3

Desarrollo de programas estructurados

Objetivos

- Comprender las técnicas fundamentales de resolución de problemas.
- Tener la capacidad de desarrollar algoritmos mediante el proceso de refinamiento descendente paso a paso.
- Tener la capacidad de utilizar la estructura de selección `if` y la estructura de selección `if/else` para elegir acciones.
- Tener la capacidad de utilizar la estructura de repetición `while` para ejecutar enunciados repetidamente en un programa.
- Comprender la repetición controlada por contador y la repetición controlada por centinela.
- Comprender la programación estructurada.
- Tener la capacidad de utilizar los operadores incrementales, decrementales, y de asignación.

El secreto del éxito es la constancia en el propósito.
Benjamin Disraeli

Movámonos todos un lugar.
Lewis Carroll

La rueda ha dado un giro completo.
William Shakespeare
King Lear

¡Cuántas manzanas habrán caído sobre la cabeza de Newton antes que comprendiera lo que le estaban sugiriendo!
Robert Frost
Comment

Sinopsis

Resumen • Terminología • Errores comunes de programación • Prácticas sanas de programación • Sugerencias de rendimiento • Observaciones de ingeniería de software • Ejercicios de autoevaluación • Respuestas a los ejercicios de autoevaluación • Ejercicios.

3.1 Introducción

Antes de escribir un programa para resolver un problema particular, es esencial tener comprensión completa del mismo, y un método planeado de forma cuidadosa para su resolución. Los dos capítulos siguientes analizan técnicas que facilitan el desarrollo de los programas de cómputo estructurados. En la Sección 4.11, presentamos un resumen de la programación estructurada, que reúne las técnicas desarrolladas tanto aquí, como en el capítulo. 4.

3.2 Algoritmos

La solución a cualquier problema de cómputo involucra la ejecución de una serie de acciones, en un orden específico. Un *procedimiento* para resolver un problema en términos de

1. las *acciones* a ejecutarse, y
2. el *orden* en el cual estas acciones deben de ejecutarse

se llama un *algoritmo*. El siguiente ejemplo demuestra la importancia de especificar de forma correcta el orden en el cual se deben de ejecutar las acciones.

Veamos el "algoritmo de levantarse" que debe de seguir un ejecutivo junior para salir de la cama y llegar al trabajo:

Salir de la cama.
Quitarse los pijamas.
Darse una ducha.
Vestirse.
Desayunar.
Utilizar el vehículo común para llegar al trabajo.

Mediante esta rutina el ejecutivo llega al trabajo bien preparado para tomar decisiones críticas. Suponga, sin embargo, que los mismos pasos se llevan a cabo en un orden ligeramente distinto:

Salir de la cama.
Quitarse los pijamas.
Vestirse.
Darse una ducha.
Desayunar.
Utilizar el vehículo común para llegar al trabajo.

En este caso, nuestro ejecutivo junior aparecerá en el trabajo totalmente mojado. La especificación del orden en un programa de cómputo en el cual los enunciados deben ser ejecutados se conoce como *control de programa*. En éste y en el capítulo siguiente, investigaremos las capacidades de control de programa de C.

3.3 Seudocódigo

El *seudocódigo* es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos. El seudocódigo que presentamos aquí es en particular útil para desarrollar algoritmos que deberán ser convertidos en programas estructurados de C. El seudocódigo es similar al inglés coloquial; es cómodo y amigable, aunque no se trate de un lenguaje verdadero de programación de computadoras.

De hecho, los programas en seudocódigo no son ejecutados sobre computadoras. Más bien, sólo ayudan al programador "a pensar" un programa, antes de intentar escribirlo en un lenguaje de programación como C. En este capítulo, damos varios ejemplos de cómo se puede utilizar con eficacia el seudocódigo, en el desarrollo de programas estructurados de C.

El seudocódigo consiste solo de caracteres, por lo que los programadores pueden de forma cómoda escribir los programas en seudocódigo en una computadora, utilizando un programa de edición. Sobre demanda la computadora puede desplegar o imprimir una copia nueva en seudocódigo del programa. Un programa preparado cuidadosamente en seudocódigo, puede ser convertido con fácilidad en el programa C correspondiente. Esto se lleva a cabo en muchos casos sólo remplazando enunciados en seudocódigo por sus equivalentes en C.

El seudocódigo incluye sólo enunciados de acción aquellos que deben ser ejecutados cuando el programa haya sido convertido de seudocódigo a C, y luego ejecutado en C. Las declaraciones no son enunciados ejecutables. Son mensajes para el compilador. Por ejemplo, la declaración

```
int i;
```

sólo le indica al compilador el tipo de la variable `i`, así como instruye al compilador que reserve espacio en memoria para esta variable. Pero esta declaración no causa ninguna acción como sería entrada, salida o cálculo para que se ejecute o ocurra al ejecutarse el programa. Algunos programadores deciden enlistar al principio de un programa en seudocódigo cada variable y mencionar de forma breve el objeto de cada una de ellas. Repetimos, el seudocódigo es una ayuda informal para el desarrollo del programa.

3.4 Estructuras de control

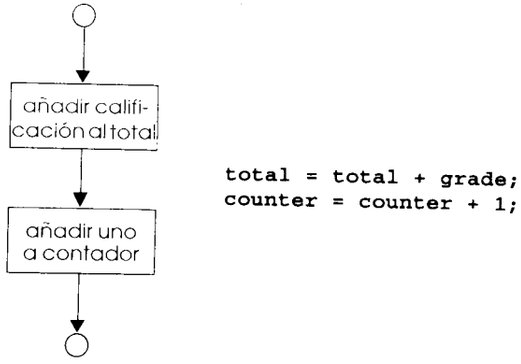
Por lo regular, en un programa los enunciados son ejecutados uno después del otro, en el orden en que aparecen escritos. Esto se conoce como *ejecución secuencial*. Varios enunciados de C, que pronto analizaremos, le permiten al programador especificar que el enunciado siguiente a ejecutar pueda ser otro diferente del que sigue en secuencia. Esto se conoce como *transferencia de control*.

Durante los años 60 , se hizo claro que el uso indiscriminado de transferencias de control era la causa de gran cantidad de dificultades experimentadas por los grupos de desarrollo de software. El dedo acusador apuntaba al *enunciado* **goto**, que le permite al programador especificar una transferencia de control a uno de amplia gama de destinos posibles, dentro de un programa. La noción de lo que se conoce como *programación estructurada* se convirtió prácticamente en sinónimo de "*eliminación de* **goto**".

Las investigaciones de Bohm y de Jacopini[1] habían demostrado que los programas podían ser escritos sin ningún enunciado **goto**. Para los programadores el reto se convirtió en modificar sus estilos a "programación sin **goto**". Y no fue sino hasta entrados los años 70' que la profesión de la programación en general empezó a tomar en serio la programación estructurada. Los resultados han sido impresionantes, ya que los grupos de desarrollo de software han informado reducciones en tiempos de desarrollo, entrega a tiempo más frecuente de sistemas y terminación dentro de presupuesto más frecuente de los proyectos de software. La clave de estos éxitos es simplemente que los programas producidos con las técnicas estructuradas, son más claros, más fáciles de depurar y de modificar, y tal vez más libres de fallas desde el primer momento.

El trabajo de Bohm y Jacopini demostró que todos los programas podrían ser escritos en términos de sólo tres *estructuras de control*, a saber, la *estructura de secuencia*, la *estructura de selección* y la *estructura de repetición*. En C la estructura de secuencia está en esencia interconstruída. A menos de que se indique lo contrario, la computadora ejecutará automáticamente enunciados C, uno después de otro, en el orden en el cual se han escrito. El segmento de *diagrama de flujo* de la figura 3.1 ilustra la estructura de secuencias de C.

Un diagrama de flujo es una representación gráfica de un algoritmo o de una porción de un algoritmo. Los diagramas de flujo se trazan utilizando ciertos símbolos de uso especial como son




```
total = total + grade;
counter = counter + 1;
```

Fig. 3.1 Estructura de secuencia de diagrama de flujo de C.

[1] Bohm, C y G. Jacopini "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, Mayo 1966, pp. 336-371.

rectángulos, diamantes, óvalos y pequeños círculos; estos símbolos están conectados entre sí por flechas, conocidas como *líneas de flujo*.

Al igual que el seudocódigo, los diagramas de flujo son útiles para el desarrollo y la representación de algoritmos, aunque la mayor parte de los programadores prefieren el seudocódigo. Los diagramas de flujo muestran con claridad cómo operan las estructuras de control; sólo para eso es que en este texto los utilizaremos.

Vea el segmento de diagrama de flujo correspondiente a la estructura de secuencia de la figura 3.1. Utilizamos un *símbolo rectángulo*, también conocido como *símbolo de acción*, para indicar cualquier tipo de acción, incluyendo un cálculo o una operación de entrada/salida. Las líneas de flujo de la figura indican el orden en el cual las acciones se ejecutarán primero, **grade** deberá ser sumado a **total** y a continuación **1** deberá ser sumado a **counter**. C nos permite tener todas las acciones que deseemos en una estructura de secuencia. Como pronto veremos, en cualquier lugar donde una acción se pueda colocar, podremos colocar varias acciones en secuencia.

Al trazar un diagrama de flujo que represente un algoritmo completo, el primer símbolo utilizado en el diagrama es un *símbolo oval* que contiene la palabra "Inicio"; y el último símbolo utilizado también es un símbolo oval que contiene la palabra "Fin". Al dibujar sólo una porción de un algoritmo, como en el caso de la figura 3.1, se omiten los símbolos ovales, y en su lugar se utilizan *símbolos de pequeños círculos*, que también se conocen como *símbolos de conexión*.

Quizás el símbolo de diagrama de flujo más importante es el *símbolo diamante*, también conocido como *símbolo de decisión*, mismo que indica donde se debe tomar una decisión. Analizaremos el símbolo diamante en la siguiente sección.

C proporciona tres tipos de estructuras de selección. La estructura de selección **if** (Sección 3.5), ya sea ejecute (elige) una acción, si una condición es verdadera, o pasa por alto la acción, si la condición es falsa. La estructura de selección **if/else** (Sección 3.6) ejecuta una acción, si la condición es verdadera, o ejecuta una acción diferente, si la condición es falsa. La estructura de selección **switch** (que se estudia en el capítulo 4) ejecuta una de entre muchas acciones diferentes, dependiendo del valor de una expresión.

La estructura **if** se llama *estructura de una sola selección*, porque selecciona o ignora una acción. La estructura **if/else** se conoce como *estructura de doble selección*, porque selecciona entre dos acciones distintas. La estructura **switch** se conoce como una *estructura de selección múltiple*, porque selecciona entre muchas acciones diferentes.

C proporciona tres tipos de estructuras de repetición, es decir **while** (Sección 3.7), así como **do/while** y **for** (ambas analizadas en el capítulo 4).

Es todo. C tiene sólo siete estructuras de control: de secuencia, tres tipos de selección y tres de repetición. Cada programa de C se forma al combinar tantos de cada tipo de estructura de control como sean apropiados, en relación con el algoritmo que resuelve el programa. Como en el caso de la estructura de secuencia de la figura 3.1, veremos que cada estructura de control contiene dos símbolos de círculo pequeños, uno en el punto de entrada a la estructura de control y uno en el punto de salida. Estas *estructuras de control de una sola entrada/una sola salida* facilitan la construcción de programas. Las estructuras de control pueden ser agregadas unas a otras, conectando el punto de salida de una estructura de control con el punto de entrada de la siguiente. Esto es muy parecido a la forma en que los niños apilan bloques de construcción, por lo que llamamos lo anterior *apilamiento de estructuras de control*. Aprenderemos que sólo hay otra forma en que las estructuras de control puedan quedar conectadas—un método conocido como *anidar estructuras de control*. Por lo tanto, cualquiera que sea el programa C que tengamos que construir en el futuro, podrá ser elaborado partiendo de sólo siete tipos diferentes de estructuras de control, combinadas de dos distintas formas.

3.5 La estructura de selección if

Una estructura de selección se utiliza para elegir entre cursos alternativos de acción. Por ejemplo, suponga que en un examen 60 es la calificación de aprobado. El enunciado en seudocódigo

If student's grade is greater than or equal to 60
Print "Passed"

determina si la condición "la calificación del estudiante es mayor que o igual a 60" es verdadera o falsa. Si la condición es verdadera, entonces se imprime "Aprobó" y el siguiente enunciado en seudocódigo en orden se "ejecuta" (recuerde que el seudocódigo no es un lenguaje de programación verdadero). Si la condición resulta falsa, se ignora la impresión, y se ejecuta el siguiente enunciado del seudocódigo en su orden. Note que la segunda línea de esta estructura de selección está con sangría. Esta sangría es opcional, pero es muy recomendada, ya que auxilia a enfatizar la estructura inherente de los programas estructurados. Aplicaremos convenciones de sangría de forma cuidadosa a lo largo de este texto. El compilador de C ignora los *espacios en blanco* como son los espacios en blanco, los tabuladores y las nuevas líneas que son utilizadas para sangrías y para el espaciamiento vertical.

Prácticas sanas de programación 3.1

La aplicación consistente de reglas convencionales responsables para las sangrías mejora en forma importante la legibilidad de los programas. Sugerimos un tabulador de tamaño fijo de aproximadamente 1/4 de pulgada o de tres espacios por cada sangría.

El enunciado anterior en seudocódigo *If* puede ser escrito en C como

```
If (grade >= 60)
  printf("Passed\n");
```

Advierta que el código C sigue de cerca al seudocódigo. Esta es una de las propiedades del seudocódigo que lo hace una herramienta tan útil para el desarrollo de programas.

Práctica sana de programación 3.2

A menudo se utiliza el seudocódigo durante el proceso de diseño para "pensar en voz alta" un programa. Posteriormente el programa en seudocódigo se convierte a C.

El diagrama de flujo de la figura 3.2 ilustra la estructura **if** de una sola selección. Este diagrama de flujo contiene el símbolo quizás de mayor importancia en la diagramación de flujo el *símbolo diamante*, también conocido como *símbolo de decisión*, y que indica que una decisión debe ser tomada. El símbolo de decisión contiene una expresión, como es una condición, que puede resultar verdadera o falsa. Dos líneas de flujo parten del símbolo de decisión. Una indica la dirección a tomarse cuando la expresión dentro del símbolo es verdadera; la otra indica la dirección a tomarse cuando la expresión es falsa. Aprendimos en el capítulo 2 que las decisiones pueden ser tomadas, basadas en condiciones que contengan operadores relacionales o de igualdad. De hecho, se puede tomar una decisión basándose en cualquier expresión si la expresión se iguala a cero, se considera como falsa, y si la expresión se iguala a no cero, se considera como verdadera.

Note que la estructura **if**, también, es una estructura de una entrada/una salida. Pronto aprenderemos que los diagramas de flujo para las demás estructuras de control también contienen (además de símbolos de pequeños círculos y líneas de flujo) sólo símbolos rectangulares para indicar acciones a ejecutarse, y símbolos diamante para indicar decisiones a tomarse. Este es el modelo de programación acción/decisión, sobre el cual estamos haciendo énfasis.

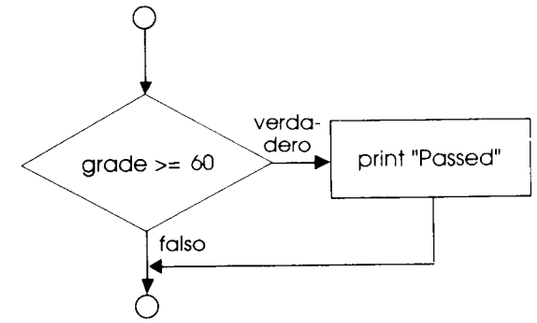


Fig. 3.2 Diagrama de flujo en C de estructura de una selección.

Podemos visualizar siete contenedores, cada uno de ellos conteniendo sólo estructuras de control de uno de los siete tipos. Estas estructuras de control están vacías. No hay nada escrito en los rectángulos y en los diamantes. La tarea de los programadores, entonces, es ensamblar un programa utilizando tantos de cada tipo de estructuras de control como el algoritmo demande, combinando estas estructuras de control, en sólo dos formas posibles (apilando o anidado), y a continuación rellenando las acciones y las decisiones de forma adecuada para el algoritmo. Analizaremos la variedad de formas en las cuales las acciones y las decisiones pueden quedar escritas.

3.6 La estructura de selección if/else

La estructura de selección **if** ejecuta una acción indicada sólo cuando la condición es verdadera; de lo contrario la acción es pasada por alto. La estructura de selección **if/else** permite que el programador especifique que se ejecuten acciones distintas cuando la condición sea verdadera que cuando la condición sea falsa. Por ejemplo, el enunciado en seudocódigo

If student's grade is greater than or equal to 60
Print "Passed"
else
Print "Failed"

imprime *Passed*, si la calificación del alumno es mayor que o igual a 60 e imprime *Failed* si la calificación del alumno es menor de 60. En cualquiera de los casos, después de haber terminado la impresión, se ejecutará el siguiente enunciado del seudocódigo. Advierta que el cuerpo de *else* también queda con sangría.

Práctica sana de programación 3.3

Haga sangrías en ambos cuerpos de los enunciados de una estructura **if/else**.

Cualquier regla convencional de sangrías que escoja deberá aplicarse con cuidado a lo largo de sus programas. Un programa que no siga reglas de espaciamiento uniformes es difícil leer.

Práctica sana de programación 3.4

Si existen varios niveles de sangría, cada nivel deberá tener sangría con una cantidad igual de espacio.

El seudocódigo anterior correspondiente a la estructura *If/else* puede ser escrito en C como

```
if (grade  >= 60)
   printf("Passed\n");
else
   printf("Failed\n");
```

El diagrama de flujo de la figura 3.3 ilustra muy bien el flujo de control de la estructura **if/else**. Otra vez, advierta que (además de los pequeños círculos y flechas) los únicos símbolos en el diagrama de flujo son rectángulos (para las acciones) y un diamante (para una toma de decisiones). Continuamos enfatizando este modelo de computación de acción/decisión. Imagínese otra vez un contenedor profundo, que contenga tantas estructuras de doble selección vacías como sean necesarias, para elaborar cualquier programa en C. Otra vez la tarea del programador es ensamblar estas estructuras de selección (mediante apilamiento y anidación), con cualesquiera otras estructuras de control requeridas por el algoritmo, y rellenar los rectángulos y diamantes vacíos con acciones y decisiones, apropiadas al algoritmo que se está implantando.

C tiene el *operador condicional (?:)* que está relacionado de cerca con la estructura **if/else**. El operador condicional es el único *operador ternario* de C —utiliza tres operandos. Los operandos, junto con el operador condicional, conforman una *expresión condicional*. El primer operando es una condición, el segundo operando es el valor de toda la expresión condicional si la condición es verdadera, y el tercer operando es el valor de toda la expresión condicional si la condición es falsa. Por ejemplo, el enunciado **printf**

```
printf("%s\n", grade >= 60 ? "Passed" : "Failed");
```

contiene una expresión condicional, que evalúa la cadena literal como **"Passed"** si la condición **grade >= 60** es verdadera y evalúa la cadena literal como **"Failed"** si la condición es falsa.

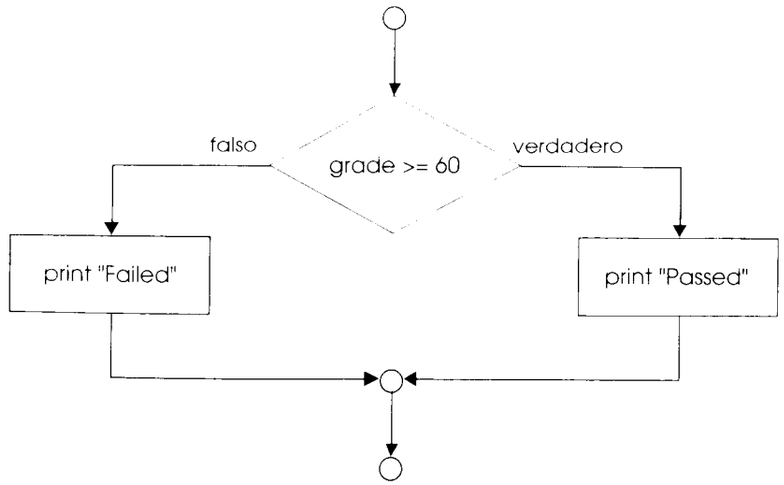


Fig. 3.3 Diagrama de flujo en C de la estructura de doble selección **if/else**.

La cadena de control de formato de **printf** contiene la especificación de conversión **%s**, para la impresión de una cadena de caracteres. Por lo tanto, el enunciado anterior **printf** lleva a cabo en esencia lo mismo que el enunciado anterior **if/else**.

Los valores en una expresión condicional también pueden ser acciones a ejecutar. Por ejemplo, la expresión condicional

```
grade  >= 60 ? printf("Passed\n") : printf("Failed\n");
```

Se lee como sigue "si **grade** es mayor que o igual a **60**, entonces **printf("Passed\n")**, y de lo contrario **printf("Failed\n")**". Esto, también, resulta comparable a la estructura anterior **if/else**. Veremos que los operadores condicionales pueden ser utilizados en algunas situaciones donde los enunciados **if/else** no pueden ser usados.

Las *estructuras* ***if/else*** *anidadas* prueban para muchos casos, colocando estructuras **if/else** dentro de estructuras **if/else**. Por ejemplo, el siguiente enunciado en seudocódigo imprimirá **A** para calificaciones de examen mayores que o iguales a **90**, **B** para calificaciones mayores que o iguales a **80**, **C** para calificaciones mayores que o iguales a **70**, **D** para calificaciones mayores que o iguales a **60**, y **F** para todas las demás calificaciones.

```
(If) Si la calificación del alumno es mayor que o igual a 90
     Imprima "A"
(else) de lo contrario
     (If) Si la calificación del alumno es mayor que o igual a 80
          Imprima "B"
     (else) de lo contrario
          (If) Si la calificación del alumno es mayor que o igual a 70
               Imprima "C"
          (else) de lo contrario
               (If) Si la calificación del alumno es mayor que o igual a 60
                    Imprima "D"
               (else) de lo contrario
                    Imprima "F"
```

Este seudocódigo puede ser escrito en C como

```
if (grade >= 90)
   printf("A\n");
else
   if (grade >= 80)
      printf("B\n");
   else
      if (grade >= 70)
         printf("C\n");
      else
         if (grade >= 60)
            printf("D\n");
         else
            printf("F\n");
```

Si la variable **grade** es mayor que o igual a 90, las primeras cuatro condiciones resultarán ciertas, pero sólo se ejecutará el enunciado **printf** después de la primera prueba. Después de que se

haya ejecutado ese **printf**, la parte **else** del enunciado "exterior" **if/else** será pasado por alto. Muchos programadores de C prefieren escribir la estructura **if** anterior, como sigue:

```
if (grade >= 90)
   printf("A\n");
else if (grade >= 80)
   printf("B\n");
else if (grade >= 70)
   printf("C\n");
else if (grade >= 60)
   printf("D\n");
else
   printf("F\n");
```

Por lo que se refiere al compilador de C, ambas formas resultan equivalentes. Esta última forma es popular porque evita las profundas sangrías del código hacia la derecha. A menudo dichas sangrías dejan poco espacio sobre la línea, obligando a la división de líneas y, por lo tanto, reduciendo la legibilidad del programa.

La estructura de selección **if** espera un enunciado dentro de su cuerpo. Para incluir varios enunciados en el cuerpo de un **if**, encierre el conjunto de enunciados entre llaves ({ y }). Un conjunto de enunciados contenidos dentro de un par de llaves se conoce como un *enunciado compuesto*.

Observación de ingeniería de software 3.1

Un enunciado compuesto puede ser colocado en cualquier parte de un programa, donde pueda ser colocado un enunciado sencillo.

El siguiente ejemplo incluye un enunciado compuesto en la parte **else** de una estructura **if/else**.

```
if (grade >= 60)
   printf("Passed.\n");
else {
   printf("Failed.\n");
   printf("You must take this course again.\n");
}
```

En este caso, si la calificación es menor que 60, el programa ejecutará los dos enunciados **printf** existentes en el cuerpo de **else** e imprimirá

```
Failed.
You must take this course again.
```

Advierta las llaves que rodean ambos enunciados en la cláusula **else**. Estas llaves son importantes. Sin ellas, el enunciado

```
printf("You must take this course again.\n");
```

quedaría fuera del cuerpo de la parte **else** del **if**, y sería ejecutada, independientemente de si la calificación es menor que 60.

Error común de programación 3.1

Olvidar una o ambas de las llaves que delimitan un enunciado compuesto.

Un error de sintaxis será detectado por el compilador. Un error lógico hará su efecto durante la ejecución. Un error lógico fatal hará que un programa falle y que termine de forma prematura. Un error lógico no fatal permitirá que continúe el programa, pero produciendo resultados incorrectos.

Error común de programación 3.2

*Colocar en una estructura **if** un punto y coma después de la condición, llevará a un error lógico en estructuras **if** de una sola selección y a un error de sintaxis en estructuras **if** de doble selección.*

Práctica sana de programación 3.5

Algunos programadores prefieren escribir las llaves de principio y de terminación de los enunciados compuestos antes de escribir en el interior de dichas llaves los enunciados individuales. Esto les ayuda a evitar la omisión de una o ambas de las llaves.

Observación de ingeniería de software 3.2

Igual que un enunciado compuesto puede ser colocado en cualquier lugar donde pudiera colocarse un enunciado sencillo, también es posible que no haya ningún enunciado de ningún tipo, es decir, un enunciado vacío. El enunciado vacío se representa colocando un punto y coma (;) donde por lo regular debería estar el enunciado.

En esta sección, hemos presentado la noción de un enunciado compuesto. Un enunciado compuesto puede contener declaraciones (como lo hace el cuerpo de **main**, por ejemplo). Si es así, el enunciado compuesto se conoce como un *bloque*. Las declaraciones dentro de un bloque deben colocarse al principio de éste, antes de cualquier enunciado de acción. En el capítulo 5 analizaremos la utilización de los bloques. Hasta entonces, el lector deberá tratar de evitar de utilizar bloques (a excepción del cuerpo de **main**, naturalmente).

3.7 La estructura de repetición while

Una *estructura de repetición* le permite al programador especificar que se repita una acción, en tanto cierta condición se mantenga verdadera. El enunciado en seudocódigo

(While) En tanto queden elementos en mi lista de compras
Adquirir elemento siguiente y tacharlo de la lista

describe la repetición que ocurre durante una salida de compras. La condición, "there are more items on my shopping list" puede ser verdadera o falsa. Si es verdadera, entonces la acción, "Purchase next item and cross it off my list" se ejecutará. Esta acción se ejecutará en forma repetida, en tanto la condición sea verdadera. El enunciado o enunciados contenidos en la estructura de repetición *while* constituyen el cuerpo del *while*. El cuerpo de la estructura *while* puede ser un enunciado sencillo o un enunciado compuesto.

Eventualmente, la condición se hará falsa (cuando se haya adquirido el último elemento de la lista de compras y se haya tachado de la misma). Llegado a este punto, la repetición se termina, y se ejecutará el enunciado en seudocódigo que sigue de inmediato después de la estructura de repetición.

Error común de programación 3.3

*No incluir en el cuerpo de una estructura **while** una acción que haga que la condición existente en **while** en algun momento se convierta en falsa. Por lo regular, esta estructura de repetición no terminará jamás —un error conocido como "ciclo infinito".*

Error común de programación 3.4

Escribir la palabra clave `while` *con una* `W` *como* `While` *(recuerde que C es un lenguaje que es sensible a la caja tipográfica). Todas las palabras reservadas de C, como* `while`, `if` *y* `else`, *sólo contienen letras minúsculas.*

Como ejemplo de un **while** real, considere un segmento de programa diseñado para encontrar la primera potencia de 2 superior a 1000. Suponga la variable de entero **product** inicializada a 2. Cuando la estructura de repetición **while** siguiente termine de ejecutarse, **product** contendrá la respuesta deseada:

```
product = 2;

while (product <= 1000)
product = 2 * product;
```

El diagrama de flujo de la figura 3.4 enseña con claraidad el flujo de control en la estructura de repetición **while.** Una vez más, advierta que (en adición a los pequeños círculos y flechas) el diagrama de flujo contiene un símbolo rectangular y un diamante. Imagine, de nuevo, un contenedor profundo de estructuras vacías **while,** que pueden ser apiladas y anidadas con otras estructuras de control, para formar una implantación estructurada del flujo de control de un algoritmo. Los rectángulos y diamantes vacíos, se llenarán entonces con decisiones y acciones apropiadas. El diagrama de flujo muestra con claridad la repetición. La línea de flujo que parte del rectángulo retorna a la decisión misma, que es probada cada vez dentro del ciclo, hasta que de forma eventual la decisión se convierte en falsa. Llegado este momento, se sale de la estructura **while** y el control pasa al siguiente enunciado del programa.

Cuando se escribe la estructura **while,** el valor de **product** es 2. La variable **product** se multiplica repetidamente por 2, asumiendo los valores 4, 8, 16, 32, 64, 128, 256, 512 y 1024 sucesivamente. Cuando **product** se convierte en 1024, la condición en la estructura **while, product <= 1000,** se hace falsa. Con ello se termina la repetición y el valor final de **product** es 1024. La ejecución del programa continúa con el enunciado que sigue después de **while.**

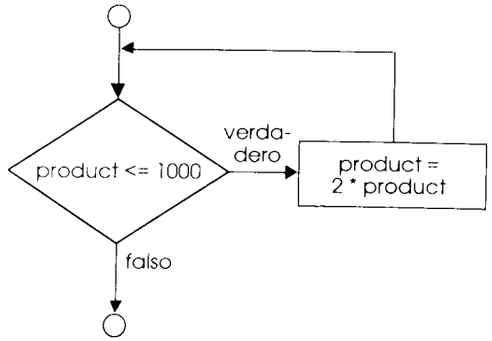


Fig. 3.4 Diagrama de flujo de la estructura de repetición **while.**

3.8 Cómo formular algoritmos: Estudio de caso 1 (repetición controlada por contador)

A fin de ilustrar cómo se desarrollan los algoritmos, resolveremos algunas variantes de un programa de promedios de clase. Considere el siguiente enunciado de programa:

> *Una clase de diez alumnos hizo un examen. Las calificaciones (enteros en el rango 0 a 100) correspondientes a este examen están a su disposición. Determine el promedio de la clase en este examen.*

El promedio de la clase es igual a la suma de las calificaciones dividida por el número de alumnos. El algoritmo para resolver este problema en una computadora, debe introducir cada una de las calificaciones, ejecutar el cálculo de promedio e imprimir el resultado.

Utilicemos el seudocódigo, enlistemos las acciones a ejecutarse, y especifiquemos el orden en el cual estas acciones deberán ser ejecutadas. Utilizaremos *repetición controlada por contador,* para introducir las calificaciones una a la vez. Esta técnica utiliza una variable llamada *contador* para definir el número de veces que deberá ejecutarse un conjunto de enunciados. En este ejemplo, la repetición terminará cuando el contador exceda de 10. En esta sección presentamos el algoritmo en seudocódigo (figura 3.5), así como el programa en C correspondiente (figura 3.6). En la siguiente sección, mostraremos cómo se desarrollan los algoritmos en seudocódigo. La repetición controlada por contador se conoce a menudo como *repetición definida* porque, antes de que se inicie la ejecución del ciclo, el número de repetición es conocido.

Advierta las referencias en el algoritmo a un total y a un contador. Un *total* es una variable, utilizada para acumular la suma de una serie de valores. Un contador es una variable, utilizada para contar en este caso, para contar el número de calificaciones capturadas. Por lo regular las variables utilizadas para almacenar totales, deberán ser inicializadas a cero antes de ser utilizadas en un programa; de lo contrario la suma incluiría el valor anterior almacenado en la posición en memoria de ese total. Las variables de contador se inicializan a cero o a uno, dependiendo en su uso (presentaremos ejemplos mostrando cada uno de ellos). Una variable sin inicializar contiene un *valor "basura"* que es el valor almacenado por última vez, en la posición de memoria reservada para la misma.

Error común de programación 3.5

Si no se inicializa un contador o un total, los resultados de su programa probablemente estarán incorrectos. Esto es un ejemplo de un error lógico.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 3.5 Algoritmo en seudocódigo, que utiliza repetición controlada por contador para resolver el problema de promedio de clase.

```
/* Class average program with
   counter-controlled repetition */
#include <stdio.h>

main()
{
   int counter, grade, total, average;

   /* initialization phase */
   total = 0;
   counter = 1;

   /* processing phase */
   while (counter <= 10) {
      printf("Enter grade: ");
      scanf("%d", &grade);
      total = total + grade;
      counter = counter + 1;
   }

   /* termination phase */
   average = total / 10;
   printf("Class average is %d\n", average);

   return 0;   /* indicate program ended successfully */
}
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Fig. 3.6 Programa en C y ejecución de muestra del problema de promedio de clase utilizando repetición controlada por contador.

Práctica sana de programación 3.6

Inicialice contadores y totales.

Advierta que el cálculo de promedios incluído en el programa produjo un resultado entero. De hecho, en este ejemplo, la suma de las calificaciones es 817, misma que al dividirse entre 10 debería resultar en 81.7, es decir, un número con un punto decimal. Veremos cómo enfrentar estos números (conocidos como números de punto flotante) en la sección siguiente.

3.9 Cómo formular algoritmos con refinamiento descendente paso a paso: Estudio del caso 2 (repetición controlada por centinela)

Generalicemos el problema de promedios de clase: considere el problema siguiente:

Desarrolle un programa de promedios de clase que pueda procesar un número arbitrario de calificaciones, cada vez que se ejecute el programa.

En el primer ejemplo de promedio de clase, se sabía por anticipado el número de calificaciones (10). En este ejemplo, no se da ninguna indicación de cuantas calificaciones se tomarán. El programa debe ser capaz de procesar un número arbitrario de calificaciones. ¿Cómo podrá el programa determinar cuándo parar la captura de calificaciones? ¿Cuándo sabrá que debe calcular e imprimir el promedio de clase?

Una forma de resolver este problema es utilizar un valor especial llamado un *valor centinela* (también conocido como *valor señal,* un *valor substituto,* o un *valor bandera*) que indicará "fin de la captura de datos". El usuario escribirá calificaciones hasta que haya capturado todas las calificaciones legítimas. Entonces escribirá un valor centinela, a fin de indicar que ha sido introducida la última calificación. La repetición controlada por centinela a menudo se llama *repetición indefinida,* porque antes de que se empiece a ejecutar el ciclo el número de repetición no es conocido.

Claramente, el valor centinela deberá ser seleccionado de tal forma que no se confunda con algún valor de entrada aceptable. Dado que normalmente las calificaciones de un examen son enteros no negativos, para este problema, -1 resulta un valor centinela aceptable. Entonces, una ejecución del programa de promedios de clase pudiera procesar un flujo de entradas como 95, 96, 75, 74, 89, y -1. El programa a continuación calcularía e imprimiría el promedio de clase para las calificaciones 95, 96, 75, 74, y 89 (-1 es el valor centinela y, por lo tanto, no debe entrar en el cálculo de promedio).

Error común de programación 3.6

Seleccionar un valor centinela que también pudiera resultar un valor legítimo de datos.

Enfocamos el programa de promedios de clase con una técnica conocida como *refinación descendente paso a paso,* técnica que resulta esencial al desarrollo de los programas bien estructurados. Empezamos con una representación en seudocódigo de lo más *general*:

Determinar el promedio de clase correspondiente al examen.

Lo más general es un enunciado que representa la función general del programa. Como tal, lo más general es en efecto, una representación completa de todo el programa. Desafortunadamente, lo más general (como en este caso) rara vez transmite una cantidad suficiente de detalle a partir del cual se pueda escribir un programa en C.

Por lo cual empezamos con el proceso de refinación. Dividimos lo más general en una serie de tareas más pequeñas y las enlistamos en el orden en el cual deberán ser ejecutadas. Esto da como resultado el siguiente primer refinamiento:

Inicializar variables
Captura, suma y cuenta de las calificaciones del examen
Calcular e imprimir el promedio de clase

Aquí, sólo se ha utilizado la estructura de secuencia —los pasos enlistados deberán ser ejecutados en orden, uno después del otro.

Observación de ingeniería de software 3.3

Cada refinamiento, al igual que la misma parte más general, debe ser una especificación completa del algoritmo; cambiando sólo el nivel de detalle.

Para seguir al siguiente nivel de refinamiento, es decir, el *segundo nivel de refinamiento,* nos comprometeremos a variables específicas. Necesitamos un total acumulado de los números, una cuenta de cuántos números han sido procesados, una variable para recibir el valor de cada calificación conforme es introducida, y una variable que contenga el promedio calculado. El enunciado en seudocódigo

Inicializar variables

pudiera ser refinado como sigue:

Inicializar total a cero
Inicializar contador a cero

Note que sólo el total y el contador deben ser inicializados; las variables promedio y calificación (para el promedio calculado y la captura del usuario, respectivamente) no necesitan ser inicializados, porque sus valores se escribirán mediante el proceso de lectura destructiva, analizado en el capítulo 2. El enunciado en seudocódigo

Entrada, suma y conteo de las calificaciones del examen

requiere de una estructura de repetición (es decir, de un ciclo), que introduzca sucesivamente cada calificación. Dado que no sabemos por anticipado cuantas calificaciones serán procesadas, utilizaremos la repetición controlada por centinela. El usuario escribirá grados legítimos uno por uno. Una vez que haya escrito el último grado legítimo, el usuario escribirá el valor centinela. Después de haber capturado cada calificación el programa hará una prueba buscando este valor, y dará por terminado el ciclo, cuando el centinela haya sido escrito. La refinación del enunciado precedente en seudocódigo, resulta entonces

Escriba la primera calificación
(While) en tanto el usuario no haya introducido todavía el centinela
    Añada esta calificación al total acumulado
    Añada uno al contador de calificaciones
    Introduzca la siguiente calificación (posiblemente el centinela)

Note que en seudocódigo, para formar el cuerpo de la estructura *while,* no utilizamos llaves alrededor de un conjunto de enunciados. Simplemente hacemos una sangría de todos estos enunciados bajo *while,* para mostrar que todos ellos corresponden al *while.* Repetimos, el seudocódigo es sólo una ayuda informal de desarrollo de programas.

El enunciado en seudocódigo

Calcular e imprimir el promedio de clase

puede ser refinado como sigue:

(If) si el contador no es igual a cero
    Haga que el promedio sea igual al total dividido por el contador
    Imprima el promedio
(else) de lo contrario
    Imprima "No se han escrito calificaciones"

Note que estamos teniendo cuidado aquí de probar la posibilidad de división entre cero un *error fatal* que, si no es detectado causaría que el programa fallara (a menudo llamado "*colapso del programa*"). El segundo refinamiento completo se muestra en la figura 3.7.


```
Initialize total to zero
initialize counter to zero

Input the first grade
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"
```

Fig. 3.7 Algoritmo en seudocódigo, que utiliza repetición controlada por centinela para resolver el problema de promedio de clase.

Error común de programación 3.7

Cualquier intento de dividir entre cero causará un error fatal.

Práctica sana de programación 3.7

Al ejecutar una división por una expresión cuyo valor pudiera ser cero, pruebe de forma explícita este caso y manéjelo apropiadamente en su programa (como sería impresión de un mensaje de error), en vez de permitir que ocurra un error fatal.

En las figuras 3.5 y 3.7, incluimos dentro del seudocódigo algunas líneas totalmente en blanco para mayor legibilidad. De hecho, las líneas en blanco separan estos programas en sus diferentes fases.

Observación de ingeniería de software 3.4

Muchos programas pueden ser divididos lógicamente en tres fases: una fase de inicialización, que inicializa las variables del programa; una fase de proceso, que captura valores de datos y ajusta las variables de programa correspondientemente; y una fase de terminación, que calcula e imprime los resultados finales.

El algoritmo en seudocódigo de la figura 3.7 resuelve el problema más general de promedio de clase. Este algoritmo fue desarrollado después de sólo dos niveles de refinamiento. A veces se requieren de más niveles.

Observación de ingeniería de software 3.5

El programador termina el proceso de refinamiento descendente paso a paso cuando el algoritmo en seudocódigo queda especificado con suficiente detalle para que el programador pueda convertirlo de seudocódigo a C. A partir de ahí la implantación del programa C es por lo regular sencilla.

En la figura 3.8 aparecen el programa C y una ejecución de muestra. A pesar de que sólo se han escrito calificaciones en enteros, el cálculo de promedio probable producirá un número decimal con un punto decimal. El tipo `int` no puede representar tal número. Para manejar números con puntos decimales el programa introduce el tipo de datos `float` (también conocidos como *números de punto flotante*) así como introduce un operador especial conocido como un *operador*

```
/* Class average program with
   sentinel-controlled repetition */
#include <stdio.h>

main()
{
   float average;              /* new data type */
   int counter, grade, total;

   /* initialization phase */
   total = 0;
   counter = 0;

   /* processing phase */
   printf("Enter grade, -1 to end: ");
   scanf("%d", &grade);

   while (grade != -1) {
      total = total + grade;
      counter = counter + 1;
      printf("Enter grade, -1 to end: ");
      scanf("%d", &grade);
   }

   /* termination phase */
   if (counter != 0) {
      average = (float) total / counter;
      printf("Class average is %.2f", average);
   }
   else
      printf("No grades were entered\n");

   return 0;   /* indicate program ended successfully */
}
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

Fig. 3.8 Programa en C y ejecución de muestra para el programa de promedio de clase mediante repetición controlada por centinela.

"cast" para manejar el cálculo de promedio. Estas características se explican en detalle después de presentar el programa.

Note el enunciado compuesto en el ciclo **while** de la figura 3.8. De nuevo, para que se ejecuten los cuatro enunciados dentro del ciclo son necesarias las llaves. Sin las llaves, los últimos

tres enunciados en el cuerpo del ciclo quedarían fuera del mismo, haciendo que la computadora interpretase este código en forma incorrecta, como sigue:

```
While(grade != -1)
   total = total + grade;
counter = counter + 1;
printf("Enter grade, -1 to end: ");
     scanf("%f", &grade);
```

Esto causaría un ciclo infinito, en tanto el usuario no introdujera -1 para la primera calificación.

Práctica sana de programación 3.8

En un ciclo controlado por centinela, las indicaciones solicitando la introducción de datos, deberían recordar de forma explícita al usuario cual es el valor centinela.

Los promedios no siempre resultan en valores enteros. A menudo, un promedio es un valor como 7.2 o -93.5, conteniendo una parte fraccionaria. Estos valores se conocen como números de punto flotante, y se representan por el tipo de datos **float**. La variable **average** se declara del tipo **float**, a fin de capturar el resultado fraccionario de nuestro cálculo. Sin embargo, el resultado del cálculo **total / counter** es un entero, porque tanto **total** como **counter** son ambas variables enteras. La división de dos enteros resulta en una *división de enteros,* en la cual se pierde la parte fraccionaria del cálculo (es decir, *se trunca*). Dado que este cálculo se ejecuta primero, se pierde la parte fraccionaria, antes de que el resultado sea asignado a **average**. A fin de producir un cálculo de punto flotante con valores enteros, debemos crear valores temporales que sean números de punto flotante para el cálculo. C proporciona un *operador cast unario* para esta tarea. El enunciado

```
average = (float) total / counter;
```

Incluye el operador cast (**float**) que crea una copia temporal de punto flotante de su propio operando, **total**. El uso de un operador cast de esta forma se conoce como *conversión explícita*. El valor almacenado en **total** sigue siendo un entero. El cálculo ahora consiste en un valor de punto flotante (la versión temporal **float** de **total**) dividida por el valor entero almacenado en **counter**. El compilador de C sólo sabe como evaluar expresiones en donde los tipos de datos de los operandos sean idénticos. A fin de asegurarse que los operandos sean del mismo tipo, el compilador lleva a cabo una operación denominada *promoción* (también conocida como *conversión implícita*) sobre los operandos seleccionados. Por ejemplo, en una expresión que contenga los tipos de datos **int** y **float**, el estándar ANSI especifica que se hacen copias de los operandos **int** y se *promueven* a **float**. En nuestro ejemplo, después de hacer una copia de **counter** y promoverla a **float**, se lleva a cabo el cálculo y el resultado de la división de punto flotante se asigna a **average**. El estándar ANSI incluye un conjunto de reglas para la promoción de operandos de diferentes tipos. En el capítulo 5 se presenta un análisis de todos los tipos estándar de datos y su orden de promoción.

Los operadores cast están disponibles para cualquier tipo de datos. El operador cast se forma colocando paréntesis alrededor del nombre de un tipo de datos. El operador cast es un *operador unario*, es decir, un operador que útiliza un operando. En el capítulo 2, estudiamos los operadores aritméticos binarios. C también acepta versiones unarias de los operadores más (**+**) y menos (**-**), de tal forma que el programador puede escribir expresiones como **-7** o **+5**. Los operadores cast se asocian de derecha a izquierda, y tienen la misma precedencia que los otros operadores unarios,

como el unario + y el unario -. Esta precedencia es de un nivel superior al de los *operadores multiplicativos* *, /, y %, y de un nivel menor que el de los paréntesis.

El programa de la figura 3.8 utiliza un especificador de conversión para **printf** igual a **%.2f** para la impresión del valor de **average**. La **f** indica que deberá ser impreso un valor de punto flotante. El **.2** es la *precisión* con la cual dicho valor se deberá desplegar. Indica que el valor será desplegado con 2 decimales a la derecha del punto decimal. Si se utiliza el especificador de conversión **%f** (sin especificar precisión), se utilizará la *precisión preestablecida* por omisión de 6 exactamente igual a si se utilizara un especificador de conversión **%.6f**. Cuando los valores de punto flotante se imprimen con precisión, el valor impreso se *redondea* al número indicado de posiciones decimales. El valor en memoria se conserva sin alteración. Cuando son ejecutados los enunciados siguientes, se imprimen los valores 3.45 y 3.4

```
printf("%.2f\n", 3.446);   /* prints 3.45 */
printf("%.1f\n", 3.446);   /* prints 3.4  */
```

Error común de programación 3.8

Es incorrecto utilizar precisión en una especificación de conversión en la cadena de control de formato de un enunciado **scanf**. *Las precisiones se utilizan sólo en la especificación de conversión de* **printf**.

Error común de programación 3.9

Utilizar números de punto flotante, de tal forma de que se suponga que están representados con precisión, puede llevar a resultados incorrectos. Los números de punto flotante son representados sólo en forma aproximada por la mayor parte de las computadoras.

Práctica sana de programación 3.9

No compare valores de punto flotante buscando igualdad.

A pesar del hecho de que los números de punto flotante no son siempre "100% precisos", tienen muchas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal "normal" de 98.6°F(37°C), no es necesario ser preciso hasta un gran número de dígitos. Cuando vemos la temperatura en un termómetro y la leemos como 98.6, pudiera en realidad ser 98.5999473210643. El punto aquí es que el expresar este número como 98.6 es suficiente para la mayor parte de las aplicaciones. Diremos más sobre este tema posteriormente.

Otra forma en que se desarrollan los números de punto flotante es mediante la división. Cuando dividimos 10 entre 3, el resultado es 3.3333333...con la secuencia de 3 repitiéndose en forma infinita. La computadora asignará cierta cantidad de espacio para contener un valor como éste, por lo que claramente el valor de punto flotante almacenado sólo puede ser una aproximación.

3.10 Cómo formular algoritmos con refinamiento descendente paso a paso: Estudio de caso 3 (estructuras de control anidadas)

Trabajemos otro problema completo. Volveremos a formular el algoritmo utilizando seudocódigo y refinamiento descendente paso a paso, y escribiremos el programa en C correspondiente. Hemos visto que las estructuras de control pueden ser apiladas una encima de otra (en secuencia), de la misma forma que un niño apila bloques. En este estudio de caso veremos la única otra forma estructurada que en C pueden conectarse estructuras de control, es decir mediante el *anidar* una estructura de control dentro de otra.

Considere el siguiente enunciado de problema:

Una universidad ofrece un curso que prepara alumnos para el examen estatal de licenciatura para corredores de bienes raíces. El año pasado, varios de los alumnos que terminaron este curso hicieron el examen de licenciatura. Naturalmente, la universidad desea saber qué tan bien salieron sus alumnos en el examen. Se le ha pedido a usted que escriba un programa para resumir los resultados. Se le ha dado una lista de estos diez alumnos. A continuación de cada nombre se ha escrito un 1 si el alumno pasó el examen y un 2 si no lo pasó.

Su programa deberá analizar los resultados del examen, como sigue:

1. Introducir cada resultado de prueba (es decir, 1 o 2). Desplegar en pantalla el mensaje "introducir resultado" cada vez que el programa solicite otro resultado de prueba.
2. Contar el número de resultados de prueba de cada tipo.
3. Desplegar un resumen de los resultados de prueba, indicando el número de alumnos que pasaron y el número de alumnos que reprobaron.
4. Si más de 8 alumnos pasaron el examen, imprima el mensaje "aumente la colegiatura".

Después de leer de forma cuidadosa el enunciado del problema, hacemos las siguientes observaciones:

1. El programa debe procesar 10 resultados de prueba. Se utilizará un ciclo controlado por contador.
2. Cada resultado de prueba es un número ya sea 1 ó 2. Cada vez que el programa lea un resultado de prueba, el programa debe determinar si el número es 1 ó 2. Probaremos buscando un 1 en nuestro algoritmo. Si el número no es un 1, supondremos que se trata de 2. (Un ejercicio al final de este capítulo analiza las consecuencias de esta suposición).
3. Se utilizarán dos contadores uno para contar el número de estudiantes o alumnos que pasaron el examen y uno para contar el número de alumnos que reprobaron el examen.
4. Después que el programa haya procesado todos los resultados, debe decidir si más de 8 alumnos pasaron el examen.

Procedamos con refinamiento descendente paso a paso. Empezamos con una representación general en seudocódigo:

Analice los resultados de examen y decida si debe aumentarse la colegiatura

De nuevo, es importante enfatizar que lo general es una representación completa del programa, pero es probable que se requerirán varios refinamientos, antes de que el seudocódigo pueda ser convertido naturalmente en un programa C. Nuestro primer refinamiento es:

Inicializar variables
Introduzca las diez calificaciones de examen, cuente los aprobados y los reprobados
Imprima un resumen de los resultados de examen, decida si debe aumentarse la colegiatura

Aquí, también, aunque tenemos una representación completa de la totalidad del programa, se requiere de aún más refinamiento. Debemos ahora fijar ciertas variables específicas. Se requieren de contadores para registrar los aprobados y los reprobados, un contador se utilizará para controlar el proceso del ciclo, y se necesita una variable para almacenar la entrada del usuario. El enunciado en seudocódigo.

Inicializar variables

puede ser refinado como sigue:

Inicializar aprobados a cero
Inicializar reprobados a cero
Inicializar alumnos a uno

Note que sólo los contadores y los totales se inicializan . El enunciado en seudocódigo

Introduzca las diez calificaciones de examen, cuente los aprobados y los reprobados

requiere de un ciclo que introduzca en lo sucesivo el resultado de cada examen. Aquí se sabe por anticipado que existirán precisamente diez resultados de exámenes, por lo que es apropiado utilizar un ciclo controlado por contador. Dentro del ciclo (es decir, *anidado* dentro del ciclo) una estructura de doble selección determinará si cada resultado de examen es un aprobado o un reprobado, e incrementará de forma correspondiente los contadores apropiados. Por lo tanto, el refinamiento del enunciado precedente en seudocódigo es

While En tanto el contador de alumnos sea menor o igual a diez
Introduzca el resultado del siguiente examen
If Si el alumno aprobó
Añada uno a aprobados
else De lo contrario
Añada uno a reprobados
Añada uno a contador de alumnos

Note la utilización de líneas en blanco para destacar la estructura de control *if/else*, a fin de mejorar la legibilidad del programa. El enunciado en seudocódigo

Imprima un resumen de los resultados de examen y decida si debe aumentarse la colegiatura

puede ser refinado como sigue:

Imprima el número de aprobados
Imprima el número de reprobados
Si ocho o más estudiantes aprobaron
Imprima "Aumentar colegiatura"

La segunda refinación completa aparece en la figura 3.9. Note que también se utilizan líneas en blanco para destacar la estructura *while* mejorando la legibilidad del programa.

Este seudocódigo está lo suficiente refinado para su conversión a C. El programa C y dos ejecuciones de muestra aparecen en la figura 3.10. Note que se ha aprovechado una característica de C, que permite que la inicialización sea incorporada en las declaraciones. Esta inicialización ocurre en tiempo de compilación.

Sugerencia de rendimiento 3.1

La inicialización de variables en el mismo momento que son declaradas reducen el tiempo de ejecución de un programa.

Observación de ingeniería de software 3.6

La experiencia ha mostrado que la parte más difícil para resolver un programa en una computadora es desarrollar el algoritmo de su solución. Una vez que se haya especificado el algoritmo correcto, el proceso de producir un programa C operacional por lo regular resulta simple.

Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
Input the next exam result

If the student passed
Add one to passes
else
Add one to failures

Add one to student counter

Print the number of passes
Print the number of failures
If eight or more students passed
Print "Raise tuition"

Fig. 3.9 Seudocódigo para el problema de resultados del examen.

Observación de ingeniería de software 3.7

Muchos programadores escriben programas sin jamás utilizar herramientas de desarrollo de programas, como es el seudocódigo. Sienten que su meta final es resolver el problema sobre una computadora, y que la escritura de seudocódigo sólo retarda la producción de los resultados finales.

3.11 Operadores de asignación

C dispone de varios operadores de asignación para la abreviatura de las expresiones de asignación. Por ejemplo, el enunciado

```
c = c + 3;
```

puede ser abreviado utilizando el *operador de asignación* `+=` como

```
c += 3;
```

El operador `+=` añade el valor de la expresión, a la derecha del operador, al valor de la variable a la izquierda del operador, y almacena el resultado en la variable a la izquierda del operador. Cualquier enunciado de la forma

variable = variable operador expresión;

donde *operador* es uno de los operadores binarios `+`, `-`, `*`, `/` o `%` (u otros que discutiremos en el capítulo 10), pueden ser escritos de la forma

variable operador= expresión;

```
/* Analysis of examination results */
#include <stdio.h>

main()
{
   /* initializing variables in declarations */
   int passes = 0, failures = 0, student = 1, result;

   /* process 10 students; counter-controlled loop */
   while (student <= 10) {
      printf("Enter result (1=pass,2=fail): ");
      scanf("%d", &result);

      if (result == 1)          /* if/else nested in while */
         passes = passes + 1;
      else
         failures = failures + 1;

      student = student + 1;
   }

   printf("Passed %d\n", passes);
   printf("Failed %d\n", failures);

   if (passes > 8)
      printf("Raise tuition\n");

   return 0;    /* successful termination */
}
```

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=paSS,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4
```

Fig. 3.10 Programa en C y ejecuciones de muestra para el problema de resultados de examen (parte 1 de 2).

Por lo tanto la asignación `c += 3` añade `3` a `c`. En la figura 3.11 aparecen los operadores de asignación aritméticos, con expresiones de muestra utilizando estos operadores y con explicaciones.

Sugerencia de rendimiento 3.2

Una expresión con un operador de asignación (como en `c += 3`) se compila más aprisa que la expresión equivalente expandida (`c = c + 3`) porque en la primera expresión `c` se evalúa únicamente una vez, en tanto que en la segunda se evalúa dos veces.


```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

Fig. 3.10 Programa C y ejecuciones de muestra para el problema de resultados de examen (parte 2 de 2).

Operador de asignación	Expresión de muestra	Explicación	Asignación
Assume : `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to `c`
`-=`	`d -= 4`	`d = d - 4`	1 to `d`
`*=`	`e *= 5`	`e = e * 5`	20 to `e`
`/=`	`f /= 3`	`f = f / 3`	2 to `f`
`%=`	`g %= 9`	`g = g % 9`	3 to `g`

Fig. 3.11 Operadores de asignación aritméticos.

Sugerencia de rendimiento 3.3

Muchas de las sugerencias de rendimiento que mencionamos en este texto dan como resultado mejoras nominales, por lo que el lector pudiera estar tentado a ignorarlos. El hecho es que el efecto acumulativo de todas estas mejoras de rendimiento pueden hacer que un programa ejecute en forma significativamente más rápida. Además, se obtiene una mejora significativa cuando una mejora, supuestamente sólo nominal, es introducida en un ciclo que pudiera repetirse un gran número de veces.

3.12 Operadores incrementales y decrementales

C también tiene el *operador incremental* unario, `++`, y el *operador decremental* unario `--`, que se resumen en la figura 3.12. Si una variable `c` es incrementada en 1, el operador incremental `++` puede ser utilizado en vez de las expresiones `c = c + 1` o bien `c += 1`. Si los operadores incrementales o decrementales son colocados antes de una variable, se conocen como los *operadores de preincremento* o de *predecremento*, respectivamente. Si los operadores incremen-

Operador	Expresión de muestra	Explicación
++	++a	Se incrementa **a** en 1 y a continuación se utiliza el nuevo valor de **a** en la expresión en la cual resida **a**.
++	a++	Utiliza el valor actual de **a** en la expresión en la cual reside **a**, y después se incrementa **a** en 1
--	--b	Se decrementa **b** en 1 y a continuación se utiliza el nuevo valor de **b** en la expresión en la cual reside **b**.
--	b--	Se utiliza el valor actual de **b** en la expresión en la cual reside **b**, y después se decrementa a **b** en 1.

Fig. 3.12 Los operadores incrementales y decrementales.

tales o decrementales se colocan después de una variable, se conocen como los *operadores de postincremento* o de *postdecremento*, respectivamente. El preincrementar (o predecrementar) una variable hace que la variable primero se incremente (o decremente) en 1, y a continuación el nuevo valor de la variable se utilizará en la expresión en la cual aparece. Si se postincrementa (o postdecrementa) la variable hace que el valor actual de la variable se utilice en la expresión en la cual aparece, y a continuación el valor de la variable se incrementará (o decrementará) en 1.

El programa de la figura 3.13 demuestra la diferencia entre las versiones de preincremento y de postincremento del operador ++. Postincrementar la variable c hará que se incremente, después de su uso en el enunciado **printf**. Preincrementar la variable c hará que se incremente, antes de su uso en el enunciado **printf**.

El programa despliega el valor de c antes y después de haber utilizado el operador ++. El operador decremental (--) funciona en forma similar.

Práctica sana de programación 3.10

Los operadores unarios deben ser colocados de forma directa al lado de sus operandos, sin espacios intermedios.

Los tres enunciados de asignación de la figura 3.10

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

pueden ser escritos de manera más concisamente con operadores de asignación, como

```
passes += 1;
failures += 1;
student += 1;
```

con operadores de preincremento como sigue

```
++passes;
++failures;
++student;
```



```
/* Preincrementing and postincrementing */
#include <stdio.h>

main()
{
   int c;

   c = 5;
   printf("%d\n", c);
   printf("%d\n", c++);        /* postincrement */
   printf("%d\n\n", c);

   c = 5;
   printf("%d\n", c);
   printf("%d\n", ++c);        /* preincrement */
   printf("%d\n", c);

   return 0;   /* successful termination */
}
```

```
5
5
6

5
6
6
```

Fig. 3.13 Muestra de la diferencia entre preincrementar y postincrementar.

o con operadores postincrementales, como sigue

```
passes++;
failures++;
student++;
```

Es importante hacer notar que al incrementar o decrementar una variable en un enunciado por sí mismo, las formas de preincremento y postincremento tienen el mismo efecto. Es sólo cuando la variable aparece en el contexto de una expresión más grande que el preincrementar y postincrementar tienen efectos distintos (y similar para predecrementar y postdecrementar).

Sólo se puede utilizar un nombre simple de variable como operando de un operador incremental o decremental.

Error común de programación 3.10

Intentar utilizar el operador incremental o decremental en una expresión distinta a la de un nombre simple de variable, es decir escribiendo ++(x+1) es un error de sintaxis.

Práctica sana de programación 3.11

Por lo regular el estándar ANSI no especifica el orden en el cual serán evaluados los operandos de un operador (aunque veremos excepciones a lo anterior en el caso de unos cuantos operadores en el capítulo 4). Por lo tanto, el programador deberá de evitar utilizar enunciados con operadores incrementales o decrementales, en los cuales una variable particular sujeta a incremento o decremento aparezca más de una vez.

La tabla en la figura 3.14 muestra la precedencia y asociatividad de los operadores presentados hasta este punto. Los operadores se muestran de arriba a abajo en orden decreciente de precedencia. La segunda columna describe la asociatividad de los operadores en cada uno de los niveles de precedencia. Note que el operador condicional (`?:`), los operadores unarios e incremental (`++`), decremental (`--`), más (`+`), menos (`-`) y los cast, así como los operadores de asignación `=`, `+=`, `-=`, `*=`, `/=` y `%=` se asocian de derecha a izquierda. La tercera columna identifica los diferentes grupos de operadores. Todos los demás operadores de la figura 3.14 se asocian de izquierda a derecha.

Resumen

- La solución a cualquier problema de cómputo involucra la ejecución de una serie de acciones en un orden específico. Un procedimiento para resolver un problema en términos de las acciones a ejecutarse y del orden en el cual estas acciones deben de ejecutarse se conoce como un algoritmo.
- La especificación en un programa de cómputo del orden en el cual los enunciados deben ser ejecutados, se conoce como control de programa.
- El seudocódigo es un lenguaje artificial e informal, que auxilia a los programadores en el desarrollo de algoritmos. Es similar al inglés coloquial. De hecho los programas en seudocódigo no se pueden ejecutar en las computadoras. Más bien, el seudocódigo tan sólo ayuda al programador a "pensar" un programa, antes de intentar su escritura en un lenguaje de programación, como es C.
- El seudocódigo está formado sólo por caracteres, por lo que los programadores pueden escribir los programas en seudocódigo en la computadora, editarlos y guardarlos.

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
`?:`	derecha a izquierda	condicional
`= += -= *= /= %=`	derecha a izquierda	de asignación

Fig. 3.14 Precedencia de los operadores utilizados hasta ahora en el texto.

- El seudocódigo comprende solamente enunciados ejecutables. Las declaraciones son mensajes al compilador, indicándole los atributos de variables, y diciéndole que reserve espacio para las mismas.
- Una estructura de selección se utiliza para elegir entre cursos alternativos de acción.
- La estructura de selección **`if`** ejecuta una acción indicada, sólo cuando la condición es verdadera.
- La estructura de selección **`if/else`** define acciones diferentes a ejecutarse cuando la condición es verdadera, y cuando la condición es falsa.
- Una estructura de selección anidada **`if/else`** puede probar muchos casos diferentes. Si más de una condición es verdadera, sólo serán ejecutados los enunciados después de la primera condición verdadera.
- Siempre que deba ejecutarse más de un enunciado, donde por lo regular un enunciado se espera, estos enunciados deberán ser colocados entre llaves, para formar un enunciado compuesto. Un enunciado compuesto puede ser colocado en cualquier sitio donde pueda colocarse un enunciado simple.
- Un enunciado vacío, que indica que no debe tomarse ninguna acción, se identifica colocando un punto y coma (`;`) donde debería estar el enunciado.
- Una estructura de repetición define que una acción deberá ser repetida, en tanto cierta condición siga siendo verdadera.
- El formato para la estructura de repetición **`while`** es

```
while (condición)
   enunciado
```

- El enunciado (o enunciado compuesto o bloque), contenido en la estructura de repetición **`while`** constituye el cuerpo del ciclo.
- Por lo regular, alguna acción especificada dentro del cuerpo de un **`while`** eventualmente debe hacer que la condición se convierta en falsa. De lo contrario, el ciclo no terminará nunca —un error conocido como ciclo infinito.
- Los ciclos controlados por contador utilizan una variable como contador, para determinar cuando debe terminar el ciclo.
- Un total es una variable que acumula la suma de una serie de números. Por lo regular, los totales deberán ser inicializados a cero, antes de ejecutar un programa.
- Un diagrama de flujo es una representación gráfica de un algoritmo. Los diagramas de flujo se dibujan utilizando ciertos símbolos especiales como son óvalos, rectángulos, diamantes y pequeños círculos conectados por flechas, llamadas líneas de flujo. Los símbolos indican acciones a ejecutarse. Las líneas de flujo indican el orden en el cual se deberán ejecutar las acciones.
- El símbolo oval, también conocido como símbolo de terminación, indica el principio y el final de todo algoritmo.
- El símbolo rectángulo, también conocido como símbolo de acción, indica cualquier tipo de cálculo, o de operación de entrada/salida. Los símbolos rectángulo corresponden a las acciones ejecutadas por enunciados de asignación, o por operaciones de entrada/salida, llevadas a cabo casi siempre por funciones estándar de biblioteca como **`printf`** y **`scanf`**.

- El símbolo diamante, también conocido como símbolo de decisión, indica que se debe tomar una decisión. El símbolo de decisión contiene una expresión, que puede ser o verdadera o falsa. Dos líneas de flujo parten de este símbolo. Una de estas líneas de flujo indica la dirección a tomar cuando la condición es verdadera; la otra indica la dirección a tomar cuando la condición es falsa.
- Un valor que contiene una parte fraccionaria es conocido como un número de punto flotante, y se representa por el tipo de datos **`float`**.
- La división de dos enteros resulta en una división de enteros, en el cual se perderá (es decir, quedará truncada) cualquier parte fraccionaria resultante del cálculo.
- C dispone de un operador unario cast (**`float`**) para crear una copia temporal de punto flotante de su operando. El uso de esta forma de un operador cast, se conoce como conversión explícita. Los operadores cast están disponibles para cualquier tipo de datos.
- El compilador de C sólo sabe evaluar expresiones en las cuales son idénticos los tipos de datos de los operandos. Para asegurarse que los operandos sean del mismo tipo, el compilador ejecuta una operación conocida como promoción (también llamada conversión implícita), sobre los operandos seleccionados. El estándar o norma ANSI especifica que se hagan copias de los operandos **`int`** y se promuevan a **`float`**. El estándar ANSI proporciona un conjunto de reglas para la promoción de operandos de diferentes tipos.
- Los valores de punto flotante son sacados en un enunciado **`printf`** con un número específico de dígitos después del punto decimal, mediante el uso de una precisión en el especificador de conversión **`%f`**. El valor **`3.456`**, salido con el especificador de conversión **`%.2f`** aparecerá desplegado como **`3.46`**. Si se utiliza el especificador de conversión **`%f`** (sin especificar precisión), será utilizada la precisión preestablecida por omisión de 6.
- C contiene varios operadores de asignación, que ayudan a abreviar ciertos tipos comunes de expresiones aritméticas de asignación. Estos operadores son: **`+=`**, **`-=`**, **`*=`**, **`/=`**, y **`%=`**. En general, cualquier enunciado de la forma

  variable = *variable operador expresión*;

  donde **`operador`** es uno de los operadores **`+`**, **`-`**, **`*`**, **`/`**, o **`%`**, puede ser escrito de la forma

  variable operador = *expresión*;

- C proporciona el operador incremental **`++`**, y el operador decremental **`--`**, para incrementar o decrementar una variable en 1. Estos operadores pueden ser prefijos o postfijos a una variable. Si el operador es prefijo a la variable, esta será incrementada o decrementada primero por 1, y a continuación utilizada en su expresión. Si el operador es postfijo a la variable, esta será utilizada en su expresión, y a continuación incrementada o decrementada en 1.

Terminología

acción
símbolo de acción
algoritmo
operadores aritméticos de asignación: **`+=`**, **`-=`**, **`*=`**, **`/=`** y **`%=`**
símbolo de flecha
bloque
cuerpo de un cliclo
"colapso"
operador cast
enunciado compuesto
operador condicional (**`?:`**)
estructura de control
contador
repetición controlada por contador
"choque"
decisión
símbolo de decisión
operador decremental (**`--`**)
precisión preestablecida o por omisión
repetición definida
símbolo diamante
división entre cero
estructura de doble selección
valor sustituto
enunciado vacío (**`;`**)
"entrada de fin de datos"
símbolo de terminación
conversión explícita
error fatal
primera refinación
valor bandera
`float`
número de punto flotante
diagrama de flujo
símbolo de diagrama de flujo
línea de flujo
valor "basura"
eliminación **`goto`**
enunciado **`goto`**
estructura de selección **`if/else`**
estructura de selección **`if`**
conversión implícita
operador incremental (**`++`**)
repetición indefinida
ciclo infinito
inicialización
fase de inicialización
división de enteros
error lógico
ciclar
estructura de múltiple selección
operadores multiplicativos
estructuras de control anidadas
estructuras **`if/else`** anidadas
error no fatal
orden de acciones
símbolo oval
operador postdecremental
operador postincremental
precisión
operador predecremental
operador preincremental
fase de procesamiento
control de programa
promoción
seudocódigo
símbolo rectángulo
repetición
estructuras de repetición
redondeo
segunda refinación
selección
estructura de selección
valor centinela
ejecución secuencial
estructura en secuencia
valor señal
estructuras de control de una entrada/una salida
estructura de una selección
estructuras de control apiladas
pasos
refinación paso a paso
programación estructurada
error de sintaxis
condición de terminación
fase de terminación
símbolo de terminación
operador ternario
general
refinación descendente paso a paso
total
transferencia de control
truncamiento
estructura de repetición **`while`**
caracteres de espacio en blanco

Errores comunes de programación

3.1 Olvidar una o ambas de las llaves que delimitan un enunciado compuesto.
3.2 Colocar en una estructura **`if`** un punto y coma despues de la condición, llevará a un error lógico en estructuras **`if`** de una sola selección y a un error de sintaxis en estructuras **`if`** de doble selección.

3.3 No incluir en el cuerpo de una estructura `while` una acción que haga que la condición existente en `while` eventualmente se convierta en falsa. Por lo regular, esta estructura de repetición no terminará jamás un error conocido como "ciclo infinito".
3.4 Escribir la palabra reservada `while` con una `W` como `While` (recuerde que C es un lenguaje que es sensible a la caja tipográfica). Todas las palabras reservadas de C, como `while`, `if` y `else`, contienen sólo letras minúsculas.
3.5 Si no se inicializa un contador o un total, los resultados de su programa es probable que estarán incorrectos. Esto es un ejemplo de un error lógico.
3.6 Seleccionar un valor centinela que también pudiera resultar un valor legítimo de datos.
3.7 Cualquier intento de dividir entre cero causará un error fatal.
3.8 Es incorrecto utilizar precisión en una especificación de conversión en la cadena de control de formato de un enunciado `scanf`. Las precisiones se utilizan sólo en la especificación de conversión de `printf`.
3.9 Utilizar números de punto flotante, de tal forma de que se suponga que están representados con precisión, puede llevar a resultados incorrectos. Los números de punto flotante son representados sólo en forma aproximada por la mayor parte de las computadoras.
3.10 Intentar utilizar el operador incremental o decremental en una expresión distinta a la de un nombre simple de variable, es decir, escribiendo `++(x+1)` es un error de sintaxis.

Prácticas sanas de programación

3.1 La aplicación consistente de reglas convencionales responsables para las sangrías mejora en forma importante la legibilidad de los programas. Sugerimos un tabulador de tamaño fijo de aproximadamente 1/4 de pulgada o de tres espacios por cada sangría.
3.2 A menudo se utiliza el seudocódigo durante el proceso de diseño para "pensar en voz alta" un programa. Posteriormente el programa en seudocódigo se convierte a C.
3.3 Haga sangrías en ambos cuerpos de los enunciados de una estructura `if/else`.
3.4 Si existen varios niveles de sangría, cada nivel deberá tener sangría con una cantidad igual de espacio.
3.5 Algunos programadores prefieren escribir las llaves de principio y de terminación de los enunciados compuestos antes de escribir en el interior de dichas llaves los enunciados individuales. Esto les ayuda a evitar la omisión de una o ambas de las llaves.
3.6 Inicialice contadores y totales.
3.7 Al ejecutar una división por una expresión cuyo valor pudiera ser cero, pruebe de forma explícita este caso y manéjelo de manera apropiada en su programa (como sería impresión de un mensaje de error), en vez de permitir que ocurra un error fatal.
3.8 En un ciclo controlado por centinela, las indicaciones solicitando la introducción de datos, deberían recordar de forma explícita al usuario cual es el valor centinela.
3.9 No compare valores de punto flotante buscando igualdad.
3.10 Los operadores unarios deben ser colocados en directo al lado de sus operandos, sin espacios intermedios.
3.11 Por lo regular el estándar ANSI no especifica el orden en el cual serán evaluados los operandos de un operador (aunque veremos excepciones a lo anterior en el caso de unos cuantos operadores en el capítulo 4). Por lo tanto, el programador deberá de evitar utilizar enunciados con operadores incrementales o decrementales, en los cuales una variable particular sujeta a incremento o decremento aparezca más de una vez.

Sugerencias de rendimiento

3.1 La inicialización de variables en el mismo momento que son declaradas reducen el tiempo de ejecución de un programa.

3.2 Una expresión con un operador de asignación (como en `c += 3`) se compila más aprisa que la expresión equivalente expandida (`c = c + 3`) porque en la primera expresión *c* se evalúa una vez, en tanto que en la segunda se evalúa dos veces.
3.3 Muchos de las sugerencias de rendimiento que mencionamos en este texto dan como resultado mejoras nominales, por lo que el lector pudiera estar tentado a ignorarlos. El hecho es que el efecto acumulativo de todas estas mejoras de rendimiento pueden hacer que un programa ejecute en forma significativamente más rápida. Además, se obtiene una mejora significativa cuando una mejora, supuestamente solo nominal, es introducida en un ciclo que pudiera repetirse un gran número de veces.

Observaciones de ingeniería de software

3.1 Un enunciado compuesto puede ser colocado en cualquier parte de un programa, donde pueda ser colocado un enunciado sencillo.
3.2 Igual que un enunciado compuesto puede ser colocado en cualquier lugar donde pudiera colocarse un enunciado sencillo, también es posible que no haya ningún enunciado de ningún tipo, es decir, un enunciado vacío. El enunciado vacío se representa colocando un punto y coma (;) donde por lo regular debería estar el enunciado.
3.3 Cada refinamiento, al igual que la misma parte más general, debe ser una especificación completa del algoritmo; cambiando sólo el nivel de detalle.
3.4 Muchos programas pueden ser divididos de forma lógica en tres fases: una fase de inicialización, que inicializa las variables del programa; una fase de proceso, que captura valores de datos y ajusta las variables de programa correspondiente; y una fase de terminación, que calcula e imprime los resultados finales.
3.5 El programador termina el proceso de refinamiento descendente paso a paso cuando el algoritmo en seudocódigo queda especificado con suficiente detalle para que el programador pueda convertirlo de seudocódigo a C. A partir de ahí la implantación del programa C es sencilla.
3.6 La experiencia ha mostrado que la parte más difícil para resolver un programa en una computadora es desarrollar el algoritmo de su solución. Una vez que se haya especificado el algoritmo correcto, el proceso de producir un programa C operacional resulta simple.
3.7 Muchos programadores escriben programas sin jamás utilizar herramientas de desarrollo de programas, como es el seudocódigo. Sienten que su meta final es resolver el problema sobre una computadora, y que la escritura de seudocódigo sólo retarda la producción de los resultados finales.

Ejercicios de autoevaluación

3.1 Responda a cada una de las preguntas siguientes.
a) Un procedimiento para resolver un problema en términos de las acciones a ejecutarse y del orden en el cual deberán dichas acciones ejecutarse, se conoce como un _____.
b) Especificar el orden de ejecución de los enunciados por parte de la computadora, se llama _____.
c) Todos los programas pueden ser escritos en función de tres estructuras de control: _____, _____, y _____.
d) La estructura de selección _____ se utiliza para ejecutar una acción cuando una condición es verdadera, y otra acción cuando la condición es falsa.
e) Varios enunciados agrupados juntos en llaves (`{` y `}`) se conocen como un _____.
f) La estructura de repetición _____ especifica que un enunciado o grupo de enunciados debe ser ejecutado de forma repetidamente, en tanto cierta condición se mantenga verdadera.
g) La repetición de un conjunto de instrucciones un número específico de veces, se conoce como una repetición _____.
h) Cuando no se conoce por anticipado cuantas veces debe repetirse un conjunto de enunciados, se puede utilizar un valor _____ para terminar la repetición.

3.2 Escriba cuatro enunciados diferentes en C, que cada uno de ellos añada uno a la variable entera **x**.

3.3 Escriba un enunciado en C para llevar a cabo cada uno de los siguientes:
a) Asignar la suma de **x** y **y** a **z**, y después del cálculo incrementar el valor de **x** en 1.
b) Multiplicar la variable **product** por 2, utilizando el operador ***=**.
c) Multiplicar la variable **product** por 2, utilizando los operadores **=** y *****.
d) Verificar si el valor de la variable **count** es mayor que 10. Si es así, imprima "**Count is greater than 10**".
e) Decremente la variable **x** en 1 y a continuación réstela de la variable **total**.
f) Añada la variable **x** a la variable **total**, y a continuación decremente **x** en 1.
g) Calcule el residuo, después que **q** se haya dividido por **divisor**, y asigne el resultado a **q**. Escriba este enunciado de dos formas distintas.
h) Imprima el valor **123.4567** con **2** dígitos de precisión ¿Cuál será el valor impreso?
i) Imprima el valor de punto flotante **3.14159** con tres dígitos a la derecha del punto decimal. ¿Cuál será el valor impreso?.

3.4 Escriba un enunciado C, que ejecute cada una de las tareas siguientes:
a) Declare las variables **sum** y **x** del tipo **int**.
b) Inicialice la variable **x** a 1.
c) Inicialice la variable **sum** a **0**.
d) Añada la variable **x** a la variable **sum** y asigne el resultado a la variable **sum**.
e) Imprima "**The sum is: **" seguido por el valor de la variable **sum**.

3.5 Combine los enunciados que escribió en el Ejercicio 3.4 en un programa que calcule la suma de los enteros de 1 a 10. Utilice la estructura **while** para ciclar a través del cálculo y de los enunciados de incremento. El ciclo deberá terminar cuando el valor de **x** se convierta en 11.

3.6 Determinar los valores de cada variable, después de que se haya ejecutado el cálculo. Suponga que cuando empieza cada enunciado, todas las variables tienen el valor 5.
a)
```
product *= x++;
```
b)
```
result = ++x + x;
```

3.7 Escriba enunciados simples en C que:
a) Introduzcan la variable entera **x** mediante **scanf**.
b) Introduzcan la variable entera **y** mediante **scanf**.
c) Inicialicen la variable entera **i** a **1**.
d) Inicialicen la variable entera **power** a **1**.
e) Multipliquen la variable **power** por **x** y asignen el resultado a **power**.
f) Incrementen la variable **y** a **1**.
g) Prueben **y** para ver si es menor o igual a **x**.
h) Saquen la variable entera **power** mediante **printf**.

3.8 Escriba un programa en C que utilice los enunciados del Ejercicio 3.7 para calcular **x** elevada a la potencia **y**. El programa deberá contener una estructura de control de repetición **while**.

3.9 Identifique y corrija los errores en cada uno de los siguientes:
a)
```
while (c  <=  5) {
      product  *= c;
      ++c;
```
b)
```
scanf("%.4F", &value);
```
c)
```
if (gender == 1)
      printf("Woman\n");
else;
      printf("Man\n");
```


3.10 Indique qué es lo que no está bien en la siguiente estructura de repetición **while**:
```
while (z = 0)
   sum += z;
```

Respuestas a los ejercicios de autoevaluación

3.1 a) Algoritmo. b) Control de programa. c) Secuencia, selección, repetición. d) **if/else**. e) Enunciado compuesto. f) **while**. g) Controlado por contador. h) Centinela.

3.2
```
x = x + 1;
x += 1;
++x;
x++;
```

3.3 a)
```
z = x++ + y;
```
b)
```
product *= 2;
```
c)
```
product = product * 2;
```
d)
```
if (count > 10)
      printf("Count is greater than 10.\n");
```
e)
```
total -= --x;
```
f)
```
total += x--;
```
g)
```
q %= divisor;
q = q % divisor;
```
h)
```
printf("%.2f", 123.4567);
```
123.46 is displayed.
i)
```
printf("%.3f\n", 3.14159);
```
3.142 is displayed.

3.4 a)
```
int sum, x;
```
b)
```
x = 1;
```
c)
```
sum = 0;
```
d)
```
sum += x;
```
or
```
sum = sum + x;
```
e)
```
printf("The sum is: %d\n", sum);
```

3.5
```
/* Calculate the sum of the integers from 1 to 10 */
#include <stdio.h>

main()
{
   int sum, x;

   x = 1;
   sum = 0;
   while (x <= 10) {
      sum += x;
      ++x;
   }

   printf("The sum is: %d\n", sum);
}
```

3.6 a)
```
product = 25, x = 6;
```
b)
```
result = 12, x = 6;
```

3.7

a) `scanf("%d", &x);`
b) `scanf("%d", &y);`
c) `i = 1;`
d) `power = 1;`
e) `power *= x;`
f) `y++;`
g) `if(y <= x)`
h) `printf("%d", power);`

3.8

```
/* raise x to the y power */
#include <stdio.h>

main()
   int x,y,i,power;

   i = 1;
   power = 1;
   scanf("%d", &x);
   scanf("%d", &y);

   while (i <= y) {
      power *= x;
      ++i;
   }

   printf("%d", power);
   return 0;
}
```

3.9 a) Error: falta la llave derecha de cierre del cuerpo **`while`**.
Corrección: Añada la llave de cierre derecha, después del enunciado **`++c;`**.
b) Error: precisión utilizada en una especificación de conversión **`scanf`**.
Corrección: Eliminar **`.4`** de la especificación de conversión **`scanf`**.
c) Error: punto y coma después de la parte **`else`** de la estructura **`if/else`** da como resultado un error lógico.
El segundo **`printf`** se ejecutará siempre.
Corrección: eliminar el punto y coma después de **`else`**.

3.10 El valor de la variable **`z`** no se cambia nunca en la estructura **`while`**. Por lo tanto se ha creado un ciclo infinito. Para evitar el ciclo infinito, **`z`** debe ser decrementado, para que de forma eventual se convierta en **`0`**.

Ejercicios

3.11 Identifique y corrija los errores en cada uno de los siguientes (Nota: pudiera haber más de un error en cada segmento de código):

a)
```
if (age >= 65);
   printf("Age is greater than or equal to 65\n");
else
   printf("Age is less than 65\n");
```

b)
```
int x = 1, total;

while (x <= 10) {
   total += x;
   ++x;
}
```


c)
```
While (x <= 100)
   total += x;
   ++x;
```

d)
```
while (y > 0) {
   printf("%d\n", y);
   ++y;
}
```

3.12 Llene los espacios vacíos en cada uno de los siguientes:
a) La solución a cualquier problema involucra la ejecución de una serie de acciones en un _____ específico.
b) Un sinónimo de procedimiento es _____.
c) Una variable que acumula la suma de varios números es un _____.
d) El proceso de definir ciertas variables en valores específicos al principio de un programa, se conoce como _____.
e) Un valor especial utilizado para indicar "entrada de fin de datos" se conoce como un valor _____, _____, _____, o _____.
f) Un _____ es una representación gráfica de un algoritmo.
g) En un diagrama de flujo, el orden en el cual deben ser ejecutados los pasos se indican por los símbolos _____.
h) El símbolo de terminación indica el _____, así como el _____ de todo algoritmo.
i) Los símbolos rectángulo corresponden a cálculos que por lo regular se ejecutan por enunciados _____ y por operaciones de entrada/salida _____ que se llevan normalmente a cabo mediante llamadas a las _____ y _____ de las funciones estándar de biblioteca.
j) El elemento escrito dentro de un símbolo decisión se conoce como una _____.

3.13 ¿Qué es lo que imprime el siguiente programa?

```
#include <stdio.h>

main()
{
   int x = 1, total;

   while (x <= 10) {
      y = x * x;
      printf("%d\n", y);
      total += y;
   }

   printf("Total is %d\n", total);
   return 0;
}
```

3.14 Escriba un enunciado en seudocódigo que indique cada uno de los siguientes:
a) Despliegue el mensaje "**`Enter two numbers`**".
b) Asigne la suma de las variables **`x`**, **`y`** y **`z`** a la variable **`p`**.
c) La condición siguiente debe de ser probada en una estructura de selección **`if/else`**: el valor actual de la variable **`m`** es mayor que dos veces el valor actual de la variable **`v`**.
d) Obtener del teclado valores para las variables **`s`**, **`r`** y **`t`**.

3.15 Formule un algoritmo en seudocódigo para cada uno de los siguientes:
a) Obtenga dos números del teclado, calcule la suma de los números y despliegue el resultado.
b) Obtenga dos números del teclado, y determine y despliegue cuál (si alguno) es el mayor de los dos números.

c) Obtenga una serie de números positivos del teclado, y determine y despliegue la suma de los números. Suponga que el usuario escribe el valor centinela `-1` para indicar "entrada de fin de datos".

3.16 Indique cuál de los siguientes es verdadero y cual es falso. Si el enunciado es falso, explique por qué.

a) La experiencia ha demostrado que la parte más difícil de la resolución de un problema en una computadora es el producir un programa C funcional.
b) Un valor centinela debe ser un valor que no pueda ser confundido con un valor de datos legítimo.
c) Las líneas de flujo indican las acciones a ejecutarse.
d) Las condiciones escritas dentro de los símbolos de decisión siempre contienen operadores aritméticos (es decir `+`, `-`, `*`, `/` y `%`).
e) En la refinación descendente paso a paso, cada refinación es una representación completa del algoritmo.

Para los ejercicios 3.17 a 3.21, lleve a cabo cada uno de estos pasos:

1. Lea el enunciado del problema.
2. Formule el algoritmo utilizando refinación descendente paso a paso.
3. Escriba un programa en C.
4. Pruebe, depure y ejecute el programa en C.

3.17 En razón del alto precio de la gasolina, los conductores están preocupados con el kilometraje que obtienen de sus automóviles. Un conductor ha llevado registro de varios tanques de gasolina, anotando las millas manejadas y los galones utilizados en cada uno de los tanques. Desarrolle un programa en C que introduzca las millas manejadas y los galones utilizados para cada tanque. El programa debe calcular y desplegar las millas por galón obtenidas de cada tanque. Después de procesar toda la información de entrada, el programa deberá calcular e imprimir las millas por galón combinadas, obtenida de todos los tanques.

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles / gallon for this tank was 22.421875

Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles / gallon for this tank was 19.417475

Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles / gallon for this tank was 24.000000

Enter the gallons used (-1 to end): -1
The overall average miles/gallon was 21.601423
```

3.18 Desarrolle un programa en C que determine si un cliente de una tienda departamental ha excedido el límite de crédito en una cuenta de cargo. Para cada uno de los clientes, están disponibles los siguientes hechos:

1. Número de la cuenta.
2. Saldo al principio del mes.
3. Total de todos los elementos cargados por el cliente este mes.
4. Total de todos los créditos aplicados este mes a la cuenta de este cliente.
5. Límite permitido de crédito.

El programa deberá introducir cada uno de estos hechos, calcular el nuevo saldo (= saldo inicial + cargos - créditos), y determinar si el nuevo saldo excede el límite de crédito del cliente. Para aquellos clientes cuyo límite de crédito esté excedido, el programa deberá desplegar el número de cuenta del cliente, el límite de crédito, el nuevo saldo y el mensaje "límite de crédito excedido".

```
Enter account number (-1 to end): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
Account:       100
Credit limit: 5500.00
Balance:      5894.78
Credit Limit Exceeded.

Enter account number (-1 to end): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00

Enter account number (-1 to end): 300
Enter beginning balance: 500.00
Enter total charges: 274.73
Enter total credits: 100.00
Enter credit limit: 800.00

Enter account number (-1 to end): -1
```

3.19 Una gran empresa química paga a su personal de ventas en base a comisiones. El personal de ventas recibe $200 por semana más 9% de las ventas brutas de esa semana. Por ejemplo, una persona de ventas que vende $5000 de productos químicos en una semana, recibe $200 más 9 % de $5000, o sea un total de $650. Desarrolle un programa en C que introduzca las ventas brutas de cada vendedor correspondiente a la última semana, y calcule y despliegue las ganancias de dicho vendedor. Procese las cifras vendedor por vendedor.

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 1234.56
Salary is: $311.11

Enter sales in dollars (-1 to end): 1088.89
Salary is: $298.00

Enter sales in dollars (-1 to end): -1
```

3.20 El interés simple de un préstamo se calcula mediante la fórmula

```
interest = principal * rate * days / 365
```

La fórmula anterior supone que `rate` es la tasa de interés anual y, por lo tanto, incluye la división entre 365 (días). Desarrolle un programa en C que introduzca `principal`, `rate` y `days` para varios prestamos diferentes, y que calcule y despliegue el interés simple para cada uno de ellos, mediante el uso de la fórmula anterior.

```
Enter loan principal (-1 to end): 1000.00
Enter interest rate: .1
Enter term of the loan in days: 365
The interest charge is $100.00

Enter loan principal (-1 to end): 1000.00
Enter interest rate: .08375
Enter term of the loan in days: 224
The interest charge is $51.40

Enter loan principal (-1 to end): 10000.00
Enter interest rate: .09
Enter term of the loan in days: 1460
The interest charge is $3600.00

Enter loan principal (-1 to end): -1
```

3.21 Desarrolle un programa en C que determine la nómina bruta para cada uno de varios empleados. La empresa paga "tiempo normal" para las primeras 40 horas trabajadas de cada empleado y paga "tiempo y medio" para todas las horas trabajadas en exceso de 40 horas. Se le proporciona una lista de los empleados de la empresa, el número de horas que cada empleado trabajó la última semana, y la tasa horaria de cada empleado. Su programa deberá introducir esta información para cada uno de los empleados, y determinar y desplegar la nómina bruta de cada uno de ellos.

```
Enter # of hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter # of hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter # of hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter # of hours worked (-1 to end): -1
```

3.22 Escriba un programa en C que demuestre la diferencia entre predecrementar y postdecrementar, utilizando el operador decremental `--`.

3.23 Escriba un programa en C que utilice ciclos para imprimir los números del 1 al 10, lado a lado en el mismo renglón, con tres espacios entre cada uno de ellos.

3.24 El proceso de encontrar el número más grande (es decir, el máximo de un grupo de números), es utilizado con frecuencia en aplicaciones de computación. Por ejemplo, un programa que determina el ganador de un concurso de ventas, introduciría el número de unidades vendidas por cada vendedor. El vendedor que hubiera vendido la mayor cantidad de unidades, ganaría el concurso. Escriba un programa primero en seudocódigo, y a continuación en C, que introduzca una serie de 10 números, y determine e imprima el mayor de los mismos. *Sugerencia:* su programa debería de utilizar tres variables como sigue:

`counter:` Un contador para contar hasta 10 (es decir, para controlar cuántos números han sido introducidos, y para determinar cuándo se han procesado todos los 10 números).
`number:` El número actual introducido al programa.
`largest:` El número más grande encontrado hasta ahora.

3.25 Escriba un programa en C, que utilice ciclos para imprimir la siguiente tabla de valores:

```
N       10*N       100*N      1000*N

1       10         100        1000
2       20         200        2000
3       30         300        3000
4       40         400        4000
5       50         500        5000
6       60         600        6000
7       70         700        7000
8       80         800        8000
9       90         900        9000
10      100        1000       10000
```

El carácter de tabulador, `\t`, puede ser utilizado en un enunciado `printf` para separar las columnas mediante tabuladores.

3.26 Escriba un programa en C que utilice ciclo para producir la siguiente tabla de valores:

```
A       A+2        A+4        A+6

3       5          7          9
6       8          10         12
9       11         13         15
12      14         16         18
15      17         19         21
```

3.27 Utilizando un método similar al del Ejercicio 3.26, encuentre los *dos* valores más grandes de los 10 números. Nota: Sólo puede introducir una vez cada número.

3.28 Modifique el programa de la figura 3.10 para validar sus entradas. En cualquier entrada, si el valor introducido es diferente que uno o dos, siga ciclando hasta que el usuario introduzca un valor correcto.

3.29 ¿Qué es lo que imprime el programa siguiente?

```
#include <stdio.h>

main()
{
   int count = 1;

   while (count <= 10) {
      printf("%s\n", count % 2 ? "****" : "++++++++");
      ++count;
   }

   return 0;
}
```

3.30 ¿Qué es lo que imprime el programa siguiente?

```
#include <stdio.h>

main()
{
   int row = 10, column;

   while (row >= 1) {
      column = 1;

      while (column <= 10) {
         printf("%s", row % 2 ? "<" : ">");
         ++column;
      }

      --row;
      printf("\n");
   }

   return 0;
}
```

3.31 *(Problema del else colgante)*. Determine la salida para cada uno de los siguientes, cuando **x** es **9** y **y** es **11**, y cuando **x** es **11** y **y** es **9**. Advierta que en un programa en C el compilador ignora la sangría. También, el compilador de C siempre asocia un **else** con el **if** anterior, a menos de que se le indique lo contrario mediante la colocación de las llaves **{}**. En vista de que a primera vista, el programador tal vez no estaría seguro de, cuál **if** coincide con qué **else**, esto se conoce como el problema del "else colgante". Hemos eliminado la sangría del código siguiente, para que el problema se haga más interesante. (*Sugerencia:* aplique las reglas de sangría que ha aprendido).

a)
```
if (x < 10)
if (y > 10)
printf("*****\n");
else
printf("#####\n");
printf("$$$$$\n");
```

b)
```
if (x < 10) {
if (y > 10)
printf("*****\n");
}
else {
printf("#####\n");
printf("$$$$$\n");
}
```

3.32 *(Otro problema de else colgante)*. Modifique el código que sigue para producir la salida mostrada. Utilice las técnicas apropiadas de sangrías. No puede hacer ningún cambio a excepción de inserción de llaves. En un programa C el compilador ignorará la sangría. Hemos eliminado las sangrías del código siguiente par hacer más atractivo el problema. Nota: pudiera ser posible que no se requiera de ninguna modificación.


```
if (y == 8)
if (x == 5)
printf("@@@@@\n");
else
printf("#####\n");
printf("$$$$$\n");
printf("&&&&&\n");
```

a) Suponiendo **x** = **5** y **y** = **8**, se produce la siguiente salida.

```
@@@@@
$$$$$
&&&&&
```

b) Suponiendo **x** = **5** y **y** = 8, se produce la siguiente salida.

```
@@@@@
```

c) Suponiendo **x** = **5** y **y** = **8**, se produce la siguiente salida.

```
@@@@@
&&&&&
```

d) Suponiendo **x** = **5** y **y** = **7**, se produce la siguiente salida. Nota: los tres enunciados **printf** últimos son todos parte de un enunciado compuesto.

```
#####
$$$$$
&&&&&
```

3.33 Escriba un programa que lea el lado de un cuadrado y a continuación lo imprima en forma de asteriscos. Su programa deberá poder funcionar para cuadrados de todos tamaños entre 1 y 20. Por ejemplo, si su programa lee un tamaño de 4, debería imprimir

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

3.34 Modifique el programa que escribió en el ejercicio 3.23, de tal forma que imprima un cuadrado hueco. Por ejemplo, si su programa lee un tamaño 5, deberá imprimir

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

3.35 Un palíndromo es un número o una frase de texto, que se lee igual hacia adelante y hacia atrás. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos son palíndromos: 12321, 55555, 45554 y 11611. Escriba un programa que lea un entero de cinco dígitos y que determine si es o no un palíndromo. (*Sugerencia:* utilice los operadores de división y de módulo para separar los números en sus dígitos individuales).

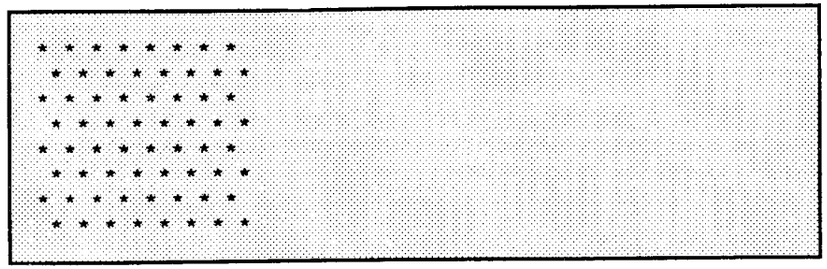
3.36 Introduzca un entero que contenga sólo 0s y 1s (es decir, un entero "binario") e imprima su equivalente decimal. (*Sugerencia:* utilice los operadores de módulo y de división para detectar los dígitos del número "binario" uno por uno, de derecha a izquierda. Al igual que en el sistema numérico decimal, donde el dígito más a la derecha tiene un valor posicional de 1, y el siguiente dígito a la izquierda tiene un valor posicional de 10, y a continuación de 100, y a continuación de 1000, etcétera, en un sistema numérico binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la derecha tiene un valor posicional de 2, y a continuación de 4, y a continuación de 8, etcétera. Por lo tanto, el número decimal 234 puede ser interpretado como 4 * 1 + 3 * 10 + 2 * 100. El equivalente decimal del número 1101 binario es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 o bien, 1 + 0 +4 + 8, es decir 13).

3.37 De forma continua escuchamos decir que las computadoras son muy rápidas. ¿Cómo puede usted determinar la verdadera velocidad de operación de su máquina? Escriba un programa con un ciclo **while** que cuente de 1 hasta 3,000,000 de uno en uno. Cada vez que el contador llegue a un múltiplo de 1,000,000, imprima este número en la pantalla. Utilice su reloj para medir cuánto tiempo tarda cada repetición de 1,000,000 del ciclo.

3.38 Escriba un programa que imprima 100 asteriscos, uno por uno. Después de cada décimo asterisco, su programa deberá imprimir un carácter de nueva línea. (*Sugerencia:* cuente de 1 a 100. Utilice un operador de módulo para reconocer cada vez que el contador llegue a un múltiplo de 10.)

3.39 Escriba un programa que lea un entero, y determine e imprima cuántos dígitos de ese entero son 7s.

3.40 Escriba un programa que despliegue el siguiente patrón.



Su programa sólo puede usar tres enunciados `printf`, y uno de la forma

```
printf("* ");
```

otro de la forma

```
printf(" ");
```

y uno de la forma

```
printf("\n");
```

3.41 Escriba un programa que continúe imprimiendo múltiplos del entero 2, es decir, 2, 4, 8, 16, 32, 64, etcétera. Su ciclo no deberá de terminar (es decir, usted debe de crear un ciclo infinito). ¿Qué ocurre cuando ejecuta este programa?

3.42 Escriba un programa que lea el radio de un círculo (como valor `float`) y que calcule e imprima el diámetro, la circunferencia y el área. Utilice para el valor de 3.14159, π.

3.43 ¿Qué es lo que no está bien en el enunciado siguiente?. Vuelva a escribir el enunciado, para que ejecute lo que probablemente el programador intentaba hacer.

```
printf ("%d", ++(x + y));
```

3.44 Escriba un programa que lea tres valores `float` no cero, y que determine e imprima si pueden representar los lados de un triángulo.

3.45 Escriba un programa que lea tres enteros no ceros y que determine e imprima si pueden ser los lados de un triángulo rectángulo.

3.46 Una empresa desea transmitir datos mediante el teléfono, pero están preocupados de que sus teléfonos pudieran estar intervenidos. Todos sus datos se transmiten como enteros de cuatro dígitos. Le han solicitado a usted que escriba un programa que cifre sus datos, de tal forma de que puedan ser transmitidos con mayor segururidad. Su programa debe leer un entero de cuatro dígitos y cifrarlo como sigue: remplazar cada dígito por *(la suma del dígito más 7) módulo 10.* A continuación, intercambiar el primer dígito con el tercero, y el segundo con el cuarto. A continuación imprimir el entero cifrado. Escriba un programa por separado, que introduzca un entero de cuatro dígitos cifrado, y que lo descifre para formar el número original.

3.47 El factorial de un entero no negativo n se escribe como $n!$ (se dice "factorial de n") y se define como sigue:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ (para valores de n mayores que o igual a 1)

y

$n! = 1$ (para $n = 0$).

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, lo que significa 120.

a) Escriba un programa que lea un entero no negativo, y que calcule e imprima su factorial.

b) Escriba un programa que estime el valor de la constante matemática e, utilizando la fórmula

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Escriba un programa que calcule el valor de e^x, utilizando la fórmula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$