CAPÍTULO 3

Asignación, formateo y entrada interactiva

TEMAS

3.1 OPERACIONES DE ASIGNACIÓN
COERCIÓN
VARIACIONES DE ASIGNACIÓN
ACUMULACIÓN
CONTEO

3.2 DAR FORMATO A NÚMEROS PARA LA SALIDA DEL PROGRAMA

3.3 EMPLEO DE LA BIBLIOTECA DE FUNCIONES MATEMÁTICAS
MOLDES

3.4 ENTRADA DE DATOS AL PROGRAMA USANDO EL OBJETO `cin`
UNA PRIMERA MIRADA A LA VALIDACIÓN DE ENTRADAS DEL USUARIO

3.5 CONSTANTES SIMBÓLICAS
COLOCACIÓN DE INSTRUCCIONES

3.6 APLICACIONES
APLICACIÓN 1: LLUVIA ÁCIDA
APLICACIÓN 2: APROXIMACIÓN A LA FUNCIÓN EXPONENCIAL

3.7 ERRORES COMUNES DE PROGRAMACIÓN

3.8 RESUMEN DEL CAPÍTULO

3.9 UN ACERCAMIENTO MÁS A FONDO: ERRORES DE PROGRAMACIÓN
CONSIDERACIÓN DE LAS OPCIONES DE CARRERA: INGENIERÍA MECÁNICA

En el capítulo anterior se exploró cómo se despliegan los resultados usando el objeto `cout` *de C++ y cómo se almacenan y se procesan los datos numéricos usando variables e instrucciones de asignación. En este capítulo se completará la introducción a C++ presentando capacidades de procesamiento y entrada adicionales.*

3.1 OPERACIONES DE ASIGNACIÓN

Ya se han encontrado instrucciones de asignación simples en el capítulo 2. Dichas instrucciones son las más básicas de C++ tanto para asignar valores a las variables como para llevar a cabo cálculos. Esta instrucción tiene la sintaxis:

variable = expresión;

La expresión más simple en C++ es una sola constante. En cada una de las siguientes instrucciones de asignación, el operando a la derecha del signo de igual es una constante:

```
largo = 25;
ancho = 17.5;
```

En cada una de estas instrucciones de asignación el valor de la constante a la derecha del signo de igual se asigna a la variable a la izquierda del signo de igual. Es importante señalar que el signo de igual en C++ no tiene el mismo significado que un signo de igual en álgebra. El signo de igual en una instrucción de asignación le indica a la computadora que determine primero el valor del operando a la derecha del signo de igual y luego almacene (o asigne) ese valor en las ubicaciones asociadas con la variable a la izquierda del signo de igual. En este sentido, la instrucción de C++ `largo = 25`; se lee "a largo se le asignó el valor 25". Los espacios en blanco en la instrucción de asignación se insertan sólo para legibilidad.

Recuerde que una variable puede ser inicializada cuando se declara. Si no se hace una inicialización dentro de la instrucción de declaración, a la variable debe asignársele un valor con una instrucción de asignación u operación de entrada antes que se use en cualquier cálculo. Pueden usarse, por supuesto, instrucciones de asignación subsiguientes para cambiar el valor asignado a una variable. Por ejemplo, suponga que las siguientes instrucciones se ejecutan una tras otra y que no se inicializó pendiente cuando fue declarada:

```
pendiente = 3.7;
pendiente = 6.28;
```

La primera instrucción de asignación le da el valor de 3.7 a la variable nombrada `pendiente`.[1] La siguiente instrucción de asignación causa que la computadora asigne un valor de 6.28 a `pendiente`. El 3.7 que estaba en `pendiente` es sobrescrito con el nuevo valor de 6.28 debido a que una variable sólo puede almacenar un valor a la vez. A veces es útil pensar en la variable a la izquierda del signo de igual como un cajón de estacionamiento temporal en un estacionamiento enorme. Del mismo modo en que un cajón de estacionamiento individual sólo puede ser usado por un automóvil a la vez, cada variable sólo puede almacenar un valor a la vez. "Estacionar" un valor nuevo en una variable causa de manera automática que el programa elimine cualquier valor estacionado ahí con anterioridad.

[1]En vista que ésta es la primera vez que se asigna un valor de manera explícita a esta variable con frecuencia se le denomina inicialización. Esto se deriva del uso histórico que expresa que una variable era inicializada la primera vez que se le asignaba un valor. Bajo este uso es correcto decir qué "pendiente fue inicializada a 3.7". Desde un punto de vista de la implementación, sin embargo, este último planteamiento es incorrecto. Esto se debe a que la operación de asignación es manejada en forma diferente por el compilador de C++ que una inicialización realizada cuando se crea una variable por una instrucción de declaración. Esta diferencia sólo es importante cuando se usan características de clase de C++ y se explica con detalle en la sección 9.1.

Además de ser una constante, el operando a la derecha del signo de igual en una instrucción de asignación puede ser una variable o cualquier otra expresión válida de C++. Una **expresión** es cualquier combinación de constantes, variables y llamadas a funciones que pueden evaluarse para producir un resultado. Por tanto, la expresión en una instrucción de asignación puede usarse para realizar cálculos usando los operadores aritméticos introducidos en la sección 2.4. Son ejemplos de instrucciones de asignación que usan expresiones que contienen estos operadores

```
suma = 3 + 7;
dif = 15 - 6;
producto = .05 * 14.6;
conteo = contador + 1;
totalnuevo = 18.3 + total;
impuestos = .06 * cantidad;
pesoTotal = factor * peso;
promedio = suma / elementos;
pendiente = (y2 - y1) / (x2 - x1);
```

Como siempre en una instrucción de asignación, el programa calcula primero el valor de la expresión a la derecha del signo de igual y luego almacena este valor en la variable a la izquierda del signo de igual. Por ejemplo, en la instrucción de asignación `pesoTotal = factor * peso;` la expresión `factor * peso` se evalúa primero para producir un resultado. Este resultado, el cual es un número, se almacena luego en la variable `pesoTotal`.

Al escribir expresiones de asignación, debe tener en cuenta dos consideraciones importantes. En vista que la expresión a la derecha del signo de igual se evalúa primero, a todas las variables usadas en la expresión debe habérseles dado valores válidos para que el resultado tenga sentido. Por ejemplo, la instrucción de asignación `pesoTotal = factor * peso;` causa que un número válido sea almacenado en `pesoTotal` sólo si el programador tiene cuidado primero de asignar números válidos a `factor` y `peso`. Por tanto la secuencia de instrucciones

```
factor = 1.06;
peso = 155.0;
pesoTotal = factor * peso;
```

nos indica los valores que se están usando para obtener el resultado que se almacenará en `pesoTotal`. La figura 3.1 ilustra los valores almacenados en las variables `factor`, `peso` y `pesoTotal`.

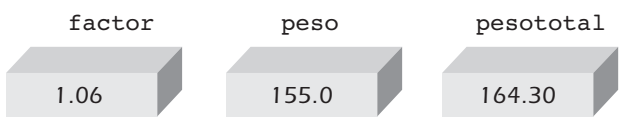


Figura 3.1 *Valores almacenados en las variables.*

La segunda consideración a tener en cuenta es que en vista que el valor de una expresión es almacenado en la variable a la izquierda del signo de igual, sólo una variable puede escribirse en esta posición. Por ejemplo, la instrucción de asignación

```
cantidad + 1892 = 1000 + 10 * 5;
```

es inválida. La expresión del lado derecho da por resultado el número entero 1050, el cual sólo puede ser almacenado en una variable. Debido a que `cantidad + 1892` no es un nombre de variable válido, el compilador no sabe dónde almacenar el valor calculado.

El programa 3.1 ilustra el uso de instrucciones de asignación para calcular el volumen de un cilindro. Como se ilustra en la figura 3.2, el volumen de un cilindro está determinado por la fórmula *volumen en* = $\pi r^2 h$, donde r es el radio del cilindro, h es la altura y π es la constante 3.1416 (con una precisión de cuatro cifras decimales).

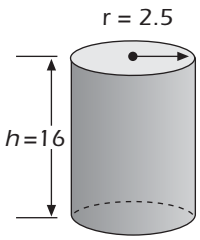


Figura 3.2 *Determinar el volumen de un cilindro.*

Programa 3.1

```
//   este programa calcula el volumen de un cilindro,
//      dados su radio y altura
#include <iostream>
using namespace std;

int main()
{
  double radio, altura, volumen;

  radio = 2.5;
  altura = 16.0;
  volumen = 3.1416 * radio * radio * altura;
  cout << "El volumen del cilindro es " << volumen << endl;

  return 0;
}
```

Cuando el programa 3.1 es compilado y ejecutado, la salida es

```
El volumen del cilindro es 314.16
```

Considere el flujo de control que usa la computadora para ejecutar el programa 3.1. La ejecución del programa comienza con la primera instrucción dentro del cuerpo de la función `main()` y continúa en forma secuencial, instrucción por instrucción, hasta que se encuentre

la llave de cierre de main. Este flujo de control se aplica a todos los programas. La computadora funciona con una instrucción a la vez, ejecutando esa instrucción sin saber cuál será la siguiente instrucción. Esto explica por qué todos los operandos usados en una expresión deben tener valores asignados a ellos antes que se evalúe la expresión. Cuando la computadora ejecuta la instrucción

```
volumen = 3.1416 * radio * radio * altura;
```

en el programa 3.1, usa cualquier valor que esté almacenado en las variables `radio` y `altura` en el momento en que se ejecuta la instrucción de asignación.[2] Si no se han asignado valores de manera específica a estas variables antes que se usen en la instrucción de asignación, la computadora usa los valores que tengan estas variables cuando se haga referencia a ellas. (En algunos sistemas todas las variables son inicializadas de manera automática en cero.) La computadora no "ve hacia delante" para verificar si se asignan valores a estas variables más adelante en el programa.

Es importante percatarse que en C++ el signo de igual, =, usado en instrucciones de asignación es en sí un operador, el cual difiere de la forma en que la mayor parte de otros lenguajes de alto nivel procesa este símbolo. En C++ (como en C), el símbolo = se llama **operador de asignación**, y una expresión que usa este operador, como `interés = principal * tasa`, es una **expresión de asignación**. En vista que el operador de asignación tiene una precedencia menor que cualquier otro operador aritmético, el valor de cualquier expresión a la derecha del signo de igual será evaluado primero, antes de la asignación.

Como todas las expresiones, las expresiones de asignación en sí mismas tienen un valor. El valor de la expresión de asignación completa es el valor asignado a la variable en el lado izquierdo del operador de asignación. Por ejemplo, la expresión `a = 5` asigna un valor de *5* a la variable a y produce que la expresión en sí tenga un valor de *5*. El valor de la expresión siempre puede verificarse usando una instrucción como

```
cout << "El valor de la expresión es " << (a = 5);
```

Aquí, el valor de la expresión en sí es desplegado y no el contenido de la variable a. Aunque tanto el contenido de la variable como la expresión tienen el mismo valor, vale la pena observar que se está tratando con dos entidades distintas.

Desde una perspectiva de programación, es la asignación real de un valor a una variable la que es significativa en una expresión de asignación; el valor final de la expresión de asignación en sí es de poca consecuencia. Sin embargo, el hecho que las expresiones de asignación tengan un valor tiene implicaciones que deben considerarse cuando se presenten los operadores relacionales en C++.

Cualquier expresión que se termine con un punto y coma se convierte en una instrucción de C++. El ejemplo más común de esto es la instrucción de asignación, la cual tan sólo es una expresión de asignación terminada con un punto y coma. Por ejemplo, terminar la expresión de asignación `a = 33` con un punto y coma produce la instrucción de asignación `a = 33;` la cual puede usarse en un programa en una sola línea.

En vista que el signo de igual es un operador en C++, son posibles en la misma expresión asignaciones múltiples o una instrucción equivalente. Por ejemplo, en la expresión `a = b = c = 25` todos los operadores de asignación tienen la misma precedencia. En vista que el ope-

[2]En vista que C++ no tiene un operador de exponenciación, el cuadrado del radio se obtiene por el término radio * radio. En la sección 3.3 se introduce la función de potencia `pow()` de C++, la cual permite elevar un número a una potencia.

rador de asignación tiene una asociatividad de derecha a izquierda, la evaluación final procede en la secuencia

```
c = 25
b = c
a = b
```

En este caso, esto tiene el efecto de asignar el número 25 a cada una de las variables en forma individual y puede representarse como

```
a = (b = (c = 25))
```

Añadir un punto y coma a la expresión original produce la instrucción de asignación múltiple

```
a = b = c = 25;
```

Esta última instrucción asigna el valor de 25 a las tres variables individuales equivalentes al siguiente orden:

```
c = 25;
b = 25;
a = 25;
```

Coerción

Algo que se debe tener en cuenta cuando se trabaje con instrucciones de asignación es el tipo de datos asignado a los valores en ambos lados de la expresión, porque ocurren conversiones de tipos de datos a lo largo de los operadores de asignación. En otras palabras, el valor de la expresión en el lado derecho del operador de asignación será convertido en el tipo de datos de la variable a la izquierda del operador de asignación. Este tipo de conversión se conoce como **coerción** porque el valor asignado a la variable en el lado izquierdo del operador de asignación es forzado al tipo de datos de la variable a la que es asignado. Un ejemplo de una coerción ocurre cuando se asigna un valor de número entero a una variable de número real; esto causa que el entero se convierta en un valor real. Por tanto, asignar un valor entero a una variable real causa que el entero sea convertido en un valor real. Del mismo modo, asignar un valor real a una variable entera fuerza la conversión de un valor real a un entero, lo cual produce la pérdida de la parte fraccionaria del número debido a truncamiento. Por ejemplo, si `temp` es una variable entera, la asignación `temp = 25.89` causa que el valor entero 25 se almacene en la variable entera `temp`.[3]

Un ejemplo más completo de conversiones de tipos de datos, las cuales incluyen conversión en modo mixto y conversión de asignación, es la evaluación de la expresión

```
a = b * d
```

[3]Es evidente que la porción entera correcta sólo se conserva cuando está dentro del rango de enteros permitidos por el compilador.

Punto de Información

lvalues y rvalues

Los términos `lvalue` y `rvalue` se utilizan con frecuencia en la tecnología de programación. Ambos términos son independientes del lenguaje y significan lo siguiente: un **`lvalue`** puede tener un valor asignado mientras que un **`rvalue`** no puede tenerlo.

Tanto en C como en C++ esto significa que un `lvalue` puede aparecer tanto en el lado izquierdo como en el derecho de un operador de asignación mientras que un `rvalue` sólo puede aparecer en el lado derecho de un operador de asignación. Por ejemplo, cada variable que hemos encontrado puede ser un `lvalue` o un `rvalue`, mientras que un número sólo puede ser un `rvalue`. Sin embargo, no todas las variables pueden ser `lvalue` y `rvalue`. Por ejemplo, un tipo de arreglo, el cual se introduce en el capítulo 11, no puede ser un `lvalue` o un `rvalue`, mientras que los elementos individuales del arreglo pueden ser ambos.

donde `a` y `b` son variables enteras y `d` es una variable de precisión simple. Cuando se evalúa la expresión en modo mixto `b * d`,[4] el valor de `d` usado en la expresión es convertido a un número de precisión doble para propósitos de cálculo. (Es importante señalar que el valor almacenado en `d` se mantiene como un número de precisión simple.) En vista que uno de los operandos es una variable de precisión doble, el valor de la variable entera `b` es convertido en un número de precisión doble para el cálculo (una vez más, el valor almacenado en `b` sigue siendo un entero) y el valor resultante de la expresión `b * d` es un número de precisión doble. Por último, se aplica la conversión del tipo de datos por medio del operador de asignación. En vista que el lado izquierdo del operador de asignación es una variable entera, el valor de precisión doble de la expresión (`b * d`) será truncado a un valor entero y almacenado en la variable `a`.

Variaciones de asignación

Auque sólo se permite una variable inmediatamente a la izquierda del signo de igual en una expresión de asignación, la variable a la izquierda del signo de igual también puede usarse a la derecha del signo de igual. Por ejemplo, la expresión de asignación `suma = suma + 10` es válida. Es claro que, como una expresión algebraica, `suma` nunca podría ser igual a sí misma más 10. Pero en C++, la expresión `suma = suma + 10` no es una ecuación, es una expresión que se evalúa en dos pasos importantes. El primer paso es calcular el valor de `suma + 10`. El segundo paso es almacenar el valor calculado en `suma`. Vea si puede determinar la salida del programa 3.2.

[4]Si es necesario, repase las reglas en la sección 2.4 para la evaluación de expresiones de modo mixto.

Programa 3.2

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

int main()
{
  int suma;

  suma = 25;
  cout << "El número almacenado en suma es " << suma << endl;
  suma = suma + 10;
  cout << "El número almacenado ahora en suma es " << suma << endl;

  return 0;
}
```

La instrucción de asignación `suma = 25;` le indica a la computadora que almacene el número 25 en suma, como se muestra en la figura 3.3.

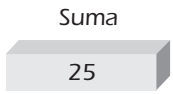


Figura 3.3 *El entero 25 es almacenado en* `suma`.

La primera instrucción `cout` causa que el valor almacenado en suma sea desplegado por el mensaje `El número almacenado en suma es 25`. La segunda instrucción de asignación en el programa 3.2, `suma = suma + 10;` causa que el programa recupere el 25 almacenado en `suma` y agregue 10 a este número, produciendo el número 35. Entonces el número 35 se almacena en la variable en el lado izquierdo del signo de igual, la cual es la variable `suma`. El 25 que estaba en suma tan sólo se sobrescribe con el nuevo valor de 35, como se muestra en la figura 3.4.

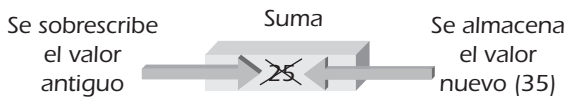


Figura 3.4 `suma = suma + 10;` *causa que se almacene un nuevo valor en* `suma`.

Las expresiones de asignación como `suma = suma + 25`, que usan la misma variable en ambos lados del operador de asignación, pueden escribirse usando los siguientes atajos de **operadores de asignación**:

```
+=   -=   *=   /=   %=
```

Por ejemplo, la expresión `suma = suma + 10` puede escribirse como `suma += 10`. Del mismo modo, la expresión `precio *= tasa` es equivalente a la expresión `precio = precio * tasa`.

Al utilizar estos operadores de asignación es importante observar que la variable de la izquierda del operador de asignación se aplica a la expresión de la derecha completa. Por ejemplo `precio * = tasa + 1` es equivalente a la expresión `precio = precio * (tasa + 1)`, no `precio = precio * tasa + 1.`

Acumulación

Las expresiones de asignación como `suma += 10` o su equivalente, `suma = suma + 10`, son muy comunes en programación. Estas expresiones se requieren para acumular subtotales cuando los datos se introducen un número a la vez. Por ejemplo, si se desea sumar los números 96, 70, 85 y 60 en forma de calculadora, podrían usarse las siguientes instrucciones:

Instrucción	Valor en `suma`
`suma = 0;`	0
`suma = suma + 96;`	96
`suma = suma + 70;`	166
`suma = suma + 85;`	251
`suma = suma + 60;`	311

La primera instrucción inicializa `suma en 0`. Esto elimina cualquier número ("valor inservible") almacenado en `suma` que pudiera invalidar el total final. Conforme se agrega cada número, el valor almacenado en `suma` se incrementa en forma correspondiente. Después de completar la última instrucción, `suma` contiene el total de todos los números agregados. El programa 3.3 ilustra el efecto de estas instrucciones al desplegar el contenido de `suma` después de hacer cada adición.

Programa 3.3

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

int main()
{
  int suma;

  suma = 0;
  cout << "El valor de suma se estableció en forma inicial en " << suma << endl;
  suma = suma + 96;
  cout << "  suma ahora es " << suma << endl;
  suma = suma + 70;
  cout << "  suma ahora es " << suma << endl;
  suma = suma + 85;
  cout << "  suma ahora es " << suma  << endl;
  suma = suma + 60;
  cout << "  La suma final es " << suma << endl;

  return 0;
}
```

La salida desplegada por el programa 3.3 es:

```
El valor de suma se estableció en forma inicial en 0
  suma ahora es 96
  suma ahora es 166
  suma ahora es 251
  La suma final es 311
```

Aunque el programa 3.3 no es un programa práctico (es más fácil sumar los números en forma manual), ilustra el efecto subtotalizador del uso repetido de instrucciones que tienen la forma

variable = variable + Valornuevo;

Se encontrarán muchos usos para este tipo de **instrucción de acumulación** cuando nos familiaricemos más con las instrucciones de repetición introducidas en el capítulo 5.

Conteo

Una instrucción de asignación que es muy similar a la instrucción de acumulación es la instrucción de conteo. Las instrucciones de conteo tienen la forma

variable = variable + númeroFijo;

Son ejemplos de instrucciones de conteo

```
i = i + 1;
n = n + 1;
contador = contador + 1;
j = j + 2;
m = m + 2;
kk = kk + 3;
```

En cada uno de estos ejemplos se usa la misma variable en ambos lados del signo de igual. Después de ejecutar la instrucción, el valor de la variable respectiva se incrementa en una cantidad fija. En los primeros tres ejemplos las variables `i, n` y `contador` fueron incrementadas en uno. En los siguientes dos ejemplos las variables respectivas se han incrementado en dos y en el ejemplo final la variable `kk` se ha incrementado en tres.

Para el caso especial en que una variable es incrementada o disminuida en uno, C++ proporciona dos operadores unitario. Usando el operador de incremento,[5] `++`, la expresión `variable = variable + 1` puede ser reemplazada por la expresión `variable++` o `++variable`. Son ejemplos del operador de incremento

Expresión	Alternativa
`i = i + 1`	`i++ o ++i`
`n = n + 1`	`n++ o ++n`
`contador = contador + 1`	`contador++ o ++contador`

El programa 3.4 ilustra el uso del operador de incremento.

[5]Como una nota histórica, el ++ en C++ se inspiró en el símbolo del operador de incremento. Se usó para indicar que C++ fue el siguiente incremento en el lenguaje C.

Programa 3.4

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

int main()
{
  int contador;

  contador = 0;
  cout << "El valor inicial de contador es " << contador << endl;
  contador++;
  cout << "   contador es ahora " << contador << endl;
  contador++;
  cout << "   contador es ahora " << contador << endl;
  contador++;
  cout << "   contador es ahora " << contador << endl;
  contador++;
  cout << "   contador es ahora " << contador << endl;

  return 0;
}
```

La salida desplegada por el programa 3.4 es:

```
El valor inicial de contador es 0
   contador es ahora 1
   contador es ahora 2
   contador es ahora 3
   contador es ahora 4
```

Cuando el operador `++` aparece antes de una variable se llama **operador de prefijo para incremento**; cuando aparece después de una variable se llama **operador de postfijo para incremento**. La distinción entre un operador de prefijo de incremento y uno de postfijo es importante cuando la variable que es incrementada se usa en una expresión de asignación. Por ejemplo, la expresión `k = ++n` hace dos cosas en una expresión. Al principio, el valor de `n` es incrementado en uno y luego el valor nuevo de `n` es asignado a la variable `k`. Por tanto, la instrucción `k = ++n;` es equivalente a las dos instrucciones

```
n = n + 1;   // incrementa n primero
k = n;       // asigna el valor de n a
```

La expresión de asignación `k = n++`, la cual usa un operador de sufijo para incremento, invierte este procedimiento. Un postfijo para incremento opera después que se ha comple-

tado la asignación. Por tanto, la instrucción `k = n++;` asigna primero el valor actual de `n` a `k` y luego incrementa el valor de `n` en uno. Esto es equivalente a las dos instrucciones

```
k = n;          // asigna el valor de n a k
n = n + 1;      // y luego incrementa n
```

Además del operador de incremento, C++ también proporciona un operador de decremento, `--`. Como podría esperarse, las expresiones `variable--` y `--variable` son equivalentes a la expresión `variable = variable - 1`.

Son ejemplos del operador de decremento:

Expresión	Alternativa
`i = i - 1`	`i-- o --i`
`n = n - 1`	`n-- o --n`
`contador = contador - 1`	`contador-- o --contador`

Cuando el operador `--` aparece antes de una variable se llama **operador de prefijo para decremento**. Cuando el decremento aparece después de una variable se llama **operador de postfijo para decremento**. Por ejemplo, las expresiones `n--` y `--n` reducen el valor de `n` en uno. Estas expresiones son equivalentes a la expresión más larga `n = n - 1`. Sin embargo, como con los operadores de incremento, los operadores de prefijo y postfijo para decremento producen resultados diferentes cuando se usan en expresiones de asignación. Por ejemplo, la expresión `k = --n` disminuye primero el valor de `n` en uno antes de asignar el valor de `n` a `k`, mientras la expresión `k = n--` signa primero el valor actual de `n` a `k` y luego reduce el valor de `n` en uno.

Ejercicios 3.1

1. Escriba una instrucción de asignación para calcular la circunferencia de un círculo que tiene un radio de 3.3 pulgadas. La ecuación para determinar la circunferencia, *c,* de un círculo es $c = 2\pi r$, donde r es el radio y π es igual a 3.1416.
2. Escriba una instrucción de asignación para calcular el área de un círculo. La ecuación para determinar el área, *a*, de un círculo es $a = \pi r^2$, donde r es el radio y $\pi = 3.1416$.
3. Escriba una instrucción de asignación para convertir temperatura en grados Fahrenheit a grados Celsius. La ecuación para esta conversión es Celsius = 5/9 (Fahrenheit – 32).
4. Escriba una instrucción de asignación para calcular la distancia de un viaje redondo, *d*, en pies, de un viaje de *s* millas en un solo sentido.
5. Escriba una instrucción de asignación para calcular el tiempo transcurrido, en minutos, necesario para hacer un viaje. La ecuación para calcular el tiempo transcurrido es *tiempo = distancia total / velocidad promedio*. Suponga que la distancia debe ser en millas y la velocidad promedio en millas/hora.

6. Escriba una instrucción de asignación para calcular el *enésimo* término en una secuencia aritmética. La fórmula para calcular el valor, v, del *enésimo* término es $v = a + (n - 1)d$, donde a = el primer número en la secuencia y d = la diferencia entre cualesquiera dos números en la secuencia.

7. Escriba una instrucción de asignación para calcular la expansión lineal en una viga de acero como una función del aumento de temperatura. La fórmula para la expansión lineal, l, es $l = l_o[1+\propto(T_f - T_o)]$, donde l_o es el largo de la viga a la temperatura T_o, $\propto$ es el coeficiente de expansión lineal y T_f es la temperatura final de la viga.

8. La ley de Coulomb establece que la fuerza F, actuando entre dos esferas cargadas eléctricamente, está dada por la fórmula $F = kq_1q_2/r^2$, donde q_1 es la carga en la primera esfera, q_2 es la carga en la segunda esfera, r es la distancia entre los centros de las dos esferas y k es una constante de proporcionalidad. Escriba una instrucción de asignación para calcular la fuerza, F.

9. Escriba una instrucción de asignación para determinar el momento de flexión máximo, M, de una viga. La fórmula para el momento de flexión máximo es, $M = XW(L - X) / L$, donde X es la distancia desde el extremo de la viga donde se aplica un peso, W, y L es el largo de la viga.

10. Determine la salida del siguiente programa:

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

int main() // un programa que ilustra el truncamiento de enteros
{
   int num1, num2;

   num1 = 9/2;
   num2 = 17/4;
   cout << "el primer entero desplegado es " << num1 << endl;
   cout << "el segundo entero desplegado es " << num2 << endl;

  return 0;
}
```

11. Determine y corrija los errores en los siguientes programas.

    a.
```
#include <iostream>
  using namespace std;
  int main()
  {
    ancho = 15
    area = largo * ancho;
    cout << "El area es " << area

  }
```

b.
```
#include <iostream>
using namespace std;
int main()
{
  int largo, ancho, area;
  area = largo * ancho;
  largo = 20;
  ancho = 15;
  cout << "El area es " << area;

  return 0;
```

c.
```
#include <iostream.h>

 int main()
 {
   int largo = 20; ancho = 15, area;
   largo * ancho = area;
   cout << "El area es " , area;

   return 0;
 }
```

12. Por error un estudiante reordenó las instrucciones en el programa 3.3 como sigue:

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

int main()
{
  int suma;
  suma = 0;
  suma = suma + 96;
  suma = suma + 70;
  suma = suma + 85;
  suma = suma + 60;
  cout << "El valor de suma se estableció en forma inicial en "
          << suma << endl;
  cout << "  suma ahora es " << suma << endl;
  cout << "  suma ahora es " << suma << endl;
  cout << "  suma ahora es " << suma << endl;
  cout << "  La suma final es " << suma << endl;

  return 0;
}
```

Determine la salida que produce este programa.

13. Usando el programa 3.1, determine el volumen de cilindros que tienen los siguientes radios y alturas.

Radio (pulg.)	Altura (pulg.)
1.62	6.23
2.86	7.52
4.26	8.95
8.52	10.86
12.29	15.35

14. El área de una elipse (véase la figura 3.5) está dada por la fórmula Área = πab.

Usando esta fórmula, escriba un programa C++ para calcular el área de una elipse que tenga un eje menor, *a*, de 2.5 pulgadas y un eje mayor, *b*, de 6.4 pulgadas.

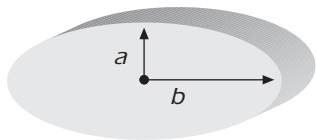


Figure 3.5 El eje menor *a* y el eje mayor *b* de una elipse.

15. Modifique el programa 3.1 para calcular el peso, en libras, del cilindro de acero cuyo volumen fue encontrado por el programa. Para determinar el peso la fórmula es peso = 0.28 $(\pi)(r^2)(h)$, donde *r* es el radio (en pulgadas) y *h* es la altura (en pulgadas) del cilindro.

16. La circunferencia de una elipse (véase la figura 3.5) está dada por la fórmula:

$$\text{Circunferencia} = \pi\sqrt{(a+b)^2}$$

Usando esta fórmula, escriba un programa en C++ para calcular la circunferencia de una elipse que tiene un radio menor de 2.5 pulgadas y un radio mayor de 6.4 pulgadas. (*Sugerencia:* la raíz cuadrada puede obtenerse elevando la cantidad $2[a^2 + b^2]$ a la potencia 0.5.)

17. a. La resistencia combinada de tres resistores conectados en paralelo, como se muestra en la figura 3.6, está dada por la ecuación

$$\text{Resistencia combinada} = \frac{1}{\frac{1}{R_1}+\frac{1}{R_2}+\frac{1}{R_3}}$$

Escriba un programa en C++ para calcular y desplegar la resistencia combinada cuando los tres resistores $R_1 = 1000$, $R_2 = 1000$, y $R_3 = 1000$ están conectados en paralelo. Su programa deberá producir el despliegue "La resistencia combinada, en ohmios, es xxxxx", donde las x son reemplazadas por el valor de la resistencia combinada calculada por su programa.

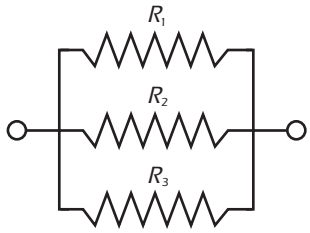


Figura 3.6 Tres resistores conectados en paralelo.

b. ¿Cómo sabe que el valor calculado por su programa es correcto?
c. Una vez que ha verificado la salida producida por su programa, modifíquela para determinar la resistencia combinada cuando los resistores $R_1 = 1500$, $R_2 = 1200$, y $R_3 = 2000$ están conectados en paralelo.

18. a. Escriba un programa en C++ para calcular y desplegar el valor de la pendiente de la línea que conecta dos puntos cuyas coordenadas son (3, 7) y (8, 12). Use el hecho que la pendiente entre dos puntos que tienen coordenadas (x_1, y_1) y (x_2, y_2) es pendiente = $(y_2 - y_1) / (x_2 - x_1)$. Su programa deberá producir el despliegue "La pendiente es xxxx", donde las x son reemplazadas por el valor calculado por su programa.
b. ¿Cómo sabe que el resultado producido por su programa es correcto?
c. Una vez que ha verificado la salida producida por su programa, modifíquela para determinar la pendiente de la línea que conecta los puntos (2, 10) y (12, 6).

19. a. Escriba un programa en C++ para calcular y desplegar las coordenadas del punto medio de la línea que conecta los dos puntos dados en el ejercicio 18a. Use el hecho que las coordenadas del punto medio entre dos puntos que tienen coordenadas (x_1,y_1) y (x_2,y_2) son $[(x_1 + x_2)]/2$, $[(y_1 + y_2)]/2)$. Su programa deberá producir el siguiente despliegue:

```
La coordenada x del punto medio es xxx
La coordenada y del punto medio es xxx
```

donde las x son reemplazadas con los valores calculados por su programa.
b. ¿Cómo sabe que los valores del punto medio calculados por su programa son correctos?
c. Una vez que ha verificado la salida producida por su programa, modifíquelo para determinar las coordenadas del punto medio de la línea que conecta los puntos (2, 10) y (12, 6).

20. a. Para el circuito eléctrico mostrado en la figura 3.7, las corrientes en los ramales, i_1, i_2, e i_3 pueden determinarse usando las fórmulas

$$i_1 = \frac{E_2R_3 + E_1(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$i_2 = \frac{E_1R_3 + E_2(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$i_3 = i_1 - i_2$$

Usando estas fórmulas escriba un programa en C++ para calcular las corrientes en los ramales cuando R_1 = 10 ohmios, R_2 = 4 ohmios, R_3 = 6 ohmios, E_1 = 12 voltios y E_2 = 9 voltios. El despliegue producido por su programa deberá ser

```
La corriente en el ramal 1 es xxxx
La corriente en el ramal 2 es xxxx
La corriente en el ramal 3 es xxxx
```

donde las x son reemplazadas por los valores determinados en su programa.

b. ¿Cómo sabe que las corrientes en circuito calculadas por su programa son correctas?

c. Una vez que ha verificado la salida producida por su programa, modifíquela para determinar las corrientes en los ramales para los siguientes valores: R_1 = 1500, R_2 = 1200, R_3 = 2000, E_1 = 15 y E_2 = 12.

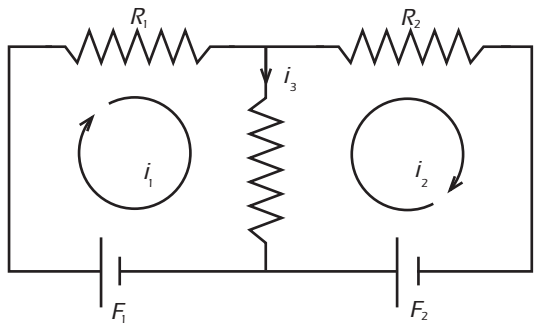


Figura 3.7 Un circuito eléctrico.

3.2 DAR FORMATO A NÚMEROS PARA LA SALIDA DEL PROGRAMA

Además de desplegar resultados correctos, es importante en extremo que un programa presente sus resultados en forma atractiva. La mayor parte de los programas son juzgados por la facilidad de introducción de datos percibida y el estilo y presentación de su salida. Por ejemplo, desplegar un resultado monetario como 1.897 no cumple con las convenciones aceptadas para los informes. El despliegue deberá ser $1.90 o $1.89, dependiendo si se usa redondeo o truncamiento.

El formato de los números desplegados por `cout` puede controlarse por manipuladores de ancho de campo incluidos en cada flujo de salida. La tabla 3.1 enumera los manipuladores disponibles que más se usan para este propósito.[6]

[6]Como se señaló en el capítulo 2, el manipulador `endl` inserta una `nueva línea` y luego vacía el flujo.

Tabla 3.1 Manipuladores de flujo más comunes

Manipulador	Acción
`setw(n)`	Establece el ancho de campo en *n*.
`setprecision(n)`	Establece la precisión del punto flotante en *n* lugares. Si se designa el manipulador `fixed`, *n* especifica el número total de dígitos desplegados después del punto decimal; de otra manera, *n* especifica el número total de dígitos significativos desplegados (números enteros más dígitos fraccionarios).
`setfill('x')`	Establece el carácter de relleno a la izquierda por omisión en *x*. (El carácter de relleno principal por omisión es un espacio, el cual es la salida para rellenar el frente de un campo de salida siempre que el ancho del campo es mayor que el valor que se está desplegando.)
`setiosflags(flags)`	Establece el formato de los indicadores (véase la tabla 3.3 para las configuraciones de los indicadores).
`scientific`	Establece la salida para desplegar números reales en notación científica.
`showbase`	Despliega la base usada para los números. Se despliega un 0 a la izquierda para los números octales y un 0x a la izquierda para números hexadecimales.
`showpoint`	Siempre despliega seis dígitos en total (combinación de partes enteras y fraccionarias). Rellena con ceros a la derecha si es necesario. Para valores enteros mayores, revierte a notación científica.
`showpos`	Despliega todos los números positivos con un signo de + a la izquierda.
`boolalpha`	Despliega valores booleanos como verdadero y falso, en lugar de como 1 y 0.
`dec`	Establece la salida para un despliegue decimal por omisión.
`endl`	Da salida a un carácter de línea nueva y despliega todos los caracteres en el búfer.
`fixed`	Siempre muestra un punto decimal y usa seis dígitos por omisión después del punto decimal. Rellena con ceros a la derecha si es necesario.
`flush`	Despliega todos los caracteres en el búfer.
`left`	Justifica a la izquierda todos los números.
`hex`	Establece la salida para un despliegue hexadecimal.
`oct`	Establece la salida para un despliegue octal.
`uppercase`	Despliega dígitos hexadecimales y el exponente en notación científica en mayúsculas.
`right`	Justifica a la derecha todos los números (éste es el valor por omisión).
`noboolalpha`	Despliega valores booleanos como 1 y 0, en lugar de verdadero y falso.
`noshowbase`	No despliega números octales con un 0 a la izquierda y los números hexadecimales con un 0x a la izquierda.
`noshowpoint`	No usa un punto decimal para números reales sin partes fraccionarias, no despliega ceros a la derecha en la parte fraccionaria de un número y despliega un máximo de sólo seis dígitos decimales.
`noshowpos`	No despliega signos de + a la izquierda (éste es el valor por omisión).
`nouppercase`	Despliega dígitos hexadecimales y el exponente en notación científica en minúsculas.

Por ejemplo, la instrucción `cout << "La suma de 6 y 5 es" << setw(3) << 21;` crea esta impresión:

```
La suma de 6 y 5 es 21
```

El manipulador de ancho de campo `setw(3)` incluido en el flujo de datos pasado a `cout` se usa para establecer el ancho del campo desplegado. El 3 en este manipulador establece el ancho de campo por omisión para el siguiente número en el flujo para que tenga un ancho de tres espacios. Esta configuración del ancho de campo causa que 21 se imprima en un campo de tres espacios, el cual incluye un espacio en blanco y el número 21. Como se ilustra, los enteros están justificados a la derecha dentro del campo especificado.

Los manipuladores de ancho de campo son útiles para imprimir columnas de números de modo que los números en cada columna se alineen en forma correcta. Por ejemplo, el programa 3.5 ilustra cómo se alinearía una columna de números enteros en ausencia de manipuladores de ancho de campo.

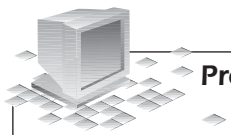
Programa 3.5

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

int main()
{
  cout << 6 << endl
       << 18 << endl
       << 124 << endl
       << "---\n"
       << (6+18+124) << endl;

  return 0;
}
```

La salida del programa 3.5 es la siguiente:

```
6
18
124
---
148
```

En vista que no se incluyeron manipuladores de ancho de campo en el programa 3.5, el objeto `cout` asigna suficiente espacio para cada número conforme lo recibe. Para forzar a los números a alinearse con el dígito de las unidades se requiere un ancho de campo suficiente para el número más grande desplegado. Para el programa 3.5, un ancho de tres bastaría. El uso de este ancho de campo se ilustra en el programa 3.6.



Programa 3.6

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

int main()
{
  cout << setw(3) << 6 << endl
       << setw(3) << 18 << endl
       << setw(3) << 124 << endl
       << "---\n"
       << (6+18+124) << endl;

  return 0;
}
```

La salida del programa 3.6 es

```
  6
 18
124
---
148
```

El manipulador de ancho de campo debe incluirse para cada ocurrencia de un número insertado en el flujo de datos enviado a `cout`; este manipulador particular sólo se aplica a la siguiente inserción de datos inmediata. Los otros manipuladores permanecen en efecto hasta que se cambian.

Cuando se usa un manipulador que requiere un argumento debe incluirse el archivo de encabezado `iomanip` como parte del programa. Esto se logra con el comando preprocesador `#include <iomanip>`, el cual se enlista como la segunda línea en el programa 3.6.

Dar formato completo a números de punto flotante requiere el uso de tres manipuladores de ancho de campo. El primer manipulador establece el ancho total del despliegue, el segundo fuerza el despliegue de un punto decimal y el tercer manipulador determina cuántos dígitos significativos se desplegarán a la derecha del punto decimal. Por ejemplo, analice la siguiente instrucción:

```
cout << "|" << setw(10) << fixed << setprecision(3) << 25.67 << "|";
```

Causa la siguiente impresión:

```
|     25.670|
```

El símbolo de barra, |, en el ejemplo se usa para delimitar (marcar) el principio y el fin del campo de despliegue. El manipulador `setw` le indica a `cout` que despliegue el número en un campo total de 10, el manipulador `fixed` fuerza de manera explícita el despliegue de un punto decimal y designa que el manipulador `setprecision` se usa para designar el número de dígitos que se va a desplegar después del punto decimal. En este caso, `setprecision` especifica un despliegue de tres dígitos después del punto decimal. Sin la designación explícita de un punto decimal (el cual también puede designarse como setiosflags(ios::fixed)), el manipulador `setprecision` especifica el número total de dígitos desplegados, el cual incluye las partes enteras y fraccionarias del número.

Para todos los números (enteros, de precisión simple y de precisión doble), `cout` ignora la especificación del manipulador `setw` si el campo especificado total es demasiado pequeño, y asigna suficiente espacio para la parte entera del número que se va a imprimir. La parte fraccionaria de los números de precisión simple y de precisión doble es desplegada hasta la precisión establecida con el manipulador `setprecision`. (En ausencia de un manipulador `setprecision`, la precisión por omisión se establece en seis lugares decimales.) Si la parte fraccionaria del número que se va a desplegar contiene más dígitos de los indicados en el manipulador `setprecision`, el número se redondea al número indicado de lugares decimales; si la parte fraccionaria contiene menos dígitos que los especificados, el número es desplegado con menos dígitos. La tabla 3.2 ilustra el efecto de varias combinaciones de manipuladores de formato. Una vez más, por claridad, se usa el símbolo de barra, |, para delinear el principio y el fin de los campos de salida.

Tabla 3.2 Efecto de los manipuladores de formato

Manipuladores	Número	Despliegue	Comentarios
`setw(2)`	`3`	`| 3|`	El número cabe en el campo
`setw(2)`	`43`	`|43|`	El número cabe en el campo
`setw(2)`	`143`	`|143|`	El ancho de campo se ignora
`setw(2)`	`2.3`	`|2.3|`	El ancho de campo se ignora

`setw(5)`
`fixed`
`setprecision(2)`
	`2.366`	`| 2.37|`	Ancho de campo de cinco con dos dígitos decimales

`setw(5)`
`fixed`
`setprecision(2)`
	`42.3`	`|42.30|`	El número cabe en el campo con precisión especificada

`setw(5)` `setprecision(2)`	`142.364`	`|1.4e+002|`	El ancho de campo se ignora y se usa notación científica con el manipulador `setprecision` especificando el número total de dígitos significativos (enteros más fraccionarios)

Punto de Información

¿Qué es un indicador o bandera?

En la programación actual, el término indicador o **bandera** se refiere a un elemento, como una variable o argumento, que establece una condición por lo general considerada activa o inactiva. Aunque se desconoce el origen exacto de este término en programación, es probable que se haya originado del uso de banderas reales para señalar una condición, como las banderas de alto, siga, precaución y ganador que se usan por lo común en las carreras de automóviles.

De manera similar, cada argumento de indicador para la función del manipulador `setiosflags()` activa una condición específica. Por ejemplo, el indicador `ios::dec` establece el formato de despliegue decimal, y el indicador `ios::oct` activa el formato de despliegue octal. En vista que estas condiciones son mutuamente excluyentes (sólo una condición puede estar activa a la vez), activar uno de estos indicadores desactiva de manera automática los otros indicadores.

Los indicadores que no son mutuamente excluyentes, como `ios::dec`, `ios::showpoint` e `ios::fixed` pueden establecerse como activas de manera simultánea. Esto puede hacerse usando tres llamadas `setiosflag()` individuales o combinando todos los argumentos en una llamada como sigue:

```
cout << setiosflags ios::dec | ios::fixed | ios::showpoint);
```

Tabla 3.2 Efecto de los manipuladores de formato (continuación)

Manipuladores	Número	Despliegue	Comentarios

`setw(5)`
`fixed`
`setprecision(2)`
	`142.364`	`|142.36|`	Se ignora el ancho de campo pero se usa la especificación de precisión. Aquí el manipulador `setprecision` especifica el número de dígitos fraccionarios

`setw(5)`
`fixed`
`setprecision(2)`
	`142.366`	`|142.37|`	Se ignora el ancho de campo pero se usa la especificación de precisión. Aquí el manipulador `setprecision` especifica el número de dígitos fraccionarios. (Nótese el redondeo del último dígito decimal)

`setw(5)` `fixed`<| `setprecision(2)`	`142`	`|  142|`	Se usa el ancho de campo, los manipuladores fixed y `setprecision` irrelevantes, porque el número es un entero.

Además de los manipuladores `setw` y `setprecision`, también está disponible un manipulador de justificación de campo. Como se ha visto, los números enviados a `cout` se despliegan por lo normal con justificación a la derecha en el campo de despliegue, mientras las cadenas se despliegan justificadas a la izquierda. Para alterar la justificación por omisión para un flujo de datos, puede usarse el manipulador `setiosflags`. Por ejemplo, analice la siguiente instrucción:

```
cout << "|" << setw(10) << setiosflags(ios::left) << 142 << "|";
```

Esto causa el siguiente despliegue justificado a la izquierda:

```
|142        |
```

Como se ha visto, ya que los datos pasados a `cout` pueden continuarse a lo largo de múltiples líneas, el despliegue anterior también sería producido por la instrucción:

```
cout << "|" << setw(10)
        << setiosflags(ios::left)
        << 142 << "|";
```

Como siempre, el manipulador de ancho de campo sólo está para el siguiente conjunto sencillo de datos desplegado por `cout`. La justificación a la derecha para las cadenas en un flujo se obtiene con el manipulador `setiosflags(ios::right)`. El símbolo `ios` tanto en el nombre de la función como en el argumento `ios::right` se deriva de las primeras letras de las palabras "input output stream" (flujo de entrada y salida).

Además de los indicadores de izquierda y derecha que pueden usarse con el manipulador `setiosflags()`, pueden usarse otros indicadores para afectar la salida. Los indicadores más usados para este manipulador se enumeran en la tabla 3.3. Los indicadores en esta tabla proporcionan de manera efectiva una forma alternativa para establecer los manipuladores equivalentes enumerados en la tabla 3.1.

Tabla 3.3 Indicadores de formato para usar con `setiosflags()`

Indicador	Significado
`ios::fixed`	Siempre muestra el punto decimal con seis dígitos después del punto decimal. Rellena con ceros a la derecha si es necesario. Este indicador tiene precedencia si se establece con el indicador `ios::showpoint`.
`ios::scientific`	Usa despliegue exponencial en la salida.
`ios::showpoint`	Siempre despliega un punto decimal y seis dígitos significativos en total (combinación de partes enteras y fraccionarias). Rellena con ceros a la derecha después del punto decimal si es necesario. Para valores enteros más grandes, revierte a notación científica a menos que esté establecido el indicador `ios::fixed`.
`ios::showpos`	Despliega un signo + a la izquierda cuando el número es positivo.
`ios::left`	Justifica a la izquierda la salida.
`ios::right`	Justifica a la derecha la salida.

Punto de Información

Dar formato a los datos en el flujo de `cout`

Los datos de punto flotante en un flujo de salida `cout` pueden formatearse en formas precisas. Uno de los requerimientos de formato más comunes es desplegar números en un formato monetario con dos dígitos después del punto decimal, como 123.45. Esto puede hacerse con la siguiente instrucción:

```
cout << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint)
     << setprecision(12);
```

El primer indicador en el manipulador, `ios::fixed`, hace que todos los números de punto flotante colocados en el flujo `cout` se desplieguen en notación decimal. Este indicador también impide el uso de notación científica. El siguiente indicador, `ios::showpoint`, le indica al flujo que siempre despliegue un punto decimal. Por último, el manipulador `setprecision` le indica al flujo que siempre despliegue dos valores decimales después del punto decimal. En lugar de usar manipuladores, también puede usar los métodos de flujo de `cout setf()` y `precision()`. Por ejemplo, el formato anterior puede lograrse también usando el código:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

Nótese la sintaxis aquí: el nombre del objeto, `cout`, se separa del método con un punto. Ésta es la forma estándar de especificar un método y conectarlo con un objeto específico. El estilo que seleccione es cuestión de preferencia.

Además, los indicadores usados tanto en el método `setf()` como en el manipulador `setiosflags()` pueden combinarse usando el operador a nivel de bit Or, | (que se explica en la sección 15.2). Usando este operador, las siguientes dos instrucciones son equivalentes.

```
cout << setiosflags(ios::fixed | ios::showpoint);
cout.setf(ios::fixed | ios::showpoint);
```

El estilo que seleccione es cuestión de preferencia.

Debido a que los indicadores en la tabla 3.3 se usan como argumentos para el método del manipulador `setiosflags()` y debido a que los términos "argumento" y "parámetro" son sinónimos, otro nombre para un método manipulador que use argumentos es el de **manipulador parametrizado**. El siguiente es un ejemplo de métodos de manipulador parametrizado:

```
cout << setiosflags(ios::showpoint) << setprecision(4);
```

Esto hace que todos los números de punto flotante subsiguientes sean enviados al flujo de salida para ser desplegados con un punto decimal y cuatro dígitos decimales. Si el número tiene menos de cuatro dígitos decimales, se llenará con ceros a la derecha.

Además de dar salida a los enteros en notación decimal, los manipuladores `oct` y `hex` permiten conversiones a formato octal y hexadecimal, respectivamente. El programa 3.7 ilustra el uso de estos indicadores. Debido a que el formato decimal es el despliegue por omisión, el manipulador `dec` no se requiere en el primer flujo de salida.

Programa 3.7

```
// un programa que ilustra conversiones de salida
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << "El valor decimal (base 10) de 15 es " << 15 << endl;
  cout << "El valor octal (base 8) de 15 es "
       << showbase << oct << 15 <<endl;
  cout << "El valor hexadecimal (base 16) de 15 es "
         << showbase << hex << 15 << endl;

  return 0;
}
```

La salida producida por el programa 3.7 es la siguiente:

```
El valor decimal (base 10) de 15 es 15
El valor octal (base 8) de 15 es 017
El valor hexadecimal (base 16) de 15 es 0xf
```

El despliegue de valores enteros en uno de los tres sistemas numéricos posibles (decimal, octal y hexadecimal) no afecta la manera en que se almacena el número dentro de una computadora. Todos los números se almacenan usando los códigos internos propios de la computadora. Los manipuladores enviados a `cout` le indican al objeto cómo convertir el código interno con el propósito de desplegar la salida.

Además de desplegar enteros en forma octal o hexadecimal, pueden escribirse las constantes en números enteros en un programa en estos formatos. Para designar una constante entera octal, el número debe tener un cero a la izquierda. El número 023, por ejemplo, es un número octal en C++. Los números hexadecimales se denotan usando un 0x a la izquierda. El uso de constantes enteras octales y hexadecimales se ilustra en el programa 3.8.

Programa 3.8

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

int main()
{
  cout << "El valor decimal de 025 es " << 025 << endl;
       << "El valor decimal de 0x37 es "<< 0x37 << endl;

  return 0;
}
```

La salida producida por el programa 3.8 es la siguiente:

```
El valor decimal de 025 es 21
El valor decimal de 0x37 es 55
```

La relación entre la entrada, almacenamiento y despliegue de enteros se ilustra en la figura 3.8.

Por último, los manipuladores especificados en las tablas 3.1 y 3.2 pueden establecerse usando los métodos de clase `ostream` enumerados en la tabla 3.4.

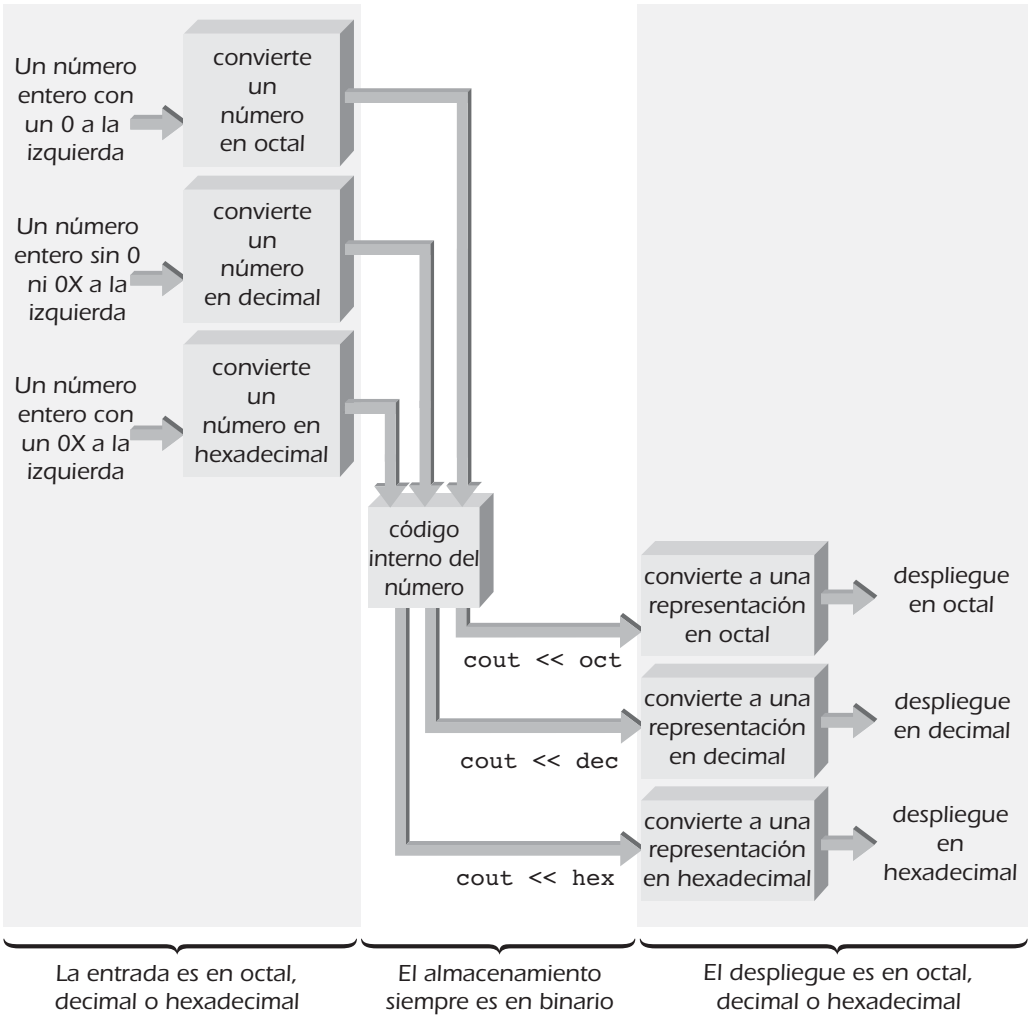


Figura 3.8 *Entrada, almacenamiento y despliegue de números enteros.*

Tabla 3.4 Métodos de clase `ostream`

Método	Comentario	Ejemplo
`precision(n)`	Equivalente a `setprecision()`	`cout.precision(2)`
`fill('x')`	Equivalente a `setfill()`	`cout.fill('*')`
`setf(ios::fixed)`	Equivalente a `setiosflags(ios::fixed)`	`cout.setf(ios::fixed)`
`setf(ios::showpoint)`	Equivalente a `setiosflags(ios::showpoint)`	`cout.setf(ios::showpoint)`
`setf(iof::left)`	Equivalente a `left`	`cout.setf(ios::left)`
`setf(ios::right)`	Equivalente a `right`	`cout.setf(ios::right)`
`setf(ios::flush)`	Equivalente a `endl`	`cout.setf(ios::flush)`

En la columna de ejemplos de la tabla 3.4, el nombre del objeto, `cout`, se separa del método con un punto. Ésta es la forma estándar de llamar a un método de clase y proporcionarle el objeto sobre el que va a operar.

Ejercicios 3.2

1. Determine la salida del siguiente programa:

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

int main()  // un programa que ilustra el truncamiento de
                números enteros
{
  cout << "respuesta1 es el entero " <<  9/4
       << "\nrespuesta2 es el entero " <<  17/3 << endl;

  return 0;
}
```

2. Determine la salida del siguiente programa:

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

int main()  // un programa que ilustra el operador
{
  cout << "El residuo de 9 dividido entre 4 es " <<  9 % 4
       << "\nEl residuo de 17 dividido entre 3 es " << 17 % 3 << endl;

  return 0;
}
```

3. Escriba un programa en C++ que despliegue los resultados de las expresiones 3.0 * 5.0, 7.1 * 8.3 – 2.2 y 3.2 / (6.1 * 5). Calcule el valor de estas expresiones en forma manual para verificar que los valores desplegados son correctos.

4. Escriba un programa en C++ que despliegue los resultados de las expresiones 15 / 4, 15 % 4 y 5 * 3 – (6 * 4). Calcule el valor de estas expresiones en forma manual para verificar que el despliegue producido por su programa es correcto.

5. Determine los errores en cada una de las siguientes instrucciones:

   a. `cout << "\n << " 15)`
   b. `cout << "setw(4)" <<  33;`
   c. `cout << "setprecision(5)" <<  526.768;`
   d. `"Hello World!" >> cout;`
   e. `cout << 47 << setw(6);`
   f. `cout << set(10) << 526.768 << setprecision(2);`

6. Determine y escriba el despliegue producido por las siguientes instrucciones:

   a. `cout << "|" << 5 <<"|";`
   b. `cout << "|" << setw(4) << 5 << "|";`
   c. `cout << "|" << setw(4) << 56829 << "|";`
   d.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 5.26 << "|";
```
   e.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 5.267 << "|";
```
   f.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 53.264 << "|";
```
   g.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 534.264 << "|";
```
   h.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 534. << "|";
```

7. Escriba el despliegue producido por las siguientes instrucciones.

   a.
```
cout << "El número es " << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 26.27 << endl;
cout << "El número es " << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 682.3 << endl;
cout << "El número es " << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 1.968 << endl;
```
   b.
```
cout << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 26.27 << endl;
cout << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 682.3 << endl;
cout << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 1.968 << endl;
cout << "------\n";
cout << setw(6) << setiosflags(ios::fixed)
     << setprecision(2)
     << 26.27 + 682.3 + 1.968 << endl;
```

c.
```
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 26.27 << endl;
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 682.3 << endl;
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 1.968 << endl;
cout << "-----\n";
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2)
     << 26.27 + 682.3 + 1.968 << endl;
```

d.
```
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 36.164 << endl;
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 10.003 << endl;
cout << "-----" << endl;
```

8. La siguiente tabla enumera la correspondencia entre los números decimales 1 a 15 y su representación octal y hexadecimal.

Decimal:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Octal:	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
Hexadecimal:	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Usando la tabla anterior, determine la salida del siguiente programa.

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

int main()
{
  cout << "\nEl valor de 14 en octal es " << oct << 14
       << "\nEl valor de 14 en hexadecimal es " << hex << 14
       << "\nEl valor de 0xA en decimal es " << dec << 0xA
       << "\nEl valor de 0xA en octal es " << oct << 0xA
       << endl;

  return 0;
}
```

9. La resistencia combinada de tres resistores conectados en paralelo, como se muestra en la figura 3.9, está dada por la ecuación

$$\text{Resistencia combinada} = \frac{1}{\frac{1}{R_1}+\frac{1}{R_2}+\frac{1}{R_3}}$$

Usando esta fórmula, escriba un programa en C++ para calcular y desplegar la resistencia combinada cuando los tres resistores $R_1 = 1000$, $R_2 = 1000$ y $R_3 = 1000$ están conectados en paralelo. La salida deberá producir el despliegue

```
La resistencia combinada es xxxx.xx ohmios,
```

donde xxxx.xx denota que el valor calculado deberá colocarse en un ancho de campo de 7 columnas, con dos posiciones a la derecha del punto decimal.

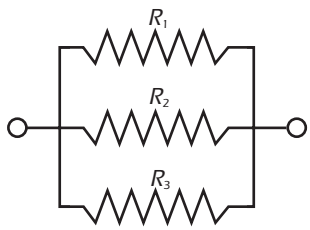


Figura 3.9 *Tres resistores conectados en paralelo.*

10. Escriba un programa en C++ para calcular y desplegar el valor de la pendiente de la línea que conecta los dos puntos cuyas coordenadas son (3, 7) y (8, 12). Use el hecho que la pendiente entre dos puntos que tienen coordenadas (x1, y1) y (x2, y2) es pendiente = (y2 – y1) / (x2 – x1). El despliegue producido por su programa deberá ser: El valor de la pendiente es xxx.xx, donde xxx.xx denota que el valor calculado deberá ser colocado en un ancho de campo suficiente para tres lugares a la izquierda del punto decimal y dos lugares a la derecha de éste.

11. Escriba un programa en C++ para calcular y desplegar las coordenadas del punto medio de la línea que conecta los dos puntos cuyas coordenadas son (3, 7) y (8, 12). Use el hecho que las coordenadas del punto medio entre dos puntos que tienen coordenadas (x1, y1) y (x2, y2) son ((X1 + X2)/2, (Y1 + Y2)/2). El despliegue producido por su programa deberá ser:

```
La coordenada x del punto medio es xxx.xx
La coordenada y del punto medio es xxx.xx
```

donde xxx.xx denota que el valor calculado deberá colocarse en un ancho de campo suficiente para tres lugares a la izquierda del punto decimal y dos lugares a la derecha de éste.

12. Escriba un programa en C++ para calcular y desplegar el momento de flexión máxima, *M*, de una viga, la cual está sostenida en ambos extremos (véase la figura 3.10). La fórmula para el momento de flexión máximo es, $M = XW(L - X) / L$, donde *X* es la distancia del extremo de la viga en que se coloca un peso, *W* y *L* es el largo de la viga. El despliegue producido por su programa deberá ser

```
El momento de flexión máxima es xxxx.xxxx
```

donde xxxx.xxxx denota que el valor calculado deberá colocarse en un ancho de campo suficiente para cuatro lugares a la derecha y a la izquierda del punto decimal.

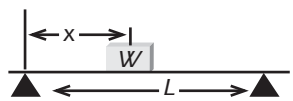


Figura 3.10 *Cálculo del momento de flexión máxima.*

13. Para el circuito eléctrico mostrado en la figura 3.11, las corrientes en los ramales, i_1, i_2 e i_3 pueden determinarse usando las fórmulas

$$i_1 = \frac{E_2R_3 + E_1(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$i_2 = \frac{E_1R_3 + E_2(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$i_3 = i_1 - i_2$$

Usando estas fórmulas, escriba un programa en C++ para calcular las corrientes en los ramales cuando R_1 = 10 ohmios, R_2 = 4 ohmios, R_3 = 6 ohmios, E_1 = 12 voltios y E_2 = 9 voltios. El despliegue producido por su programa deberá ser

```
La corriente en el ramal 1 es xx.xxxxx
La corriente en el ramal 2 es xx.xxxxx
La corriente en el ramal 3 es xx.xxxxx
```

donde xx.xxxxx denota que el valor calculado deberá colocarse en un ancho de campo suficiente para dos lugares a la izquierda del punto decimal y cinco lugares a la derecha de éste.

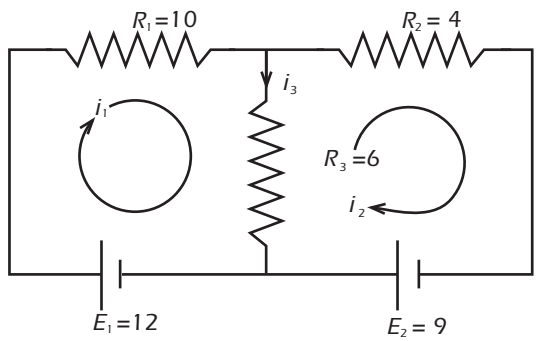


Figura 3.11 *Cálculo de corrientes en circuito en un circuito eléctrico.*

3.3 EMPLEO DE LA BIBLIOTECA DE FUNCIONES MATEMÁTICAS

Como se ha visto, las instrucciones de asignación pueden usarse para ejecutar cálculos aritméticos. Por ejemplo, la instrucción de asignación

```
voltios = resistencia * corriente;
```

multiplica el valor en `corriente` por el valor en `resistencia` y asigna el valor resultante a `voltios`. Aunque la adición, sustracción, multiplicación y división se logran con facilidad usando operadores aritméticos de C++, no existen operadores para elevar un número a una potencia, encontrar la raíz cuadrada de un número o determinar valores trigonométricos.

Para facilitar estos cálculos, C++ proporciona funciones preprogramadas estándares que pueden incluirse en un programa.

Antes de usar una de las funciones matemáticas de C++, necesita saber

- El nombre de la función matemática deseada
- Qué hace la función matemática
- El tipo de datos requerido por la función matemática
- El tipo de datos del resultado devuelto por la función matemática
- Cómo incluir la biblioteca

Para ilustrar el uso de las funciones matemáticas de C++, considere la función matemática llamada `sqrt`, la cual calcula la raíz cuadrada de un número. La raíz cuadrada de un número se calcula usando la expresión

```
sqrt(número)
```

donde el nombre de la función, en este caso `sqrt`, es seguido por paréntesis que contienen el número cuya raíz cuadrada se desea calcular. El propósito de los paréntesis que siguen al nombre de la función es proporcionar un embudo por el que puedan pasar los datos a la función (véase la figura 3.12). Los elementos que pasan a la función por medio de los paréntesis se llaman argumentos de la función y constituyen sus datos de entrada. Por ejemplo, las siguientes expresiones se usan para calcular la raíz cuadrada de los argumentos 4., 17.0, 25., 1043.29 and 6.4516, respectivamente:

```
sqrt(4.)
sqrt(17.0)
sqrt(25.)
sqrt(1043.29)
sqrt(6.4516)
```

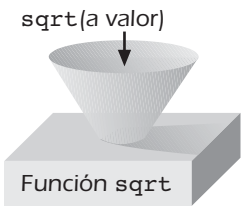


Figura 3.12 *Transmisión de datos a la función* `sqrt()`.

Hay que observar que el argumento para la función `sqrt()` debe ser un valor real. Éste es un ejemplo de las capacidades de sobrecarga de la función de C++. La sobrecarga de la función permite que el mismo nombre de la función sea definido para argumentos con diferentes tipos de datos. En este caso en realidad hay tres funciones de raíz cuadrada nombradas `sqrt()`, una definida para argumentos en número de punto flotante, de doble precisión y de

doble precisión largos. La función `sqrt` correcta es invocada dependiendo del tipo de valor que se le da. La función `sqrt()` determina la raíz cuadrada de su argumento y devuelve el resultado como un número doble. Los valores devueltos por las expresiones anteriores son

Expresión	Valor devuelto
`sqrt(4.)`	2
`sqrt(17.0)`	4.12311
`sqrt(25.)`	5
`sqrt(1043.29)`	32.3
`sqrt(6.4516)`	2.54

Además de la función `sqrt`, la tabla 3.5 enumera las funciones matemáticas de C++ más usadas. Tener acceso a estas funciones en un programa requiere que se incluya con la función el archivo de encabezado matemático llamado `cmath`, el cual contiene declaraciones apropiadas para la función matemática. Esto se hace colocando la siguiente instrucción preprocesadora al principio de cualquier programa que use una función matemática:

```
#include <cmath>
```
← sin punto y coma

Aunque algunas de las funciones matemáticas enumeradas requieren más de un argumento, todas las funciones, por definición, pueden devolver en forma directa como máximo un valor. Además, todas las funciones enumeradas están sobrecargadas: esto significa que puede usarse el mismo nombre de función con argumentos con números enteros y reales. La tabla 3.6 ilustra el valor devuelto por funciones selectas usando argumentos de ejemplo.

Tabla 3.5 Funciones comunes de C++

Nombre de la función	Descripción	Valor devuelto
`abs(a)`	valor absoluto	mismo tipo de datos que el argumento
`pow(a1,a2)`	a1 elevado a la potencia a2	tipo de datos del argumento a1
`sqrt(a)`	raíz cuadrada de un número real	precisión doble
`sin(a)`	seno de a (a en radianes)	doble
`cos(a)`	coseno de a (d en radianes)	doble
`tan(a)`	tangente de a (d en radianes)	doble
`log(a)`	logaritmo natural de a	doble
`log10(a)`	logaritmo común (base 10) de a	doble
`exp(a)`	e elevado a la potencia a	doble

Table 3.6 Ejemplos de funciones selectas

Ejemplo	Valor devuelto
`abs(-7.362)`	7.362
`abs(-3)`	3
`pow(2.0,5.0)`	32
`pow(10,3)`	1000
`log(18.697)`	2.92836
`log10(18.697)`	1.27177
`exp(-3.2)`	0.040762

En cada caso que se usa una función matemática, ésta se activa al dar el nombre de la función y transmitirle datos dentro del paréntesis que sigue al nombre de la función (véase la figura 3.13).

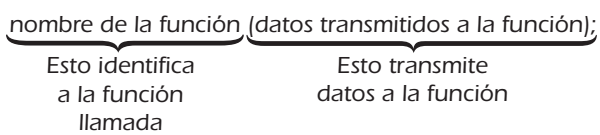


Figura 3.13 *Uso y transmisión de datos a una función.*

Los argumentos que se transmiten a una función no necesitan ser constantes simples. Las expresiones también pueden ser argumentos, siempre que la expresión pueda calcularse para producir un valor del tipo de datos requerido. Por ejemplo, los siguientes argumentos son válidos para las funciones dadas:

```
sqrt(4.0 + 5.3 * 4.0)          abs(2.3 * 4.6)
sqrt(16.0 * 2.0 - 6.7)         sin(theta - phi)
sqrt(x * y - z/3.2)            cos(2.0 * omega)
```

Las expresiones entre paréntesis se evalúan primero para producir un valor específico. Por tanto, tendrían que asignarse valores a las variables theta, phi, x, y, z y omega antes de usarse en las expresiones anteriores. Después que se calcula el valor del argumento, éste se transmite a la función.

Las funciones también pueden incluirse como parte de expresiones más grandes. Por ejemplo,

```
   4 * sqrt(4.5 * 10.0 - 9.0) - 2.0
=  4 * sqrt(36.0) - 2.0
=  4 * 6.0 - 2.0
=  24.0 - 2.0
=  22.0
```

La evaluación paso por paso de una expresión como

```
3.0 * sqrt(5 * 33 - 13.71) / 5
```

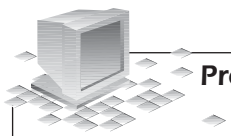
es

Paso	Resultado
1. Realizar la multiplicación en el argumento	`3.0 * sqrt(165 - 13.71) / 5`
2. Completar el cálculo del argumento	`3.0 * sqrt(151.29) / 5`
3. Devolver un valor de la función	`3.0 * 12.3 / 5`
4. Realizar la multiplicación	`36.9 / 5`
5. Realizar la división	`7.38`

El programa 3.9 ilustra el uso de la función `sqrt` para determinar el tiempo que tarda una pelota en golpear el suelo después de haber sido dejada caer desde una torre de 800 pies. La fórmula matemática usada para calcular el tiempo, en segundos, que tarda en caer una distancia determinada, en pies, es

```
tiempo = sqrt(2 * distancia / g)
```

donde *g* es la constante gravitacional igual a 32.2 pies/s^2.



Programa 3.9

```
#include <iostream>  // esta línea puede colocarse en segundo lugar en vez
                        de en primero
#include <cmath>     // esta línea puede colocarse en primer lugar en vez
                        de en segundo
using namespace std;

int main()
{
  int altura;
  double tiempo;

  altura = 800;
  tiempo = sqrt(2 * altura / 32.2);
  cout << "Tardará " << tiempo << " segundos en caer "
       << altura << " pies.\n";

  return 0;
}
```

La salida producida por el programa 3.9 es

```
Tardará 7.04907 segundos en caer 800 pies.
```

Como se usa en el programa 3.9, el valor devuelto por la función `sqrt` es asignado a la variable `tiempo`. Además de asignar el valor devuelto de una función a una variable, el valor devuelto puede incluirse dentro de una expresión más grande, o incluso usarse como un argumento para otra función. Por ejemplo, la expresión

```
sqrt( sin( abs(theta) ) )
```

es válida. En vista que están presentes paréntesis, el cálculo procede de los pares interiores hacia los pares de paréntesis exteriores. Por tanto, el valor absoluto de `theta` se calcula primero y se usa como un argumento para la función sin. El valor devuelto por la función `sin` se usa luego como un argumento para la función `sqrt()`.

Hay que observar que los argumentos de todas las funciones trigonométricas (`sin`, `cos`, etc.) deben expresarse en radianes. Por tanto, para obtener el seno de un ángulo que está dado en grados, primero debe convertirse el ángulo a una medida en radianes. Esto se logra con facilidad al multiplicar el ángulo por el término (3.1416/180.). Por ejemplo, para obtener el seno de 30 grados, puede usarse la expresión `sin (30 * 3.1416/180.)`

Moldes

Ya se ha visto la conversión del tipo de datos de un operando dentro de expresiones aritméticas en modo mixto (sección 2.4) y mediante operadores de asignación (sección 3.1). Además de estas conversiones del tipo de datos implícitas, C++ también proporciona conversiones de tipo explícitas especificadas por el usuario. El operador usado para forzar la conversión de un valor a otro tipo es el **operador de molde** (cast). C++ proporciona operadores de molde en tiempo de compilación y en tiempo de ejecución.

El molde en tiempo de compilación es un operador unitario que tiene la sintaxis `tipoDeDatos(expresión)`, donde `tipoDeDatos` es el tipo de datos deseado al que se convierte la expresión entre paréntesis. Por ejemplo, la siguiente expresión

```
int (a * b)
```

asegura que el valor de la expresión `a * b` es convertido en un valor de número entero.[7]

Con la introducción del estándar más reciente de C++, se incluyeron moldes en tiempo de ejecución. En este tipo de molde, la conversión de tipo solicitada es verificada en tiempo de ejecución y se aplica si la conversión produce un valor válido. Aunque se dispone de cuatro tipos de moldes en tiempo de ejecución, el molde más usado y que corresponde al molde en tiempo de compilación tiene la sintaxis siguiente:

```
staticCast<tipo-de-datos> (expresión)
```

Por ejemplo, el molde en tiempo de ejecución `staticCast<int>(a * b)` es equivalente al molde en tiempo de compilación `int (a* b)`.

[7]La sintaxis de tipo molde en C, en este caso `(int)(a * b)`, también funciona en C++.

Ejercicios 3.3

1. Escriba las llamadas de función para determinar:
   a. La raíz cuadrada de 6.37.
   b. La raíz cuadrada de x – y.
   c. El seno de 30 grados.
   d. El seno de 60 grados.
   e. El valor absoluto de $a^2 - b^2$.
   f. El valor de e elevado a la tercera potencia.

2. Para $a = 10.6$, $b = 13.9$, $c = -3.42$, determine los siguientes valores:
   a. `int (a)`
   b. `int (b)`
   c. `int (c)`
   d. `int (a + b)`
   e. `int (a) + b + c`
   f. `int (a + b) + c`
   g. `int (a + b + c)`
   h. `float (int (a)) + b`
   i. `float (int (a + b))`
   j. `abs(a) + abs(b)`
   k. `sqrt(abs(a - b))`

3. Escriba instrucciones de C++ para lo siguiente:
   a. b = seno x – cos x
   b. $b = \text{seno}^2 x - \cos^2 x$
   c. *área* = (c * b * seno a)/2
   d. $c = \sqrt{a^2 + b^2}$
   e. $p = \sqrt{|m - n|}$
   f. $suma = \dfrac{a\left(r^n - 1\right)}{r - 1}$

4. Escriba, compile y ejecute un programa en C++ que calcule y devuelva la raíz cuarta del número 81.0, la cual es 3. Cuando haya verificado que su programa funciona en forma correcta, úselo para determinar la raíz cuarta de 1,728.896400. Su programa deberá usar la función `sqrt()`.

5. Escriba, compile y ejecute un programa en C++ que calcule la distancia entre dos puntos cuyas coordenadas son (7, 12) y (3, 9). Use el hecho que la distancia entre dos puntos que tienen coordenadas (x_1, y_1) y (x_2, y_2) es *distancia* = $sqrt([x_1 - x_2]^2 + [y_1 - y_2]^2)$. Cuando haya verificado que su programa funciona en forma correcta, calculando la distancia entre los dos puntos en forma manual, use su programa para determinar la distancia entre los puntos (–12, –15) y (22, 5).

6. Si se coloca una escalera de 20 pies en un ángulo de 85 grados sobre un lado de un edificio, como se ilustra en la figura 3.14, la altura a la que la escalera toca el edificio puede calcularse como *altura* = 20 * seno 85°. Calcule esta altura en forma manual y luego escriba, compile y ejecute un programa en C++ que determine y despliegue el valor de la altura. Cuando haya verificado que su programa funciona en forma correcta, úselo para determinar la altura de una escalera de 25 pies colocada en un ángulo de 85 grados.

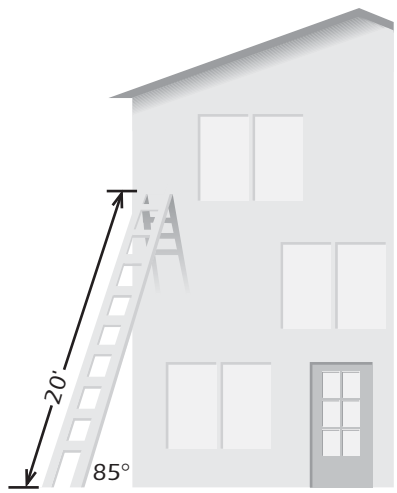


Figura 3.14 Calcular la altura de una escalera contra un edificio.

7. La altura máxima alcanzada por una pelota lanzada con una velocidad inicial v, en metros/segundo, en un ángulo de θ está dada por la fórmula *altura* = (.5 * v^2 * seno^2 θ) / 9.8. Usando esta fórmula, escriba, compile y ejecute un programa en C++ que determine y despliegue la altura máxima alcanzada cuando la pelota es lanzada a 5 millas/hora en un ángulo de 60 grados. (*Sugerencia:* Asegúrese de convertir la velocidad inicial en las unidades correctas. Hay 1609 metros en una milla.) Calcule la altura máxima en forma manual y verifique el resultado producido por su programa. Después de haber verificado que su programa funciona en forma correcta, úselo para determinar la altura alcanzada por una pelota lanzada a 7 millas/hora en un ángulo de 45 grados.

8. Para valores pequeños de x, el valor de seno(x) puede aproximarse con la serie de potencias:

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \cdots$$

Como con la función sin, el valor de x debe expresarse en radianes. Usando esta serie de potencias, escriba, compile y ejecute un programa en C++ que aproxime el seno de 180/3.1416 grados, lo cual es igual a un radián. Además, haga que su programa use la función sin para calcular el seno y desplegar tanto los valores calculados como la diferencia absoluta de los dos resultados. Verifique la aproximación producida por su programa en forma manual. Después que haya verificado que su programa funciona en forma correcta, úselo para aproximar el valor del seno de 62.2 grados.

9. Las coordenadas polares de un punto consisten en la distancia, r, de un origen especificado y un ángulo, θ, con respecto al eje x. Las coordenadas (x y y) del punto se relacionan con sus coordenadas polares por las fórmulas

$$x = r \cos \theta$$
$$y = r \text{ seno } \theta$$

Usando estas fórmulas, escriba un programa en C++ que calcule las coordenadas (x, y) del punto cuyas coordenadas polares son $r = 10$ y $\theta = 30$ grados. Verifique los resultados producidos por su programa calculando los resultados en forma manual. Después de haber verificado que su programa funciona en forma correcta, úselo para convertir las coordenadas polares $r = 12.5$ y $\theta = 67.8°$ en coordenadas rectangulares.

10. Un modelo del crecimiento de la población mundial, en miles de millones de personas, desde 2000 está dado por la ecuación:

$$Población = 6.0\ e^{0.02[año\ -\ 2000]}$$

Usando esta fórmula, escriba, compile y ejecute un programa en C++ para estimar la población mundial en el año 2005. Verifique el resultado desplegado por su programa calculando la respuesta en forma manual. Después que haya verificado que su programa funciona en forma correcta, úselo para estimar la población mundial en el año 2012.

11. Un modelo para estimar el número de gramos de un cierto isótopo radiactivo que restan después de N años está dado por la fórmula

$$Material\ remanente = (material\ original)\ e^{-0.00012N}$$

Usando esta fórmula, escriba, compile y ejecute un programa en C++ para determinar la cantidad de material radiactivo remanente después de 1000 años, suponiendo una cantidad inicial de 100 gramos. Verifique el despliegue producido por su programa usando un cálculo manual. Después que haya verificado que su programa funciona en forma correcta, úselo para determinar la cantidad de material radiactivo remanente después de 275 años, suponiendo una cantidad inicial de 250 gramos.

12. El número de años que se requiere para que se descomponga un cierto isótopo de uranio a la mitad de una cantidad original está dado por la

$$Vida\ media = \ln(2)/k$$

donde k es igual a 0.00012. Usando esta fórmula, escriba, compile y ejecute un programa en C++ que calcule y despliegue la vida media de este isótopo de uranio. Verifique el resultado producido por su programa usando un cálculo manual. Después que haya verificado que su programa funciona en forma correcta, úselo para determinar la vida media de un isótopo de uranio que tenga $k = 0.00026$.

13. La amplificación de circuitos electrónicos se mide en unidades de decibeles, las cuales se calculan como

$$10\ LOG\ (P_o/P_i)$$

donde P_o es la potencia de la señal de salida y P_i es la potencia de la señal de entrada. Usando esta fórmula, escriba, compile y ejecute un programa en C++ que calcule y despliegue la amplificación en decibeles en la que la potencia de salida es 50 veces

la potencia de entrada. Verifique el resultado desplegado por su programa usando un cálculo manual. Después de haber verificado que su programa funciona en forma correcta, úselo para determinar la amplificación de un circuito cuya potencia de salida es 4.639 vatios y la potencia de entrada es 1 vatio.

14. La intensidad de un sonido se mide en unidades de decibeles, las cuales se calculan como

```
10 LOG (SL/RL)
```

donde *SL* es la intensidad del sonido que se está midiendo y *RL* es un sonido de referencia del nivel de intensidad. Usando esta fórmula, escriba un programa en C++ que calcule y despliegue el ruido en decibeles de una calle transitada que tiene una intensidad de sonido de 10 000 000 *RL*. Verifique el resultado producido por su programa usando un cálculo manual. Después que haya verificado que su programa funciona en forma correcta, úselo para determinar el nivel de sonido, en decibeles, de los siguientes sonidos:

a. Un susurro con una intensidad de sonido de 200 *RL*
b. Una banda de rock tocando con una intensidad de sonido de 1 000 000 000 000 *RL*
c. Un avión despegando con una intensidad de sonido de 100 000 000 000 000 *RL*

15. Cuando una pelota de hule especial se deja caer desde una altura dada (en metros) su velocidad de impacto (en metros/segundo) cuando golpea el suelo está dada por la fórmula *velocidad* = sqrt*(2 * g * altura)*. La pelota rebota entonces a 2/3 de la altura desde la cual cayó la última vez. Usando esta información, escriba, pruebe y ejecute un programa en C++ que calcule y despliegue la velocidad de impacto de los primeros tres rebotes y la altura alcanzada en cada rebote. Pruebe su programa usando una altura inicial de 2.0 metros. Ejecute el programa dos veces y compare los resultados de soltar la pelota en la Tierra (g = 9.81 m/s^2) y en la Luna (g = 1.67 m/s^2).

16. a. Una balanza tiene pesas de los siguientes tamaños: 100 lb., 50 lb., 10 lb., 5 lb. y 1 lb. El número de pesas de 100 lb. y 50 lb. requeridas para pesar un objeto cuyo peso es de `PESO` libras puede calcularse usando las siguientes instrucciones de C++:

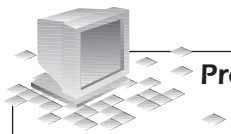
```
// Determine la cantidad de pesas de 100 lb.
      w100 = int(WEIGHT/100)
// Determine la cantidad de pesas de 50 lb.
      w50 = int((WEIGHT - w100 * 100)/50)
```

Usando estas instrucciones como punto de partida, escriba un programa en C++ que calcule la cantidad de cada tipo de pesas necesarias para pesar un objeto de 789 lb.

b. Sin compilar ni ejecutar su programa, compruebe el efecto, en forma manual, de cada instrucción en el programa y determine qué está almacenado en cada variable conforme se encuentra cada instrucción.
c. Cuando haya verificado que su algoritmo funciona en forma correcta, compile y ejecute su programa. Verifique que los resultados producidos por su programa son correctos. Después que haya verificado que su programa funciona en forma correcta, úselo para determinar las pesas requeridas para pesar un objeto de 626 lb.

3.4 ENTRADA DE DATOS AL PROGRAMA USANDO EL OBJETO cin

Los datos para programas que sólo se van a ejecutar una vez pueden incluirse en forma directa en el programa. Por ejemplo, si se desea multiplicar los números 30.0 y 0.05, podría usarse el programa 3.10.



Programa 3.10

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

int main()
{
  double num1, num2, producto;

  num1 = 30.0;
  num2 = 0.05;
  producto = num1 * num2;
  cout << "30.0 por 0.05 es " << producto << endl;

  return 0;
}
```

La salida desplegada por el programa 3.10 es

```
30.0 por 0.05 es 1.5
```

El programa 3.10 puede acortarse, como se ilustra en el programa 3.11. Ambos programas, sin embargo, tienen el mismo problema básico de que deben rescribirse a fin de multiplicar diferentes números. Ambos programas carecen de la facilidad para introducir números diferentes con los cuales operar.

Programa 3.11

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

int main()
{
  cout << "30.0 por 0.05 es " << 30.0 * 0.05 << endl;

  return 0;
}
```

Con excepción de la práctica proporcionada al programador por escribir, introducir y ejecutar el programa, es evidente que los programas que hacen el mismo cálculo sólo una vez, con el mismo conjunto de números, no son muy útiles. Después de todo, es más simple usar una calculadora para multiplicar dos números que introducir y ejecutar el programa 3.10 o 3.11.

Esta sección presenta el objeto `cin`, el cual se usa para introducir datos en un programa mientras se está ejecutando. Del mismo modo que el objeto `cout` despliega una copia del valor almacenado dentro de una variable, el objeto `cin` permite al usuario introducir un valor en la terminal (véase la figura 3.15). El valor se almacena entonces en forma directa en una variable.

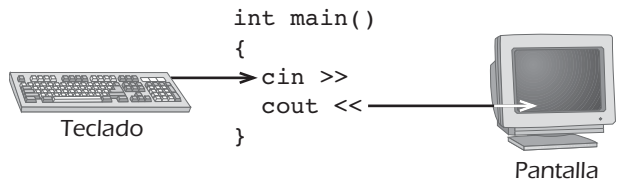


Figura 3.15 `cin` *se usa para introducir datos; cout se usa para desplegar datos.*

Cuando se encuentra una instrucción como `cin >> num1;` la computadora detiene la ejecución del programa y acepta datos del teclado. Cuando se escribe un elemento de datos, el objeto `cin` almacena el elemento en la variable mostrada después del operador de extracción ("obtener de"), `>>`. El programa continúa luego su ejecución con la siguiente instrucción después de la llamada a `cin`. Para ver esto, considere el programa 3.12.

Programa 3.12

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

int main()
{
  double num1, num2, producto;

  cout << "Por favor introduzca un número: ";
  cin  >> num1;
  cout << "Por favor introduzca otro número: ";
  cin  >> num2;
  producto = num1 * num2;
  cout << num1 << " por " << num2 << " es " << producto << endl;

  return 0;
}
```

La primera instrucción `cout` en el programa 3.12 imprime una cadena que le indica a la persona en la terminal qué deberá introducir. Cuando se usa una cadena de salida de esta manera se llama ***indicador de comandos***. En este caso el indicador de comandos le indica al usuario que introduzca un número. La computadora entonces ejecuta la siguiente instrucción, la cual usa un objeto `cin`. El objeto `cin` pone a la computadora en un estado de pausa temporal (o espera) tanto tiempo como le tome al usuario introducir un valor. Luego el usuario le señala al objeto `cin` que se terminó la entrada de datos al oprimir la tecla de retorno después que se ha introducido el valor. El valor introducido se almacena en la variable a la derecha del símbolo de extracción, y la computadora sale de su estado de pausa. Luego procede la ejecución del programa con la siguiente instrucción, la cual en el programa 3.12 es otra instrucción que usa `cout`. Esta instrucción causa que se despliegue el siguiente mensaje. La siguiente instrucción usa entonces `cin` para poner de nuevo a la computadora en un estado de espera temporal mientras el usuario introduce un segundo valor. Este segundo número se almacena en la variable `num2`.

La siguiente ejecución se hizo usando el programa 3.12.

```
Por favor introduzca un número: 30
Por favor introduzca otro número: 0.05
30 por 0.05 es 1.5
```

En el programa 3.12, cada vez que se invoca `cin` se usa para almacenar un valor en una variable. Sin embargo, el objeto `cin` puede usarse para introducir y almacenar tantos valores como símbolos de extracción, `>>`, y variables haya para contener los datos introducidos. Por ejemplo, la instrucción

```
cin >> num1 >> num2;
```

produce dos valores que se leen de la terminal y se asignan a las variables `num1` y `num2.` Si los datos introducidos en la terminal fueran

```
0.052  245.79
```

las variables `num1` y `num2` contendrían los valores 0.052 y 245.79, respectivamente. Hay que observar que cuando se introducen cifras como 0.052 y 245.79 debe haber al menos un espacio entre ellas. El espacio entre las cifras introducidas indica con claridad dónde termina una cifra y comienza la siguiente. Insertar más de un espacio entre cifras no tiene efecto en `cin.`

El mismo espaciado también es aplicable al introducir datos de caracteres; es decir, el operador de extracción, `>>`, se saltará los espacios en blanco y almacenará el siguiente carácter que no sea un espacio en blanco en una variable de carácter. Por ejemplo, en respuesta a las instrucciones

```
char ch1, ch2, ch3;  // declara tres variables de carácter
cin >> ch1 >> ch2 >> ch3;  // acepta tres caracteres
```

La entrada

```
a  b  c
```

causa que la letra `a` sea almacenada en la variable `ch1`, sea almacenada en la variable `ch2,` y la letra `c` sea almacenada en la variable `ch3`. En vista que una variable de carácter sólo puede usarse para almacenar un carácter,

```
abc
```

también puede usarse la entrada.

Puede utilizarse cualquier cantidad de instrucciones que usen el objeto `cin` e introducirse cualquier cantidad de valores usando una sola instrucción `cin`. El programa 3.13 ilustra el uso del objeto `cin` para introducir tres números desde el teclado. El programa calcula luego y despliega el promedio de los números introducidos.

Programa 3.13

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

int main()
{
  int num1, num2, num3;
  double promedio;

  cout << "Introduzca tres números enteros: ";
  cin >> num1 >> num2 >> num3;
  promedio =  (num1 + num2 + num3) / 3.0;
  cout << "El promedio de los números es " << promedio << endl;

  return 0;
}
```

La siguiente muestra de ejecución se hizo usando el programa 3.13:

```
Introduzca tres números enteros: 22 56 73
El promedio de los números es 50.3333
```

Hay que observar que los datos introducidos en el teclado para esta muestra de ejecución consisten en la entrada

```
22 56 73
```

En respuesta a este flujo de entrada, el programa 3.13 almacena el valor 22 en la variable `num1`, el valor 56 en la variable `num2`, y el valor 73 en la variable `num3` (véase la figura 3.16). En vista que el promedio de tres números enteros puede ser un número de punto flotante, la variable promedio, la cual se usa para almacenar el promedio, es declarada como una variable de punto flotante. Hay que observar también que los paréntesis son necesarios en la instrucción de asignación `promedio = (num1 + num2 + num3) / 3.0;`. Sin estos paréntesis, el único valor que se dividiría entre tres sería el entero en num3 (porque la división tiene una precedencia mayor que la adición).

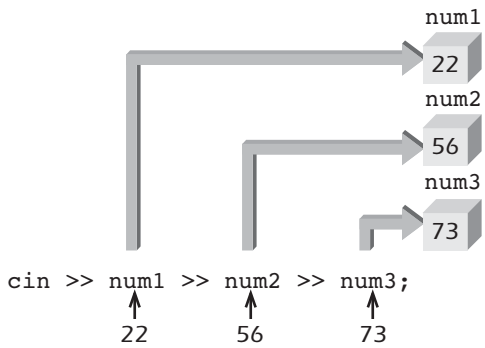


Figura 3.16 *Introducción de datos en las variables* `num1`, `num2`, *and* `num3`.

La operación de extracción `cin` como la operación de inserción `cout` es lo bastante "lista" para hacer unas cuantas conversiones de tipo de datos. Por ejemplo, si se introduce un número entero en lugar de un número de precisión doble, el entero será convertido al tipo de datos correcto.[8] Del mismo modo, si se introduce un número de precisión doble cuando se espera un entero, sólo se usará la parte entera del número. Por ejemplo, suponga que se introducen los siguientes números en respuesta a la instrucción `cin >> num1 >> num2 >> num3;`, donde `num1` y `num3` han sido declarados como variables de precisión doble y `num2` es una variable entera:

```
56  22.879  33.923
```

El *56* será convertido en 56.0 y almacenado en la variable `num1`. La operación de extracción continúa extrayendo datos del flujo de entrada que se le envía, esperando un valor entero. Por lo que respecta a `cin` el punto decimal después de 22 en el número 22.879 indica el final de un entero y el inicio de un número decimal. Por tanto, el número 22 es asignado a `num2`. Al continuar procesando su flujo de entrada, `cin` toma .879 como el siguiente número de punto flotante y lo asigna `num3`. Por lo que respecta a `cin` es entrada extra y es ignorado. Sin embargo, si al principio no se introducen suficientes datos, el objeto `cin` continuará haciendo que la computadora esté en pausa hasta que se hayan introducido suficientes datos.

Una primera mirada a la validación de entradas del usuario

Un programa bien construido debería validar las entradas del usuario y asegurar que no se caiga el programa o produzca una salida sin sentido debido a una entrada inesperada. El término *validar* significa verificar que el valor introducido corresponde al tipo de datos de la variable a la que es asignado el valor dentro de una instrucción `cin` y también que el valor está dentro de un rango aceptable de valores apropiados para la aplicación. Los programas que detectan y responden en forma efectiva a una entrada inesperada del usuario se conocen de manera formal como **programas robustos** y de manera informal como programas "a prueba de balas". Una de sus labores como programador es producir tales programas. Tal como están escritos, los programas 3.12 y 3.13 no son programas robustos. Veamos por qué.

[8]En sentido estricto, lo que viene del teclado no es ningún tipo de datos, como un `int` o `double`, sino es tan sólo una secuencia de caracteres. La operación de extracción maneja la conversión desde secuencia de caracteres hacia un tipo de datos definido.

El primer problema con estos programas se hace evidente cuando un usuario introduce un valor no numérico. Por ejemplo, considere la siguiente muestra de ejecución usando el programa 3.13:

```
Introduzca tres números enteros: 10 20.68 20
El promedio de los números es -2.86331e+008
```

Esta salida ocurre debido a que la conversión del segundo número introducido produce que el valor entero 20 sea asignado a `num2` y el valor –858993460 sea asignado a `num3`.[9] Este último valor corresponde a un carácter inválido, el punto decimal, al que se le asigna un valor entero esperado. El promedio de los números 10, 20 y –858993460 es calculado en forma correcta como –286331143.3, el cual es desplegado en notación científica con seis dígitos significativos como –2.86331e+08. Por lo que respecta al usuario de promedio, esto se reportará como un error del programa. Este mismo problema ocurre siempre que se introduce un valor no entero en cualquiera de las primeras dos entradas. (No ocurre para cualquier valor numérico introducido como la tercera entrada porque la parte entera de la última entrada es aceptada y se ignora la entrada restante.) Como programador, su respuesta inicial puede ser "El programa pide en forma clara que se introduzcan valores enteros". Ésta, sin embargo, es la respuesta de un programador inexperto. Los programadores profesionales entienden que es su responsabilidad asegurar que un programa anticipe y maneje en forma apropiada todas las entradas que introducirá un usuario. Esto se logra pensando qué puede salir mal con su programa mientras lo desarrolla y luego haciendo que otra persona o grupo lo pruebe.

En enfoque básico para manejar la introducción de datos inválidos se conoce como ***validación de entradas del usuario***, lo cual significa validar los datos introducidos durante la introducción o inmediatamente después que los datos han sido introducidos y luego proporcionarle al usuario una forma de reintroducir cualesquiera datos inválidos. La validación de entradas del usuario es una parte esencial de cualquier programa viable desde el punto de vista comercial; si se hace en forma correcta, protege al programa de intentar procesar datos que pueden causar problemas de cálculo. Se verá cómo proporcionar este tipo de validación después de presentar las instrucciones de selección y repetición de C++ en los capítulos 4 y 5, respectivamente.

Ejercicios 3.4

1. Para las siguientes instrucciones de declaración, escriba una instrucción con el objeto `cin` que cause que la computadora entre en pausa mientras el usuario introduce los datos apropiados.

   a.
   ```
   int primernum;
   ```
   b.
   ```
   double calificacion;
   ```
   c.
   ```
   double segundonum;
   ```
   d.
   ```
   char valorclave;
   ```
   e.
   ```
   int mes, anios;
   double promedio;
   ```
   f.
   ```
   char ch;
   int num1,num2;
   double calificacion1, calificacion2;
   ```
   g.
   ```
   double interés, principal, capital;
   double precio, rédito;
   ```

[9] Algunos sistemas aceptarán el .68 como la tercera entrada. En todos los casos el último valor 20 es ignorado.

h.
```
char ch, letra1, letra2;
int num1,num2,num3;
```
i.
```
double temp1,temp2,temp3;
double voltios1, voltios2;
```

2. a. Escriba un programa en C++ que despliegue primero el siguiente indicador:

```
Introduzca la temperatura en grados Celsius:
```

Haga que su programa acepte un valor introducido desde el teclado y convierta la temperatura introducida a grados Fahrenheit, usando la fórmula *Fahrenheit* = (9.0 / 5.0) * *Celsius* + 32.0. Su programa deberá desplegar entonces la temperatura en grados Fahrenheit, usando un mensaje de salida apropiado.

b. Compile y ejecute el programa escrito para el ejercicio 2a. Verifique su programa calculando, en forma manual y luego usando su programa, el equivalente en Fahrenheit de los siguientes datos de prueba:

Conjunto de datos de prueba 1: 0 grados Celsius
Conjunto de datos de prueba 2: 50 grados Celsius
Conjunto de datos de prueba 3: 100 grados Celsius

Cuando esté seguro que su programa funciona en forma correcta, úselo para completar la siguiente tabla:

Celsius	Fahrenheit
45	
50	
55	
60	
65	
70	

3. Escriba, compile y ejecute un programa en C++ que despliegue el siguiente indicador:

```
Introduzca el radio de un círculo:
```

Después de aceptar un valor para el radio, su programa deberá calcular y desplegar el área del círculo. (*Sugerencia: área = 3.1416 * radio*^{2}) Con propósitos de prueba, verifique su programa usando una entrada de prueba de un radio de 3 pulgadas. Des-

pués de determinar en forma manual que el resultado producido por su programa es correcto, use su programa para completar la siguiente tabla:

Radio (pulg.)	Área (pulg.^{2})
1.0	
1.5	
2.0	
2.5	
3.0	
3.5	

4. *a.* Escriba, compile y ejecute un programa en C++ que despliegue los siguientes indicadores:

```
Introduzca las millas recorridas:
Introduzca los galones de gasolina consumidos:
```

Después que se despliegue cada indicador, su programa deberá usar una instrucción `cin` para aceptar datos desde el teclado para el indicador desplegado. Después que se haya introducido el número de galones de gasolina consumidos, su programa deberá calcular y desplegar las millas por galón obtenidas. Este valor deberá ser incluido en un mensaje apropiado y calculado usando a ecuación *millas por galón = millas/galones* consumidos. Verifique su programa usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: Millas = 276, Gasolina = 10 galones
Conjunto de datos de prueba 2: Millas = 200, Gasolina = 15.5 galones

Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

Millas recorridas	Galones consumidos	MPG
250	16.00	
275	18.00	
312	19.54	
296	17.39	

b. Para el programa escrito para el ejercicio 4a, determine cuántas ejecuciones de verificación se requieren para asegurar que el programa funciona en forma correcta y dé una razón que apoye su respuesta.

5. a. Escriba, compile y ejecute un programa en C++ que despliegue los siguientes indicadores:

```
Introduzca un número:
Introduzca un segundo número:
Introduzca un tercer número:
Introduzca un cuarto número:
```

Después que se despliega cada indicador, su programa deberá usar una instrucción cin para aceptar un número desde el teclado para el indicador desplegado. Después que se ha introducido el cuarto número, su programa deberá calcular y desplegar el promedio de los números. El promedio deberá incluirse en un mensaje apropiado. Verifique el promedio desplegado por su programa usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: 100, 100, 100, 100
Conjunto de datos de prueba 2: 100, 0, 100, 0

Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

Números	Promedio
92, 98, 79, 85	
86, 84, 75, 86	
63, 85, 74, 82	

b. Repita el ejercicio 5a, asegurándose que usa el mismo nombre de variable, numero, para cada entrada de número. También use la variable `suma` para la suma de los números. (*Sugerencia:* para hacer esto, puede usar la instrucción `suma = suma + numero` después que se ha aceptado cada número. Repase el material sobre acumulación presentado en la sección 2.3.)

6. a. Escriba, compile y ejecute un programa en C++ que calcule y despliegue el valor de polinomio de segundo orden $ax^2 + bx + c$ para valores introducidos por el usuario de los coeficientes a, b, c y la variable x. Haga que su programa despliegue primero un mensaje informando al usuario que realizará el programa, y luego despliegue indicadores apropiados para avisar al usuario que introduzca los datos deseados. (*Sugerencia:* use un indicador como `Introduzca el coeficiente del término x al cuadrado:`)

b. Verifique el resultado producido por su programa para el ejercicio 6a usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: $a = 0$, $b = 0$, $c = 22$, $x = 56$
Conjunto de datos de prueba 2: $a = 0$, $b = 22$, $c = 0$, $x = 2$
Conjunto de datos de prueba 3: $a = 22$, $b = 0$, $c = 0$, $x = 2$
Conjunto de datos de prueba 4: $a = 2$, $b = 4$, $c = 5$, $x = 2$
Conjunto de datos de prueba 5: $a = 5$, $b = -3$, $c = 2$, $x = 1$

Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

a	b	c	x	valor del polinomio
2.0	17.0	–12.0	1.3	
3.2	2.0	15.0	2.5	
3.2	2.0	15.0	–2.5	
–2.0	10.0	0.0	2.0	
–2.0	10.0	0.0	4.0	
–2.0	10.0	0.0	5.0	
–2.0	10.0	0.0	6.0	
5.0	22.0	18.0	8.3	
4.2	–16	–20	–5.2	

7. El número de bacterias, B, en un cierto cultivo que es sometido a refrigeración puede aproximarse por la ecuación $B = 300000\ e^{-0.032t}$, donde e es el número irracional 2.71828 (redondeado a cinco lugares decimales), conocido como número de Euler, y t es el tiempo, en horas, que se ha refrigerado el cultivo. Usando esta ecuación, escriba, compile y ejecute un programa simple en C++ que indique al usuario que introduzca un valor de tiempo, calcule el número de bacterias en el cultivo y despliegue el resultado. Con propósitos de prueba, verifique su programa usando una entrada de prueba de 10 horas. Cuando haya verificado la operación de su programa, úselo para determinar el número de bacterias en el cultivo después de 12, 18, 24, 36, 48 y 72 horas.

8. Escriba, compile y ejecute un programa que calcule y despliegue el valor de la raíz cuadrada de un número real introducido por el usuario. Verifique su programa calculando las raíces cuadradas de los siguientes datos: 25, 16, 0 y 2. Cuando complete su verificación, use su programa para determinar la raíz cuadrada de 32.25, 42, 48, 55, 63 y 79.

9. Escriba, compile y ejecute un programa que calcule y despliegue la raíz cuarta de un número introducido por el usuario. Recuerde del álgebra elemental que la raíz cuarta de un número puede encontrarse elevando el número a la potencia ¼. (*Sugerencia:* no utilice la división de números enteros; ¿puede ver por qué?) Verifique su programa calculando la raíz cuarta de los siguientes datos: 81, 16, 1 y 0. Cuando haya completado su verificación, use su programa para determinar la raíz cuarta de 42, 121, 256, 587, 1240 y 16 256.

10. Para el circuito en serie mostrado en la figura 3.17, la baja del voltaje, V_2, a través del resistor, R_2, y de la potencia, P_2, enviada al resistor está dada por las ecuaciones $V_2 = I\ R_2$ y $P_2 = I\ V_2$, donde $I = E\ /(R_1 + R_2)$. Usando estas ecuaciones, escriba, compile y ejecute un programa en C++ que indique al usuario que introduzca los valores de E,

R_1 y R_2, calcule la baja de voltaje y la potencia enviadas a R_2, y despliegue los resultados. Verifique su programa usando los datos de prueba: E = 10 voltios, R_1 = 100 ohmios y R_2 = 200 ohmios. Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

E (voltios)	R_1 (ohmios)	R_2 (ohmios)	Baja de voltaje (voltios)	Potencia enviada (vatios)
10	100	100		
10	100	200		
10	200	200		
20	100	100		
20	100	200		
20	200	200		

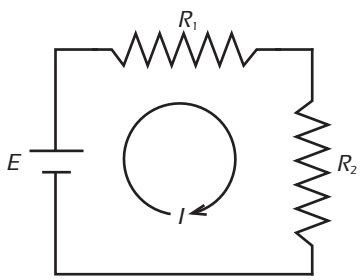


Figura 3.17 *Cálculo de la baja de voltaje.*

11. Escriba, compile y ejecute un programa en C++ que calcule la resistencia combinada de tres resistores paralelos. Los valores de cada resistor deberían ser aceptados usando una instrucción `cin` (use la fórmula para la resistencia combinada dada en el ejercicio 9 de la sección 3.2). Verifique la operación de su programa usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: R_1 = 1000, R_2 = 1000 y R_3 = 1000.
Conjunto de datos de prueba 2: R_1 = 1000, R_2 = 1500 y R_3 = 500.

Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

R1 (voltios)	R2 (ohmios)	R3 (ohmios)	Resistencia combinada (ohmios)
3000	3000	3000	
6000	6000	6000	
2000	3000	1000	
2000	4000	5000	
4000	2000	1000	
10000	100	100	

12. Usando instrucciones `input`, escriba, compile y ejecute un programa en C++ que acepte las coordenadas (*x* y *y*) de dos puntos. Haga que su programa determine y despliegue los puntos medios de los dos puntos (use la fórmula dada en el ejercicio 11 de la sección 3.2). Verifique su programa usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: Punto 1 = (0, 0) y Punto 2 = (16, 0)
Conjunto de datos de prueba 2: Punto 1 = (0, 0) y Punto 2 = (0, 16)
Conjunto de datos de prueba 3: Punto 1 = (0, 0) y Punto 2 = (–16, 0)
Conjunto de datos de prueba 4: Punto 1 = (0, 0) y Punto 2 = (0, –16)
Conjunto de datos de prueba 5: Punto 1 = (–5, –5) y Punto 2 = (5, 5)

Cuando haya completado su verificación, use su programa para completar la siguiente tabla.

Punto 1	Punto 2	Punto medio
(4, 6)	(16, 18)	
(22, 3)	(8, 12)	
(–10, 8)	(14, 4)	
(–12, 2)	(14, 3.1)	
(3.1, –6)	(20, 16)	
(3.1, –6)	(–16, –18)	

13. Escriba, compile y ejecute un programa en C++ que calcule y despliegue el valor del flujo de corriente a través de un circuito RC. El circuito consiste en una batería que está conectada en serie a un interruptor, un resistor y un capacitor. Cuando el interruptor se cierra, la corriente, *i*, que fluye a través del circuito está dada por la ecuación:

$$i = (E/R)\, e^{-t/\tau}$$

donde E es el voltaje de la batería (en voltios), R es la resistencia (en ohmios), τ es la constante de tiempo y t es el tiempo (en segundos) desde que el interruptor fue cerrado.

El programa deberá indicar al usuario que introduzca valores apropiados y use instrucciones de entrada para aceptar los datos. Al construir los indicadores, use instrucciones como `Introduzca el voltaje de la batería`. Verifique la operación de su programa calculando, en forma manual, la corriente para los siguientes datos de prueba:

Conjunto de datos de prueba 1: Voltaje = 20 voltios, R = 10 ohmios, τ = 0.044, t = 0.023 segundos.

Conjunto de datos de prueba 2: Voltaje = 35 voltios = 35, R = 10 ohmios, τ= 0.16, t = 0.067 segundos.

Cuando haya completado su verificación, use su programa para determinar el valor de la corriente para los siguientes casos:

a. Voltaje = 35 voltios, R = 10 ohmios, τ = 0.16, t = 0.11 segundos.
b. Voltaje = 35 voltios, R = 10 ohmios, τ = 0.16, t = 0.44 segundos.
c. Voltaje = 35 voltios, R = 10 ohmios, τ = 0.16, t = 0.83 segundos.
d. Voltaje = 15 voltios, R = 10 ohmios, τ = 0.55, t = 0.11 segundos.
e. Voltaje = 15 voltios, R = 10 ohmios, τ = 0.55, t = 0.44 segundos.
f. Voltaje = 15 voltios, R = 10 ohmios, τ = 0.55, t = 0.067 segundos.
g. Voltaje = 6 voltios, R = 1000 ohmios, τ = 2.6, t = 12.4 segundos.

14. El programa 3.12 indica al usuario que introduzca dos números, donde el primer valor introducido es almacenado en `num1` y el segundo valor es almacenado en `num2`. Usando este programa como punto de partida, escriba un programa que intercambie los valores almacenados en las dos variables.

15. Escriba un programa en C++ que indique al usuario que introduzca un número. Haga que su programa acepte el número como un entero y lo despliegue de inmediato usando una llamada al objeto `cout`. Ejecute su programa tres veces. La primera vez que ejecute el programa introduzca un número entero válido, la segunda vez introduzca un número de precisión doble y la tercera vez introduzca un carácter. Usando el despliegue de salida, vea qué número aceptó en realidad su programa de los datos que introdujo.

16. Repita el ejercicio 15 pero haga que su programa declare la variable usada para almacenar el número como una variable de precisión doble. Ejecute el programa tres veces. La primera vez introduzca un entero, la segunda vez introduzca un número de precisión doble y la tercera vez introduzca un carácter. Usando el despliegue de salida, siga la pista de cuál número aceptó en realidad su programa de los datos que introdujo. ¿Qué sucedió, y por qué?

17. a. ¿Por qué cree que los programas de aplicaciones exitosos contienen verificaciones de validez extensas de los datos de entrada? (*Sugerencia:* revise los ejercicios 16 y 17.)
b. ¿Cuál piensa que es la diferencia entre una verificación del tipo de datos y una verificación de lo razonable que son esos datos?
c. Suponga que un programa requiere que el usuario introduzca un día, mes y año. ¿Cuáles son algunas verificaciones que podría hacer en los datos introducidos?

3.5 CONSTANTES SIMBÓLICAS

Ciertas constantes usadas dentro de un programa tienen un significado más general que es reconocido fuera del contexto del programa. Los ejemplos de estos tipos de constantes incluyen el número 3.1416, el cual es π con una precisión de cuatro lugares decimales; 32.2 pies/sec^2, lo cual es la constante gravitacional; y el número 2.71828, el cual es el número de Euler con una precisión de cinco lugares decimales.

El significado de otras constantes que aparecen en un programa se define estrictamente dentro del contexto de la aplicación que se está programando. Por ejemplo, al determinar el peso de objetos de varios tamaños, la densidad del material que se está usando adquiere un significado especial. Por sí mismos los números de densidad son bastante ordinarios, pero en el contexto de una aplicación particular tienen un significado especial. Números como éstos son conocidos a veces por los programadores como ***números mágicos***. Cuando el mismo número mágico aparece de manera repetida dentro del mismo programa se vuelve una fuente potencial de error, por lo que se tendrá que cambiar la constante. Sin embargo, múltiples cambios están sujetos a error, con un solo valor que se pase por alto y no sea cambiado, el resultado obtenido cuando se ejecuta el programa será incorrecto y la fuente de error difícil de localizar.

Para evitar el problema de tener un número mágico diseminado en un programa en muchos lugares y permitir la identificación clara de constantes universales, como π, C++ permite al programador darle a estas constantes su propio nombre simbólico. Entonces, en lugar de usar el número en todo el programa, se usa en cambio un nombre simbólico. Si alguna vez se tiene que cambiar el número, el cambio sólo necesita hacerse una vez en el punto donde el nombre simbólico es equiparado con el valor numérico real. Equiparar números con nombres simbólicos se logra usando el calificador de declaración `const`. El calificador `const` especifica que el identificador declarado sólo puede leerse después que es inicializado; no puede cambiarse. Tres ejemplos que usan este calificador son

```
const double PI = 3.1416;
const double DENSIDAD = 0.238;
const int MAXNUM = 100;
```

La primera instrucción de declaración crea una constante de doble precisión llamada PI y la inicializa con el valor 3.1416, mientras la segunda instrucción de declaración crea la constante de precisión doble llamada DENSIDAD y la inicializa con 0.238. Por último, la tercera declaración crea una constante entera llamada MAXNUM y la inicializa con el valor 100.

Una vez que se crea e inicializa un identificador `const`, el valor almacenado en él no puede cambiarse. Por tanto, para propósitos prácticos, el nombre de la constante y su valor se vinculan por la duración del programa que los declara.

Aunque hemos escrito los identificadores `const` en letras mayúsculas, podrían haberse usado letras minúsculas. Sin embargo, es común en C++, usar letras mayúsculas para los identificadores `const` a fin de identificarlos con facilidad. Luego, siempre que un programador vea letras mayúsculas en un programa, sabrá que el valor de la constante no puede cambiarse.

Una vez declarado, puede usarse un identificador `const` en cualquier instrucción C++ en lugar del número que representa. Por ejemplo, las instrucciones de asignación

```
circum = 2 * PI * radio;
peso = DENSIDAD * volumen;
```

son válidas. Por supuesto que estas instrucciones deben aparecer después de las declaraciones para todas sus variables. En vista que una declaración `const` equipara de manera efectiva un valor constante con un identificador, y el identificador puede ser usado como un reemplazo directo para su constante inicializada, estos identificadores se conocen por lo general como ***constantes simbólicas*** o ***constantes nombradas***. Se usarán estos términos en forma intercambiable.

Colocación de instrucciones

En esta etapa se han introducido una variedad de tipos de instrucciones. La regla general en C++ para la colocación de las instrucciones es tan sólo que una variable o constante simbólica debe declararse antes que pueda ser usada. Aunque esta regla permite que tanto las directivas del preprocesador como las instrucciones de declaración sean colocadas a lo largo de un programa, hacerlo así resultará en una estructura de programa muy pobre. Como una buena forma de programación, deberá usarse el siguiente orden en las instrucciones:

```
directivas del preprocesador

int main()
{
  constantes simbólicas
  declaraciones de la función principal

  otras instrucciones ejecutables

  return valor
}
```

Conforme se introduzcan nuevos tipos de instrucción se expandirá esta estructura de colocación para acomodarlos. Hay que observar que las instrucciones de comentario pueden entremezclarse con libertad en cualquier parte dentro de esta estructura básica.

El programa 3.14 ilustra esta estructura básica y usa una constante simbólica para calcular el peso de un cilindro de acero. La densidad del acero es 0.284 lb/pulg^3.

Programa 3.14

```
// Este programa determina el peso de un cilindro de acero
//   al multiplicar el volumen del cilindro por su densidad
//  El volumen del cilindro está dado por la fórmula PI * pow(radio,2) * altura.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  const double PI = 3.1416;
  const double DENSIDAD = 0.284;
  double radio, altura, peso;

  cout << "Introduzca el radio del cilindro (en pulgadas): ";
  cin  >> radio;
  cout << "Introduzca la altura del cilindro (en pulgadas): ";
  cin  >> altura;
  weight = DENSIDAD * PI * pow(radio,2) * altura;
  cout << setiosflags(ios:: fixed)
         << setiosflags(ios::showpoint)
         << setprecision(4)
         << "El cilindro pesa " << peso << " libras" << endl;

  return 0;
}
```

Se puede observar en el programa 3.14 que se han definido dos constantes simbólicas: PI y DENSIDAD. La siguiente ejecución se hizo para determinar el peso de un cilindro con un radio de 3 pulgadas y una altura de 12 pulgadas.

```
Introduzca el radio del cilindro (en pulgadas): 3
Introduzca la altura del cilindro (en pulgadas): 12
El cilindro pesa 96.3592 libras
```

La ventaja de usar la constante nombrada PI en el programa 3.14 es que identifica con claridad el valor de 3.1416 en términos reconocibles por la mayoría de las personas. La ventaja de usar la constante nombrada DENSIDAD es que permite que el programador cambie el valor de la densidad por otro material sin tener que buscar por todo el programa para ver dónde se usa la densidad. Por supuesto, si van a ser considerados muchos materiales diferentes, la densidad deberá cambiarse de una constante simbólica a una variable. Surge una interrogante natural, entonces, sobre la diferencia entre constantes simbólicas y variables.

El valor de una variable puede alterarse en cualquier parte dentro de un programa. Por su naturaleza, una constante nombrada es un valor constante que no debe alterarse después que se ha definido. Nombrar una constante en lugar de asignar el valor a una variable asegura que el valor en la constante no pueda ser alterada en lo subsiguiente. Siempre que aparece una constante nombrada en una instrucción tiene el mismo efecto que la constante que representa. Por tanto, DENSIDAD en el programa 3.14 tan sólo es otra forma de representar el número 0.284. En vista que DENSIDAD y el número 0.284 son equivalentes, el valor de DENSIDAD no puede cambiarse después dentro del programa. Una vez que se ha definido DENSIDAD como una constante, una instrucción de asignación como

```
DENSIDAD = 0.156;
```

carece de significado y producirá un mensaje de error, porque DENSIDAD no es una variable. En vista que DENSIDAD es sólo un sustituto para el valor 0.284, esta instrucción es equivalente a escribir la expresión inválida 0.284 = 0.156. Además de usar una instrucción `const` para nombrar constantes, como en el programa 3.14, esta instrucción también puede utilizarse para equiparar el valor de una expresión constante con un nombre simbólico. Una expresión constante es una expresión que consta sólo de operadores y constantes. Por ejemplo, la instrucción

```
const double GRAD_A_RAD = 3.1416/180.0;
```

equipara el valor de la expresión constante 3.1416/180.0 con el nombre simbólico `GRAD_A_RAD`. El nombre simbólico, como siempre, puede ser usado en cualquier instrucción después de su definición. Por ejemplo, en vista que la expresión 3.1416/180.0 se requiere para convertir grados a radianes, el nombre simbólico seleccionado para este factor de conversión puede usarse en forma conveniente siempre que se requiera una conversión de este tipo. Por tanto, en la instrucción de asignación

```
altura = distancia * sin(angulo * GRAD_A_RAD);
```

la constante simbólica `GRAD_A_RAD` se usa para convertir el valor de la medida de un ángulo en radianes.

Una constante nombrada definida con anterioridad también puede usarse en una instrucción `const` subsiguiente. Por ejemplo, la siguiente secuencia de instrucciones es válida:

```
const double PI = 3.1416;
const double  GRAD_A_RAD = PI / 180.0;
```

En vista que la constante 3.1416 había sido equiparada con el nombre simbólico PI, puede usarse de manera legítima en cualquier definición subsiguiente, aun dentro de otra instrucción `const`. El programa 3.15 usa la constante nombrada `GRAD_A_RAD` para convertir un ángulo introducido por un usuario, en grados, en su medida equivalente en radianes para que la use la función `sin`.

Programa 3.15

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

int main()
{
  const double PI = 3.1416;
  const double GRAD_A_RAD = PI/180.0;
  double angulo;

  cout << "Introduzca el ángulo (en grados): ";
  cin >> angulo;
  cout << setiosflags(ios:: fixed)
       << setiosflags(ios::showpoint)
       << setprecision(4)
       << "El seno del ángulo es " << sin(angulo * GRAD_A_RAD) << endl;

  return 0;
}
```

La siguiente muestra de ejecución se hizo usando el programa 3.15.

```
Introduzca el ángulo (en grados): 30
El seno del ángulo es 0.5000
```

Aunque se ha usado el calificador `const` para construir constantes simbólicas, este tipo de datos se encontrará una vez más en el capítulo 6, donde se mostrará que son útiles como argumentos de función al asegurar que el argumento no es modificado dentro de la función.

Ejercicios 3.5

1. Modifique el programa 3.9 para usar la constante nombrada GRAV en lugar del valor 32.2 usado en el programa. Compile y ejecute su programa para verificar que produce el mismo resultado mostrado en el texto.
2. Vuelva a escribir el siguiente programa para usar la constante nombrada FACTOR en lugar de la expresión (5.0/9.0) contenida dentro del programa.

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

int main()
```

```
{
  double fahren, celsius;
  cout << "Introduzca una temperatura en grados Fahrenheit: ";
  cin  >> fahren;
  celsius = (5.0/9.0) * (fahren - 32.0);
  cout << "La temperatura Celsius equivalente es "
       << celsius << endl;

  return 0;
}
```

3. Vuelva a escribir el siguiente programa para usar la constante simbólica PRIMA en lugar del valor 0.04 contenido dentro del programa.

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

int main()
{
  float prima, cantidad, interés;
  prime = 0.04;      // tasa de interés de la prima
  cout << <Introduzca la cantidad: ";
  cin  >> cantidad;
  interés = prima * cantidad;
  cout << "El interés ganado es"
       << interés << " dólares" << endl;

  return 0;
}
```

4. Vuelva a escribir el siguiente programa de modo que la variable `voltios` sea cambiada a una constante simbólica.

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

int main()
{
  double corriente, resistencia, voltios;

  voltios = 12;
  cout << " Introduzca la resistencia: ";
  cin >> resistencia;
  corriente = voltios / resistencia;
  cout << "La corriente es " << corriente << endl;

  return 0;
}
```

3.6 APLICACIONES

En esta sección se presentan dos aplicaciones para ilustrar más a fondo tanto el uso de la instrucción `cin` para aceptar datos introducidos por el usuario como el uso de la biblioteca de funciones para realizar cálculos.

Aplicación 1: Lluvia ácida

El uso de carbón como la fuente principal de energía por vapor comenzó con la Revolución Industrial. En la actualidad el carbón es una de las fuentes principales de generación de energía eléctrica en muchos países industrializados.

Desde mediados del siglo XIX se ha sabido que el oxígeno usado en el proceso de combustión se combina con el carbono y el azufre en el carbón para producir dióxido de carbono y dióxido de azufre. Cuando estos gases se liberan en la atmósfera el dióxido de azufre se combina con el agua y el oxígeno en el aire para formar ácido sulfúrico, el cual es transformado en iones hidronio y sulfatos separados (véase la figura 3.18). Son los iones hidronio en la atmósfera que caen a la tierra, como componentes de la lluvia, los que cambian los niveles de acidez de lagos y bosques.

El nivel de ácido de la lluvia y lagos se mide en una escala de pH usando la fórmula

$$\text{pH} = -\log_{10} (\text{concentración de iones hidronio})$$

donde la concentración de iones hidronio se mide en unidades de moles/litro. Un valor de pH de 7 indica un valor neutral (ni ácido ni alcalino), mientras niveles por debajo de 7 indican la presencia de un ácido, y niveles por encima de 7 indican la presencia de una sustancia alcalina. Por ejemplo, el ácido sulfúrico tiene un valor de pH de aproximadamente 1, la lejía tiene un valor de pH de aproximadamente 13, y el agua de manera típica tiene un valor de pH de 7. La vida marina por lo general no puede sobrevivir en agua con un nivel de pH por debajo de 4.

Usando la fórmula para el pH, se escribirá un programa en C++ que calcula el nivel de pH de una sustancia con base en un valor introducido por un usuario para la concentración de iones hidronio. Usando el procedimiento de desarrollo descrito en la sección 2.6 tenemos los siguientes pasos.

Paso 1 *Analizar el problema*

Aunque el planteamiento del problema proporciona información técnica sobre la composición de la lluvia ácida, desde un punto de vista de programación éste es un problema bastante simple. Aquí sólo se requiere una salida (un nivel de pH) y una entrada (la concentración de iones hidronio).

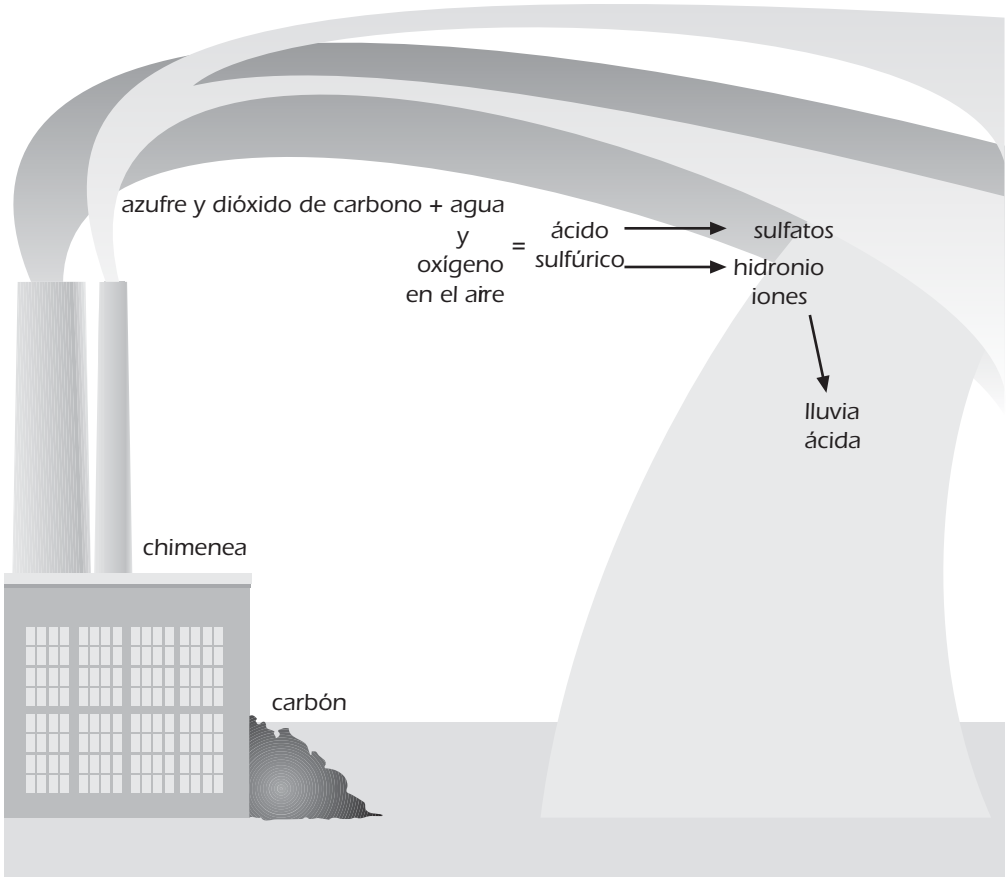


Figura 3.18 *La formación de lluvia ácida.*

Paso 2 Desarrollar una solución

El algoritmo requerido para transformar la entrada en la salida requerida es un uso bastante sencillo de la fórmula del pH que se proporciona. La representación en seudocódigo del algoritmo completo para introducir los datos de entrada, procesar los datos para producir la salida deseada y desplegar la salida es:

> ***Desplegar un indicador para introducir un nivel de concentración de iones.***
> ***Leer un valor para el nivel de concentración.***
> ***Calcular un nivel de pH usando la fórmula dada.***
> ***Desplegar el valor calculado.***

Para asegurar que entendemos la fórmula usada en el algoritmo, haremos un cálculo manual. El resultado de este cálculo puede usarse luego para verificar el resultado producido por el programa. Suponiendo una concentración de hidronio de 0.0001 (cualquier valor es útil), el nivel de pH se calcula como $-\log_{10} 10^{-4}$. Ya sea que sepa que el logaritmo de 10 elevado a una po-

tencia es la potencia misma, o usando una tabla de logaritmos, el valor de esta expresión es $-(-4) = 4$.

Paso 3 *Codificar la solución*

El programa 3.16 describe el algoritmo seleccionado en C++. La elección de los nombres de las variables es arbitrario.

Programa 3.16

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

int main()
{
  double hidronio, nivelpH;

  cout << "Introduzca la concentración de iones hidronio: ";
  cin  >> hidronio;
  nivelpH = -log10(hidronio);
  cout << "El nivel de pH es " << nivelpH << endl;

  return 0;
}
```

El programa 3.16 comienza con dos instrucciones de preprocesador `#include`, seguidos por la función `main()`. Dentro de `main()`, una instrucción de declaración declara dos variables de punto flotante, `hidronio` y `nivelpH`. El programa despliega entonces un indicador solicitando datos de entrada del usuario. Después que se despliega el indicador, se usa una instrucción `cin` para almacenar los datos introducidos en la variable `hidronio`. Por último, se calcula un valor para `nivelpH`, usando la función logarítmica de biblioteca, y se despliega. Como siempre, el programa es terminado con una llave de cierre.

Paso 4 *Probar y corregir el programa*

Una ejecución de prueba del programa 3.16 produjo lo siguiente:

```
Introduzca el nivel de concentración de iones hidronio: 0.0001
El nivel de pH es 4
```

Debido a que el programa realiza un cálculo sencillo, y el resultado de esta ejecución de prueba concuerda con nuestro cálculo manual previo, el programa se ha probado por completo. Ahora puede usarse para calcular el nivel de pH de otras concentraciones de hidronio con confianza en que los resultados producidos son precisos.

Aplicación 2: Aproximación a la función exponencial

La función exponencial e^x, donde e se conoce como el número de Euler (y tiene el valor 2.718281828459045. . .) aparece muchas veces en descripciones de fenómenos naturales. Por ejemplo, la descomposición radiactiva, el crecimiento de la población y la curva normal (en forma de campana) usada en aplicaciones estadísticas pueden describirse usando esta función.

El valor de e^x puede aproximarse usando la serie[10]

$$1+\frac{x}{1}+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\frac{x^5}{120}+\frac{x^6}{720}\cdots$$

Usando este polinomio como base, escriba un programa que aproxime e elevado a un valor de x introducido por un usuario utilizando los primeros cuatro términos de esta serie. Para cada aproximación despliegue el valor calculado por la función exponencial de C++, `exp()`, el valor aproximado y la diferencia absoluta entre los dos. Asegúrese de verificar su programa usando un cálculo manual. Una vez que esté completa la verificación, use el programa para aproximar e^4. Usando el procedimiento de desarrollo descrito en la sección 2.6 se llevan a cabo los siguientes pasos.

Paso 1 Analizar el problema

El planteamiento del problema especifica que se van a hacer cuatro aproximaciones, usando uno, dos, tres y cuatro términos del polinomio de aproximación, respectivamente. Para cada aproximación se requieren tres valores de salida: el valor de e^x producido por la función exponencial, el valor aproximado y la diferencia absoluta entre los dos valores. La figura 3.19 ilustra, en forma simbólica, la estructura del despliegue de salida requerido.

```
                e^x                    Aproximación       Diferencia
valor de la función en biblioteca  1er valor aproximado  1a diferencia
valor de la función en biblioteca   2o valor aproximado  2a diferencia
valor de la función en biblioteca  3er valor aproximado  3a diferencia
valor de la función en biblioteca   4o valor aproximado  4a diferencia
```

Figura 3.19 *Despliegue de salida requerido.*

La salida indicada en la figura 3.19 puede usarse para darse una "idea" de cómo se verá el programa. Al presentarse que cada línea en el despliegue sólo puede ser producida al ejecutar una instrucción `cout`, deberá quedar claro que deben ejecutarse cuatro de estas instrucciones. Además, en vista que cada línea de salida contiene tres valores calculados, cada instrucción `cout` tendrá tres elementos en su lista de expresión.

La única entrada al programa consiste en el valor de x. Por supuesto, esto requerirá un solo indicador y una instrucción `cin` para introducir el valor necesario.

[10]La fórmula de la que se deriva ésta es

$$e^x=\frac{x^0}{0!}+\frac{x^1}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots+\frac{x^n}{n!}$$

Paso 2 Desarrollar una solución

Antes que puedan calcularse los elementos de salida, será necesario hacer que el programa le indique al usuario que introduzca un valor de x y luego haga que el programa acepte el valor introducido. El despliegue de salida real consiste de dos líneas seguidas por cuatro líneas de datos calculados. Las líneas de título pueden producirse usando dos instrucciones `cout`. Ahora veamos cómo se producen los datos reales que se están desplegando.

El primer elemento en la primera línea de salida de datos ilustrada en la figura 3.19 puede obtenerse usando la función de biblioteca `exp()`. El segundo elemento en esta línea, la aproximación a e^x, puede obtenerse usando el primer término en el polinomio que se dio en la especificación del programa. Por último, el tercer elemento en la línea puede calcularse usando la función de biblioteca `abs()` en la diferencia entre los primeros dos elementos. Cuando se calculan todos estos elementos, puede usarse una sola instrucción `cout` para desplegar los tres resultados en la misma línea.

La segunda línea de salida ilustrada en la figura 3.19 despliega el mismo tipo de elementos que la primera línea, excepto que la aproximación a e^x requiere el uso de dos términos del polinomio de aproximación. Hay que observar que además que el primer elemento en la segunda línea, el valor obtenido por la función `exp()`, es el mismo que el primer elemento en la primera línea. Esto significa que este elemento no tiene que recalcularse y tan sólo puede desplegarse una segunda vez el valor calculado para la primera línea. Una vez que se han calculado los datos para la segunda línea, puede usarse una sola instrucción `cout` para desplegar los valores requeridos.

Por último, sólo el segundo y tercer elementos en las últimas dos líneas de salida mostradas en la figura 3.19 necesitan recalcularse, en vista que el primer elemento en estas líneas es el mismo que se calculó antes para la primera línea.

Por tanto, para este problema, el algoritmo completo descrito en seudocódigo es

Desplegar un indicador para el valor de entrada de x
Leer el valor de entrada
Desplegar las líneas de encabezado
Calcular el valor exponencial de x usando la función exp()
Calcular la primera aproximación
Calcular la primera diferencia
Imprimir la primera línea de salida
Calcular la segunda aproximación
Calcular la segunda diferencia
Imprimir la segunda línea de salida
Calcular la tercera aproximación
Calcular la tercera diferencia
Imprimir la tercera línea de salida
Calcular la cuarta aproximación
Calcular la cuarta diferencia
Imprimir la cuarta línea de salida

Para asegurar que entendemos el procesamiento usado en el algoritmo se hará un cálculo manual. El resultado de este cálculo puede utilizarse luego para verificar el resultado producido por el programa que escribimos. Para propósitos de prueba se usará un valor de 2 para x, el cual produce las siguientes aproximaciones.

Usando el primer término del polinomio la aproximación es

$$e^2 = 1$$

Usando los primeros dos términos del polinomio la aproximación es

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

Usando los primeros tres términos del polinomio la aproximación es

$$e^2 = 3 + \frac{2^2}{2} = 5$$

Usando los primeros cuatro términos del polinomio la aproximación es

$$e^2 = 5 + \frac{2^3}{6} = 6.3333$$

Hay que observar que al usar cuatro términos del polinomio no fue necesario recalcular el valor de los primeros tres términos; en cambio, se usó el valor calculado antes.

Paso 3 Codificar la solución

El programa 3.17 representa una descripción del algoritmo seleccionado en C++.

Programa 3.17

```
// este programa aproxima la funcion e elevada a la potencia x
// usando uno, dos, tres y cuatro terminos de un polinomio de aproximacion
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x, val_func, aprox, diferencia;

  cout << "\nIntroduzca un valor de x: ";
  cin  >> x;

    // imprimir las dos lineas del titulo
  cout << " e a la x          Aproximacion       Diferencia\n";
  cout << "-------------     -------------      -------------\n";

  val_func = exp(x);      // utilizar la funcion de biblioteca

    // calcular la primera aproximacion
```

(*continúa*)

(Continuación)

```
    aprox = 1;
    diferencia = abs(val_func - aprox);
    cout << setw(10) << setiosflags(ios::showpoint) << val_func
         << setw(18) << aprox
         << setw(18) << diferencia << endl;

      // calcular la segunda aproximacion
    aprox = aprox + x;
    diferencia = abs(val_func - aprox);
    cout << setw(10) << setiosflags(ios::showpoint) << val_func
         << setw(18) << aprox
         << setw(18) << diferencia << endl;

      // calcular la tercera aproximacion
    aprox = aprox + pow(x,2)/2.0;
    diferencia = abs(val_func - aprox);
    cout << setw(10) << setiosflags(ios::showpoint) << val_func
         << setw(18) << aprox
         << setw(18) << diferencia << endl;

      // calcular la cuarta aproximacion
    aprox = aprox + pow(x,3)/6.0;
    diferencia = abs(val_func - aprox);
    cout << setw(10) << setiosflags(ios::showpoint) << val_func
         << setw(18) << aprox
         << setw(18) << diferencia << endl;

    return 0;
}
```

Al revisar el programa 3.17 se puede observar que el valor de entrada de x se obtiene primero. Luego se imprimen las dos líneas del título antes que se haga cualquier cálculo. Luego se calcula el valor de e^x usando la función de biblioteca `exp()` y se asigna a la variable `val_func`. Esta asignación permite que este valor sea usado en los cuatro cálculos de diferencias y se despliegue cuatro veces sin necesidad de recalcularse.

En vista que la aproximación a e^x se "acumula" usando cada vez más términos del polinomio de aproximación, sólo se calcula el término nuevo para cada aproximación y se agrega a la aproximación previa. Por último, para permitir que las mismas variables se usen de nuevo, los valores en ellas se imprimen de inmediato antes que se haga la siguiente aproximación. La siguiente es una muestra de ejecución producida por el programa 3.17.

```
Introduzca un valor de x: 2
 e a la x          Aproximación        Diferencia
-------------     -------------     -------------
  7.38906           1.00000           6.38906
  7.38906           3.00000           4.38906
  7.38906           5.00000           2.38906
  7.38906           6.33333           1.05572
```

Paso 4 Probar y corregir el programa

Las primeras dos columnas de datos de salida producidos por la muestra de ejecución concuerdan con nuestro cálculo manual. Una comprobación manual de la última columna verifica que también contiene en forma correcta la diferencia en valores entre las primeras dos columnas.

Debido a que el programa sólo ejecuta nueve cálculos, y el resultado de la ejecución de prueba concuerda con nuestros cálculos manuales, parece que el programa se ha probado por completo. Sin embargo, es importante entender que esto se debe a nuestra elección de los datos de prueba. Seleccionar un valor de 2 para x nos obligó a verificar que el programa estaba, de hecho, calculando 2 elevado a las potencias requeridas. Una elección de 0 o 1 para nuestro cálculo manual no nos habría proporcionado la verificación que necesitamos. ¿Puede ver por qué sucede así?

Usar estos últimos dos valores no probaría de manera adecuada si el programa usó la función `pow()` en forma adecuada, ¡o ni siquiera si la usó en absoluto! Es decir, podría haberse construido un programa incorrecto que no usara la función `pow()` para producir valores correctos para = 0 y x = 1, pero no para otros valores de x. Sin embargo, en vista que los datos de prueba que usamos verifican de manera adecuada el programa, podemos usarlo con confianza en los resultados producidos. Es claro, sin embargo, que la salida demuestra que para lograr cualquier nivel de precisión con el programa se requerirían más de cuatro términos.

Ejercicios 3.6

1. Introduzca, compile y ejecute el programa 3.16 en su computadora.
2. **a.** Introduzca, compile y ejecute el programa 3.17 en su computadora.
   b. Determine cuántos términos del polinomio de aproximación deberían usarse para lograr un error de menos de 0.0001 entre la aproximación y el valor de e^2 determinado por la función `exp()`.
3. Por error un estudiante escribió el programa 3.17 como sigue:

```
// este programa aproxima la funcion e elevada a la potencia x
// usando uno, dos, tres y cuatro terminos de un polinomio de aproximacion
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x, Val_func, aprox, diferencia;
```

```
    // imprimir las dos lineas del titulo
  cout << " e  a  la x       Aproximacion       Diferencia\n";
  cout << "-------------     -------------      -------------\n";

  cout << "\nIntroduzca un valor de x: ";
  cin >> x;
  val_func = exp(x);      // usar la funcion de biblioteca

    // calcular la primera aproximacion
  aprox = 1;
  diferencia = abs(val_func - aprox);
  cout << setw(10) << setiosflags(ios::showpoint) << val_func
       << setw(18) << aprox
       << setw(18) << diferencia << endl;

    // calcular la segunda aproximacion
  aprox = aprox + x;
  diferencia = abs(val_func - aprox);
  cout << setw(10) << setiosflags(ios::showpoint) << val_func
       << setw(18) << aprox
       << setw(18) << diferencia << endl;

    // calcular la tercera aproximacion
  aprox = aprox + pow(x,2)/2.0;
  diferencia = abs(val_func - aprox);
  cout << setw(10) << setiosflags(ios::showpoint) << val_func
       << setw(18) << aprox
       << setw(18) << diferencia << endl;

    // calcular la cuarta aproximacion
  aprox = aprox + pow(x,3)/6.0;
  diferencia = abs(val_func - aprox);
  cout << setw(10) << setiosflags(ios::showpoint) << val_func
       << setw(18) << aprox
       << setw(18) << diferencia << endl;

  return 0;
}
```

Determine la salida que producirá este programa.

4. El valor de π puede aproximarse con la serie

$$4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots\right)$$

Usando esta fórmula, escriba un programa que calcule y despliegue el valor de π usando 2, 3 y 4 términos de la serie.

5. a. La fórmula para la desviación normal estándar, z, usada en aplicaciones estadísticas es

$$z = \frac{x - \mu}{\sigma}$$

donde μ se refiere a un valor medio y σ a una desviación estándar. Usando esta fórmula, escriba un programa que calcule y despliegue el valor de la desviación normal estándar cuando $x = 85.3$, $\mu = 80$ y $\sigma = 4$.

b. Vuelva a escribir el programa elaborado en el ejercicio 5a para aceptar los valores de x, μ y σ como entradas del usuario mientras se está ejecutando el programa.

6. a. La ecuación de la curva normal (en forma de campana) utilizada en aplicaciones estadísticas es

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{-\left(\frac{1}{2}\right)\left[(x-\mu)/\sigma\right]^2}$$

Usando esta ecuación, y suponiendo que $\mu = 90$ y $\sigma = 4$, escriba un programa que determine y despliegue el valor de y cuando $x = 80$.

b. Vuelva a escribir el programa elaborado en el ejercicio 6a para aceptar los valores de x, μ y σ como entradas del usuario mientras se está ejecutando el programa.

7. a. Escriba, compile y ejecute un programa en C++ que calcule y despliegue el aumento de voltaje de un amplificador de tres etapas a una frecuencia de 1000 Hertz. Los aumentos de voltaje de las etapas son:

Aumento de la etapa 1: $23/[2.3^2 + (0.044f)^2]^{1/2}$
Aumento de la etapa 2: $12/[6.7^2 + (0.34f)^2]^{1/2}$
Aumento de la etapa 3: $17/[1.9^2 + (0.45f)^2]^{1/2}$

donde f es la frecuencia en Hertz. El aumento de voltaje del amplificador es el producto de los aumentos de las etapas individuales.

b. Vuelva a hacer el ejercicio 7a suponiendo que la frecuencia se introducirá cuando el programe esté en ejecución.

8. El volumen de petróleo almacenado en un tanque cilíndrico enterrado en el subsuelo a 200 pies de profundidad está determinado por la medición de la distancia de la parte superior del tanque a la superficie del petróleo. Conociendo esta distancia y el radio del tanque, el volumen de petróleo en el tanque puede determinarse usando la fórmula *volumen* = π *radio*^{2} (200 – *distancia*). Usando esta información, escriba, compile y ejecute un programa en C++ que acepte las mediciones del radio y la distancia, calcule el volumen de petróleo en el tanque y despliegue los dos valores de entrada y el volumen calculado. Verifique los resultados de su programa haciendo un cálculo manual usando los siguientes datos de prueba: radio igual a 10 pies y distancia igual a 12 pies.

9. El perímetro, el área de superficie aproximada y el volumen aproximado de una alberca están dados por las siguientes fórmulas:

perímetro = 2(largo + ancho)
*volumen = largo * ancho * profundidad promedio*
*área de superficie subterránea = 2(largo + ancho)profundidad promedio + largo * ancho*

Usando estas fórmulas como base, escriba un programa en C++ que acepte las medidas de largo, ancho y profundidad promedio, y luego calcule el perímetro, el volumen y el área de superficie subterránea de la alberca. Al escribir su programa haga los siguientes dos cálculos inmediatamente después que se han introducido los datos de entrada: *largo* * *ancho* y *largo* + *ancho*. Los resultados de estos dos cálculos deberán usarse luego, según sea apropiado, en las instrucciones de asignación para determinar el perímetro, volumen y área de superficie subterránea sin recalcularlas para cada ecuación. Verifique los resultados de su programa haciendo un cálculo manual usando los siguientes datos de prueba: largo igual a 25 pies, ancho igual a 15 pies y profundidad promedio igual a 5.5 pies. Cuando haya verificado que su programa funciona, úselo para completar la siguiente tabla.

Largo	Ancho	Profundidad	Perímetro	Volumen	Área de superficie subterránea
25	10	5.0			
25	10	5.5			
25	10	6.0			
25	10	6.5			
30	12	5.0			
30	12	5.5			
30	12	6.0			
30	12	6.5			

3.7 Errores comunes de programación

Al usar el material presentado en este capítulo, esté consciente de los siguientes errores posibles:

1. Olvidar asignar o inicializar valores para todas las variables antes que éstas se usen en una expresión. Tales valores pueden ser asignados por instrucciones de asignación, ser inicializados dentro de una instrucción de declaración o asignados en forma interactiva introduciendo los valores usando el objeto `cin`.
2. Utilizar una función matemática de biblioteca sin incluir la declaración de preprocesador `#include <cmath>` (y en un sistema basado en UNIX olvidar incluir el argumento `-lm` en la línea de comandos `cc`).
3. Utilizar una función de biblioteca sin proporcionar el número correcto de argumentos que tengan el tipo de datos apropiado.
4. Aplicar el operador de incremento o decremento a una expresión. Por ejemplo, la expresión

   ```
   (count + n)++
   ```

es incorrecta. Los operadores de incremento y decremento sólo pueden aplicarse a variables individuales.

5. Olvidar separar todas las variables transmitidas a `cin` con un símbolo de extracción, `>>`.
6. No estar dispuesto a probar un programa a fondo. Después de todo, en vista que usted escribió el programa, supone que es correcto o lo habría cambiado antes de compilarlo. Es difícil en extremo retroceder y probar con honestidad su propio software. Como programador deberá recordar en forma constante que un programa no es correcto por el solo hecho que usted piense que lo es. Encontrar errores en su propio programa es una experiencia seria, pero le ayudará a convertirse en un programador maestro.
7. Un error más exótico y menos común ocurre cuando se usan los operadores de incremento y decremento con variables que aparecen más de una vez en la misma expresión. Este error ocurre debido a que C++ no especifica el orden en el que se tiene acceso a los operandos dentro de una expresión. Por ejemplo, el valor asignado a resultado en la instrucción

   ```
   resultado = i + i++;
   ```

   es dependiente del compilador. Si su compilador tiene acceso primero al primer operando, `i`, la instrucción anterior es equivalente a

   ```
   result = 2 * i;
   i++;
   ```

   Sin embargo, si su compilador tiene acceso primero al segundo operando, `i++`, el valor del primer operando será alterado antes que se use la segunda vez y el valor $2i + 1$ es asignado al resultado. Por consiguiente, como regla general, no use el operador de incremento o decremento en una expresión cuando la variable sobre la que opera aparece más de una vez en la expresión.

3.8 RESUMEN DEL CAPÍTULO

1. Una expresión es una secuencia de uno o más operandos separados por operadores. Un operando es una constante, una variable u otra expresión. Un valor se asocia con una expresión.
2. Las expresiones se evalúan de acuerdo con la precedencia y asociatividad de los operadores usados en la expresión.
3. El símbolo de asignación, =, es un operador. Las expresiones que usa este operador asignan un valor a una variable; además, la expresión en sí adquiere un valor. En vista que la asignación es una operación en C++, son posibles múltiples usos del operador de asignación en la misma expresión.

4. El operador de incremento, `++`, agrega uno a una variable, mientras el operador de decremento, `--`, resta uno de una variable. Ambos operadores pueden ser usados como prefijos o posfijos. En la operación de prefijo la variable es aumentada (o disminuida) antes que su valor sea usado. En la operación de posfijo la variable es aumentada (o disminuida) después que se usa su valor.

5. C++ proporciona funciones de biblioteca para calcular raíz cuadrada, logaritmos y otros cálculos matemáticos. Cada programa que utilice una de estas funciones matemáticas debe incluir la instrucción `#include <cmath>` o tener una declaración de función para la función matemática antes de llamarla.

6. Todas las funciones matemáticas de biblioteca operan sobre sus argumentos para calcular un solo valor. Para usar una función de biblioteca de manera efectiva, debe saber lo que hace la función, el nombre de la función, el número y tipos de datos de los argumentos esperados por la función y el tipo de datos del valor devuelto.

7. Los datos transmitidos a una función se llaman argumentos de la función. Los argumentos son transmitidos a una función de biblioteca al incluir cada argumento, separado por comas, dentro de los paréntesis que siguen al nombre de la función. Cada función tiene sus propios requisitos para el número y tipos de datos de los argumentos que deben proporcionarse.

8. Las funciones pueden incluirse dentro de expresiones más grandes.

9. El objeto `cin` se usa para introducir datos. Este objeto acepta un flujo de datos del teclado y asigna los datos a variables. La forma general de una instrucción que utiliza `cin` es:

   ```
   cin >> var1 >> var2 . . . >> varn;
   ```

   El símbolo de extracción, `>>`, debe usarse para separar los nombres de las variables.

10. Cuando encuentra una instrucción `cin` la computadora suspende de manera temporal la ejecución de más instrucciones hasta que se hayan introducido suficientes datos para el número de variables contenidas en la instrucción `cin`.

11. Es una buena práctica de programación desplegar un mensaje, antes de una instrucción `cin`, que alerte al usuario sobre el tipo y número de elementos de datos que deben introducirse. Dicho mensaje se llama indicador.

12. Los valores pueden equipararse a una sola constante, usando la palabra clave `const`. Esto crea una constante nombrada que es de sólo lectura después que es inicializada dentro de la instrucción de declaración. Esta declaración tiene la sintaxis

    `const` *Tipodedatos NombreSimbólico* = *valorInicial;*

    y permite que se use la constante en lugar del valor inicial en cualquier parte del programa después de la declaración.

3.9 UN ACERCAMIENTO MÁS A FONDO: ERRORES DE PROGRAMACIÓN

El ideal en la programación es producir programas legibles libres de errores que funcionen en forma correcta y puedan modificarse o cambiarse con un mínimo de pruebas. Puede trabajar hacia este ideal teniendo en cuenta los diferentes tipos de errores que pueden ocurrir, cuándo se detectan de manera típica y cómo corregirlos.

Puede detectar un error en cuatro formas:

1. Antes que un programa sea compilado
2. Mientras el programa se compila
3. Mientras el programa se ejecuta
4. Después que el programa se ha ejecutado y se ha examinado la salida

Y, por extraño que parezca, en algunos casos, un error puede no detectarse en absoluto.

El método para detectar errores antes que se compile un programa se llama **verificación de escritorio**. La verificación de escritorio, la cual por lo general se lleva a cabo mientras se encuentra sentado ante un escritorio con el código enfrente de usted, se refiere al proceso de verificar el código fuente en busca de errores inmediatamente después que ha sido mecanografiado.

Los errores detectados por el compilador se conocen de manera formal como **errores en tiempo de compilación**, y los errores que ocurren mientras el programa se ejecuta se conocen de manera formal como errores en tiempo de ejecución. Otros nombres para los errores en tiempo de compilación son **errores** de **sintaxis** y **errores** de **análisis** gramatical, términos que enfatizan el tipo de error que es detectado por el compilador.

En este momento, es probable que haya encontrado numerosos errores en tiempo de compilación. Aunque los programadores principiantes tienden a frustrarse por ellos, los programadores experimentados entienden que el compilador está realizando una verificación valiosa, y que corregir los errores que detecte el compilador por lo general es fácil. Debido a que estos errores ocurren mientras se está desarrollando el programa y no mientras un usuario intenta realizar una tarea importante, nadie excepto el programador sabe que ocurrieron; los arregla y se van.

Los errores en tiempo de ejecución son más problemáticos debido a que ocurren mientras un usuario ejecuta el programa; en la mayor parte de los sistemas comerciales, el usuario no es el programador. Aunque muchos tipos de errores pueden causar un error en tiempo de ejecución, como una falla en el hardware, desde un punto de vista de programación la mayor parte de los errores en tiempo de ejecución se conocen como errores de lógica o lógica defectuosa, lo cual abarca no haber pensado lo que el programa debería hacer o no anticipar cómo un usuario puede hacer que falle el programa. Por ejemplo, si un usuario introduce datos que producen un intento de dividir un número entre cero, ocurre un error en tiempo de ejecución. Como programador, la única forma de protegerse contra errores en tiempo de ejecución es anticipar todo lo que podría hacer una persona para causar errores y someter su programa a una prueba rigurosa. Aunque los programadores principiantes tienden a culpar al usuario por un error causado al introducir datos incorrectos, los profesionales no lo hacen. Entienden que un error en tiempo de ejecución es un defecto en el producto final que puede causar daños a la reputación del programa y el programador.

Para prevenir errores en tiempo de compilación y en tiempo de ejecución, es más fructífero distinguir entre ellos basándose en lo que los causa. Como se ha señalado, los errores de

compilación también se llaman errores de sintaxis, lo cual se refiere a errores en la estructura u ortografía de una instrucción. Por ejemplo, examine las siguientes instrucciones:

```
cout << "Hay cuatro errores de sintaxis aqui\n
cot " Puede encontralos";
```

Contienen cuatro errores de sintaxis. Estos errores son los siguientes:

1. Faltan las comillas que cierran en la línea 1.
2. Falta un punto y coma para terminar en la línea 1.
3. La palabra clave `cout` está mal escrita en la línea 2.
4. Falta el símbolo de inserción, <<, en la línea 2.

Todos estos errores serán detectados por el compilador cuando el programa es compilado. Esto sucede con todos los errores de sintaxis porque violan las reglas básicas de C++; si no son descubiertos por la verificación de escritorio, el compilador los detecta y despliega un mensaje de error.[11] En algunos casos, el mensaje de error es claro y el error es obvio; en otros casos, se requiere un poco de trabajo detectivesco para entender el mensaje de error desplegado por el compilador. Debido a que los errores de sintaxis son el único tipo de error que puede detectarse en el momento de la compilación, los términos errores en tiempo de compilación y errores de sintaxis se usan de manera indistinta. En sentido estricto, sin embargo, tiempo de compilación se refiere al momento en que se detecta el error y sintaxis se refiere al tipo de error detectado.

El error en la palabra "encontralos" en la segunda instrucción no es un error de sintaxis. Aunque este error de ortografía producirá que se despliegue una línea de salida indeseable, no es una violación de las reglas sintácticas de C++. Es un **error tipográfico**, conocido por lo común como "error de dedo".

Un error lógico puede causar un error en tiempo de ejecución o producir resultados incorrectos. Estos errores se caracterizan por una salida errónea, inesperada o involuntaria que es un resultado directo de algún defecto en la lógica del programa. Estos errores, los cuales nunca son detectados por el compilador, pueden detectarse en la verificación de escritorio, al probar el programa, por accidente cuando un usuario obtiene una salida errónea mientras el programa se está ejecutando, o no detectarse en absoluto. Si el error es detectado mientras el programa está en ejecución, puede ocurrir un error en tiempo de ejecución que produce que se genere un mensaje de error, la terminación prematura del programa, o ambos.

El error de lógica más grave es causado por una comprensión incorrecta de los requerimientos totales del programa, debido a que la lógica dentro de un programa se refleja en la lógica con la que es codificado. Por ejemplo, si el propósito de un programa es calcular la fuerza de soporte de carga de una viga de acero y el programador no entiende por completo cómo se va a hacer el cálculo, qué entradas son necesarias para realizar el cálculo o qué condiciones especiales existen (como la forma en que la temperatura afecta a la viga), ocurrirá un error de lógica. Debido a que estos errores no son detectados por el compilador y con frecuencia pueden pasarse por alto en el tiempo de ejecución, siempre son más difíciles de detectar que los errores de sintaxis. Si son detectados, un error de lógica de manera típica aparece en una de dos formas predominantes. En un caso, el programa se ejecuta has-

[11] Sin embargo, puede ser que no se detecten todos al mismo tiempo. Con frecuencia, un error de sintaxis enmascara a otro error, y el segundo error es detectado después que se corrige el primer error.

ta completarse pero produce resultados incorrectos. Por lo general, los errores de lógica de este tipo son revelados por lo siguiente:

- **No hay salida.** Esto es causado por una omisión en una instrucción de salida o una secuencia de instrucciones que elude de manera inadvertida una instrucción de salida.
- **Salida poco atractiva o mal alineada.** Esto es causado por un error en una instrucción de salida.
- **Resultados numéricos incorrectos.** Esto es causado por valores incorrectos asignados a las variables usadas en una expresión, el uso de una expresión aritmética incorrecta, una omisión de una instrucción, un error de redondeo o el uso de una secuencia de instrucciones inapropiada.

Una segunda forma en que se revelan los errores de lógica es causando un error en tiempo de ejecución. Son ejemplos de este tipo de error de lógica son los intentos de dividir entre cero u obtener la raíz cuadrada de un número negativo.

Deberá planear la prueba de su programa cuidadosamente para maximizar la posibilidad de localizar errores. Siempre tenga en cuenta que *aunque una sola prueba puede revelar la presencia de un error, no verifica la ausencia de otro error*. Es decir, el hecho que un error sea revelado por la prueba, no indica que otro error no esté al acecho en alguna otra parte en el programa; además, *el hecho que una prueba no revele errores no significa que no haya errores*.

Sin embargo, una vez que descubre un error debe localizar dónde ocurre y arreglarlo. En jerga de computación, un error de programa se conoce como **bug**, y el proceso de aislar, corregir y verificar la corrección se llama **depuración**.[12]

Aunque no existen reglas inflexibles para aislar la causa de un error, pueden aplicarse algunas técnicas útiles. La primera de éstas es una técnica preventiva. Con frecuencia, muchos errores son introducidos por el programador por la premura de codificar y ejecutar un programa antes de entender qué se requiere y cómo se va a lograr el resultado. Un síntoma de esta prisa por introducir un programa en la computadora es la falta de un esbozo del programa propuesto o la falta de una comprensión detallada de lo que se requiere en realidad. Muchos errores pueden eliminarse al verificar en el escritorio una copia del programa antes de introducirlo o compilarlo.

Una segunda técnica útil es imitar a la computadora y ejecutar cada instrucción en forma manual, como lo haría la computadora. Esto significa escribir cada variable, tal como se encuentra en el programa, y enumerar el valor que debería almacenarse en la variable conforme se encuentre cada entrada e instrucción de asignación. Hacer esto agudiza sus habilidades de programación porque requiere que entienda lo que causa que suceda cada instrucción en su programa. Esta verificación se llama **rastreo del programa**.

Una tercera técnica de depuración poderosa es incluir algún código temporal en su programa que despliegue los valores de variables selectas. Si los valores desplegados son incorrectos, puede determinar qué parte de su programa los generó y hacer las correcciones necesarias.

[12]La derivación de este término es interesante. Cuando un programa dejó de ejecutarse en la computadora MARK I, en la Universidad de Harvard, en septiembre de 1945, el mal funcionamiento fue rastreado hasta un insecto muerto que había entrado en los circuitos eléctricos. La programadora, Grace Hopper, registró el incidente en su bitácora como "Primer caso real de bug (insecto) encontrado".

En la misma manera, podría agregar código temporal que despliegue los valores de todos los datos de entrada. Esta técnica se conoce como **impresión en eco** y es útil para establecer que el programa está recibiendo en forma correcta e interpretando en forma correcta los datos de entrada.

La más poderosa de todas las técnicas de depuración y rastreo es usar un programa especial llamado **depurador**. Un programa depurador puede controlar la ejecución de un programa en C++, puede interrumpir el programa C++ en cualquier punto de su ejecución y desplegar los valores de todas las variables en el punto de interrupción.

Por último, ninguna exposición de la depuración está completa sin mencionar el ingrediente primario necesario para el aislamiento y corrección exitosa de los errores. Es la actitud y espíritu con que se emprende la tarea. Después de escribir un programa, es natural que suponga que es correcto. Es difícil retroceder y probar honestamente y encontrar errores en su propio software. Como programador, debe recordar en forma constante que un programa no es correcto sólo porque usted piensa que lo es. Encontrar errores en su propio programa es una experiencia seria, pero le ayudará a que se convierta en un programador maestro. El proceso puede ser emocionante y divertido si lo enfoca como una detección de problemas con usted como el detective maestro.

Consideración de opciones de carrera

Ingeniería mecánica

En general, los ingenieros mecánicos trabajan con máquinas o sistemas que producen o aplican energía. El rango de actividades tecnológicas que se consideran parte de la ingeniería mecánica quizás es más amplio que en cualquier otro campo de la ingeniería. El campo puede subdividirse más o menos en cuatro categorías.

1. Energía. Diseño de máquinas y sistemas generadores de energía como quemadores y turbinas para generar electricidad, energía solar, sistemas de calefacción e intercambio de calor.
2. Diseño. Diseño innovador de partes o componentes de máquinas desde los más intrincados y pequeños hasta los gigantescos. Por ejemplo, los ingenieros mecánicos trabajan al lado de los ingenieros eléctricos para diseñar sistemas de control automático como los robots.
3. Automotriz. Diseño y prueba de vehículos de transporte y las máquinas usadas para fabricarlos.
4. Calefacción, ventilación, aire acondicionado y refrigeración. Diseño de sistemas para controlar nuestro ambiente tanto en interiores como en exteriores y para controlar la contaminación.

Los ingenieros mecánicos por lo general tienen estudios sólidos en materias como termodinámica, transferencia de calor, estática y dinámica y mecánica de fluidos.