continuación de ciclo y termine la repetición. El programa continúa ejecutando el primer enunciado que encuentra después de la estructura **for** (en este caso, el enunciado **return** al final del programa).

La figura 4.3 analiza más de cerca la estructura **for** de la figura 4.2. Advierta que la estructura **for** "lo hace todo" especifica cada uno de los elementos necesarios para la repetición controlada por contador con una variable de control. Si en el cuerpo del **for** existe más de un enunciado, se requerirán de llaves para definir el cuerpo del ciclo.

Note que la figura 4.2 utiliza la condición de continuación de ciclo **counter <= 10**. Si el programador hubiera escrito en forma incorrecta **counter < 10**, entonces el ciclo sólo se hubiera ejecutado 9 veces. Este es un error común de lógica, conocido como un *error de diferencia por uno.*

Error común de programación 4.2

Usar un operador relacional incorrecto o usar un valor final incorrecto de un contador de ciclo, en la condición de una estructura **while** *o* **for**, *puede causar errores de diferencia por uno.*

Práctica sana de programación 4.6

Usar el valor final en la condición de una estructura **while** *o* **for** *y utilizar el operador relacional* **<=** *auxiliará a evitar errores de diferencia por uno. Para un ciclo que se utilice para imprimir los valores del 1 al 10, por ejemplo, la condición de confirmación de ciclo debería ser* **counter <= 10** *en vez de* **counter < 11** *o bien* **counter < 10**.

El formato general de la estructura **for** es

```
for (expresión1; expresión2; expresión3)
   enunciado
```

donde *expresión1* inicializa la variable de control de ciclo, *expresión2* es la condición de continuación del ciclo, y *expresión3* incrementa la variable de control. En la mayor parte de los

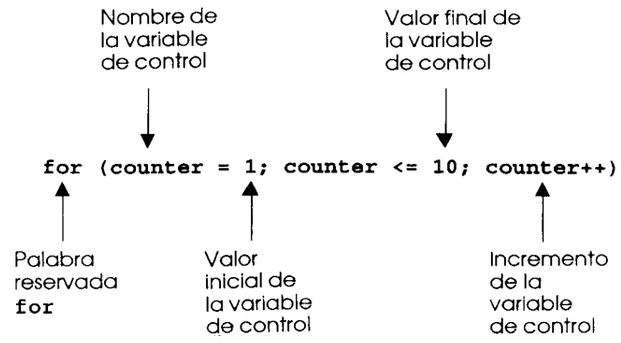


Fig. 4.3 Componentes de un encabezado típico **for**.

casos, la estructura **for** puede representarse mediante una estructura equivalente **while**, como sigue:

```
expresión1;
while (expresión2) {
   enunciado
   expresión3;
}
```

Existe una excepción para esta regla, misma que se analizará en la Sección 4.9.

A menudo, *expresión1* y *expresión3* son expresiones en listas separadas por coma. Las comas se utilizan aquí como *operadores coma,* garantizando que las listas de expresiones se evalúen de izquierda a derecha. El valor y el tipo de una lista de expresiones separada por coma, es el valor y el tipo de la expresión más a la derecha de dicha lista. El operador coma es utilizado prácticamente sólo en estructuras **for**. Su uso principal es permitir al programador utilizar múltiples expresiones de inicialización y/o múltiples incrementos. Por ejemplo, en una sola estructura **for** pudieran existir dos variables de control que deban ser inicializadas e incrementadas.

Práctica sana de programación 4.7

Coloque sólo expresiones que involucren las variables de control en las secciones de inicialización y de incremento de una estructura **for**. *Las manipulaciones de las demás variables deberían de aparecer, ya sea antes del ciclo (si se ejecutan una vez, como los enunciados de incialización) o dentro del cuerpo del ciclo (si se ejecutan una vez en cada repetición, como son los enunciados incrementales o decrementales).*

Las tres expresiones de la estructura **for** son opcionales. Si se omite *expresión2*, C supondrá que la condición es verdadera, creando por lo tanto un ciclo infinito. También se puede omitir la *expresión1*, si la variable de control se inicializa en alguna otra parte del programa. La *expresión3* podría también omitirse, si el incremento se calcula mediante enunciados en el cuerpo de la estructura **for**, o si no se requiere de ningún incremento. La expresión incremental en la estructura **for** actúa como un enunciado de C independiente, al final del cuerpo del **for**. Por lo tanto, las expresiones

```
counter = counter + 1
counter += 1
++counter
counter++
```

son equivalentes todas ellas, en la porción incremental de la estructura **for**. Muchos programadores C prefieren la forma **counter++**, porque el incremento ocurre después de que se haya ejecutado el cuerpo del ciclo. Por lo tanto, la forma postincremental parece más natural. Dado que la variable que se postincrementa o se preincrementa aquí no aparece en una expresión, ambas formas de incremento surten el mismo efecto. Los dos puntos y comas en la estructura **for** son requeridos.

Error común de programación 4.3

Usar comas en vez de puntos y coma en un encabezado **for**.

Error común de programación 4.4

Colocar un punto y coma de inmediato a la derecha de un encabezado **for** *hace del cuerpo de esta estructura* **for** *un enunciado vacío. Por lo regular esto es un error lógico*

4.5 La estructura for: notas y observaciones

1. Tanto la inicialización, como la condición de continuación de ciclo, y el incremento pueden contener expresiones aritméticas. Por ejemplo, suponga que `x = 2` y `y = 10`, el enunciado

   ```
   for (j = x;  j <= 4 * x * y; j += y / x)
   ```

   es equivalente al enunciado

   ```
   for (j = 2;  j <= 80;  j += 5)
   ```

2. El "incremento" puede ser negativo (en cuyo caso realmente se trata de un decremento, y el ciclo de hecho contará hacia atrás).
3. Si la condición de continuación de ciclo resulta falsa al inicio, la porción del cuerpo del ciclo no se ejecutará. En vez de ello, la ejecución seguirá adelante con el enunciado que siga a la estructura **for**.
4. La variable de control con frecuencia se imprime o se utiliza en cálculos en el cuerpo de un ciclo, pero esto no es necesario. Es común utilizar la variable de control para controlar la repetición, aunque jamás se mencione la misma dentro del cuerpo del ciclo.
5. El diagrama de flujo de la estructura **for** es muy similar al de la estructura **while**. Por ejemplo, el diagrama de flujo del enunciado **for**

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

   se muestra en la figura 4.4. Este diagrama de flujo deja claro que la incialización ocurre una vez y que el incremento ocurre después de que se ejecuta el enunciado del cuerpo. Advierta que (independiente de los pequeños círculos y flechas), el diagrama de flujo contiene sólo los símbolos de rectángulo y diamante. Imagine, otra vez, que el programador tiene acceso a un contenedor profundo de estructuras **for** vacías —tantas como necesite el programador para apilarlas y anidarlas con otras estructuras de control, para formar una implantación estructurada del flujo de control de un algoritmo. Y otra vez, los rectángulos y los diamantes se rellenarán con acciones y decisiones apropiadas al algoritmo.

Práctica sana de programación 4.8

Aunque el valor de la variable de control puede ser modificado en el cuerpo de un ciclo **for**, *ello podría llevarnos a errores sutiles. Lo mejor es no cambiarlo.*

4.6 Ejemplos utilizando la estructura for

Los siguientes ejemplos muestran métodos de variar en una estructura **for** la variable de control.

a) Varíe la variable de control de `1` a `100` en incrementos de `1`.

```
for (i = 1; i <= 100; i++)
```

b) Varíe la variable de control de `100` a `1` en incrementos de `-1` (decrementos de `1`).

```
for (i = 100; i >= 1; i--)
```

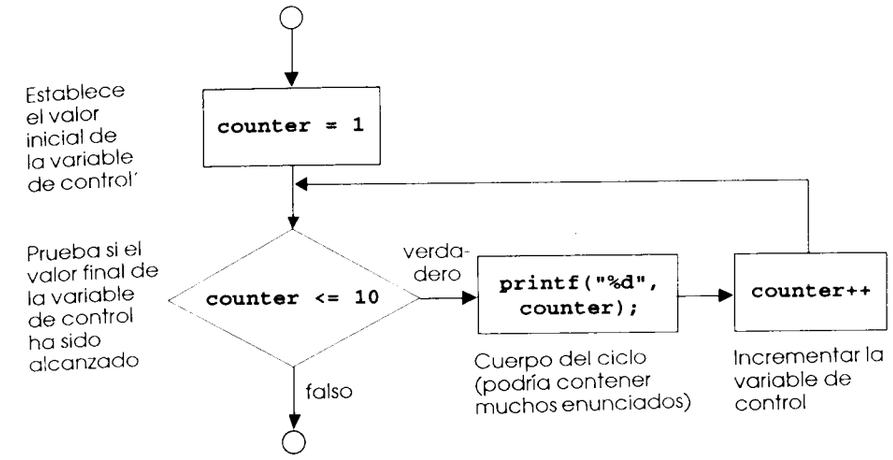



Fig. 4.4 Diagrama de flujo de una estructura **for** típica

c) Variar la variable de control de `7` a `77` en pasos de `7`.

```
for (i = 7; i <= 77; i += 7)
```

d) Variar la variable de control de `20` a `2` en pasos de `-2`

```
for (i = 20; i >= 2; i -= 2)
```

e) Variar la variable de control a lo largo de la siguiente secuencia de valores: **2, 5, 8, 11, 14, 17, 20.**

```
for (j = 2; j <= 20; j += 3)
```

f) Variar la variable de control de acuerdo a la siguiente secuencia de valores: **99, 88, 77, 66, 55, 44, 33, 22, 11, 0.**

```
for (j = 99; j >= 0; j -=11)
```

Los siguientes dos ejemplos proporcionan aplicaciones simples de la estructura **for**. El programa de la figura 4.5 utiliza la estructura **for** para sumar todos los enteros pares desde `2` hasta `100`.

Note que el cuerpo de la estructura **for** de la figura 4.5 podría de hecho combinarse en la parte más a la derecha del encabezado **for** mediante el uso del operador coma, como sigue:

```
for (number = 2; number <= 100; sum += number, number += 2)
   ;
```

La inicialización `sum = 0` también podría ser combinada en la sección de inicialización del **for**.

Práctica sana de programación 4.9

Aunque los enunciados que anteceden a un **for**, *y los enunciados del cuerpo de* **for**, *a menudo pueden ser combinados en un encabezado* **for**, *evite hacerlo, porque hace el programa más difícil de leer.*

```
/* Summation with for */
#include <stdio.h>

main()
{
   int sum = 0, number;

   for (number = 2; number <= 100; number += 2)
      sum += number;

   printf("Sum is %d\n", sum);

   return 0;
}
```

```
Sum is 2550
```

Fig. 4.5 Suma utilizando `for`.

Práctica sana de programación 4.10

Limite, si es posible, a una sola línea el tamaño de los encabezados de estructuras de control.

El ejemplo siguiente calcula el interés compuesto, utilizando la estructura **`for`**. Considere el siguiente enunciado de problema:

Una persona invierte $1000.00 en una cuenta de ahorros, que reditúa un interés del 5 %. Suponiendo que todo el interés se queda en depósito dentro de la cuenta, calcule e imprima la cantidad de dinero en la cuenta, al final de cada año, durante 10 años. Para la determinación de estas cantidades utilice la fórmula siguiente:

$$a = p(1 + r)^n$$

donde

p es la cantidad originada invertida (es decir, el principal)
r es la tasa anual de interés
n es el número de años
a es la cantidad en depósito al final del año *n*.

Este problema incluye un ciclo, que ejecuta el cálculo indicado para cada uno de los 10 años en los cuales el dinero se queda en depósito. La solución aparece en la figura 4.6.

La estructura **`for`** ejecuta 10 veces el cuerpo del ciclo, variando la variable de control de 1 a 10, en incrementos de 1. Aunque C no incluye un operador de exponenciación, podemos, sin embargo, utilizar para este fin la función estándar de biblioteca **`pow`**. La función **`pow(x, y)`** calcula el valor de **`x`**, elevado a la potencia **`y`**. Toma dos argumentos del tipo **`double`** y devuelve un valor **`double`**. El tipo **`double`** es un tipo de punto flotante parecido a **`float`**, pero una variable de tipo **`double`** puede almacenar un valor de una magnitud mucho mayor y con mayor precisión que **`float`**. Note que, siempre que una función matemática como es **`pow`** sea utilizada, deberá incluirse el archivo de cabecera **`math.h`**. De hecho, este programa funcionaría mal sin la inclusión


```
/* Calculating compound interest */
#include <stdio.h>
#include <math.h>

main()
{
   int year;
   double amount, principal = 1000.0, rate = .05;

   printf("%4s%21s\n", "Year", "Amount on deposit");

   for (year = 1; year <= 10; year++) {
      amount = principal * pow(1.0 + rate, year);
      printf("%4d%21.2f\n", year, amount);
   }

   return 0;
}
```

```
Year   Amount on deposit
1            1050.00
2            1102.50
3            1157.62
4            1215.51
5            1276.28
6            1340.10
7            1407.10
8            1477.46
9            1551.33
10           1628.89
```

Fig. 4.6 Cómo calcular interés compuesto utilizando `for`.

de **`math.h`**. La función **`pow`** requiere dos argumentos **`double`**. Note que **`year`** es un entero. El archivo **`math.h`** incluye información que le indica al compilador que convierta el valor de **`year`** a una representación temporal **`double`**, antes de llamar a la función. Esta información queda contenida en algo que se conoce como *función prototipo* **`pow`**. Las funciones prototipo son una nueva característica de importancia de ANSI C y se explican en el capítulo 5. En este capítulo 5 proporcionamos un resumen de la función **`pow`** y de otras funciones matemáticas de biblioteca.

Note que hemos declarado las variables **`amount`**, **`principal`** y **`rate`** como del tipo **`double`**. Hemos hecho lo anterior por razones de simplicidad, porque estamos manejando fracciones de dólares.

Práctica sana de programación 4.11

No utilice variables del tipo `float` o `double` para llevar a cabo cálculos monetarios. La falta de precisión de los números de punto flotante pueden causar errores, que resultarán en valores monetarios incorrectos. En los ejercicios, exploramos la utilización de enteros para la ejecución de cálculos monetarios.

A continuación una explicación sencilla de lo que podría salir mal al utilizar **float** o bien **double** para representar cantidades en dólares.

Dos cantidades en dólares **float** almacenadas en la máquina podrían ser 14.234 (que con **%.2f**, imprime como 14.23) y 18.673 (el cual con **%.2f**, imprime como 18.67). Cuando estas cantidades se suman, producen la suma 32.907, que con **%.2f**, imprime como 32.91. Por lo tanto su impresión aparecería como

```
  14.23
+ 18.67
-------
  32.91
```

la suma de los números individuales, según se ven impresos ¡debería ser 32.90! ¡Queda usted advertido!

El especificador de conversión **%21.2f** es utilizado en el programa para imprimir el valor de la variable **amount**. El **21** es el especificador de conversión que indica el ancho del campo en el cual el valor se imprimirá. Un ancho de campo de **21** define que el valor impreso aparecerá en **21** posiciones de impresión. El **2** especifica la precisión (es decir, el número de posiciones decimales). Si el número de caracteres desplegados es menor que el ancho del campo, entonces el valor quedará de forma automática *justificado a la derecha* dentro de ese campo. Esto es en particular útil para alinear valores de punto flotante que tengan la misma precisión. A fin de *justificar a la izquierda* un valor en un campo, coloque un - (signo menos) entre el % y el ancho del campo. Note que el signo menos, también puede ser utilizado para alinear a la izquierda a los enteros (como en **%-6d**) y a las cadenas de caracteres (como en **%-8s**). En el capítulo 9 analizaremos con todo detalle las poderosas capacidades de formato de **printf** y de **scanf**.

4.7 La estructura de selección múltiple switch

En el capítulo 3, analizamos la estructura de una selección **if**, y la estructura de doble selección **if/else**. En forma ocasional, un algoritmo contendrá una serie de decisiones, en las cuales una variable o expresión se probará por separado contra cada uno de los valores constantes enteros que puede asumir, y se tomarán diferentes acciones. Para esta forma de toma de decisiones se proporciona una estructura de selección múltiple **switch** C.

La estructura **switch** está formada de una serie de etiquetas **case**, y de un caso opcional **default**. El programa en la figura 4.7 utiliza **switch** para contar el número de cada distinta letra de calificación que los estudiantes alcanzaron en un examen.

En el programa, el usuario escribe las calificaciones, en letras, correspondientes a una clase. Dentro del encabezado **while**,

```
while ( ( grade = getchar() ) != EOF)
```

la asignación entre paréntesis **(grade = getchar())** se ejecuta en primer término. La función **getchar** (proveniente de la biblioteca estándar de entrada/salida) lee un carácter del teclado y almacena este carácter en la variable entera **grade**. Los caracteres se almacenan por lo regular en variables del tipo ***char***. Sin embargo, una característica importante de C, es que los caracteres pueden ser almacenados en cualquier tipo de dato entero, porque en la computadora son representados como enteros de un byte. Por lo tanto, podemos tratar a un carácter como si fuera ya sea o un entero, o un carácter, dependiendo de su uso. Por ejemplo, el enunciado

```
printf("The character (%c) has the value %d.\n", 'a', 'a');
```



```
/* Counting letter grades */
#include <stdio.h>

main()
{
   int grade;
   int aCount = 0, bCount = 0, cCount = 0,
       dCount = 0, fCount = 0;

   printf("Enter the letter grades.\n");
   printf("Enter the EOF character to end input.\n");

   while ( ( grade = getchar() ) != EOF) {

      switch (grade) {    /* switch nested in while */

         case 'A': case 'a':  /* grade was uppercase A */
            ++aCount;         /* or lowercase a */
            break;

         case 'B': case 'b':  /* grade was uppercase B */
            ++bCount;         /* or lowercase b */
            break;

         case 'C': case 'c':  /* grade was uppercase C */
            ++cCount;         /* or lowercase c */
            break;

         case 'D': case 'd':  /* grade was uppercase D */
            ++dCount;         /* or lowercase d */
            break;

         case 'F': case 'f':  /* grade was uppercase F */
            ++fCount;         /* or lowercase f */
            break;

         case '\n': case ' ': /* ignore these in input */
            break;

         default:        /* catch all other characters */
            printf("Incorrect letter grade entered.");
            printf(" Enter a new grade.\n");
            break;
      }
   }

   printf("\nTotals for each letter grade are:\n");
   printf("A: %d\n", aCount);
   printf("B: %d\n", bCount);
   printf("C: %d\n", cCount);
   printf("D: %d\n", dCount);
   printf("F: %d\n", fCount);

   return 0;
}
```

Fig. 4.7 Un ejemplo del uso de **switch** (parte 1 de 2).

```
Enter the letter grades.
Enter the EOF character to end input
A
B
C
C
A
D
F
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
B

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 4.7 Un ejemplo del uso de **switch** (parte 2 de 2).

utiliza los especificadores de conversión **%c** y **%d**, para imprimir el carácter **a** y su valor como entero, respectivamente. El resultado es

```
The character (a) has the value 97.
```

El entero 97 es la representación numérica del carácter, dentro de la computadora. Muchas computadoras hoy día utilizan el *conjunto de caracteres ASCII (American Standard Code for Information Interchange),* en el cual el 97 representa la letra minúscula **'a'**. En el Apéndice D se presenta una lista de los caracteres ASCII y sus valores decimales. Los caracteres pueden ser leídos utilizando **scanf** mediante el uso del especificador de conversión %c.

De hecho todos los enunciados de asignación en conjunto tienen un valor. Este es precisamente el valor que se asigna a la variable del lado izquierdo del signo de =. El valor de la asignación **grade = getchar()** es el carácter que es regresado por **getchar** y asignado a la variable **grade**.

El hecho que los enunciados de asignación tengan valores puede ser útil para inicializar varias variables con un mismo valor. Por ejemplo,

```
a = b = c = 0;
```

primero evalúa la asignación **c = 0** (porque el operador = se asocia de derecha a izquierda). A la variable **b**, a continuación, se le asigna el valor de la asignación **c = 0** (que es 0). Entonces, a la variable **a** se le asigna el valor de la asignación **b = (c = 0)** (lo que también es 0). En el programa, el valor de la asignación **grade = getchar()** se compara con el valor de **EOF**, (un

símbolo cuya siglas significan "fin de archivo"). Utilizamos **EOF** (que tiene por lo regular el valor de -1) como valor centinela. El usuario escribe una combinación de tecleos, dependiendo del sistema, que signifiquen "fin de archivo", es decir, "ya no tengo más datos a introducir". **EOF** es una constante simbólica entera, definida en el archivo de cabecera **<stdio.h>** (en el capítulo 6 veremos cómo son definidas las constantes simbólicas). Si el valor asignado a **grade** es igual a **EOF**, el programa se termina. En este programa hemos decidido representar caracteres como int, porque **EOF** tiene un valor entero (otra vez, lo normal es -1).

Sugerencia de portabilidad 4.1

Las combinaciones de teclas para la introducción de **EOF** *(fin de archivo) son dependientes del sistema.*

Sugerencia de portabilidad 4.2

Hacer la prueba buscando la constante simbólica **EOF**, *en vez de buscar -1, hace más portátiles a los programas. El estándar ANSI indica que* **EOF** *es un valor entero negativo (pero no necesariamente -1). Por lo tanto,* **EOF** *pudiera tener diferentes valores en diferentes sistemas*

En sistemas UNIX y en muchos otros, el indicador **EOF** se introduce escribiendo la secuencia

<return> <ctrl-d>

Esta notación significa que se oprima la tecla de entrar y de forma simultánea se presione tanto la tecla **ctrl** como la tecla **d**. En otros sistemas, como en el VAX VMS de Digital Equipment Corporation, o en el MS-DOS de Microsoft Corporation, el indicador **EOF** puede ser introducido escribiendo

<ctrl-z>

El usuario introduce las calificaciones con el teclado. Cuando se oprime la tecla de retorno (o de entrar) los caracteres son leídos por la función **getchar**, un carácter a la vez. Si el carácter introducido no es igual a **EOF**, se introduce en la estructura **switch**. La palabra reservada **switch** es seguida por el nombre de variable **grade** entre paréntesis. Esto se conoce como la *expresión de control.* El valor de esta expresión es comparado con cada una de las *etiquetas case*. Suponga que el usuario ha escrito la letra **C** como calificación. La **C** automáticamente se compara con cada uno de los **case** dentro de **switch**. Si ocurre una coincidencia (**case 'C':**), se ejecutarán los enunciados correspondientes a dicho **case**. En el caso de la letra **C**, **cCount** se incrementará en **1**, y de inmediato mediante el enunciado **break** se sale de la estructura **switch**.

El enunciado **break** causa que el control de programa continúe con el primer enunciado que sigue después de la estructura **switch**. Se utiliza el enunciado **break**, porque de lo contrario los **case**s en un enunciado **switch** se ejecutarían juntos. Si en alguna parte de una estructura **switch** no se utiliza **break**, entonces, cada vez que ocurre una coincidencia en la estructura, se ejecutarían todos los enunciados de los **case**s remanentes. (Esta característica rara vez es de utilidad, aunque resulta perfecta para programar la canción iterativa "¡The twelve Days of Christmas!"). Si no existe coincidencia, el caso **default** es ejecutado y se imprime un mensaje de error.

Cada **case** puede tener una o más acciones. La estructura **switch** es diferente de todas las demás estructuras, en el sentido de que no se requieren llaves alrededor de varias acciones en un **case** de un **switch**. La estructura general de selección múltiple **switch** (que utilice a un **break** en cada **case**) tiene un diagrama de flujo como el de la figura 4.8.

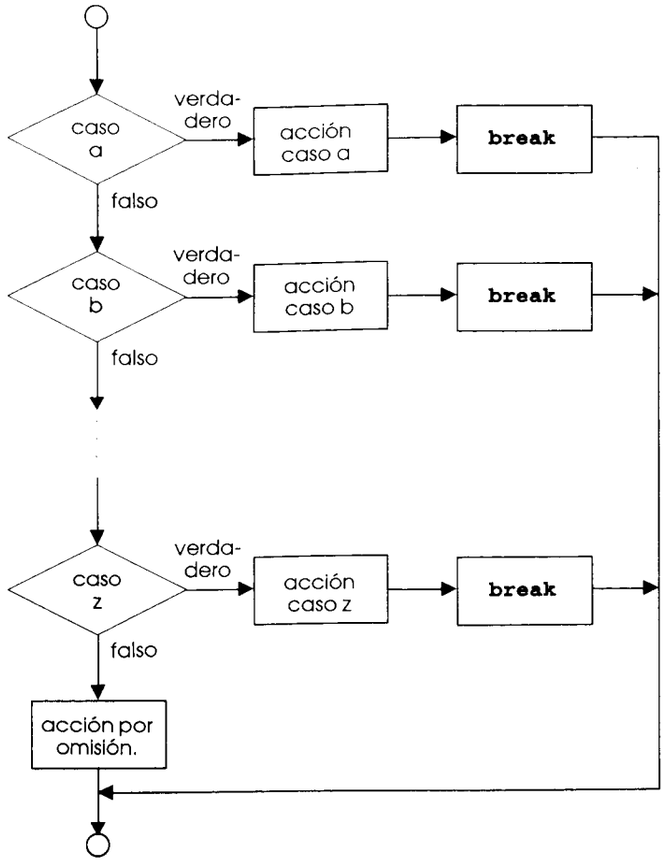


Fig. 4.8 La estructura de selección múltiple **switch**.

El diagrama de flujo deja claro que cada enunciado **break**, al final de un **case**, hace que el control dé salida de inmediato a la estructura **switch**. Otra vez, advierta que (independientemente de los pequeños círculos y flechas) el diagrama de flujo sólo contiene símbolos rectángulo y diamante. Imagínese, otra vez, que el programador tiene acceso a un contenedor profundo de estructuras **switch** vacías —tantas como pudiera necesitar el programador para apilarlas y anidarlas junto con otras estructuras de control, para formar una implantación estructurada del flujo de control de un algoritmo. Y de nuevo, los rectángulos y los diamantes a continuación serán llenados con acciones y decisiones apropiadas al algoritmo.

Error común de programación 4.5

Olvidar en una estructura **switch** *un enunciado* **break**, *cuando se requiere de uno.*

Práctica sana de programación 4.12

Proveer un caso **default** *en enunciados* **switch**. *En un* **switch** *serán ignorados los casos no aprobados de forma explícita. El caso* **default** *ayuda a evitar lo anterior, enfocando al programador sobre la necesidad de procesar condiciones excepcionales. Existen situaciones en las cuales no se requiere de procesamiento* **default**.

Práctica sana de programación 4.13

Aunque las cláusulas **case** *y la cláusula de caso* **default** *en una estructura* **switch** *pueden ocurrir en cualquier orden, se considera una práctica sana de programación colocar al final la cláusula* **default**.

Práctica sana de programación 4.14

En una estructura **switch**, *cuando la cláusula* **default** *se enlista al final, el enunciado* **break** *no es requerido. Pero algunos programadores incluyen este* **break**, *para fines de claridad y simetría con otros* **cases**.

En la estructura **switch** de la figura 4.7, las líneas

```
case '\n': case ' ':
   break;
```

hacen que el programa se salte los caracteres de nueva línea y de espacios vacíos. La lectura de caracteres uno a la vez puede causar ciertos problemas. Para que el programa lea los caracteres, deben ser enviados a la computadora oprimiendo la *tecla de entrar* del teclado. Esto hace que se coloque un carácter de nueva línea en la entrada, después del carácter que deseamos procesar. A menudo, este carácter de nueva línea debe ser procesado en forma especial, para que el programa funcione de manera correcta. Al incluir los casos precedentes en nuestra estructura **switch**, evitamos que en el caso **default** se imprima el mensaje de error, cada vez que se encuentra en la entrada un carácter de nueva línea o de espacio.

Error común de programación 4.6

No procesar los caracteres de nueva línea en la entrada al leer los caracteres uno a la vez, puede ser causa de errores lógicos.

Práctica sana de programación 4.15

Recuerde proporcionar capacidades de proceso para los caracteres de nueva línea en la entrada, cuando se procesen caracteres uno a la vez.

Note que varias etiquetas de caso enlistadas juntas (como **case 'D':case 'd':** en la figura 4.7) significan que el mismo conjunto de acciones ocurrirá para cualquiera de estos casos.

Al utilizar la estructura **switch**, recuerde que puede ser usada sólo para probar una *expresión integral constante*, es decir, cualquier combinación de constantes de carácter y de constantes enteros que tengan un valor constante entero. Una constante de carácter se representa como un carácter específico, entre comillas sencillas como es **'A'**. Los caracteres deben ser encerrados dentro de comillas sencillas para que sean reconocidos como constantes de carácter. Las constantes enteras son solo valores enteros. En nuestro ejemplo hemos utilizado constantes de carácter. Recuerde que los caracteres de hecho son valores pequeños enteros.

Los lenguajes portátiles, como C, deben tener tamaños flexibles de tipos de datos. Diferentes aplicaciones pudieran necesitar enteros de tamaños diferentes. C proporciona varios tipos de datos para representar enteros. El rango de los valores de enteros para cada tipo depende del hardware particular de cada computadora. Además de los tipos **int** y **char**, C proporciona los tipos **short** (que es una abreviatura de **short int**) y **long** (que es una abreviatura de **long int**). El estándar ANSI especifica que el rango mínimo de valores para enteros **short** es ±32767. Para la gran mayoría de los cálculos enteros, los enteros **long** son suficientes. El estándar define que el rango mínimo de valores para los enteros **long** es ±2147483647. En la mayoría de las computadoras, los **int** son equivalentes ya sea a **short** o a **long**. El estándar indica que el rango de valores para un **int**, es por lo menos igual al rango de los enteros **short** y no mayor que el rango de los enteros **long**. El tipo de dato **char** puede ser utilizado para representar enteros en el rango de ±127, o cualquiera de los caracteres del conjunto de caracteres de la computadora.

Sugerencia de portabilidad 4.3

Dado que de un sistema a otro **int** *varía en tamaño, utilice enteros* **long**, *si espera procesar enteros fuera del rango ±32767, y desearía ser capaz de ejecutar el programa en varios sistemas de cómputo diferentes.*

Sugerencia de rendimiento 4.1

En situaciones orientadas a rendimiento, donde la memoria está escasa o se requiere de velocidad, pudiera ser deseable utilizar tamaños de enteros más pequeños.

4.8 La estructura de repetición do/while

Las estructura de repetición **do/while** es similar a la estructura **while**. En la estructura **while**, la condición de continuación de ciclo se prueba al principio del ciclo, antes de ejecutarse el cuerpo del mismo. La estructura **do/while** prueba la condición de continuación del ciclo, *después* de ejecutar el cuerpo del ciclo y, por lo tanto, el cuerpo del ciclo se ejecutará por lo menos una vez. Cuando termina **do/while**, la ejecución continuará con el enunciado que aparezca después de la cláusula **while**. Note que en la estructura **do/while** no es necesario utilizar llaves, si en el cuerpo existe sólo un enunciado. Sin embargo, por lo regular las llaves se utilizan, para evitar confusión entre las estructuras **while** y **do/while**. Por ejemplo,

```
while (condición)
```

se considera normalmente como el encabezado de una estructura **while**. Un **do/while** sin llaves rodeando el cuerpo de un solo enunciado, aparece como

```
do
   enunciado
while (condición);
```

lo que podría ser motivo de confusión. La última línea **while** (*condición*) **;** pudiera ser interpretada de forma equivocada por el lector como una estructura **while** conteniendo un enunciado vacío. Por lo tanto, el **do/while** con un enunciado, a menudo se escribe como sigue, a fin de evitar confusión:


```
do {
enunciado
} while (condición);
```

Práctica sana de programación 4.16

Algunos programadores incluyen siempre llaves en una estructura **do/while**, *aún si las llaves no son necesarias. Esto ayuda a eliminar ambigüedad entre la estructura* **do/while** *con un enunciado, y la estructura* **while**.

Error común de programación 4.7

Se generarán ciclos infinitos en una estructura **while**, **for** *o bien* **do/while** *cuando la condición de continuación de ciclo nunca se convierte en falsa. A fin de evitar lo anterior, asegúrese de que no exista un punto y coma inmediatamente después del encabezado de una estructura* **while** *o* **for**. *En un ciclo controlado por contador, asegúrese que la variable de control es incrementada (o decrementada) en el cuerpo del ciclo. En un ciclo controlado por centinela, asegúrese que el valor centinela es eventualmente introducido.*

El programa de la figura 4.9 utiliza una estructura **do/while** para imprimir los números del 1 al 10. Note que la variable de control **counter** es preincrementada en la prueba de continuación de ciclo. Note también la utilización de llaves para encerrar el cuerpo de un solo enunciado del **do/while**.

La estructura del **do/while** se diagrama en la figura 4.10. Este diagrama de flujo demuestra que la condición de continuación de ciclo no se ejecutará sino hasta después de que la acción se lleve a cabo por lo menos una vez. De nuevo, note (que además de los pequeños círculos y flechas) el diagrama de flujo sólo contiene un símbolo de rectángulo y uno de diamante. Imagine, repetimos, que el programador tiene acceso a un contenedor profundo de estructuras **do/while** vacías tantas como pudiera necesitar el programador para apilarlas y anidarlas junto con otras estructuras de control, para formar una implantación estructurada del flujo de control de un algoritmo. Y otra vez, los rectángulos y diamantes serán entonces llenados con acciones y decisiones apropiadas a dicho algoritmo.

```
/* Using the do/while repetition structure */
#include <stdio.h>

main()
{
   int counter = 1;

   do {
      printf("%d  ", counter);
   } while (++counter <= 10);

   return 0;
}
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.9 Cómo usar la estructura **do/while**.

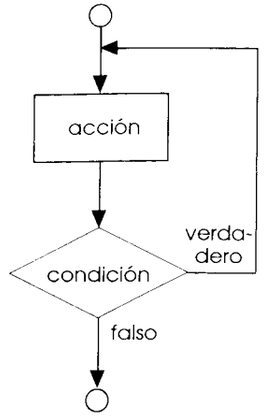


Fig. 4.10 La estructura de repetición do/while.

4.9 Los enunciados break y continue

Los enunciados **break** y **continue** se utilizan para modificar el flujo de control. El enunciado **break**, cuando se utiliza o ejecuta en una estructura **while, for, do/while,** o **switch**, causa la salida inmediata de dicha estructura. La ejecución del programa continúa con el primer enunciado después de la estructura. Los usos comunes del enunciado **break** son para escapar en forma prematura de un ciclo, o para saltar el resto de una estructura **switch** (como en la figura 4.7). La figura 4.11 demuestra el enunciado **break** en una estructura de repetición **for.** Cuando la estructura **if** detecta que **x** se ha convertido en **5**, se ejecuta **break**. Esto da por terminado el enunciado **for,** y el programa continúa con el **printf** existente después de **for**. El ciclo se ejecuta por completo cuatro veces.

El enunciado **continue**, cuando se ejecuta en una estructura **while, for, do/while,** salta los enunciados restantes del cuerpo de dicha estructura, y ejecuta la siguiente iteración del ciclo. En estructuras **while** y **do/while,** la prueba de continuación de ciclo se valúa de inmediato de que se haya ejecutado el enunciado **continue**. En la estructura **for**, se ejecuta la expresión incremental, y a continuación se valúa la prueba de continuación de ciclo. Anteriormente indicamos que la estructura **while** podría ser utilizada, en la mayor parte de los casos, para representar la estructura **for**. Una excepción ocurre cuando la expresión incremental en la estructura **while** sigue a un enunciado **continue**. En este caso, el incremento no se lleva a cabo antes de la prueba de la condición de continuación de repetición, y no se ejecuta el **while** de la misma forma que en el caso del **for**. En la figura 4.2 se utiliza el enunciado **continue** en una estructura **for,** para saltar el enunciado **printf** de la estructura, y empezar con la siguiente iteración del ciclo.

Práctica sana de programación 4.17

Algunos programadores son de la opinión que **break** *y* **continue** *violan las normas de la programación estructurada. Dado que los efectos de estos enunciados pueden ser obtenidos mediante técnicas de programación estructuradas que aprenderemos pronto, estos programadores no utilizan* **break** *ni* **continue**.


```
/* Using the break statement in a for structure */
#include <stdio.h>

main()
{
   int x;

   for (x = 1; x <= 10; x++) {

      if (x == 5)
         break;     /* break loop only if x == 5 */

      printf("%d ", x);
   }

   printf("\nBroke out of loop at x == %d\n", x);
   return 0;
}
```

```
1 2 3 4
Broke out of loop at x == 5
```

Fig. 4.11 Cómo utilizar el enunciado break en una estructura for.

```
/* Using the continue statement in a for structure */
#include <stdio.h>

main()
{
   int x;

   for (x = 1; x <= 10; x++) {

      if (x == 5)
         continue;  /* skip remaining code in loop only
                       if x == 5 */

      printf("%d ", x);
   }

   printf("\nUsed continue to skip printing the value 5\n");
   return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 4.12 Cómo utilizar el enunciado continue en una estructura for.

Sugerencia de rendimiento 4.2

Los enunciados `break` *y* `continue`*, cuando son usados de forma adecuada, se ejecutan más aprisa que las técnicas estructuradas correspondientes que pronto aprenderemos.*

Observación de ingeniería de software 4.1

Existe cierta contraposición entre alcanzar ingeniería de software de calidad y conseguir software de mejor rendimiento. A menudo una de estas metas se consigue a expensas de la otra.

4.10 Operadores lógicos

Hasta ahora hemos estudiado sólo *condiciones simples*, como son `counter < = 10`, `total >1000` y `number != sentinelValue`. Hemos expresado estas condiciones en términos de los operadores relacionales >, <, >=, y <=, y de los operadores de igualdad == y !=. Cada decisión probaba precisamente una sola condición. Si quisiéramos probar varias condiciones en el proceso de llevar a cabo una decisión, tendríamos que ejecutar estas pruebas en enunciados por separado, o en estructuras anidadas `if`, o bien `if/else`.

C proporciona *operadores lógicos,* que pueden ser utilizados para formar condiciones más complejas, al combinar condiciones simples. Los operadores lógicos son `&&` (*el AND lógico*), `||` (*OR lógico*), y `!` (*NOT lógico* también conocido como *negación lógica*). Estudiaremos ejemplos de cada uno de los anteriores.

Suponga que deseamos asegurarnos, en algún momento en un programa, que dos condiciones son *ambas* verdaderas, antes de que seleccionemos una cierta trayectoria de ejecución. En este caso podemos utilizar el operador lógico `&&`, como sigue:

```
if (gender == 1 && age = 65)
   ++seniorFemales;
```

Este enunciado `if` contiene dos condiciones simples. La condición `gender == 1` pudiera ser evaluada, por ejemplo, a fin de determinar si una persona es mujer. La condición `age >= 65` se evalúa para determinar si la persona es un ciudadano senior. Las dos condiciones simples se evalúan primero, porque las precedencias de == y de >= ambas son más altas que la precedencia de `&&`. A continuación, el enunciado `if` considera la condición combinada.

```
gender == 1 && age >= 65
```

Esta condición es verdadera si y sólo si ambas condiciones simples son verdaderas. Por último, si esta condición combinada es de hecho verdadera, entonces el contador de `seniorFemales` se incrementará en `1`. Si cualquiera o ambas de las condiciones simples son falsas, entonces el programa salta la incrementación y prosigue al enunciado que siga al `if`.

La tabla de la figura 4.13 resume el operador `&&`. La tabla muestra las cuatro posibles combinaciones de valores cero (falso) y no cero (verdadero) correspondientes a la expresión1 y a la expresión2. Estos tipos de tablas a menudo se conocen como *tablas de la verdad.* C evaluará todas las expresiones que incluyan operadores relacionales, operadores de igualdad, y operadores lógicos en 0 o en 1. Aunque C defina como 1 un valor verdadero, aceptará como verdadero *cualquier* valor no cero.

Consideremos ahora el operador `||` (OR lógico). Suponga que deseamos asegurarnos, en determinado momento de un programa, que alguna *o* ambas de dos condiciones son verdaderas,

expresión1	expresión2	expresión1 && expresión2
0	0	0
0	no cero	0
no cero	0	0
no cero	no cero	1

Fig. 4.13 Tabla de la verdad para el operador `&&` (AND lógico).

antes de que seleccionemos un cierto camino de ejecución. En este caso utilizaremos el operador `||`, como en el siguiente segmento de programa:

```
if (semesterAverage >= 90 || finalExam >= 90)
   printf("Student grade is A\n");
```

Este enunciado también contiene dos condiciones simples. La condición `semesterAverage >= 90` se evalúa para determinar si el estudiante merece un "A" en el curso, debido a un rendimiento sólido a todo lo largo del semestre. La condición `finalExam >= 90` se evalúa para determinar si el estudiante merece una calificación de "A" en el curso, debido a un rendimiento extraordinario durante el examen final. El enunciado `if`, a continuación analiza la condición combinada

```
semesterAverage >= 90 || finalExam >= 90
```

y califica al estudiante con una "A", si cualquiera o ambas de las condiciones simples es verdadera. Note que el mensaje "`Student grade is A`" no se imprimirá salvo que ambas condiciones simples resulten falsas (cero). La figura 4.14 es una tabla de la verdad correspondiente al operador OR lógico (`||`).

El operador `&&` tiene una precedencia más alta que `||`. Ambos operadores se asocian de izquierda a derecha. Una expresión que contenga `&&` o `||` sólo se evalúa en tanto no se conozca la verdad o la falsedad. Entonces, la evaluación de la condición

```
gender == 1 && age >= 65
```

se detendrá, si `gender` no es igual a `1` (es decir, que toda la expresión es falsa), y sólo continuará si `gender` es igual a `1` (es decir, toda la expresión podría aún ser verdadera si `age >= 65`).

expresión1	expresión2	expresión1 || expresión2
0	0	0
0	no cero	1
no cero	0	1
no cero	no cero	1

Fig. 4.14 Tabla de la verdad para el operador OR lógico (`||`).

Sugerencia de rendimiento 4.3

En expresiones que utilicen el operador `&&`, haga que la condición que más probabilidades tenga de ser falsa sea la que aparezca más a la izquierda. En expresiones que utilicen al operador `||`, haga que la condición que más probabilidades tenga de ser verdadera, sea la condición más a la izquierda. Esto puede reducir el tiempo de ejecución del programa.

C proporciona el signo de `!` (negación lógica) para permitir a un programador el "invertir" el significado de una condición. A diferencia de los operadores `&&` y `||`, que combinan dos condiciones (y que por lo tanto son operadores binarios), el operador de negación lógica tiene como operando una condición (y, por lo tanto, es un operador unario). El operador de negación lógico se coloca antes de una condición, cuando estemos interesados en escoger un camino de ejecución si resulta falsa la condición original (antes del operador de negación lógico), como en el siguiente segmento de programa:

```
if (!(grade == sentinelValue))
printf("The next grade is %f\n", grade);
```

Se requiere el paréntesis que rodea la condición `grade == sentinelValue`, porque el operador de negación lógico tiene una precedencia más alta que el operador de igualdad. La figura 4.15 es una tabla de la verdad correspondiente al operador lógico de negación.

En la mayor parte de los casos, expresando la condición en forma distinta y utilizando un operador relacional apropiado el programador puede evitar el uso de la negación lógica. Por ejemplo, el enunciado precedente pudiera verse también escrito como sigue:

```
if (grade != sentinelValue)
printf("The next grade is %f\n", grade);
```

La tabla en la figura 4.16 muestra la precedencia y asociatividad de los operadores de C estudiados hasta este momento. Los operadores se muestran de arriba abajo en orden decreciente de precedencia.

4.11 Confusión entre los operadores de igualdad (==) y de asignación (=)

Existe un tipo de error que los programadores de C, independiente de su experiencia, tienden a hacer con tanta frecuencia que sentimos que merece una sección por separado. Este error es el intercambio accidental de los operadores == (igualdad) y = (asignación). Lo que hace tan dañinos estos intercambios es el hecho de que, de forma ordinaria, no causan errores de sintaxis. En vez de ello, por lo regular los enunciados que contienen estos errores compilan en forma correcta, y los programas se ejecutan hasta su terminación, generando probablemente resultados incorrectos, debido a errores lógicos en tiempo de ejecución.

expresión	!expresión
0	1
nonzero	0

Fig. 4.15 Tabla de la verdad para el operador ! (negación lógica).

Operadores	Asociatividad	Tipo
`()`	de izquierda a derecha	paréntesis
`++ -- + - ! (tipo)`	de derecha a izquierda	unario
`* / %`	de izquierda a derecha	multiplicativo
`+ -`	de izquierda a derecha	aditivo
`< <= > >=`	de izquierda a derecha	relacional
`== !=`	de izquierda a derecha	igualdad
`&&`	de izquierda a derecha	AND lógico
`||`	de izquierda a derecha	OR lógico
`?:`	de derecha a izquierda	condicional
`= += -= *= /= %=`	de derecha a izquierda	asignación
`,`	de izquierda a derecha	coma

Fig. 4.16 Precedencia y asociatividad de operadores.

Existen dos aspectos de C que causan estos problemas. Uno es que cualquier expresión en C que produzca un valor, puede ser utilizada en la porción decisiva de cualquier estructura de control. Si el valor es 0, se trata como falso, y si el valor no es 0 se trata como verdadero. El segundo es que las asignaciones en C producen un valor, es decir el valor asignado a la variable en el lado izquierdo del operador de asignación. Por ejemplo, suponga que tratamos de escribir

```
if (payCode == 4)
printf("You get a bonus!");
```

pero accidentalmente escribimos

```
if (payCode = 4)
printf("You get a bonus!");
```

El primer enunciado `if` concede de forma correcta una bonificación a la persona cuyo código de nómina es igual a 4. El segundo enunciado `if` —aquel que tiene el error— evalúa la expresión de asignación en la condición `if`. Esta expresión es una asignación simple, cuyo valor es la constante 4. Dado que cualquier valor no cero se interpreta como "verdadero", la condición en este enunciado `if` será siempre verdadera, y la persona recibirá siempre una bonificación ¡independiente de cuál sea el verdadero código de nómina!

Error común de programación 4.8

Utilizar el operador `==` para asignación, o bien utilizar el operador `=` para igualdad.

Los programadores por lo regular escriben condiciones como `x == 7` con el nombre de la variable a la izquierda y la constante a la derecha. Al invertir éstas, de forma tal que la constante esté a la izquierda y el nombre de la variable a la derecha, como en `7 == x`, el programador que de forma accidental remplaza el operador `==` con el `=` quedará protegido por el compilador. El compilador

tratará lo anterior como un error de sintaxis, porque en el lado izquierdo de un enunciado de asignación sólo puede colocarse un nombre de variable. Como mínimo, esto evitará el daño potencial de un error lógico en tiempo de ejecución.

Los nombres de variables se dice que son *lvalues* (por "left values") porque pueden ser utilizados en el lado izquierdo de un operador de asignación. Las constantes se dice que son *rvalues* (por "right values") porque sólo pueden ser utilizadas en el lado derecho de un operador de asignación. Note que los lvalues también pueden ser utilizados como rvalues, pero no al revés.

Práctica sana de programación 4.18

Cuando una expresión de igualdad tiene una variable y una constante como en `x == 1`*, algunos programadores prefieren escribir la expresión con la constante a la izquierda y el nombre variable a la derecha, como protección contra un error lógico, que ocurriría cuando el programador remplaza de forma accidental el operador* `==` *por signo de* `=`*.*

El otro lado de la moneda puede resultar igual de desagradable. Suponga que el programador desea asignar un valor a una variable con un enunciado simple como

```
x = 1;
```

pero en vez de ello escribe

```
x == 1;
```

Aquí, tampoco, hay error de sintaxis. En vez de ello, el compilador simplemente evalúa la expresión condicional. Si `x` es igual a `1`, la condición es verdadera y la expresión devolverá el valor 1. Si `x` no es igual a `1`, la condición es falsa y la expresión regresará el valor 0. Independientemente del valor que se regrese, no ha habido un valor de asignación, por lo que el valor simplemente se perderá, y el valor de `x` se conservará inalterado, causando probablemente un error lógico en tiempo de ejecución. ¡Desafortunadamente, en este problema no tenemos un truco a la mano disponible para auxiliarle!

4.12 Resumen de la programación estructurada

Al igual que los arquitectos diseñan edificios, empleando la sabiduría colectiva de su profesión, así deberían los programadores diseñar los programas. Nuestro campo es más joven que el de la arquitectura, y nuestra sabiduría colectiva es considerablemente menor. Hemos aprendido mucho en apenas cinco décadas. Pero quizás de mayor importancia, hemos aprendido que la programación estructurada produce programas que son más fáciles (que los programas no estructurados) de entender y, por lo tanto, más fáciles de probar, depurar, modificar e inclusive de comprobar su corrección en un sentido matemático.

Los capítulos 3 y 4 se han concentrado en las estructuras de control de C. Se ha presentado cada estructura, su diagrama de flujo y su análisis, por separado, junto con ejemplos. Ahora, resumimos los resultados de los capítulos 3 y 4 e introducimos un conjunto sencillo de reglas para la formación y las propiedades de los programas estructurados.

La figura 4.17 resume las estructuras de control analizadas en los capítulos 3 y 4. Los pequeños círculos se utilizan en la figura para indicar un solo punto de entrada y un solo punto de salida de cada estructura. El conectar de forma arbitraria símbolos individuales de diagramas de flujo puede llevar a programas no estructurados. Por lo tanto, la profesión de la programación ha decidido combinar los símbolos de diagramación de flujo para formar un conjunto limitado de estructuras de control, y elaborar sólo programas estructurados mediante la combinación adecuada de

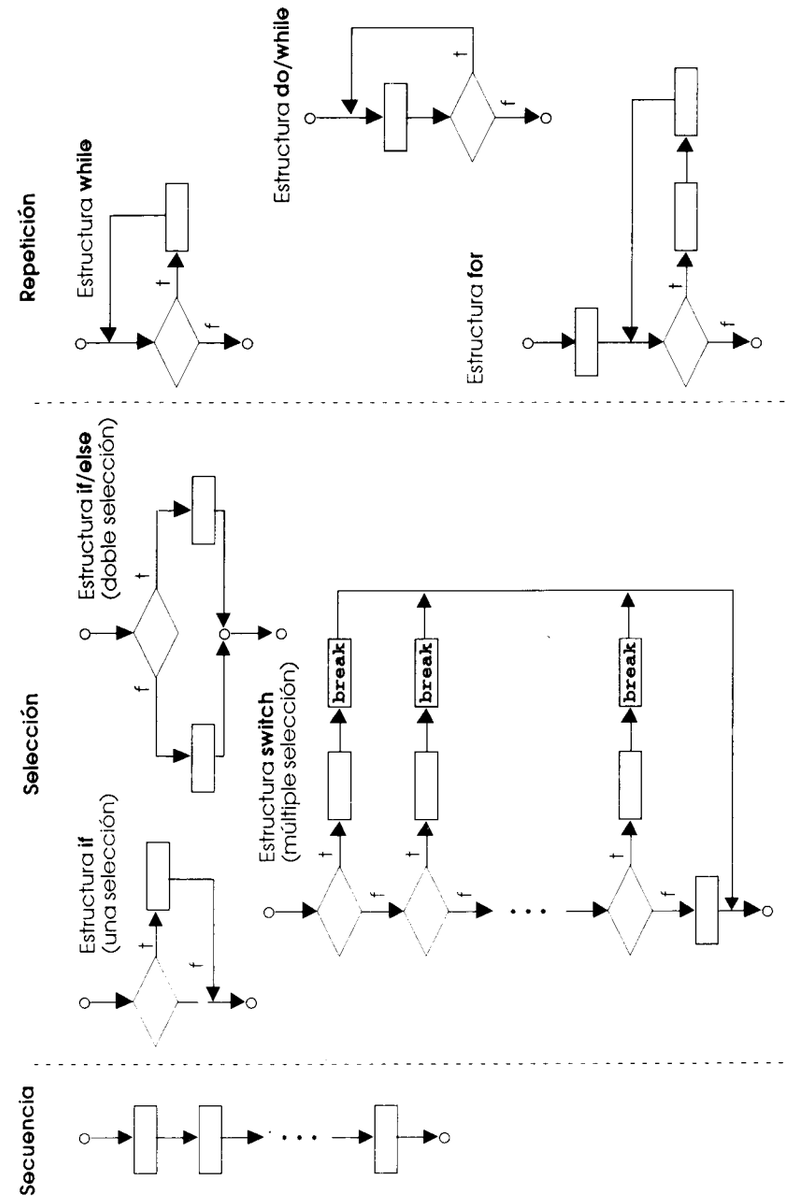


Fig. 4.17 Estructuras de una entrada/una salida de secuencia, de selección y de repetición de C.

estructuras de control en sólo dos formas simples. Por simplicidad, sólo se utilizan estructuras de control de una entrada/una salida sólo hay una forma de entrar y una de salir en cada estructura

de control. Es sencilla la conexión en secuencia de las estructuras de control para formar programas estructurales el punto de salida de una estructura de control se conecta de forma directa al punto de entrada de la siguiente estructura de control, es decir dentro de un programa las estructuras de control son simplemente colocadas una después de la otra; le hemos llamado a esto "apilar estructuras de control". Las reglas para formar programas estructurados, también permite anidar las estructuras de control.

En la figura 4.18 se muestran las reglas para la formación de programas adecuadamente estructurados. Las reglas suponen que el símbolo rectángulo de diagramación de flujo puede ser utilizado para indicar cualquier acción, incluyendo entrada/salida.

La aplicación de las reglas de la figura 4.18 siempre resulta en un diagrama de flujo estructurado con una apariencia nítida y de bloques constructivos. Por ejemplo, la aplicación repetida de la regla 2 al diagrama de flujo más simple, resulta en un diagrama de flujo estructurado, que contiene muchos rectángulos en secuencia (figura 4.20). Note que la regla 2 genera una pila de estructuras de control; por lo tanto llamaremos a la regla 2 *regla de apilamiento.*

La regla 3 se conoce como la *regla de anidamiento.* La aplicación repetida de la regla 3 al diagrama de flujo más simple resulta en un diagrama de flujo con estructuras de control nítida anidadas. Por ejemplo, en la figura 4.21, el rectángulo en el diagrama de flujo más simple, primero se remplaza con una estructura de doble selección (`if/else`). A continuación se vuelve a aplicar la regla tres a ambos rectángulos de la estructura de doble selección, remplazando cada uno de estos rectángulos por estructuras de doble selección. Los recuadros punteados alrededor de cada una de las estructuras de doble selección, representan el rectángulo que fue remplazado.

Reglas para la formación de programas estructurados

1) Empiece con el "diagrama de flujo más simple" (figura 4.19)
2) Cualquier rectángulo (acción) puede ser remplazado por dos rectángulos (acciones) en secuencia.
3) Cualquier rectángulo (acción) puede ser remplazado por cualquier estructura de control (secuencia, `if`, `if/else`, `switch`, `while`, `do/while`, o bien `for`).
4) Las reglas 2 y 3 pueden ser aplicadas tan frecuentemente como se desee y en cualquier orden.

Fig. 4.18 Reglas para la formación de programas estructurados.

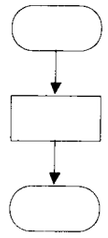


Fig. 4.19 El diagrama de flujo más sencillo.

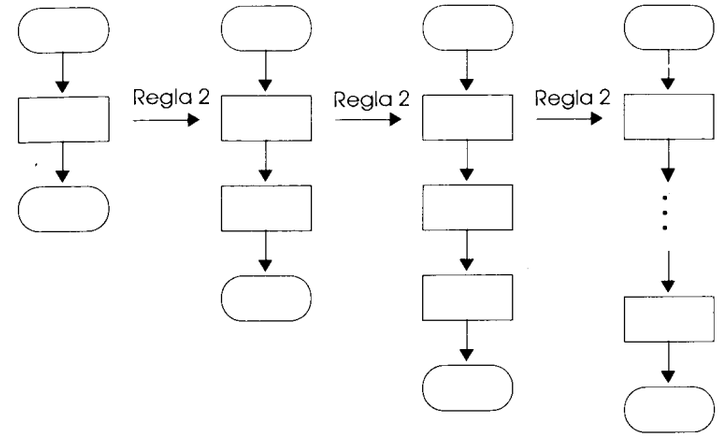


Fig. 4.20 Aplicación repetida de la regla 2 de la figura 4.18 al diagrama de flujo más sencillo.

La regla 4 genera estructuras más grandes, más complejas y más profundamente anidadas. Los diagramas de flujo que resultan de aplicar las reglas de la figura 4.18, constituyen el conjunto de todos los diagramas posibles de flujos estructurados y, por lo tanto, también el conjunto de todos los posibles programas estructurados.

Es debido a la eliminación del enunciado `goto` que estos bloques constructivos nunca se traslapan el uno sobre el otro. La belleza del enfoque estructurado, es que utilizamos un pequeño número de piezas simples de una entrada/una salida, y las ensamblamos en dos formas sencillas. En la figura 4.22 se muestran los tipos de bloques constructivos apilados, que resultan de la aplicación de la regla 2, y los tipos de bloques constructivos anidados que resultan de la aplicación de la regla 3. La figura también muestra el tipo de bloques constructivos superpuestos que no pueden aparecer en diagramas de flujo estructurados (debido a la eliminación del enunciado `goto`).

Si se siguen las reglas de la figura 4.18, no es posible crear un diagrama de flujo no estructurado (como el de la figura 4.23). Si no está seguro de cómo está estructurado un diagrama de flujo particular, aplique las reglas de la figura 4.18 a la inversa, para tratar de reducir dicho diagrama de flujo al diagrama de flujo más simple. Si el diagrama de flujo es reducible al más simple, el diagrama de flujo original está estructurado; de lo contrario no lo está.

La programación estructurada fomenta la simplicidad. Bohm y Jacopini nos han dado como resultado que sólo se requieren tres formas de control:

- Secuencia
- Selección
- Repetición

La secuencia es un asunto trivial. La selección se pone en acción en una de tres formas:

- La estructura `if` (una selección)
- La estructura `if/else` (doble selección)
- La estructura `switch` (múltiple selección)

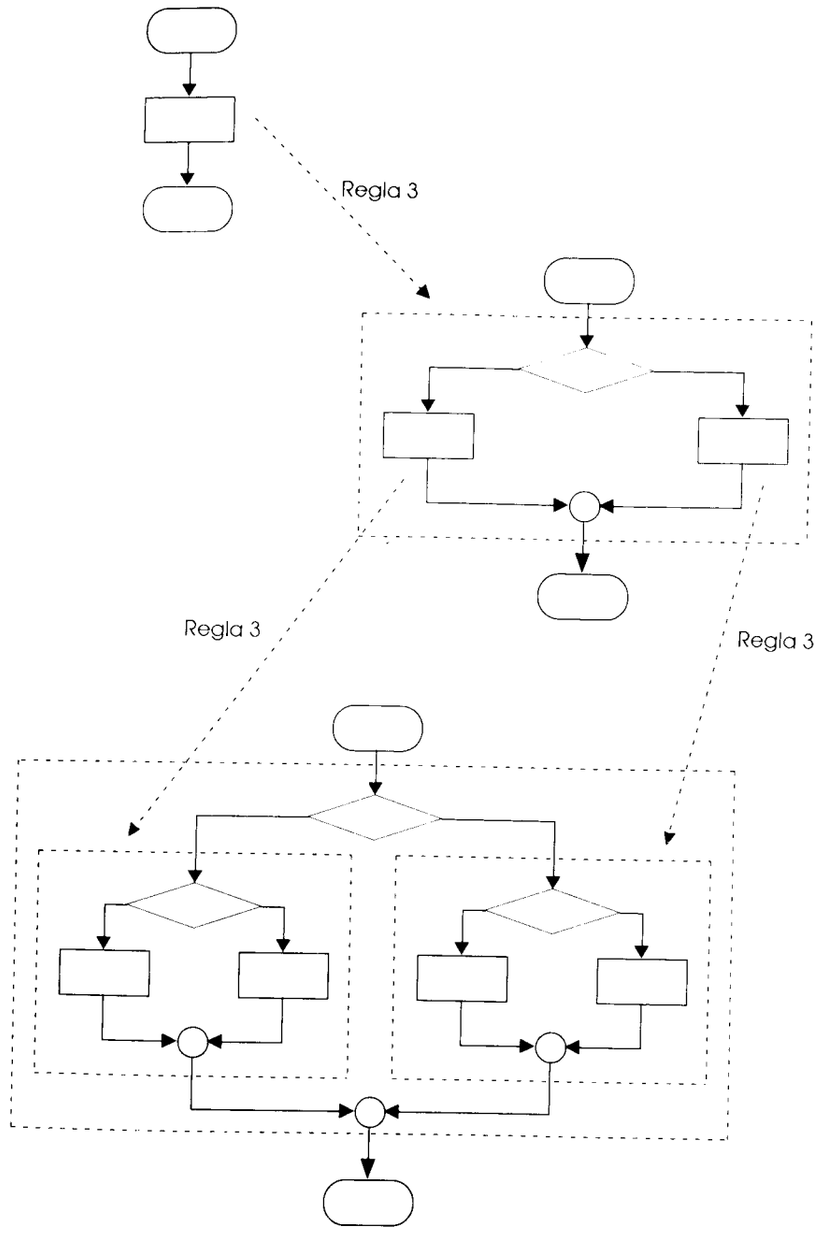


Fig. 4.21 Aplicación de la regla 3 de la figura 4.18 al diagrama de flujo más simple.

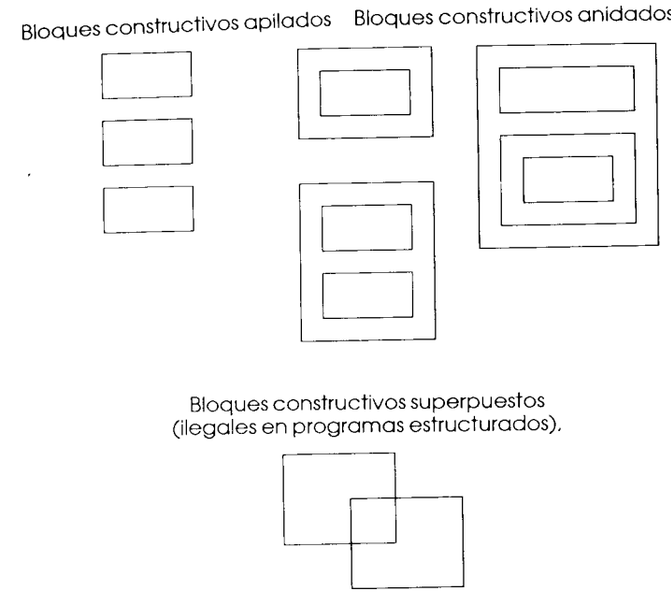


Fig. 4.22 Bloques constructivos apilados, bloques constructivos anidados, y bloques constructivos superpuestos.

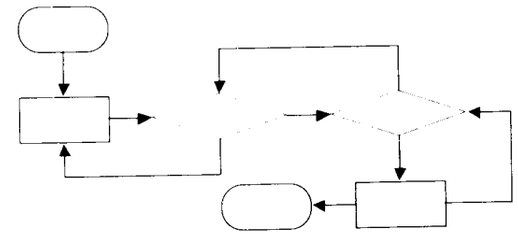


Fig. 4.23 Un diagrama de flujo no estructurado.

De hecho, es fácil comprobar que la estructura simple **`if`** es suficiente para proporcionar cualquier forma de selección —todo lo que se puede hacer con la estructura **`if/else`** y con la estructura **`switch`**, puede ser llevado a cabo con la estructura **`if`**.

La repetición se ejecuta en una de tres formas:

- La estructura **`while`**.
- La estructura **`do/while`**.
- La estructura **`for`**

Es fácil también demostrar que la estructura **while** es suficiente para proporcionar cualquier forma de repetición. Cualquier cosa que pueda ser realizada con la estructura **do/while** y la estructura **for** también puede ser realizada con la estructura **while**.

Combinando estos resultados se llega a que cualquier forma de control, que de alguna manera se requiera en un programa C, puede ser expresado en términos de tres formas de control:

- Secuencia
- Estructura **if** (selección)
- Estructura **while** (repetición)

Y estas estructuras de control sólo pueden ser combinadas de dos formas —apilamiento y anidamiento. De hecho, en realidad la programación estructurada [illegible]


```
do
   enunciado
while (condición);
```

- El enunciado **break**, cuando se ejecuta en una de la estructuras de repetición (**for**, **while** y **do/while**), causa la salida inmediata de la estructura. La ejecución continúa con el siguiente primer enunciado después del ciclo.
- El enunciado **continue**, cuando se ejecuta en una de las estructuras de repetición (**for**, **while** y **do/while**), salta todos los enunciados siguientes del cuerpo de la estructura, y continúa con la siguiente iteración del ciclo.
- El enunciado **switch** maneja una serie de decisiones en las cuales una variable o expresión particular se prueba para cada uno de los valores que puede asumir, y se toman diferentes

estructura de repetición `for`
función `getchar`
incremento de variable de control
repetición indefinida
ciclo infinito
valor inicial de variable de control
justificación a la izquierda
AND lógico (`&&`)
negación lógica (`!`)
operadores lógicos
OR lógico (`||`)
`long`
condición de continuación de ciclo
variable de control de ciclo
contador de ciclo
lvalue ("left value")
signo menos para justificación a la izquierda
selección múltiple
estructuras de control anidadas
regla de anidación
error de diferencia por uno
función `pow`
estructuras de repetición
<return><ctrl-d>
justificación a la derecha
rvalue ("valor derecho")
`short`
condición simple
estructuras de control de una entrada/ una salida
regla de apilamiento
estructura de selección `switch`
tabla de la verdad
operador unario
estructura de repetición `while`

Errores comunes de programación

4.1 Dado que los valores en punto flotante pueden ser aproximados, el control de contador de ciclos con variables de punto flotante puede dar como resultado valores de contador no precisos y pruebas no exactas de terminación.

4.2 Usar un operador relacional incorrecto o usar un valor final incorrecto de un contador de ciclo, en la condición de una estructura `while` o `for`, puede causar errores de diferencia por uno.

4.3 Usar comas en vez de puntos y coma en un encabezado `for`.

4.4 Colocar un punto y coma inmediatamente a la derecha de un encabezado `for` hace del cuerpo de esta estructura `for` un enunciado vacío. Por lo regular esto es un error lógico

4.5 Olvidar en una estructura `switch` un enunciado `break`, cuando se requiere de uno.

4.6 No procesar los caracteres de nueva línea en la entrada al leer los caracteres uno a la vez, puede ser causa de errores lógicos.

4.7 Se generarán ciclos infinitos en una estructura `while`, `for` o bien `do/while` cuando la condición de continuación de ciclo nunca se convierte en falsa. A fin de evitar lo anterior, asegúrese de que no exista un punto y coma después del encabezado de una estructura `while` o `for`. En un ciclo controlado por contador, asegúrese que la variable de control es incrementada (o decrementada) en el cuerpo del ciclo. En un ciclo controlado por centinela, asegúrese que el valor centinela es eventual introducido en algún momento.

4.8 Utilizar el operador == para asignación, o bien utilizar el operador = para igualdad.

Prácticas sanas de programación

4.1 Controlar el contador de ciclos con valores enteros.

4.2 Hacer sangría en los enunciados en el cuerpo de cada estructura de control.

4.3 Colocar una línea en blanco antes y después de cada estructura de control principal, para que se destaque en el programa.

4.4 Demasiados niveles anidados pueden dificultar la comprensión de un programa. Como regla general, procure evitar el uso de más de tres niveles de sangrías.

4.5 La combinación de espaciamiento vertical antes y después de las estructuras de control y las sangrías de los cuerpos de las estructuras de control dentro de los encabezados de las estructuras de control, le da a los programas una apariencia bidimensional que mejora de manera significativa la legibilidad del programa.

4.6 Usar el valor final en la condición de una estructura `while` o `for` y utilizar el operador relacional <= auxiliará a evitar errores de diferencia por uno. Para un ciclo que se utilice para imprimir los valores del 1 al 10, por ejemplo, la condición de confirmación de ciclo debería ser `counter <= 10` en vez de `counter < 11` o bien counter < 10.

4.7 Coloque sólo expresiones que involucren las variables de control en las secciones de inicialización y de incremento de una estructura `for`. Las manipulaciones de las demás variables deberían de aparecer, ya sea antes del ciclo (si se ejecutan una vez, como los enunciados de incialización) o dentro del cuerpo del ciclo (si se ejecutan una vez en cada repetición, como son los enunciados incrementales o decrementales).

4.8 Aunque el valor de la variable de control puede ser modificado en el cuerpo de un ciclo `for`, ello podría llevarnos a errores sutiles. Lo mejor es no cambiarlo.

4.9 Aunque los enunciados que anteceden a un `for`, y los enunciados del cuerpo de un `for`, a menudo pueden ser combinados en un encabezado `for`, evite hacerlo, porque hace el programa más difícil de leer.

4.10 Limite, si es posible, a una sola línea el tamaño de los encabezados de estructuras de control.

4.11 No utilice variables del tipo `float` o `double` para llevar a cabo cálculos monetarios. La falta de precisión de los números de punto flotante pueden causar errores, que resultarán en valores monetarios incorrectos. En los ejercicios, exploramos la utilización de enteros para la ejecución de cálculos monetarios.

4.12 Proveer un caso `default` en enunciados `switch`. En un `switch` serán ignorados los casos no probados en forma explicita. El caso `default` ayuda a evitar lo anterior, enfocando al programador sobre la necesidad de procesar condiciones excepcionales. Existen situaciones en las cuales no se requiere de procesamiento `default`.

4.13 Aunque las cláusulas `case` y la cláusula de caso `default` en una estructura `switch` pueden ocurrir en cualquier orden, se considera una práctica sana de programación colocar al final la cláusula `default`.

4.14 En una estructura `switch`, cuando la cláusula `default` se enlista al final, el enunciado `break` no es requerido. Pero algunos programadores incluyen este `break`, para fines de claridad y simetría con otros `case`s.

4.15 Recuerde proporcionar capacidades de proceso para los caracteres de nueva línea en la entrada, cuando se procesen caracteres uno a la vez.

4.16 Algunos programadores incluyen siempre llaves en una estructura `do/while`, aún si las llaves no son necesarias. Esto ayuda a eliminar ambigüedad entre la estructura `do/while` con un enunciado, y la estructura `while`.

4.17 Algunos programadores son de la opinión que `break` y `continue` violan las normas de la programación estructurada. Dado que los efectos de estos enunciados pueden ser obtenidos mediante técnicas de programación estructuradas que aprenderemos pronto, estos programadores no utilizan `break` ni `continue`.

4.18 Cuando una expresión de igualdad tiene una variable y una constante como en `x == 1`, algunos programadores prefieren escribir la expresión con la constante a la izquierda y el nombre variable a la derecha, como protección contra un error lógico, que ocurriría cuando el programador remplaza de forma accidental el operador == por signo de =.

Sugerencias de rendimiento

4.1 En situaciones orientadas a rendimiento, donde la memoria está escasa o se requiere de velocidad, pudiera ser deseable utilizar tamaños de enteros más pequeños.

4.2 Los enunciados `break` y `continue`, cuando se útilizan de forma adecuada, se ejecutan con mayor rapidez que las técnicas estructuradas correspondientes que pronto aprenderemos.

4.3 En expresiones que utilicen el operador &&, haga que la condición que más probabilidades tenga de ser falsa sea la que aparezca más a la izquierda. En expresiones que utilicen al operador ||, haga

que la condición que más probabilidades tenga de ser verdadera, sea la condición más a la izquierda. Esto puede reducir el tiempo de ejecución del programa.

Sugerencias de portabilidad

4.1 Las combinaciones de teclas para la introducción de **EOF** (fin de archivo) son sistema dependientes.

4.2 Hacer la prueba buscando la constante simbólica **EOF**, en vez de buscar -1, hace más portátiles a los programas. El estándar ANSI indica que **EOF** es un valor entero negativo (pero no necesariamente -1). Por lo tanto, **EOF** pudiera tener diferentes valores en diferentes sistemas

4.3 Dado que de un sistema a otro **int** varía en tamaño, utilice enteros **long**, si espera procesar enteros fuera del rango ±32767, y desearía ser capaz de ejecutar el programa en varios sistemas de cómputo diferentes.

Observación de ingeniería de software

4.1 Existe cierta contraposición entre alcanzar ingeniería de software de calidad y conseguir software de mejor rendimiento. A menudo una de estas metas se consigue a expensas de la otra.

Ejercicios de autoevaluación

4.1 Llene los espacios vacíos en cada uno de los enunciados siguientes.

a) La repetición controlada por contador también se conoce como repetición _____, porque se sabe por anticipado cuántas veces se ejecutará el ciclo.

b) La repetición controlada por centinela también se conoce como _____, porque no se sabe con anticipación cuántas veces se ejecutará el ciclo.

c) En la repetición controlada por contador, se utiliza un _____ para contar el número de veces que deben repetirse un grupo de instrucciones.

d) El enunciado _____, cuando se ejecuta en una estructura de repetición, hace que se ejecute de inmediato la siguiente iteracción del ciclo.

e) El enunciado _____, cuando se ejecuta en una estructura de repetición, o en un **switch**, hace que se salga de inmediato de la estructura.

f) La _____, se utiliza para probar una variable o expresión particular para cada uno de los valores enteros constantes que puede asumir.

4.2 Indique si los siguientes son verdaderos o falsos. Si la respuesta es falsa, explique por qué.

a) El caso **default** se requiere en la estructura de selección **switch**.

b) El enunciado **break** se requiere en el caso **default** de una estructura de selección **switch**.

c) La expresión **(x > y&&a < b)** es verdadera ya sea que **x > y** es verdadero o **a > b** es verdadero.

d) Una expresión que contenga el operador **||** es verdadera si alguno o ambos de sus operandos son verdaderos.

4.3 Escriba un enunciado en C o un conjunto de enunciados en C que ejecuten cada una de las tareas siguientes:

a) Sume los enteros impares entre 1 y 99 mediante una estructura **for**. Suponga que las variables enteras **sum** y **count** han sido declaradas.

b) Imprima el valor **333.546372** en un ancho de campo con **15** caracteres con precisiones de **1**, **2**, **3**, **4**, y **5**. Justifique la salida a la izquierda. ¿Cuáles son los valores que se imprimen?

c) Calcule el valor de **2.5** elevado a la potencia de **3** utilizando la función **pow**. Imprima el resultado con una precisión de **2** en un ancho de campo de **10** posiciones. ¿Cuál es el valor que se imprime?

d) Imprima los enteros de 1 a 20 utilizando el ciclo **while** y la variable contador **x**. Suponga que la variable **x** ha sido declarada, pero no inicializada. Imprima sólo 5 enteros por línea. Sugerencia: Use el cálculo **x % 5**. Cuando el valor de lo anterior es 0, imprima un carácter de nueva línea, de lo contrario imprima un carácter de tabulador.

e) Repita el Ejercicio 4.3 (d) utilizando una estructura **for**.

4.4 Encuentre el error en cada uno de los segmentos de código siguientes y explique cómo corregirlo.

a)
```
x = 1;
while (x <= 10);
   x++;
}
```

b)
```
for (y = .1; y != 1.0; y += .1)
   printf("%f\n", y);
```

c)
```
switch (n) {
   case 1:
      printf("The number is 1\n");
   case 2:
      printf("The number is 2\n");
      break;
   default;
      printf("The number is not 1 or 2\n");
      break;
}
```

d) El siguiente código debería imprimir los valores del 1 al 10.
```
n = 1;
while (n < 10)
   printf("%d ", n++);
```

Respuestas a los ejercicios de autoevaluación

4.1 a) definido. b) indefinido. c) variable de control o contador. d) **continue**. e) **break**. f) estructura de selección **switch**.

4.2 a) Falso. El caso **default** es opcional. Si no se requiere una acción por omisión, entonces no es necesario un caso **default**.

b) Falso. El enunciado **break** se utiliza para salir de la estructura **switch**. El enunciado **break** no es requerido cuando el caso **default** es el último caso.

c) Falso. Al utilizar el operador **&&** ambas expresiones relacionadas deben ser verdaderas, para que toda la expresión resulte verdadera.

d) Verdadero.

4.3 a)
```
sum = 0;
for (count = 1; count <= 99; count +=2)
sum += count;
```

b)
```
printf("%-15.1f\n", 333.546372);  /* prints 333.5     */
printf("%-15.2f\n", 333.546372);  /* prints 333.55    */
printf("%-15.3f\n", 333.546372);  /* prints 333.546   */
printf("%-15.4f\n", 333.546372);  /* prints 333.5464  */
printf("%-15.5f\n", 333.546372);  /* prints 333.54637 */
```

c)
```
printf("%10.2f\n", pow(2.5, 3));  /* prints 15.63     */
```

d)
```
x = 1;
while (x <= 20) {
   printf("%d", x);
```

```
      if (x % 5 == 0)
         printf("\n");
      else
         printf("\t");
      x++;
   }
```

o bien

```
x = 1;
while (x <= 20)
   if (x % 5 == 0)
      printf("%d\n", x++);
   else
      printf("%d\t", x++);
```

o bien

```
x = 0;
while (++x  <= 20)
   if (x % 5 == 0)
      printf("%d\n", x);
   else
      printf("%d\t", x);
```

e)
```
for (x = 1; x <= 20; x++) {
   printf("%d", x);
   if (x % 5 == 0)
      printf("\n");
   else
      printf("\t");
}
```

o bien

```
for (x = 1; x <= 20; x++)
   if (x % 5 == 0)
      printf("%d\n", x);
   else
      printf("%d\t", x);
```

4.4 a) Error: el punto y coma después del encabezado `while` genera un ciclo infinito.
Corrección: Remplace el punto y coma por una llave izquierda `{`, o bien elimine tanto el `;` como la llave derecha `}`.

b) Error: usar un número de punto flotante para control de una estructura de repetición `for`.
Corrección: utilice un entero, y ejecute el cálculo adecuado a fin de obtener los valores que desea.

```
for (y = 1; y != 10; y++)
   printf("%f\n", (float) y / 10);
```

c) Error: el enunciado `break` faltante en los enunciados para el primer `case`.
Corrección: añada un enunciado `break` al final de los enunciados para el primer `case`. Advierta que esto no es necesariamente un error, si el programador desea que `case 2:` se ejecute siempre que se ejecute `case 1:`.

d) Error: Operador relacional incorrecto, utilizado en la condición de continuación de repetición `while`. Corrección: Utilice `<=` en vez de `<` que.

Ejercicios

4.5 Encuentre el error en cada uno de los siguientes (Nota: pudiera haber más de un solo error).

a)
```
For (x = 100, x < = 1, x++)
   printf("%d\n", x);
```

b) El código siguiente debería imprimir si el entero dado es impar o par:
```
switch (value % 2) {
   case 0:
      printf("Even integer\n");
   case 1:
      printf("Odd integer\n");
}
```

c) El siguiente código debe introducir un entero y un carácter, y a continuación imprimirlos. Suponga que el usuario escribe como entrada `100 A`.
```
scanf("%d", &intVal);
charVal = getchar();
printf("Integer: %d\nCharacter: %c\n", intVal, charVal);
```

d)
```
for (x = .000001; x <= .0001; x += .000001)
   printf("%.7f\n", x);
```

e) El código siguiente deberá dar como salida los enteros impares de 999 hasta 1:
```
for (x = 999; x >= 1; x += 2)
   printf("%d\n", x);
```

f) El código siguiente debe dar como salida los enteros pares desde 2 hasta 100:
```
counter = 2;

Do {
   if (counter % 2 == 0)
      printf("%d\n", counter);

   counter += 2;
      } While (counter  100);
```

g) El código siguiente deberá sumar los enteros de 100 a 150 (suponga que `total` está inicializado a 0):
```
for (x = 100; x <= 150; x++);
   total += x;
```

4.6 Diga qué valores de la variable de control `x` se imprimen para cada uno de los enunciados `for` siguientes:

a)
```
for (x = 2; x <= 13; x += 2)
   printf("%d\n", x);
```

b)
```
for (x = 5; x <= 22; x += 7)
   printf("%d\n", x);
```

c)
```
for (x = 3; x <= 15; x += 3)
   printf("%d\n", x);
```

d)
```
for (x = 1; x <= 5; x += 7)
   printf("%d\n", x);
```

e)
```
for (x = 12; x >= 2; x -= 3)
printf("%d\n", x);
```

4.7 Escriba enunciados `for` que impriman las siguientes secuencias de valores:

a) 1, 2, 3, 4, 5, 6, 7
b) 3, 8, 13, 18, 23
c) 20, 14, 8, 2, -4, -10
d) 19, 27, 35, 43, 51

4.8 ¿Qué es lo que ejecuta el siguiente programa?

```
#include <stdio.h>

main()
{
   int i, j, x, y;

   printf("Enter integers in the range 1-20: ");
   scanf("%d%d," &x, &y);

   for (i = 1; i <= y; i++) {

      for (j = 1; j <= x; j++)
         printf("@");

      printf("\n");
   }

   return 0;
}
```

4.9 Escriba un programa que sume una secuencia de enteros. Suponga que el primer entero leído con `scanf` especifica el número de valores que faltan de introducir. Su programa deberá leer un valor cada vez que `scanf` sea ejecutado. Una secuencia de entrada típica pudiera ser

```
5 100 200 300 400 500
```

donde `5` indica que los `5` valores subsiguientes deberán de ser sumados.

4.10 Escriba un programa que calcule e imprima el promedio de varios enteros. Suponga que el último valor leído mediante `scanf` es el centinela `9999`. Una secuencia típica de entrada pudiera ser

```
10 8 11 7 9 9999
```

indicando que debe calcularse el promedio de todos los valores que preceden a `9999`.

4.11 Escriba un programa que localice el más pequeño de varios enteros. Suponga que el primer valor leído especifica el número de valores que restan.

4.12 Escriba un programa que calcule e imprima la suma de los enteros pares del 2 al 30.

4.13 Escriba un programa que calcule e imprima el producto de los enteros impares del 1 al 15.

4.14 La función *factorial* se utiliza con frecuencia en problemas de probabilidades. El factorial de un entero positivo *n* (escrito *n!* y dicho como "factorial de n") es igual al producto de los enteros positivos del 1 hasta el *n*. Escriba un programa que evalúe los factoriales de los enteros del 1 al 5. Imprima el resultado en forma tabular. ¿Qué podría impedirle que se calculara el factorial de 20?.

4.15 Modifique el programa de interés compuesto de la Sección 4.6 para repetir sus pasos para tasas de interés de 5 %, 6 %, 7 %, 8 %, 9 % y 10 por ciento. Utilice un ciclo `for` para variar la tasa de interés.

4.16 Escriba un programa que imprima los siguientes patrones por separado, uno debajo del siguiente. Utilice ciclos `for` para generar los patrones. Todos los asteriscos (`*`), deberán ser impresos por un solo

enunciado `printf` de la forma `printf("*");` (esto hace que los asteriscos se impriman uno al lado del otro). *Sugerencia*: los últimos dos patrones requieren que cada línea empiece con un número correcto de espacios en blanco.

```
(A)          (B)          (C)          (D)

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

4.17 La cobranza se hace cada vez más difícil durante periodos de recesión, por lo que las empresas pudieran restringir sus límites de crédito para impedir que sus cuentas por cobrar (el dinero que se les debe) crezca demasiado. En respuesta a una recesión prolongada, una compañía ha reducido a la mitad los límites de crédito de sus clientes. Por lo tanto, si un cliente particular tenía un límite de crédito de $2000, el límite de crédito de este cliente es ahora de $1000. Si un cliente tenía un límite de crédito de $5000, el límite de crédito de este cliente es ahora de $2500. Escriba una programa que analice el estado de crédito de tres clientes de esta empresa. Para cada cliente se le dará:

1. El número de cuenta del cliente.
2. El límite de crédito del cliente, antes de la recesión.
3. EL saldo actual del cliente (es decir, la cantidad que el cliente le debe a la empresa).

Su programa deberá calcular e imprimir el nuevo límite de crédito para cada cliente, y deberá determinar (e imprimir) qué clientes tienen saldos actuales que exceden sus nuevos límites de crédito.

4.18 Una aplicación interesante de las computadoras, es dibujar gráficos y gráficas de barra (a veces llamadas "histogramas"). Escriba un programa que lea cinco números (cada uno de ellos entre 1 y 30). Para cada uno de los números leídos, su programa deberá imprimir una línea, conteniendo dicho número en asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, deberá imprimir `*******`.

4.19 Una empresa de ventas por correo vende cinco productos distintos, cuyos precios de menudeo se muestran en la tabla siguiente:

Número de producto	Precio al menudeo
1	$ 2.98
2	4.50
3	9.98
4	4.49
5	6.87

Escriba un programa que lea una serie de pares de números, como sigue:

1. Número del producto.
2. Cantidad vendida en un día.

Su programa deberá utilizar un enunciado `switch` para ayudar a determinar el precio de menudeo de cada producto. Su programa deberá calcular y desplegar el valor total de menudeo, de todos los productos vendidos la semana pasada.

4.20 Complete las tablas de la verdad siguiente, llenando cada uno de los espacios en blanco con un 0 o un 1.

Condición1	Condición2	Condición1 && Condición2
0	0	0
0	no cero	0
no cero	0	____
no cero	no cero	____

Condición1	Condición2	Condición1 || Condición2
0	0	0
0	no cero	1
no cero	0	____
no cero	no cero	____

Condición1	!Condición1
0	1
no cero	____

4.21 Vuelva a escribir el programa de la figura 4.2, de tal forma que la inicialización de la variable `counter` sea hecha en la declaración, en vez de en la estructura `for`.

4.22 Modifique el programa de la figura 4.7, de tal forma que calcule la calificación promedio de la clase.

4.23 Modifique el programa de la figura 4.6, de tal forma que sólo utilice enteros para calcular el interés compuesto. (*Sugerencia*: trate todas las cantidades monetarias como números enteros de centavos. A continuación "divida" el resultado en su porción dólares y en su porción en centavos, mediante el uso de las operaciones de división y de módulo respectivamente. Inserte un punto).

4.24 Suponga que `i = 1, j = 2, k = 3, m = 2`. ¿Qué es lo que imprime cada uno de los enunciados siguientes?.

a) `printf("%d", i == 1);`
b) `printf("%d", j == 3);`

c) `printf("%d", i >= 1 && j > 4);`
d) `printf("%d", m <= 99 && k < m);`
e) `printf("%d", j >= i || k == m);`
f) `printf("%d", k + m < j || 3 - j >= k);`
g) `printf("%d", !m);`
h) `printf("%d", !(j - m));`
i) `printf("%d", !(k < m));`
j) `printf("%d", !(j > k));`

4.25 Imprima la tabla de equivalentes decimales, binarios, octales y hexadecimales. Si no está familiarizado con estos sistemas numéricos, lea primero el Apéndice E, si desea tratar de resolver este ejercicio.

4.26 Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

Imprima una tabla que muestre el valor de π aproximado a un término de esta serie, a dos, a tres, etcétera. ¿Cuántos términos de esta serie tendrá que utilizar antes de que empiece a tener 3.14?, ¿3.141?, ¿3.1415? ¿3.14159?

4.27 (*Ternas pitagóricas*). Un triángulo rectángulo puede tener lados que sean todos enteros. El conjunto de tres valores enteros para los lados de un triángulo rectángulo se conoce como una terna pitagórica. Estos tres lados deben de satisfacer la relación de que la suma de los cuadrados de dos de los lados es igual al cuadrado de la hipotenusa. Encuentre todos las ternas pitagóricas para lado1, lado2 e hipotenusa, todos ellos no mayores de 500. Utilice un ciclo `for` de triple anidamiento, que pruebe todas las posibilidades. Este es un ejemplo de computación de "fuerza bruta". Para muchas personas no es de forma estética agradable. Pero existen muchas razones por las cuales estas técnicas son de importancia. Primero, con la potencia de las computadoras incrementándose a una velocidad tan fenomenal, soluciones que con la tecnología de sólo hace unos años habrían tomado años e inclusive siglos de tiempo de computadora para producir, pueden ser hoy producidas en horas, minutos o inclusive segundos. ¡Chips microprocesadores recientes pueden procesar más de 100 millones de instrucciones por segundo! Y en los años 90's probablemente aparecerán chips de microprocesador de billones de instrucciones por segundo. Segundo, como aprenderá en cursos de ciencias de computación más avanzadas, hay gran número de problemas interesantes para los cuales no existe un enfoque algorítmico conocido salvo el uso de simple fuerza bruta. En este libro investigamos muchos tipos de metodologías para la resolución de problemas. Consideraremos muchos enfoques de fuerza bruta para la solución de varios problemas interesantes.

4.28 Una empresa paga a sus empleados como gerentes (que reciben un salario semanal fijo), trabajadores horarios (quienes reciben un salario horario fijo por las primeras 40 horas de trabajo, y "tiempo y medio", es decir, 1.5 veces su sueldo horario, para las horas extras trabajadas), trabajadores a comisión (quienes reciben $250 más 5.7% de sus ventas semanales brutas), o trabajadores a destajo (quienes reciben una cantidad fija de dinero por cada una de las piezas que produce cada trabajador a destajo de esta empresa que trabaja sólo un tipo de piezas). Escriba un programa para calcular la nómina semanal de cada empleado. Usted no sabe por anticipado el número de empleados. Cada tipo de empleado tiene su propio código de nómina: los gerentes tienen un código de nómina 1, los trabajadores horario tienen un código 2, los trabajadores a comisión el código 3 y los trabajadores a destajo el código 4. Utilice un `switch` para calcular la nómina de cada empleado, basado en el código de nómina de dicho empleado. Dentro del `switch`, solicite al usuario (es decir, al oficinista de nóminas) que escriba los hechos apropiados que requiere su programa, para calcular la paga o nómina de cada empleado, basado en el código de nómina de cada uno de ellos.

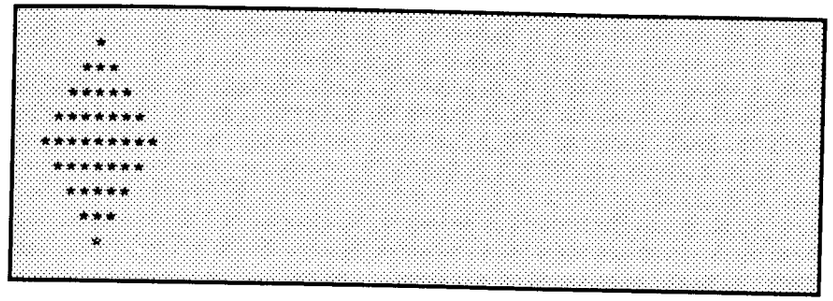
4.29 (*Leyes de De Morgan*). En este capítulo, hemos analizado los operadores lógicos `&&`, `||`, y `!`. Las leyes de De Morgan a veces pueden hacer que sea más conveniente para nosotros expresar una expresión lógica. Estas leyes dicen que la expresión `!(`*condición1* `&&` *condición2*`)` es lógicamente equivalente a la expresión `(!`*condición1* `||` `!`*condición2*`)`. También, la expresión `!(`*condición1* `||` `!`*condición2*`)` es

lógicamente equivalente a la expresión (!*condición1* && !*condición2*). Utilice las leyes de De Morgan para escribir expresiones equivalentes para cada uno de los siguientes, y a continuación escriba un programa para mostrar que tanto la expresión original como la nueva, en cada caso, son equivalentes:

a) `!(x < 5) && !(y >= 7)`
b) `!(a == b) || !(g != 5)`
c) `!((x <= 8) && (Y > 4))`
d) `!((i > 4( || (j <= 6))`

4.30 Vuelva a escribir el programa de la figura 4.7, remplazando el enunciado **`switch`** con un enunciado **`if/else`** anidado; tenga cuidado de manejar el caso **`default`** de forma adecuada. A continuación vuelva a escribir esta nueva versión, remplazando el enunciado **`if/else`** anidado con un serie de enunciados **`if`**; aquí, también, tenga cuidado cómo manejar correctamente el caso **`default`** (esto es más difícil que en el caso de la versión **`if/else`** anidada). Este ejercicio demuestra que **`switch`** es una conveniencia y que cualquier enunciado **`switch`** puede ser escrito utilizando sólo enunciados de una sola selección.

4.31 Escriba un programa que imprima la forma en diamante siguiente. Puede usted utilizar enunciados **`printf`** que impriman ya sea un asterisco (*), o un espacio en blanco. Maximice su utilización de repeticiones (utilizando estructuras **`for`** anidadas), y minimice el número de enunciados **`printf`**.



4.32 Modifique el programa que escribió en el Ejercicio 4.31 para leer un número impar del rango 1 al 19, a fin de especificar el número de renglones del diamante. Su programa a continuación deberá desplegar un diamante del tamaño apropiado.

4.33 Si está familiarizado con los números romanos, escriba un programa que imprima una tabla de todos los equivalentes de números romanos con números decimales en el rango del 1 al 100.

4.34 Escriba un programa que imprima una tabla de los equivalentes binarios, octal y hexadecimal de los números decimales en el rango 1 hasta 256. Si no está familiarizado con estos sistemas numéricos, lea primero el Apéndice E, si desea tratar de intentar hacer este ejercicio.

4.35 Describa el proceso que utilizaría para remplazar el ciclo **`do/while`** con un ciclo equivalente **`while`**. ¿Qué problema ocurre cuando usted trata de remplazar el ciclo **`while`** con un ciclo equivalente **`do/while`**? Suponga que se le ha dicho que debe eliminar un ciclo **`while`** y remplazarlo con un **`do/while`**. ¿Qué estructura de control adicional necesitaría utilizar, y cómo la utilizaría, para asegurarse que el programa resultante se comporta igual que el original?.

4.36 Escriba un programa que introduzca el año en el rango 1994 hasta 1999 y utilice una repetición de ciclo **`for`** para producir un calendario condensado e impreso de forma nítida. Tenga cuidado con los años bisiestos.

4.37 Una crítica del enunciado **`break`** y del enunciado **`continue`** es que cada uno de ellos no es estructurado. De hecho los enunciados **`break`** y **`continue`** siempre pueden ser remplazados por enunciados estructurados, aunque el hacerlo puede resultar un poco torpe. Describa en general cómo eliminaría cualquier enunciado **`break`** de un ciclo de un programa, y lo remplazaría con algún equivalente estructurado

(*Sugerencia*: el enunciado **`break`** deja un ciclo desde dentro del cuerpo del ciclo). La otra forma de salir es haciendo fallar la prueba de continuación de ciclo. Considere utilizar en la prueba de continuación de ciclo, una segunda prueba que indique (salida temprana debida a una condición "de salida"). Utilice la técnica que desarrolló aquí para eliminar el enunciado 'break' del programa de la figura 4.11.

4.38 ¿Qué es lo que hace el siguiente segmento de programa?.

```
for (i = 1; i <= 5; i++) {
    for (j = 1; j <= 3; j++) {
        for (k = 1, k <= 4; k++)
            printf("*");
            Printf("\n");

}
        printf("\n");

}
```

4.39 Describa en general cómo eliminaría cualquier enunciado **`continue`** de un ciclo de un programa y remplazar dicho enunciado con algún equivalente estructurado. Use las técnicas que desarrolló aquí para eliminar el enunciado **`continue`** del programa de la figura 4.12.

4.40 Describa en general cómo eliminaría los enunciados **`break`** de una estructura **`switch`** y los remplazaría con equivalentes estructurados. Utilice la técnica (quizás algo torpe) que desarrolló aquí para eliminar los enunciados **`break`** del programa de la figura 4.7.